Lecture Notes in Computer Science 6200

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Zohar Manna Doron A. Peled (Eds.)

Time for Verification

Essays in Memory of Amir Pnueli

Springer

Volume Editors

Zohar Manna
Stanford University, Computer Science Department
Stanford, CA 94305, USA
E-mail: zm@cs.stanford.edu

Doron A. Peled
Bar Ilan University, Department of Computer Science
Ramat Gan, 52900, Israel
E-mail: doron.peled@gmail.com

Library of Congress Control Number: 2010929623

CR Subject Classification (1998): F.2, F.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-13753-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-13753-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Amir Pnueli (1941-2009)

Preface

This volume is dedicated to the memory of Amir Pnueli: a great scientist, a colleague and a friend. Amir touched our lives in several ways. As a scientist, Amir had the exceptionally deep insight that can open a new research area with a cleverly crafted paper. Having published over 250 papers, and won the Turing Award, the highest awarded recognition in computer science, Amir is no doubt one of the most brilliant and visionary computer scientists of all times. As a colleague and research collaborator, Amir steered the entire field of research in unforeseen, original, directions. As a mentor, Amir was admired by the students that were lucky to be supervised by him. Amir will always be remembered as a colleague and a friend, who, with his kind manners and great vision has influenced and will continue to influence present and future generations of computer scientists.

April 2010

Zohar Manna
Doron Peled

Table of Contents

Modal and Temporal Argumentation Networks

Howard Barringer[1] and Dov M. Gabbay[2]

[1] School of Computer Science, University of Manchester
Oxford Road, Manchester, M13 9PL, UK
Howard.Barringer@manchester.ac.uk
[2] Department of Computer Science, King's College London
The Strand, WC2R 2LS, UK
Bar Ilan University, Israel
University of Luxembourg, Luxembourg
Dov.Gabbay@kcl.ac.uk

Abstract. The traditional Dung networks depict arguments as atomic and studies the relationships of attack between them. This can be generalised in two ways. One is to consider, for example, various forms of attack, support and feedback. Another is to add content to nodes and put there not just atomic arguments but more structure, for example, proofs in some logic or simply just formulas from a richer language. This paper offers to use temporal and modal language formulas to represent arguments in the nodes of a network. The suitable semantics for such networks is Kripke semantics. We also introduce a new key concept of usability of an argument.

1 Introduction: Concept of Usability

There are several good reasons why we should consider modal and temporal argumentation networks

1. **Time dependence of arguments.** Some arguments lose their potency with the passage of time. This is well known in political context. Politicians sometimes wait for the 'storm' to blow away, especially in matters of corruption and public protest. For example, some members of the UK Parliament (MPs) were recently exposed as claiming unjustifiably large expenses. There was a strong public protest to these findings, resulting in the resignation of some MPs. Many, however, have kept a low profile, awaiting for the public to forget. Let a represent the public protest over excessive expense claims, and b denote the standing of MPs, symbolically we may have that now a attacks b, see Figure 1, but not for long, soon a will no longer attack b but a new attack on b, from c, say political in-fighting, may occur.
   In such cases we can represent the arguments and the attacks as time dependent, $a(t), b(t)$ where t represents time. In contexts where arguments have strength (i.e. $a(t)$ is a number between 0 and 1) we can even consider the rate of change, $\frac{da}{dt}, \frac{db}{dt}$ and include it in our considerations. See [1, Section 5]. It may be convenient to represent the situation as in Figure 2. Where we do indicate attack arrows as on and off. Better still is to put labels on the attacks, e.g. $\mathbf{l}(c, b), \mathbf{l}(a, b)$, which can be on or off, and consider these labels as time dependent.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 1–25, 2010.

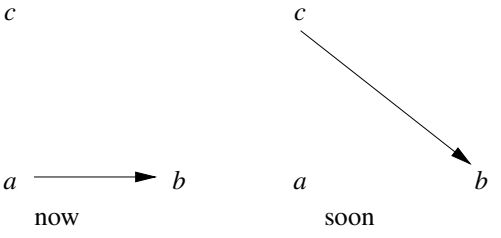


Fig. 1.

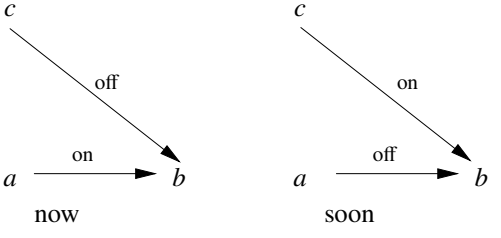


Fig. 2.

2. **Temporal facts as arguments.** Past facts or future scheduled events can also be used as arguments for the present. An argument against the trustworthiness of a person may be the facts of past betrayals. An argument in favour of a higher mortgage loan may be a scheduled increment in salary next year. Unfortunately, this does not work with UK banks. An argument against a higher mortgage loan may be the possibility of redundancy in the future. Figure 3 is an example of how a scheduled redundancy exercise in the near future can be used as an argument against a high mortgage loan now, where *d* means the event of redundancy and *m* the general argument in favour of a mortgage. We shall discuss later why it is not reasonable to encapsulate 'Future *d*' as a single argument *c* attacking *m* now. We lose

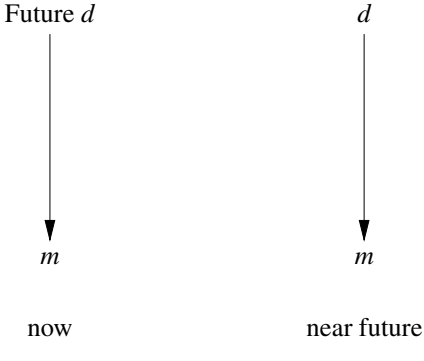


Fig. 3.

structure this way (compare with how we lose structure in propositional logic in case $\forall x A(x) \vdash A(\alpha)$. We need predicate logic to go into the structure of the sentences).

3. **Fibring arguments.** Arguments from one domain may be brought into another domain. For example, expert medical arguments may be brought in as a package into a legal argument. This may be best treated in the context of modal logic, (bringing information from the medical world into the legal world), where $\Diamond x$ means bring in information x from another world, i.e. domain. See Figure 4. Let b be the legal argument to commit the accused to a one year prison sentence for tax evasion. Let c be the medical argument that the accused has cancer. This medical argument attacks b in the legal world. A hefty fine is more appropriate. c is part of a medical network of arguments and c emerges among the winning arguments of that network. Figure 4 illustrates the situation. Of course, in the legal world $\Diamond c$ might be attacked as unacceptable evidence on the basis of some procedural errors in putting in forward (not shown in diagram).

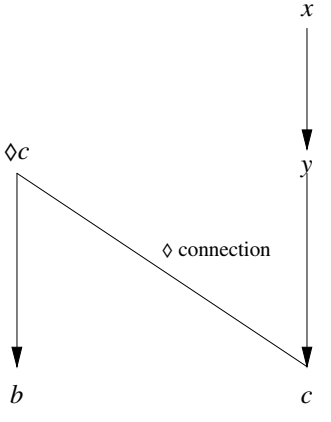


Fig. 4.

4. **Future arguments.** The possibility that an argument a may be able to defeat another argument b. We denote this by $\Diamond a$. Such possibilities are central to threats and negotiations arguments where various future scenarios are discussed. For example, don't ask for more than a 10% salary settlement as it will never be approved by the executives — there may be strong fairness arguments for claiming10% but pragmatically it will not be affordable and thus will not get approved.
5. **Past arguments.** We can use the fact that an argument c was potent in the past (denoted by Pc) to attack another current argument. Figure 5 is an example of such a configuration where $\Diamond a$ indicates that argument a is possible and Pc indicates that argument c was considered in the past, but maybe is no longer taken seriously now, yet the fact that it was a serious argument in the past is sufficient to defeat b. For example, a mother might say "I have always cared for you, you cannot abandon me now".

   For example a female employee may threaten with a possible argument claiming harassment. It may be that one cannot argue harassment now but it is not clear what

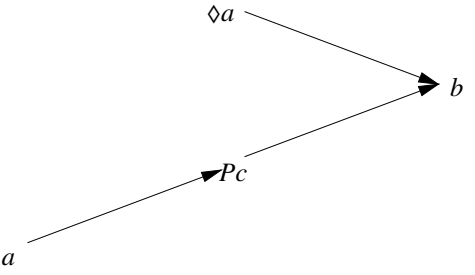


Fig. 5.

the circumstances would look like when reviewed in the future. So ◊harassment may have some force. We have had many such arguments when UK law was expected to be overruled by EU law. Many ◊(EUlaw) arguments were already defeating local UK arguments even before the EU law came into force in the UK.
Negotiations always involve evaluation of future scenarios of possible arguments and counter arguments and the possibilities of certain scenarios may be a strong argument at present.
Arguments from the point of view of tradition have always been successful in the past, e.g. we've always accepted the A'level standard as an appropriate university entrance qualification, so we continue to do so, even though many will argue that the level has dropped. We can have our doubts about the value of tradition now but yet an argument of the form "but it has always been the case that x" may still win out.
Any use of precedent is also akin to this form.

The above discussion suggests we introduce the concept of usability of arguments. We may have at a certain time or at a certain context some arguments that are talked about and are available in some real sense but these arguments cannot be used, for a variety of reasons. The formal presentation of such arguments can be to introduce them into the network but label them as unusable through a usability function h. If x is an argument then $h(x) = 1$ means it is usable and $h(x) = 0$ means it is not. The reader may ask why do we want to introduce them at all if they are not usable? Well, in the context of modal and temporal logic, it makes sense to talk about them. Maybe they were usable, maybe they will be usable or are possibly but not necessarily usable, or should have been usable, etc, etc. We give several examples.

6. **The catholic super administrator.** A UK university, operating an equal opportunities policy, advertises for a faculty administrator. There is a shortlist of three candidates and, because of a special request from one candidate, the interview date is moved.[1]

[1] As an argument for wishing a new interview date, the candidate has declared that she is getting married in the local catholic church, and these dates coincide. As a result of this correspondence, the interviewers know she is Catholic and a new bride.

The top two candidates are: a woman aged 42, who knows 15 languages and 10 computer languages and has a PhD in economics and business administration from Harvard. She has lots of experience working for government administration; the other candidate is a man of similar age, but not with as strong a background as the lady.
There is an argument for hiring the lady candidate: she is the best!
There is an argument against hiring the lady candidate: she is Catholic, aged 42, recently married, and will probably waste no time in starting a family.
This latter argument is a strong subjective argument, which, following the proper procedures, cannot be used. Indeed, one cannot mention it, let alone even think it!
h (this argument) = 0.[2]

7. **The rape.** A girl complained she was raped by a man late at night in the street. The man claimed that she gave him reason to take the view that she was willing and available. The entire incident was filmed, video and audio, by a CCTV camera.
However, this camera was installed without a licence and hence, because of a legal technicality, any evidence from the CCTV is not admissible. The evidence from the CCTV clearly and unambiguously defeated the claim of the man, but because of its inadmissability, the jury was instructed to accept the man's claim.

In both cases we present the network in the form of Figure 1 but with both admissable and inadmissable arguments, where those that are usable are marked via a function h. $h(a) = 0$ means we cannot use argument a. Therefore, b is a winning argument under h.

It is important to note that unusability is temporary and can change. Circumstances can change, the law can change, new arguments can be brought forward an what was unusable can become usable.

8. **Unusability due to defeat.** Figure 4 can be an example of unusable argument. The notation $\Diamond c$, wants to bring the cancer argument from another network, the medical network into the present network, the legal one. In the figure c is a winning argument in the medical network. c is attacked by y in the figure but is defended by x. However, it is quite possible in a complex cross network situation, that we have a $\Diamond z$ such that in the appropriate network for z, z is defeated. In this case we can view $\Diamond z$ as unusable. Again, this is not permanent and may change.
9. **Unusability due to secrecy.** It is quite possible that an argument a is defeated by an argument $a*$ which cannot be recorded explicitly in the system. In this case it may be convenient not to mention $a*$ and to simply mark a as unsuable.

2 Formal Considerations

We need to define the formal machinery and distinctions allowing us to put in context our approach to modal and temporal argumentation networks. So we define some basic notions in this section and move on to the Kripke models in the next section.

[2] There is a known case where a preferred candidate did not score as well as another candidate in an interview for a position in local government. In exceptional circumstances, the interview panel was reconvened and the outcome was that the preferred candidate's score actually fell!

Definition 1 (General Labelled Networks)

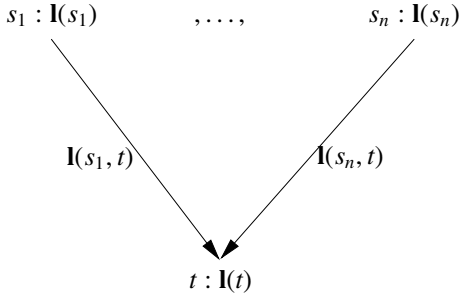
1. *A general labelled network has the form*

$$\mathbb{N} = (T, \rho, \mathbf{l}, \mathbf{f})$$

where T is a set of nodes and $\rho \subseteq T \times T$ is a binary relation on T. $\mathbf{l}$ is a labelling function on $T \cup \rho$ giving labels from a set of labels $\mathbb{L}$ (usually $\mathbb{L} = \{0, 1\}$ or $[0, 1]$). The label $\mathbf{l}(t)$, for $t \in T$ can be thought of as the strength of the node. The label $\mathbf{l}(t, s)$ for $(t, s) \in \rho$ can be thought of as the transmission label from t to s.
The functional $\mathbf{f}$ is an update functional, it updates the labelling function $\mathbf{l}$ to a new one $\mathbf{f}(\mathbf{l})$. $\mathbf{f}$ is a pair of functions, $\mathbf{f}_1, \mathbf{f}_2$, which operate on multisets of elements to give a new element. For example, the function 'maximum' or the function 'take the sum of' is such a function. We write the value $\mathbf{f}_i(x, y, z, \ldots)$ where $(x, y, z, \ldots)$ is a sequence or a multiset). Given a node t, let $\rho(t)$ be $\{s | s\rho t$ holds$\}$. Then for any t, let

$$\mathbf{f}(\mathbf{l})(t) = \mathbf{f}_1(\mathbf{l}(t), \mathbf{l}(s), \mathbf{l}(s, t), s \in \rho(t))$$
$$\mathbf{f}(\mathbf{l})(s, t) = \mathbf{f}_2(\mathbf{l}(t), \mathbf{l}(s), \mathbf{l}(s, t))$$

be new labels at t, and $at(s, t)$ given by the functional $\mathbf{f}$. $\mathbf{f}$ depends on ρ and $\mathbf{l}$, and on the labels and transmission labels, as depicted in Figure 6.



We have $\rho(t) = \{s_1, \ldots, s_n\}$
The new label is $\mathbf{f}(\mathbf{l})(t) = \mathbf{f}(\mathbf{l}(s_i), \mathbf{l}(s_i, t), i = 1, \ldots, n)$

Fig. 6.

The way $\mathbf{f}$ is calculated is not described here. The reader can compare with section 2 of our paper [1], where we give some examples of algorithms for $\mathbf{f}$ in terms of ρ and $\mathbf{l}$. $\mathbf{f}$ can be used for successive updating of the labelling of our network.

We define $\mathbf{l}_m$ for $m \geq 0$ by induction on m.

Step 0 : $\mathbf{l}_0 = \mathbf{l}$.

Step $m + 1$: $\mathbf{l}_{m+1} = \mathbf{f}(\mathbf{l}_m)$

Let ? be a fixed label in $\mathbb{L}$. We can define $\mathbf{l}_\infty$ using ? by letting $\mathbf{l}_\infty(x) = y$, if for some k we have $\mathbf{l}_m(x) = y$ for all $m \geq k$, and otherwise let $\mathbf{l}_\infty(x) =$?. x is either a node $t \in T$ or a connection $(s, t) \in \rho$.

We now have the machinery to look at argumentation networks and we use the Caminada labelling for them [9].

Definition 2 (Argumentation Model)

1. *Let $\mathbb{C}$ be the language of the classical propositional calculus with atoms Q and connectives $\neg, \wedge, \vee, \rightarrow, \top, \bot$. Q is the set of atomic arguments.*
2. *An argumentation model has the form $\mathbb{N} = (\mathbb{F}, \rho, h)$ where $\mathbb{F}$ is a set of formulas, ρ is a binary relation on $\mathbb{F}$ and h an assignment of truth values to the atoms Q. We can view h as a subset $h \subseteq Q$, and think of it as the set of usable arguments.*
3. *Given h, we can assign usability values to the formulas of $\mathbb{F}$ using the traditional truth table. We write $h(A)$ as the value of a wff A under h. h can now be regarded as a subset of $\mathbb{F}$.*
4. *A network is atomic iff $\mathbb{F} \subseteq Q$.*
5. *Note that h gives initial usability values which are not necessarily permanent and may change in the course of the recursive evaluation, see Definition 3.*

Definition 3. *Let $\mathbb{N} = (\mathbb{F}, \rho, h)$ be an argumentation model. We define an algorithm for extracting winning arguments out of $\mathbb{N}$ as follows. The definition is by levels. We define $h_m(A)$ for $A \in \mathbb{F}$ by induction on m, (compare with Definition 1).*

Level 0
$h_0(A) = h(A)$

Level $m + 1$
Let $A \in \mathbb{F}$ and let $\rho(A) = \{B_1, \ldots, B_k\}$ be all formulas B of $\mathbb{F}$ such that $B\rho A$ holds. These are the formulas which attack A according to ρ. There are several possibilities

1. *$h_m(B) = 0$ for all $B \in \rho(A)$. In this case let $h_{m+1}(A) = 1$.*
2. *For some $B \in \rho(A), h_m(B) = 1$. In this case let $h_{m+1}(A) = 0$.*
3. *$\rho(A) = \emptyset$, in which case let $h_{m+1}(A) = 1$*

Let π be the operation which defines h_{m+1} out of h_m, i.e. $h_{m+1} = \pi h_m$. Of course π depends on ρ. To be more explicit about the role of π, assume H is a function giving $\{0, 1\}$ values to all elements of $\mathbb{F}$. Using ρ and rules (1), (2), (3) above we can transform H into H'. We write $H' = \pi H$.[3]

Level ∞
Let $h_\infty(A) = y \in \{0, 1\}$ iff for some $k, h_m(A) = y$ for all $m \geq k$.
Let $h_\infty(A) =?$ (undefined) otherwise.

h_∞ is called the BG labelling of $\mathbb{F}$.

Definition 4 (Caminada Labelling). *Let $(\mathbb{F}, \rho)$ be an atomic network. A Caminada labelling on $\mathbb{F}$ is a function λ giving values in $\{0, 1, ?\}$ satisfying the following:*

1. *if $\rho(x) = \emptyset$ then $\lambda(x) = 1$*
2. *If for some $y \in \rho(x), \lambda(y) = 1$, then $\lambda(x) = 0$.*
3. *If for all $y \in \rho(x), \lambda(y) = 0$ then $\lambda(x) = 1$.*
4. *If for some $y \in \rho(x), \lambda(y) =?$ and for no $y \in \rho(x)$ do we have $\lambda(y) = 1$, then $\lambda(x) =?$.*

[3] In terms of Definition 1, H is a labelling **l** (no transmission labels) and π is a functional **f** whose algorithm uses clauses (1), (2), (3).

Lemma 1. *Let $\mathbb{N} = (\mathbb{F}, \rho, \lambda)$ be an atomic network with the Caminada labelling λ. Then there exists an assignment h_0 such that $h_\infty = \lambda$.*

Proof. Let $h^+(x) = 1$ if $\lambda(x) = 1$ or $\lambda(x) =?$.
Let $h^+(x) = 0$ if $\lambda(x) = 0$.
Let $h^-(x) = 1$ if $\lambda(x) = 1$
Let $h^-(x) = 0$ if $\lambda(x) = 0$ or $\lambda(x) =?$
Let $h_0 = h^+$.

We now prove

1. If $h_m = h^+$ then $h_{m+1} = h^-$
2. If $h_m = h^-$ then $h_{m+1} = h^+$.

Assume $h_m = h^\pm$. We calculate $h_{m+1}(x), x \in \mathbb{F}$, and show $h_{m+1} = h^\mp$.

If $\lambda(x) = 1$ then either $\rho(x) = \varnothing$ or for all $y \in \rho(x), \lambda(y) = 0$. In this case $h_m(y) = 0$ and so $h_{m+1}(x) = 1$.

If $\lambda(x) = 0$ then for some $y \in \rho(x), \lambda(y) = 1$. In this case $h_m(y) = 1$ and so $h_{m+1}(x) = 0$.

If $\lambda(x) =?$ then $\rho(x) \neq \varnothing$ and for some $y \in \rho(x), \lambda(y) =?$ and for none of the other $y \in \rho(x)$ do we have $\lambda(y) = 1$. So let $\{y_1, \ldots, y_k, y_{k+1}, \ldots, y_r\} = \rho(x)$, with $k \geq 1, \lambda(y_j) = ?, 1 \leq j \leq k$ and $\lambda(y_{k+1}) = \ldots \lambda(y_r) = 0$.

Clearly if $h_m = h^\pm$, then $h_m(y_{k+1}) = \ldots = h_m(y_r) = 0$.

If $h_m = h^+$ then $h_m(x) = 1$ and $h_m(y_1)_= \ldots h_m(y_k) = 1$ and hence $h_{m+1}(x) = 0$.

This shows that $h_{m+1} = h^-$, since x was arbitrary.

If $h_m = h^-$ then $h_m(x) = 0$ and $h_m(y_1) = \ldots = h_m(y_k) = 0$ and so for all $y \in \rho(x), h_m(y) = 0$ hence $h_{m+1}(x) = 1$.

Again since x was arbitrary we get $h_{m+1} = h^+$.

So if we start with $h_0 = h^+$, we get $h_{2m} = h^+$, $h_{2m+1} = h^-$ and so $h_\infty = \lambda$.

Lemma 2. *The converse of the previous lemma does not hold. Not every h_∞ is a Caminada labelling.*

Proof. Consider the network in Figure 7

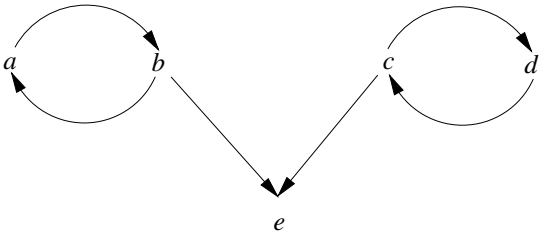


Fig. 7.

Start with $h_0(a) = h_0(b) = 1, h_0(c) = h_0(d) = 0, h_o(e) = 0$. We have $h_1(a) = h_1(b) = 0, h_1(c) = h_1(d) = 1, h_1(e) = 0$.

We also have

$$h_{2m} = h_0, h_{2m+1} = h_1.$$

Thus $h_\infty(a) = h_\infty(b) = h_\infty(c) = h_\infty(d) =?$ and $h_\infty(e) = 0$.

The Caminada labelling rules do not allow for $\lambda = h_\infty$.

Remark 1

1. The reason we could provide the example in Figure 7 is that h gave value 1 to the loop (a, b) and value 0 to the loop (c, d). So as the values in the loop oscillated, there was always one loop which attacked e. If all loops were to oscillate synchronously, $h(e)$ would have oscillated as well.
   How can we overcome this? We can use ultrafilters to get an exact value out of the oscillation. We need some concepts
   Let *Nat* be the set of natural numbers. A family of subsets $\mathbb{U}$ of numbers is called an ultrafilter if the following holds
   (a) $Nat \in \mathbb{U}, \emptyset \notin \mathbb{U}$
   (b) If $X, Y \in \mathbb{U}$ then $X \cap Y \in \mathbb{U}$
   (c) either X or $Nat - X$ is in $\mathbb{U}$.
   $\mathbb{U}$ says which sets are 'big'.
   We also note that there exists an ultrafilter $\mathbb{U}$ such that all co-finite sets are in $\mathbb{U}$.
   We now give an alternative definition of h_∞. Call it h_ω.

$$h_\omega(x) = 1 \text{ iff } U_x = \{m | h_m(x) = 1\} \in \mathbb{U}.$$

   Let us see what happens with the example of Figure 7 if we use h_ω instead of h_∞. We have
   $U_a = U_b =$ all even numbers
   $U_c = U_d =$ all odd numbers.
   One of two sets {odd numbers, even numbers} is in $\mathbb{U}$. From symmetry we can assume without loss of generality that it is the even numbers. We get

$$h_\omega(a) = h_\omega(b) = 1, h_\omega(c) = h_\omega(d) = 0.$$

   So h_ω is the same as h_0 and we have nothing.
   Let us try another angle.
2. The discrepenacy with the Caminada labelling and hence with the Dung network rules seem to arise in the case where a winning argument x according to Dung gets a value $h(x) = 0$.
   Figure 8 gives two typical examples.
   According to the Dung rules a, d, c are winning arguments. If h is such $h(a) = 0$ then b and e will be the winning arguments.
   The question we ask is can we use a device which makes the two approaches compatible?
   Suppose we say a is not usable (i.e. $h(a) = 0$) because there is an attack on a. Say $\mathbf{h}(a)$ is the argument which attacks a. $\mathbf{h}(a)$ is not in the network but the fact that a is attacked is recorded by $h(a) = 0$. There may be good reason why we don't want $\mathbf{h}(a)$ to be explicit in the network. Maybe $\mathbf{h}(a)$ is a different type of argument.

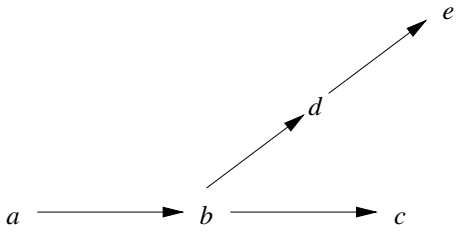


Fig. 8.

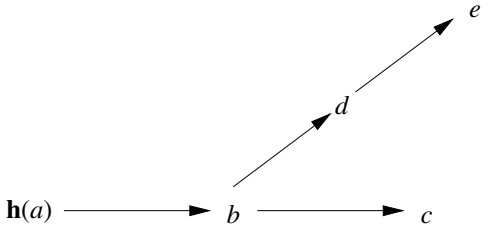


Fig. 9.

Maybe it is a secret argument. Whatever the reason is, the real network is Figure 9. The above trick works for this network. Does it work in general?

Given an atomic network $(\mathbb{F}, \rho, h)$ can we get the correct result by adding nodes $\mathbb{F}_1 = \{\mathbf{h}(x) | x \text{ such that } h(x) = 0\}$ and letting $\mathbb{F}' = \mathbb{F} \cup \mathbb{F}_1$ and $\rho' = \rho \cup \{(\mathbf{h}(x), x) \mid \mathbf{h}(x) \in \mathbb{F}_1\}$?

Do we have a general theorem that we can pair the winning subsets? I.e. can we have:

(a) For any winning set $T' \subseteq \mathbb{F}'$, $T' \cap \mathbb{F}$ is a winning set of $(\mathbb{F}, \rho, h)$.
(b) For any winning set $T \subseteq \mathbb{F}$, there exists a winning set $T' \subseteq \mathbb{F}'$ such that $T = T' \cap \mathbb{F}$.

We can hope for such results only if whenever h says an argument x is out then it is out permanently, because when we insert $\mathbf{h}(x)$ to attack x and force it out, it is out permanently.

Our algorithm in Definition 3 and later on in the section dealing with modal and temporal logic, does not keep unusable arguments out permanently, it does bring them in depending on the attack cycles.

Remark 2 (Discussion of the Dung network rules)

1. The discrepancy with the Caminada labelling is a serious one. The Caminada labelling is faithful to the Dung argumentation network rules, namely (see Definition 3).
   (a) If all attacks on a node x are defeated (are *out*) then x is *in*.
   (b) If some attacks on a node x are *in* then x is *out*.
   (c) If there are no nodes attacking x then x is *in*.

In the Dung framework these rules are *not defeasible rules*, they are *absolute*.
So consider, for example, an argumentation network with one node and one argument x. Since nothing attacks x, x is a winning argument. If we look at a classical model for x with $x = 0$, namely x is unusable for whatever reason, then the Dung rule overrides the unusability of x and x is still a winning argument.
Compare this with default logic. The default rule $\frac{x}{x}$ says that if x is consistent to add then add it by default. This will not override any given data about x.
So rules (a), (b), (c) are too strong when we give a model interpretation to the arguments. An argument can win even though it is unusable, simply because it is not attacked.

2. We call for a new critical evaluation of rules (a)–(c). We would abandon rule (c) and modify rule (a).
   The proposed BG rules for a Dung network relative to an assignment or other evidence, are (a*)–(c*) below. We call the associated update functional π^* (compare with π of Definition 3).
   a*. If all attacks on x are defeated and there is no evidence that x is unusable then x is *in*
   b*. If some attacks on x are in the x is *out*.
   c*. If there are no nodes attacking x then x is *in* only if there is no evidence that it is unusable.
   Jakobovits and Vermeir [7] have already proposed that an argument which has all of its defeaters out need not necessarily be in. This view is criticised in [2]. Our (a*) and (c*) are in line with [7].
3. Let us call an assignment h a Caminada assignment to the atoms of Q if for some Caminada labelling λ we have
   – $h(x) = 1$ if $\lambda(x) = 1$.
   – $h(x) = 0$ if $\lambda(x) = 0$
   – $\forall x(\lambda(x) =?$ implies $h(x) = 1)$.
   From Lemma 1 we know that $h_\infty = \lambda$ and hence Caminada assignments are compatible with Dung networks. If we restrict our Kripke model to Caminada assignments then maybe we will have no technical discrepancies.

Remark 3. The difference between BG and Caminada labelling can be appreciated by looking at the logic programming translation of a Dung network. Consider the network of Figure 1. Its translation is ($\neg$ is negation as failure)

1. b if $\neg a$
2. a

In our model we also give usability assignments $h(x)$ to x. So we translate as follows into a logic programming program (we regard $\mathbf{h}(x)$ as a new literal dependent on x attacking x by virtue of an argument showing that x is unusable):

1*. b if $\neg a \wedge \neg \mathbf{h}(b)$
2*. a if $\neg \mathbf{h}(a)$
3*. $\mathbf{h}(x)$ if $\neg \mathbf{zero}(x)$, for all nodes x
4*. $\mathbf{zero}(x)$, for all x such that $x =$ usable, under the assignment h.

$\mathbf{h}(x)$ and $\mathbf{zero}(x)$ are new literals, for each node x.

Note that we use $\neg\mathbf{h}(x)$ rather than $\mathbf{h}(x)$ in the program so that the program will have a corresponding Dung network. To make $\neg\mathbf{h}(x)$ fail we need to add $\mathbf{h}(x)$, for x = usable, under h.

The corresponding Dung network for Figure 1 is Figure 10 below (see footnote 1):

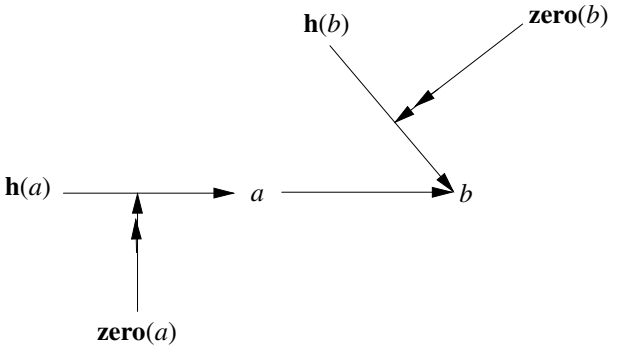


Fig. 10.

So the BG program is defined to contain the following clauses

1* If $x_1, \ldots, x_n$ attack y in the network we include the clause

$$y \text{ if } \neg x_1 \wedge \ldots \wedge \neg x_n \wedge \neg\mathbf{h}(y)$$

where $\mathbf{h}(y)$ is a new atom.

2* If y is not attacked by any node we include the clause
y if $\neg\mathbf{h}(y)$

3* Add $\mathbf{h}(y)$ if $\neg\mathbf{zero}(y)$

4* For every y such that $h(y) = 1$ we include the literal $\mathbf{zero}(y)$.[4]

3 Kripke Models for Argumentation Networks 1

We begin this section with some methodological remarks and general examples which will bring us to a point of view best suited for the presentation of modal and temporal argumentation networks.

We begin with a simple example. Consider the sentence

– John read le livre with interest.

[4] Readers familiar with attacks on attacks as developed in [1] will realise we can present a network relative to additional nodes attacking connections. The $\mathbf{zero}(y)$ nodes are added according to assignment to attack the connection $\mathbf{zero}(y) \rightarrow \mathbf{h}(y)$. In fact there is no need to do it this way. We do not need $\mathbf{zero}(y)$. We can simply augment the original network with additional nodes $\mathbf{h}(y)$ attacking the node y for all y such that $h(y) = 0$. The problem with that is that h keeps on changing and so the $\mathbf{h}(y)$ will keep on being in and out.

This sentence contains some French words. To understand the sentence we need to go to a French dictionary and come back with values. We come back with the words 'the book'.

Now consider

– John is dishonest because he did not pay yesterday.

To check the value of the above, we must go to yesterday and verify that John did not pay.

Let t label the location of the main sentence and let s label the location of where we need to go. In each case we have the following situation.

In the process of evaluating the algorithm $\mathcal{A}_t$ at t, we hit upon a unit of the form

Take x to location s, find a value $y = V_s^t(x)$ for x at s and come back and plug it into our local algorithm $\mathcal{A}_t$ and carry on.

The notation $V_s^t(x)$, is the value you get for x at location s intended to be understood and used at t (so V_s^t is a French to English 'function' in the first example, and a function reading from a payment ledger for the second example).

Let us now take another example. Consider the three argumentation networks in Figure 11. In network t the node x is not an argument but an instruction to look for an

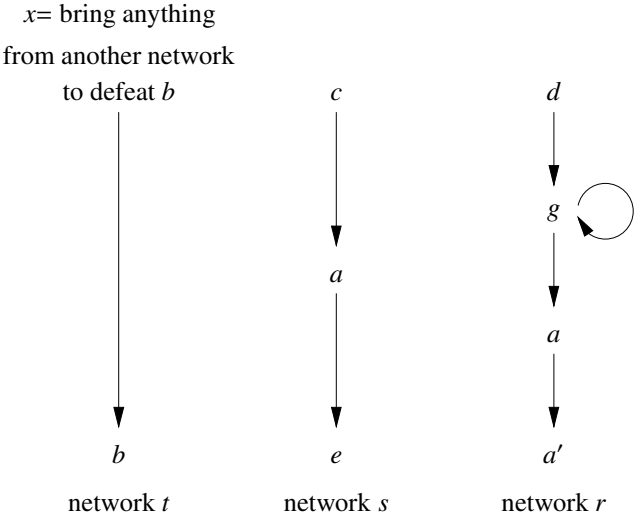


Fig. 11.

accessible network in which there is a winning argument which can defeat b. The two accessbile networks are s and r. In s, a is not a winning argument, but it is in r. Suppose a is capable of defeating b; this knowledge is not recorded in the network t but is known to us either extra-logically or intrinsically (for example, b is logically inconsistent with a). Then x will be argument a coming from network s. x can be as specific as

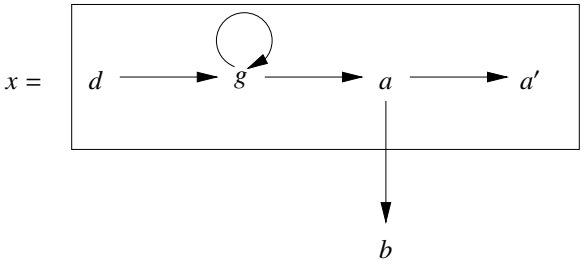


Fig. 12.

needed for a successful search. Note that we could have written Figure 12. This is a fibring of network r at position x at network t. The attack on b comes from inside the network r from node a onto node b (in network t).

We can turn the situation into modal logic by using $\Diamond x$ (or even $\Diamond a$ if we know that $x = a$ will do the job) and we get a kind of Kripke model, see for example Figure 13. The zigzag arrow $\rightsquigarrow$ is accessibility and the ordinary arrow $\rightarrow$ is attack. $\Diamond x$ (or $\Diamond a$)

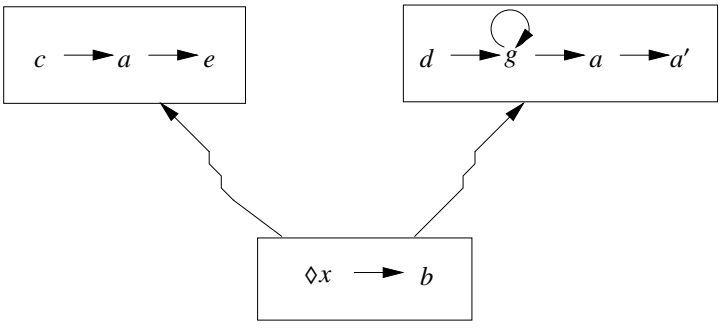


Fig. 13.

means find a winning argument a which can attack b. Here the meaning of $\Diamond a$ is administrative. $\Diamond$ is a metalevel administrative connective. $\Diamond a$ does not mean that a is a possible argument; and $\Diamond$ is not in the argument language.

Consider now the following network

$$\Diamond \text{ storm} \rightarrow b$$

in which $\Diamond$ storm represents "it is possible there will be a storm" and b represents "setting sail now". The model envisages several possible futures, if in at least one of them there is a storm then that possibility defeats b. Here the possible $\Diamond$ is not an administrative but a temporal event. The language of $\Diamond a$ is object level, and $\Diamond$ must be in the language of arguments.

Technically the mathematics of both cases, the administrative metalevel $\Diamond$ and the temporal event object level $\Diamond$, is very similar. In both cases we can put $\Diamond x$ in the nodes of an argumenation network and seek winning arguments in accessible networks.

In either case of $\Diamond x$ we need to search other networks for an appropriate winning value. So it is not clear until after the calculation and search, whether we have a usable argument here or not, especially if the family of networks is complex. Hence the need and the technical usefulness and value of a usability assignment. It simplifies matters during the calculations.

We now explore our options for Kripke models for argumentation networks. We begin with a simple first attempt which will turn out to need improvement. However, it is helpful to go through this first attempt for us to appreciate what is to be added. Consider the network of Figure 5 and let Figure 14 describe a Kripke model. The reader should bear this in mind when reading the next formal definition.

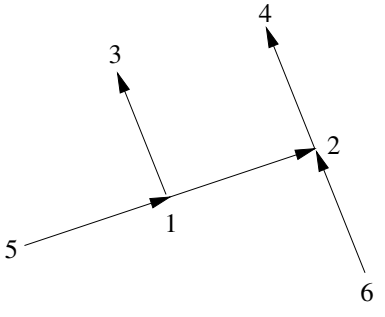


Fig. 14.

Definition 5 (Languages). *We recall here the languages of modal and temporal logic.*

1. *The classical connectives we use are* $\sim$ *(negation),* $\wedge, \vee, \rightarrow$. *We reserve* $\neg$ *for negation as failure.*
2. *In temporal logic we use the connective PA for A was true in the past and FA for A will be true in the future. A temporal model is usually presented through a non-empty set T of moments of time and an earlier-later relation* $<$ *on T.* $<$ *is usually taken as irreflexive and transitive. The classical truth conditions for P and F are*
   - $t \vDash PA$ *iff for some s such that* $s < t$, *we have* $s \vDash A$.
   - $t \vDash FA$ *iff or some s such that* $t < s$, *we have* $s \vDash A$

   Temporal logic defines
   $HA =\sim P \sim A = A$ *has always been true*
   $GA =\sim F \sim A = A$ *will always be true.*
3. *Modal logic uses* $\Diamond A$ *reading A holds in another accessible world. The set of worlds is denoted by S and has an accessibility relation* $R \subseteq S \times S$. *We have*
   - $t \vDash \Diamond A$ *iff for some s such that tRs we have* $s \vDash A$.

   $\Box A$ *usually means* $\sim \Diamond \sim A$.

4. *The usual temporal or modal logics have formulas evaluated at worlds. If we want to define the notion of modal and temporal networks we will need to deal with networks of formulas evaluated at worlds.*
5. *We can have in the language both the temporal connectives P, F and the modal connective $\Diamond$. In which case the semantics will need to have both R and $<$. We may allow for the future to be also a possibility for $\Diamond$, in which case we have:*

$$t < s \rightarrow tRs.$$

   Sometimes only P and $\Diamond$ are used in which case we can use only R and go backwards in it for evaluating P.
6. *The examples below use 'usability' instead of truth. So we have for example that "a is usable at world t" is treated mathematically the same way as we treat "a is true (holds) at t".*

Let us start with some examples in the simple language which contain arguments of the form x, (atomic), $\Diamond x$, possibility of an argument and Px, past arguments. Think of $\Diamond x$ as future possibility. So in the Kripke model, $\Diamond$ goes up in the arrow direction and P goes down in the arrow direction.

In the usual Kripke evaluation procedures for classical logic, with a set Q of atomic arguments, we have an assignment h giving a truth value at each world t for each atomic proposition q. We write $h(t) \subseteq Q$ for the set of true propositions at t. In the argumentation case we do not have atomic propositions, but we do have argumentation networks which themselves contain atomic propositions. For example the network of Figure 5 contains the atoms a, b, c.

So we give the following definition. For each node n in the Kripke model we assign a set $h(n) \subseteq$ set of atoms appearing in the network. For an atom q, we assign a usability value 1 if $q \in h(n)$ and 0 otherwise. We also use the notation $h(n, q) = 1$ or $h(n) \vDash +q$ to indicate that $q \in h(n)$, and $h(n, q) = 0$ or $h(n) \vDash -q$ to indicate that $q \notin h(n)$.

Figure 15 is an example of such an assignment, where we write $\pm q$ to indicate the value of q.

Suppose we want to know the value of the network of Figure 5 at the model at node 1. How do we evaluate Pc? We follow the traditional steps of evaluation in a Kripke model; we go down the accessibility relation and look for a world where c is usable.

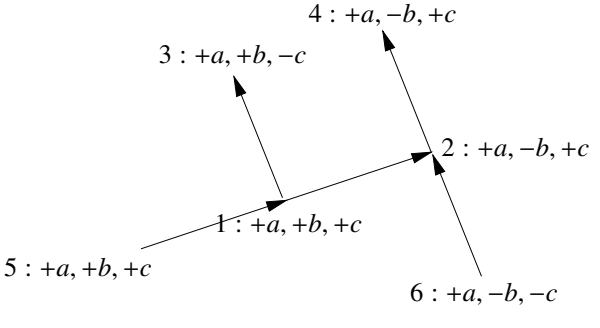


Fig. 15.

At node 5 we have $+c$, so maybe we say $+Pc$ at node 1, but we notice that a attacks Pc in the network (see Figure 5). So does $1 \models Pc$ or not? Furthermore, Pc attacks b and so at node 2 does Pb hold or not?

We have $+b$ at node 1, so we would like to say $2 \models Pb$, however b may be successfully attacked at node 1. So we may not have Pb after all, so what do we have?

We need an agreed recursive definition.

Let us see some examples where we might have a loop, and try and get a clue by working the example out.

Example 1. Consider the network and Kripke model as described in Figure 16

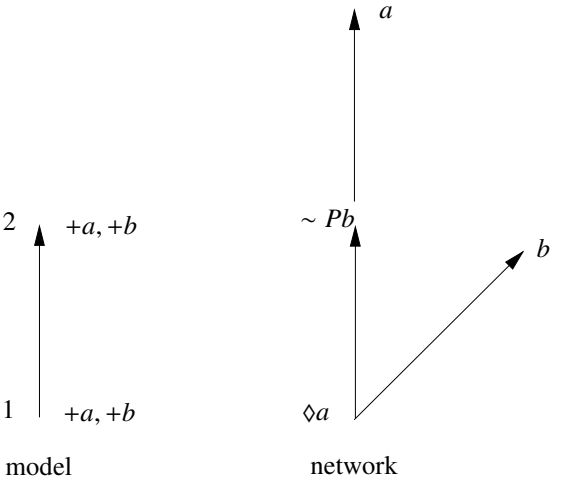


Fig. 16.

To evaluate the network at 1, we know that $\sim Pb$ = usable. But $\sim Pb$ is attacked by $\Diamond a$ and so we need the value of $\Diamond a$ at 1. For this we need the value of a at point 2. We have $+a$ at 2, but this is attacked by $\sim Pb$, which is not attacked by $\Diamond a$ at 2 since it is not usable at 2.

So we need to know the value of b at 1. We do have $+b$ at 1 but this is attacked by $\Diamond a$ so we need to know the value of a at 2 and we have a loop.

We need some process of evaluation which will give us a better chance to resolve the loops.

We do this by levels of recursive evaluation. We use two bits of notation.

1. $h_m(t, A)$ is the assignment of value at level m to the argument A. A may be atomic or a formula.
2. $t \models_m A$ is the network value of A at world t at level m. We use the BG algorithm for the functional π^* namely clauses $(1*)$, $(2*)$ and $(3*)$ of Remark 2.

So to make the meaning of h_m and $\models_m$ crystal clear: h_m says which arguments in the network are usable at level m, while $\models_m$ says which arguments are winning (not defeated) at level m. An argument defeated at $\models_m$ is considered unusable by h_{m+1}.

Let us now apply this to Figure 16's example.

Level 0

h_0 is the assignment h to the atoms as indicated in the Figure, i.e. $h_0(1,a) = h_0(1,b) = h_0(2,a) = h_0(2,b) = 1$. $\models_0$ is defined for $a\,b$ as the same value as h_0.

For $\Diamond a, Pb, h_0, \models_0$ is not necessarily defined.

It is convenient to use the notation

$h_m(t) \models +A$ to say $h_m(t,A) = 1$

$h_m(t) \models -A$ to say $h_m(t,A) = 0$.

The idea of h_m and $\models_m$ is as follows: h_m evaluates the temporal modalities using the Kripke model without regard to the argumentation network. Then $\models_m$ records the result of the attacks (i.e. the winning arguments) of the argumentation network at each world. The $\models_m$ may record the defeat of some atoms in the network, thus rendering them unusable at level $m+1$ (by virtue of being defeated at level m) thus giving rise to a new assignment which can now be used to calculate h_{m+1}, and so on.

Level 1

$$h_1(1) \models +\Diamond a, +b, + \sim Pb, +a$$
$$h_1(2) \models -\Diamond a, +b, - \sim Pb, +a.$$

Now we have networks with nodes which have values and so we calculate the winning arguments. These are the ones holding at the worlds of the Kripke model at level one. We write:

$$1 \models_1 \Diamond a, a$$
$$2 \models_2 b, a$$

Level 2

The assignment we use to calculate h_2 is the atomic part of $\models_1$, namely $1 \models_1 a$ and $2 \models_1 a, b$.

$$h_2(1) \models +\Diamond a, -b, +a, + \sim Pb$$
$$h_2(2) : -\Diamond a, +b, + \sim Pb, +a$$

After evaluation of the networks we get

$$1 \models_2 \Diamond a, a$$
$$2 :\models_2 \quad \sim Pb, b$$

Level 3

$\models_2$ gives us a new assignment to the atoms, namely $1 \models_2 a$ and $2 \models_2 b$.

$$h_3(1) \models -\Diamond a, -b, +a, + \sim Pb$$
$$h_3(2) \models -\Diamond a, +b, + \sim Pb, -a$$

calculating the surviving arguments at each world we get

$$1 \models_3 \sim Pb, b$$
$$2 \models_3 \sim Pb, b$$

Level 4
We now get the assignment from $\models_3$ for atoms as

$$1 \models_3 b, 2 \models_3 b$$

We calculate h_4

$$h_4(1) \models -\Diamond a, - \sim Pb, +b, -a$$
$$h_4(2) \models -\Diamond a, - \sim Pb, +b, -a$$

We calculate $\models_4$ at each node using network rules

$$1 \models_4 \sim Pb, b,$$
$$2 \models_4 \sim Pb, b.$$

$\models_3$ and $\models_4$ are the same. So the answer is now stable.

Remark 4. The reader may wonder what has happened to the loop we observed before and what kind of interpretation (loop resolution) we are getting. To explain that, let us look at the traditional loop in the network of Figure 17, in which a and b attach each other. We have three complete extensions $\{a\}, \{b\}, \varnothing$. Let us do the level calculation intuitively.

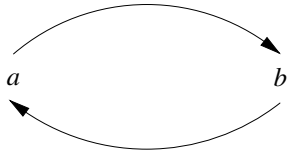


Fig. 17.

Level 0
Start with $+a, +b$.
Level 1
Attack as suggested by level 0 we get $-a, -b$.
Level 2
Attack as suggested by level 2. We get $+a, +b$.

So we are infinitely looping and we can put a question mark on a and on b, leading to $\varnothing$.

Of course everything depends on the assignment at level 0. Other possible assignments are $\{+a, -b\}, \{-a, +b\}, \{-a, -b\}$, yielding $\{a\}, \{b\}$ and $\varnothing$ respectively.

Example 2. We try and evaluate the network of Figure 5 at the model of Figure 15. We use the *BG* π^* evaluation algorithm.

We do this by levels. Let h_0 be the assignment to the atoms as indicated in Figure 15. Thus h_0 satisfies

Level 0

$$h_0(5) \models +a, +b, +c$$
$$h_0(6) \models +a, -b, -c$$
$$h_0(1) \models +a, +b, +c$$
$$h_0(2) \models +a, -b, -c$$
$$h_0(3) \models +a, +b, -c$$
$$h_0(4) \models +a, -b, +c.$$

Level 1

$$h_1(5) \models +a, +b, +c, -Pc, +\Diamond a$$
$$h_1(6) \models +a, -b, -c, -Pc, +\Diamond a$$
$$h_1(1) \models +a, +b, +c, +Pc, +\Diamond a$$
$$h_1(2) \models +a, -b, +c, -Pc, +\Diamond a$$
$$h_1(3) \models +a, +b, -c, +Pc, -\Diamond a$$
$$h_1(4) \models +a, -b, +c, +Pc, -\Diamond a$$

Now the network of Figure 5 has values for each node at each world. We can compute the winning argument at each world. Note that c does not appear as a node in the network so its value we inherit from h.

$$5 \models_1 +a, -b, +c, -Pc, +\Diamond a$$
$$6 \models_1 +a, -b, -c, -Pc, +\Diamond a$$
$$1 \models_1 +a, -b, +c, +Pc, +\Diamond a$$
$$2 \models_1 +a, -b, +c, -Pc, +\Diamond a$$
$$3 \models_1 +a, +b, -c, +Pc, -\Diamond a$$
$$4 \models_1 +a, +b, +c, +Pc, -\Diamond a$$

Level 2
We evaluate h_2 using the assignment to the atoms suggested by $\models_1$.

$$h_2(5) \models +a, -b, +c, -Pc, +\Diamond a$$
$$h_2(6) \models +a, -b, +-, -Pc, +\Diamond a$$
$$h_2(1) \models +a, -b, +c, +Pc, +\Diamond a$$
$$h_2(2) \models +a, -b, +c, -Pc, +\Diamond a$$
$$h_2(3) \models +a, +b, -c, +Pc, -\Diamond a$$
$$h_2(4) \models +a, +b, +c, +Pc, -\Diamond a$$

We now calculate $\models_2$

$$5 \models_2 +a, -b, +c, -Pc, +\Diamond a$$
$$6 \models_2 +a, -b, -c, -Pc, +\Diamond a$$
$$1 \models_2 +a, -b, +c, +Pc, +\Diamond a$$
$$2 \models_2 +a, -b, +c, -Pc, +\Diamond a$$
$$3 \models_2 +a, +b, -c, +Pc, -\Diamond a$$
$$4 \models_2 +a, +b, +c, +Pc, -\Diamond a$$

We have stability since $\models_1$ equals $\models_2$.

Definition 6 (Temporal Languages). *Let Q be a set of atoms. The basic temporal language based on Q uses the unary connectives $\mathbb{C} = \{\neg, \Diamond, P\}$. A temporal formula has the form $\alpha_1\alpha_2 \ldots \alpha_m q$, where $q \in Q$ and $\alpha_i \in \mathbb{C}, i = 1, \ldots, m$.*

The language is said to be very basic if we allow $\Diamond x$ and Px and x only.

The full temporal language also allows the use of the classical connectives $\wedge, \vee \rightarrow$ and unrestricted use of $\Diamond$ and P.

Definition 7 (Temporal Kripke Models). *Let (S, R, a) be a Kripke model and let $\mathbb{N} = (\mathbb{F}, \rho)$ be a network in the basic temporal language with $\neg, \Diamond, P$ and Q or in the full temporal language. Let h be an assignment giving for each world $t \in S$ and an atom $q \in Q$ a usability value $h(t, q) \in \{0, 1\}$.*[5]

We define by induction a sequence of new assignments $h_1, h_2, \ldots$ and semantic consequences $\models_1, \models_2 \ldots$ as follows:

1. *Let h_1 be defined as follows*
   $h_1(t, q) = h(t, q)$, for q atomic
   $h_1(t, \Diamond A) = 1$ if for some $s \in S$ such that tRs we have $h_1((s, A) = 1$.
   $h_1(t, PA) = 1$ iff for some s, such that sRt we have $h_1(s, A) = 1$.
   The definition for the classical connectives is the usual one.
2. *Let $\models_1$ be defined as follows:*
   First consider $h_1(t)$ as an assignment on $(\mathbb{F}, \rho)$ with t as a fixed parameter. We can consider $h_1(t)$ as a subset of $\mathbb{F}$. Consider the operator π of Definition 3.
   We define $\models_1$ by

$$t \models_1 x \text{ iff } x \in \pi h_1(t).$$

   The reader can compare with the construction in Example 1.

Note that if we want to use defeasible rules as discussed in Remark 2 then we use π^* of Remark 2 instead of π of Definition 3.

We now define $h_{m+1} \models_{m+1}$ for $m \geq 1$.

h_{m+1} is obtained from $\models_m$ in the same way that h_1 was obtained from h, by regarding $t \models_m q, q \in Q$ as an assignment to the atoms. Note that all we need is the values of $\models_m$ on the atoms of Q.

$\models_{m+1}$ is obtained from h_{m+1} in the same way that h_2 was obtained from h_1, i.e. for each t, we have $t \models_{m+1} A$ iff $A \in \pi h_{m+1}(t)$.

This defines $h_m \models_m$ for all $n \geq 1$.

We now define $t \models_\infty A$ for $t \in S$ and $A \in \mathbb{F}$ as follows:
$t \models_\infty A$ holds (does not hold) if for some k, $t \models_m A$ holds (resp. does not hold) for all $n \geq k$.

Otherwise if $t \models_m A$ oscillates, we say that $t \models_\infty A$ is undecided.

[5] This definition is parallel to the traditional one. $\Diamond$ goes up the accessibility relation and P goes down it. The evaluation is more complicated because the formulas are part of a network.

As noted before if we insist that at each t the assignment $h(t) \subseteq Q$ is a Caminada assignment (need not be the same one for all t), then we will have less technical discrepancies with the Dung interpretation. We need to check, however, since our language has temporal operators, whether $h_m(t)$ remains a Caminada assignment. We will examine this point later in the section. Our guess is that further adjustment will be needed.

We shall give better definitions later in the next section.

4 Kripke Models for Argumentation Networks 2

Let us assess the situation we are in. In the previous section, we offered a simple model. This model can be improved. The problem is not so much the discrepancy with the Dung approach (see Remark 3) as this is not a unique possible world problem, but the difficulty is that we need to sharpen the intuitive meaning to Definition 7. Definition 7 is mainly technically motivated by a natural formal analogue of the semantical movements in a traditional modal and temporal Kripke model. We need to clarify more sharply the meaning of usability of arguments and its connection to truth and falsity and possibility of argument and facts.

There seems to be a fundamental difference between the modal operator $\Diamond$ and the temporal operator P. P goes into the past while $\Diamond$ goes to an alternative world of different reasoning/argumentation framework. In ordinary Kripke semantics for traditional modal and temporal logics, there is no technical distinction between the two. In argumentation context we need to give different technical treatment to these two connectives. The $\Diamond$ we treat as a fibring operator (go to another context, do something there and come back with the result, see [5]), while the operator P is still treated as purely temporal.

We illustrate with an example.

We want to argue against a political candidate c. We want to bring in the past facts that he double-crossed his partners, showing lack of loyalty and trustworthiness (call this Pd, d for "dirt"). However, the situation today is such that digging up the past on a candidate is counterproductive (call this $\sim p$). It is suggested therefore to wait 6 months for the facts to emerge naturally (i.e. $\Diamond d$ where $\Diamond$ here reads future possibility).

A counterargument against waiting is that by that time criteria for judging candidates will change and the argument will be defeated, say d will be attacked by e (e can mean who cares? ; it was a long time ago!).

Figure 18 shows the situation:

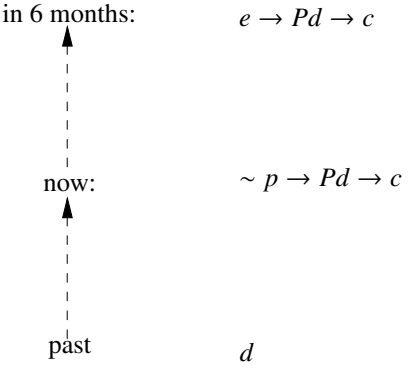


Fig. 18.

We notice the following discrepancies between the formal situation of Figure 18 and the formal definition we gave to modal and temporal argumentation networks, in Definition 7.

1. We have different networks at different possible worlds.
2. $\Diamond$ behaves like a fibring operator
3. P is purely temporal for facts.
4. With P the assignment h indicates $\pm$ usability by virtue of truth or falsity, while with $\Diamond$ the assignment h might indicate $\pm$ usability for other reasons.
5. We cannot have a proper temporal and modal treatment of arguments without looking into the details of what is the internal structure of arguments and how exactly do they attack one another in terms of such structure.
   Suppose we adopt the view that arguments are proofs and attacks disrupt such proofs. Let us examine how time T gets into the picture. Consider the following example:
   I park my car near Russell Square at 11am in the morning, do my business of the day and come back to it at 5 pm. I expect to find it there. If I don't find the car then nonmonotonic deduction allows me to conclude that the car was stolen. We can have a little logical system (nonmonotonic or Bayesian net) which compares the conclusions of "car stolen" against "car towed away by local council parking department". We can assume the latter conclusion can be defeated. We can also ignore some well known difficulties of persistence of arguments where one gets paradoxically that the car was stolen only a few seconds before my return (the stolen car paradox).
   There is one sure way to attack this argument and its conclusion. This is to prove the simple fact that I don't have a car.
   Figure 19 illustrates the situation

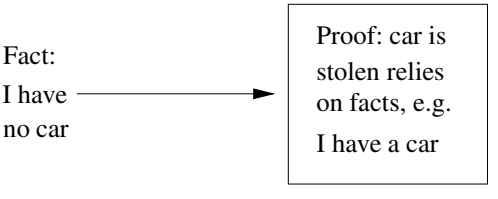


Fig. 19.

   Facts can attack arguments most effectively. Also if the facts are undecided or are not available, then we claim the argument is not usable. See section 5 for further discussion.
   So when designing a new modal and temporal logic for argumentation, we need a pure $\{P, F\}$ temporal logic just for the facts.
6. Our next question is whether an argument itself can be time dependent. This is a bit tricky. In monotonic logic the answer is no. Euclid geometric proofs are as valid and good today as they were in ancient times. But in the nonmonotonic case the answer is yes. Nonmonotonic reasoning depends on context. Today a girl in a mini-skirt

will not be considered immodest but go back 200 years and everyone at that time will nonmonotonically deduce she is 'fast'. So as we can see from this example the deduction mechanism itself can change in time. Thus we may argue for example along the lines 'you had better get yourself a decent long dress now because soon people's perception will change and you will no longer be respectable wearing a mini-skirt'.

Definition 8 (Temporal Kripke Models - Udpated)

1. *Consider a language with the classical connectives* $\{\sim, \wedge, \vee, \rightarrow\}$ *and the temporal and modal connectives* $\{P, F, \Diamond\}$. *We also assume a set* Q *of atoms, and we use the atoms to construct formulas using the connectives in the traditional manner.*
2. *A Kripke model for the above language has the form* $(S, R, \leq, a, h)$, *where* S *is a nonempty set of worlds and* $R \subseteq S \times S$ *is a binary relation for* $\Diamond$ *and* $<$ *is an irreflexive and transitive relation on* S *for* F *and* P. h *is an assignment giving for each* $t \in S$, *a subset* $h(t) \subseteq Q$. *We consider* h *as a usability function on the atomic arguments of* Q, *saying which elements of* Q *are usable at world* t.
3. *For each* t, *let* $\mathbb{N}(t) = (\mathbb{F}(t), \rho(t))$ *be an argumentation system.*
4. *We define by induction a sequence of new assignments* $h_1, h_2, \ldots$ *and semantic consequences* $\vDash_1, \vDash_2 \ldots$ *as follows:*
   (a) *Let* h_1 *be defined as follows*
   $h_1(t, q) = h(t, q)$, *for* q *atomic*
   $h_1(t, \Diamond A) = 1$ *if for some* $s \in S$ *such that* tRs *we have* $h_1((s, A) = 1$.
   $h_1(t, PA) = 1$ *iff for some* s, *such that* $s < t$ *we have* $h_1(s, A) = 1$.
   $h_1(t, FA)$ *iff for some* s *such that* $t < s$ *we have* $h_1(s, A) = 1$.
   The definition for the classical connectives is the usual one.
   (b) *Let* $\vDash_1$ *be defined as follows:*
   First consider $h_1(t)$ *as an assignment on* $(\mathbb{F}, (t), \rho(t))$ *with* t *as a fixed parameter. We can consider* $h_1(t)$ *as a subset of* $\mathbb{F}(t)$. *Consider the operator* $\pi*$ *of Remark 2.*
   We define $\vDash_1$ *by*

$$t \vDash_1 x \text{ iff } x \in \pi^* h_1(t).$$

We now define $h_{m+1} \vDash_{m+1}$ *for* $m \geq 1$.
h_{m+1} *is obtained from* $\vDash_m$ *in the same way that* h_1 *was obtained from* h, *by regarding* $t \vDash_m q, q \in Q$ *as an assignment to the atoms. Note that all we need is the values of* $\vDash_m$ *on the atoms of* Q.
$\vDash_{m+1}$ *is obtained from* h_{m+1} *in the same way that* h_2 *was obtained from* h_1, *i.e. for each* t, *we have* $t \vDash_{m+1} A$ *iff* $A \in \pi^* h_{m+1}(t)$.
This defines $h_m \vDash_m$ *for all* $n \geq 1$.
We now define $t \vDash_\infty A$ *for* $t \in S$ *and* $A \in \mathbb{F}$ *as follows:*
$t \vDash_\infty A$ *holds (does not hold) if for some* k, $t \vDash_m A$ *holds (resp. does not hold) for all* $n \geq k$.
Otherwise if $t \vDash_m A$ *oscillates, we say that* $t \vDash_\infty A$ *is undecided.*

The upshot of the above is that we need a much more complex, tailor made model for argumentation.

5 Conclusions

In this paper, we have considered a generalisation of traditional Dung argumentation networks to handle families of different argumentation networks structured according to either context or time. We allow the arguments to be either atomic or to refer to different context or time. In each context (world), we consider the valid atomic sentences and can evaluate modal or temporal connectives in such a system. This paper continues our programme in developing temporal dynamics of support and attack networks [1]. Other work handling temporal dimensions in argumentation includes that of Augusto and Simari [13] and Mann and Hunter [12]. The former embeds absolute time arguments into meta-level like predicates "occurs", "holds", "do", etc. over classical logic formulas; the latter uses classical logic sentences augmented with Allen-type intervals together with a "holds" meta predicate. Our work, on the other hand, uses non-classical modal and temporal connectives in the tradition of non-classical logic.

Acknowledgements

We are grateful to Martin Caminada, Sanjay Modgil, Jane Spurr and Leon van der Torre for penetrating comments.

References

1. Barringer, H., Gabbay, D., Woods, J.: Temporal dynamics of support and attack networks. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 59–98. Springer, Heidelberg (2005)
2. Caminada, M.W.A.: On the issue of reinstatement in argumentation. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 111–123. Springer, Heidelberg (2006), http://icr.uni.lu/~martinc/publications/JELIA_reinstatement.pdf
3. Caminada, M.W.A.: An algorithm for computing semi-stable semantics. In: Mellouli, K. (ed.) ECSQARU 2007. LNCS (LNAI), vol. 4724, pp. 222–234. Springer, Heidelberg (2007), http://icr.uni.lu/~martinc/publications/algorithm_eCSQARU.pdf
4. Caminada, M.W.A., Amgoud, L.: On the evaluation of argumentation formalisms. Artificial Intelligence 171(5-6), 286–310 (2007)
5. Gabbay, D.M.: Fibring Logics, OUP (1996)
6. Hunter, A.: Ramification analysis with structured news reports using temporal argumentation. In: Proc. of the Adventures in Argumentation Workshop (at 6th ECSQARU) (2001), http://www.cs.ucl.ac.uk/staff/a.hunter/papers/ra2.ps
7. Jakobovits, H., Vermeir, D.: Robust semantics for argumentation frameworks. Journal of Logic and Computation 9, 215–261 (1999)
8. Gabbay, D.M., d'Avila Garcez, A.S.: Logical Modes of Attack in Argumentation Networks. Studia Logica 93(2-3), 199–230 (2009)
9. Caminada, M., Gabbay, D.M.: A logical account of formal argumentation. Studia Logica 93(2-3), 109–145 (2009)
10. Gabbay, D.M.: Modal foundations for argumentation networks. Studia Logica 93(2-3), 199–230 (2009)
11. Gabbay, D.M.: Fibring argumentation frames. Studia Logica 93(2-3), 231–295 (2009)
12. Mann, N., Hunter, A.: Argumentation Using Temporal Knowledge. In: COMMA 2008, Toulouse, France, May 28-30, pp. 204–215. IOS Press, Amsterdam (2008)
13. Augusto, J.C., Simari, G.R.: A temporal argumentative system. AI Communications 12, 237–257 (1999)

Knowledge Based Scheduling of Distributed Systems

Saddek Bensalem[1], Doron Peled[2], and Joseph Sifakis[1]

[1] Centre Equation - VERIMAG, 2 Avenue de Vignate, Gières, France
[2] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. Priorities are used to control the execution of systems to meet given requirements for optimal use of resources, e.g., by using scheduling policies. For distributed systems it is hard to find efficient implementations for priorities; because they express constraints on global states, their implementation may incur considerable overhead.

Our method is based on performing model checking for knowledge properties. It allows identifying where the local information of a process is sufficient to schedule the execution of a high priority transition. As a result of the model checking, the program is transformed to react upon the knowledge it has at each point. The transformed version has no priorities, and uses the gathered information and its knowledge to limit the enabledness of transitions so that it matches or approximates the original specification of priorities.

1 Introduction

Executing transitions according to a priority policy is complicated when each process has a limited view of the situation of the rest of the system. Such limited local information can be described as the *knowledge* that processes have at each point of the execution [3,4]. Separating the design of the system into a transition system and a set of priorities can be a very powerful tool [6], yet quite challenging to implement [1]. Our solution for implementing priorities is based on model checking [2,11] of knowledge properties [10]. This analysis checks which processes possess knowledge about having a maximal priority transition enabled at the current state.

The information gathered during the model checking stage is used as a basis for a program transformation. It produces a new program without priorities, which implements or at least approximates the prioritized behaviors of the old program. At runtime, processes consult some table, constructed based upon the apriory model checking analysis, that tells them, depending on the current local information, whether a current enabled transition has a maximal priority and thus can be immediately executed. This transformation only blocks some of the transitions, based on the precalculated table. Thus, it does not introduce any new executions or deadlocks, and consequently preserves all the linear temporal logic properties [9] of the system.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 26–41, 2010.

For states where no process can locally know about having a maximal priority transition, we suggest several options. One solution is to put some semi-global observers that can observe the combined situation of several processes, obtaining in this way more knowledge regarding which process has a transition with maximal priority. Another possibility is to relax the priority policy, and allow a good approximation. The priorities discussed in this paper are inspired by the BIP system (Behavior Interaction Priority) [6].

2 Preliminaries

The model used in this paper is Petri Nets. This model has a visual representation that is helpful in presenting our examples. In addition, this model is very close to the BIP model. The method and algorithms developed here can equally apply to other models, e.g., transition systems, communicating automata, etc.

Definition 1. *A* Petri Net N *is a tuple* (P, T, E, s_0) *where*

- P *is a finite set of* places. *The* states *are defined as* $S = 2^P$.
- T *is a finite set of* transitions.
- $E \subseteq (P \times T) \cup (T \times P)$ *is a bipartite relation between the places and the transitions.*
- $s_0 \subseteq P$ *is the* initial state *(hence* $s_0 \in S$*).*

For a transition $t \in T$, we define the set of input places ${}^\bullet t$ as $\{p \in P | (p, t) \in E\}$, and *output places* $t^\bullet$ as $\{p \in P | (t, p) \in E\}$.

Definition 2. *A transition* t *is* enabled *in a state* s *if* ${}^\bullet t \subseteq s$ *and* $t^\bullet \cap s = \emptyset$.

A state s is in *deadlock* if there is no enabled transition from it. We denote the fact that t is enabled from s by $s[t\rangle$.

Definition 3. *A transition* t *can be* fired *(or* executed*) from state* s *to state* s'*, which is denoted by* $s[t\rangle s'$*, when* t *is enabled at* s*. Then,* $s' = (s \setminus {}^\bullet t) \cup t^\bullet$*. We extend this notation to* $s[t_1 t_2 \dots t_k\rangle s'$*, when there is a sequence* $s[t_1\rangle s_1[t_2\rangle s_2 \dots s_{k-1}[t_k\rangle s'$*, i.e., the system moves from* s *to* s' *by firing the sequence of transitions* $t_1 t_2 \dots t_k$.

Definition 4. *Two transitions* t_1 *and* t_2 *are* independent *if* $({}^\bullet t_1 \cup t_1{}^\bullet) \cap ({}^\bullet t_2 \cup t_2{}^\bullet) = \emptyset$*. Let* $I \subset T \times T$ *be the* independence *relation. Two transitions are* dependent *if they are not independent.*

Visually, transitions are represented as lines, places as circles, and the relation E is represented using arrows. In Figure 1, there are places $p_1, p_2, \dots, p_7$ and transitions t_1, t_2, t_3, t_4. We depict a state by putting full circles, called *tokens*, inside the places of that state. In the example in Figure 1, the initial state s_0 is $\{p_1, p_2, p_7\}$. The transitions that are enabled from the initial state are a and b. If we fire transition a from the initial state, the tokens from p_1 and p_7 will be removed, and a token will be placed in p_3. In the Petri Net in Figure 1, all the transitions are dependent on each other, since they all involve the place p_7. Removing p_7, as in Figure 2, makes both a and c become independent on both b and d.

Definition 5. *An* execution *is a maximal (i.e. it cannot be extended) alternating sequence of states $s_0 t_1 s_1 t_2 s_2 \ldots$ with s_0 the initial state of the Petri Net, such that for each states s_i in the sequence, $s_i[t_{i+1}\rangle s_{i+1}$.*

We denote the executions of a Petri Net N by $exec(N)$. A state is *reachable* in a Petri Net if it appears on at least one of its executions. We denote the reachable states of a Petri Net N by $reach(N)$.

We use places also as state predicates and denote $s \models p_i$ iff $p_i \in s$. This is extended to Boolean combinations on such predicates in a standard way. For a state s, we denote by φ_s the formula that is a conjunction of the places that are in s and the negated places that are not in s. Thus, φ_s is satisfied exactly by the state s and no other state. For the Petri Net in Figure 1 we have that the initial state s satisfies $\varphi_s = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge p_7$. For a set of states $Q \subseteq S$, we can write a *characterizing formula* $\varphi_Q = \bigvee_{s \in Q} \varphi_s$ or use any equivalent propositional formula. We say that a predicate φ is an *invariant* of a Petri Net N if $s \models \varphi$ for each $s \in reach(N)$. As usual in logic, when a formula φ_Q characterizes a set of states Q and a formula $\varphi_{Q'}$ characterizes a set of states Q', then $Q \subseteq Q'$ if and only if $\varphi_Q \rightarrow \varphi_{Q'}$.

Definition 6. *A* process *of a Petri Net N is a subset of the transitions $\pi \subseteq T$ satisfying that for each $t_1, t_2 \in \pi$, such that $(t_1, t_2) \in I$, there is no reachable state s in which both t_1 and t_2 are enabled.*

We will sometimes denote in a figure the separation of transitions of a Petri Net into different processes using dotted lines. We assume a given set of processes $\mathcal{S}$ that covers all the transitions of the net, i.e., $\bigcup_{\pi \in \mathcal{S}} \pi = T$. Note that there can be multiple ways to define a set of processes for the same Petri Net. A transition can belong to several processes, e.g., when it models a synchronization between processes. Let $proc(t)$ be the set of processes to which t belongs, i.e., $proc(t) = \{\pi | t \in \pi\}$.

Definition 7. *The* neighborhood *$ngb(\pi)$ of a process π is the set of places $\bigcup_{t \in \pi} ({}^\bullet t \cup t^\bullet)$. For a set of processes $\Pi \subseteq \mathcal{S}$, $ngb(\Pi) = \bigcup_{\pi \in \Pi} ngb(\pi)$.*

In the rest of this paper, when a formula refers to a set of processes Π, we will often replace writing the singleton process set $\{\pi\}$ by writing π instead. For the Petri Net in Figure 1, there are two executions: *acbd* and *bdac*. There are two processes: the *left* process $\pi_l = \{a, c\}$ and the *right* process $\pi_r = \{b, d\}$. The neighborhood of process π_l is $\{p_1, p_3, p_5, p_7\}$. The place p_7, belonging to the neighborhood of both processes, acts as a semaphore. It can be captured by the execution of a or of b, guaranteeing that $\neg(p_3 \wedge p_4)$ is an invariant of the system.

Definition 8. *A* Petri Net with priorities *is a pair $(N, \ll)$, where N is a Petri Net and $\ll$ is a partial order relation among the transitions T of N.*

Definition 9. *A transition t has a* maximal *priority in a state s if $s[t\rangle$ and, furthermore, there is no transition r with $s[r\rangle$ such that $t \ll r$.*

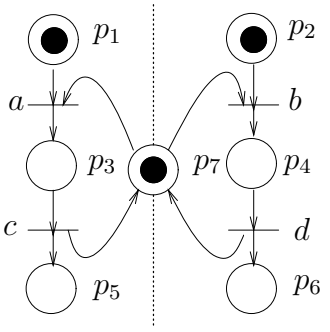


Fig. 1. A Petri Net

Definition 10. *An execution of a Petri Net with priorities is a maximal alternating sequence of states and transitions* $s_0 t_1 s_1 t_2 s_2 t_3 \ldots$ *with* s_0 *the initial state of the Petri Net. Furthermore, for each state* s_i *in the sequence it holds that* $s_i[t_{i+1}\rangle s_{i+1}$ *for* t_{i+1} *having maximal priority in* s_i*.*

To emphasize that the executions take into account the priorities, we sometimes call them *prioritized executions*. We denote the executions of a Prioritized Petri Net $(N, \ll)$ by $priorE(N, \ll)$. The set of states that appear on $priorE(N, \ll)$ will be denoted by $reach(N, \ll)$. The following is a direct consequence of the definitions:

Lemma 1. $reach(N, \ll) \subseteq reach(N)$ *and* $priorE(N, \ll) \subseteq exec(N)$.

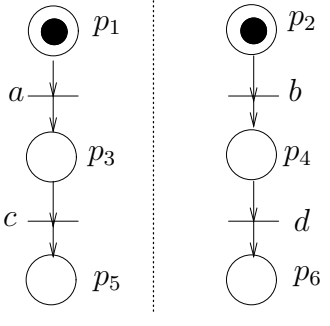


Fig. 2. A Petri Net with Priorities $a \ll d, b \ll c$

The executions of the Petri Net M in Figure 2, when the priorities $a \ll d$ and $b \ll c$ are *not* taken into account, include $abcd, acbd, bacd, badc$, etc. However, when taking the priorities into account, the prioritized executions of M are the same as the executions of the Net N in Figure 1.

Unfortunately, enforcing prioritized executions in a completely distributed way may be impossible. In Figure 2, a and c belong to one (left) process π_l, and

b and d belong to another (right) process π_r, with no interaction between the processes. Then, the left process π_l, upon having a token in p_1, cannot locally decide whether to execute a; the priorities dictate that a can be executed if d is not enabled, since a has a lower priority than d. But this information is not locally available to the left process, which cannot distinguish between the cases where the right process has a token in p_2, p_4 or p_6.

Definition 11. *The* local information *of a set of processes Π of a Petri Net N in a state s is $s|_\Pi = s \cap nbg(\Pi)$.*

That is, the local information of Π at a given state consists of the restriction of the state to the neighborhood of the transitions of Π. The local information of a process π in a state s plays the role of a *local state* of π in s. We prefer to use the term "local information" since neighborhoods of different processes may overlap on some common places rather than partitioning the global states. In the Petri Net in Figure 1, the local information of the left process in any state s consists of restriction of s to the places $\{p_1, p_3, p_5, p_7\}$. In the depicted initial state, the local information is $\{p_1, p_7\}$.

Our definition of local information is only one among possible definitions that can be used for modeling the part of the state that the system is aware of at any given moment. Consider again the Petri Net in Figure 1. The places p_1, p_3 and p_5 may represent the location counter in the left process. When there is a token in p_1 or p_3, it is reasonable to assume that the existence of a token in place p_7 (the semaphore) is known to the left process. However, it is implementation dependent whether the left process is aware of the value of the semaphore when the token is at place p_5 or not. This is because at this point, the semaphore may affect the enabledness of the right process (if it has a token in p_2) but would not have an effect on the left process. Thus, a subtly different definition of local information can be used instead. For simplicity, we will continue with the simpler definition above.

Definition 12. *Let $\Pi \subseteq \mathcal{S}$ be a set of processes. Define an equivalence relation $\equiv_\Pi \subseteq reach(N) \times reach(N)$ such that $s \equiv_\Pi s'$ when $s|_\pi = s'|_\pi$ for each $\pi \in \Pi$.*

It is easy to see that the enabledness of a transition depends only on the local information of a process that contains it, as stated in the following lemma.

Lemma 2. *If $t \in \pi$ and $s \equiv_\pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$.*

Note that this does not mean that each decision to fire a transition is made locally; rather, it states that the mutual information related to the enabledness of a transition t is made in our model a part of the local information available to each process in $proc(t)$. We cannot always make a local decision, based on the local information of processes (and sometimes sets of processes), that would guarantee only the prioritized executions in a Prioritized Petri Net $(N, \ll)$. It is possible that there are two states $s, s' \in reach(N)$ such that $s \equiv_\pi s'$, a transition $t \in \pi$ is enabled in s with maximal priority, but in s' the transition t is not maximal among the enabled transitions. This can be demonstrated on the Prioritized

Petri Net in Figure 2. There, we have that for π_l, $\{p_1, p_2\} \equiv_{\pi_l} \{p_1, p_4\}$. In the state $\{p_1, p_2\}$, a is a maximal priority enabled transition (uncomparable with b), while in $\{p_1, p_4\}$, a is not anymore maximal, as we have that $a \ll d$, and both a and d are now enabled.

In the following we will use predicates, with propositions that are the place of the Petri Net, to explain the approach and the implementation:

All the reachable states: $\varphi_{reach(N)}$.
The states where transition t is enabled: $\varphi_{en(t)}$.
At least one transition is enabled, i.e., there is no deadlock: $\varphi_{df} = \bigvee_{t \in T} \varphi_{en(t)}$.
The transition t has a maximal priority among all the enabled transitions of the system: $\varphi_{max(t)} = \varphi_{en(t)} \wedge \bigwedge_{t \ll r} \neg \varphi_{en(r)}$.
The local information of processes Π at state s: $\varphi_{s|_\Pi}$.

The corresponding sets of states can be easily computed by model checking and stored in a compact way, e.g., using BDDs.

3 Knowledge Based Approach for Priority Scheduling

The problem we want to solve is the following: given a Petri Net with priorities $(N, \ll)$, we want to obtain a Petri Net N' without priorities, such that $exec(N') \subseteq priorE(N, \ll)$. Moreover, $reach(N')$ must not introduce new deadlock states that are not in $reach(N, \ll)$.

In control theory, the transformation that takes a system and allows blocking some transitions adds a supervisor process [12], which is usually an automaton that runs synchronously with the controlled system. This (finite state) automaton observes the controlled system, progresses according to the transitions it observes, and blocks some of the enabled transitions, depending on its current state. Some of the transitions may be defined as *uncontrollable*, meaning that the controller cannot block them. Of course, the definition of uncontrollable transitions must be consistent with the priorities; if a transition is uncontrollable and is enabled in some state together with a higher priority transition, then no correct controller can be constructed. A distributed controller sets up such a supervisor per each process. In a *conjunctive supervisor* [15], in order to execute an enabled transition t that belongs to several processes, all the corresponding supervisors must agree to fire it. In a *disjunctive supervisor*, it is sufficient that at least one of the supervisors allows (supports) t.

Instead of constructing supervisors, one per single process or per a set of processes, we transform the processes of the Petri net themselves. For simplicity of the transformation, we allow Extended Petri Nets, where processes may have local variables, and each transition has an enabling condition and a transformation (see, e.g., [7]).

Definition 13. *An* Extended Petri Net *has, in addition to the Petri Net components, also finite variables V_π for each process $\pi \in \Pi$. The enabling condition*

of each transition t is augmented to include also a predicate en_t on the variables $V_t = \cup_{\pi \in proc(t)} V_\pi$. In order for t to fire, en_t must hold in addition to the usual Petri Net enabling condition on the input and output places of t. When t is executed, in addition to the usual changes to the tokens, the variables V_t are updated according to the transformation f_t that is also associated with t.

As we saw in the previous section, we may not be able to decide, based on the local information of a process or a set of processes, whether some enabled transition is maximal with respect to priority. We can exploit some model checking based analysis to identify the cases where such local decisions can be made: our approach for a local or semi-local decision on firing transitions is based on the *knowledge* of processes [3], or of sets of processes. Basically, the knowledge of a process at a given state is the properties common to all the reachable states that are consistent with the local information of that process.

Definition 14. *The processes Π (jointly)* know *a (Boolean) property ψ in a state s, denoted $s \models K_\Pi \psi$, exactly when for each s' such that $s \equiv_\Pi s'$, we have that $s' \models \psi$.*

At the moment, the definition of knowledge assumes that the processes do not maintain a log with their history. We henceforth use knowledge formulas combined with using Boolean operators and propositions. For a detailed syntactic and semantic description of logics with knowledge one can refer, e.g., to [3]. In this paper We neither define nor use the nesting of knowledge operators, e.g., $K_{\Pi_1}(K_{\Pi_2}(\varphi))$, nor the notion of "common" knowledge $C_\Pi \varphi$.

The following lemmas follow immediate from the definitions:

Lemma 3. *If $s \models K_\Pi \varphi$ and $s \equiv_\Pi s'$, then $s' \models K_\Pi \varphi$.*

Lemma 4. *The processes Π know ψ at state s exactly when $(\varphi_{reach(N)} \wedge \varphi_{s|_\Pi}) \rightarrow \psi$ is a propositional tautology.*

Now, given a Petri Net, one can perform *model checking* in order to calculate whether $s \models K_\pi \psi$. Note that implementing Lemma 4, say with BDDs, is *not* the most space efficient way of checking knowledge properties, since $\varphi_{reach(N)}$ can be exponentially big in the size of the description of the Petri Net. In a (polynomial) space efficient check, we enumerate all the states s' such that $s \equiv_\pi s'$, check reachability of s' using binary search and, if reachable, check whether $s' \models \psi$.

4 The Supporting Process Policy

The *supporting process policy*, described below, transforms a Prioritized Petri Net $(N, \ll)$ into a priorityless Extended Petri Net N' that implements or at least approximates the priorities of the original net. This transformation augments the states with additional information, and adds conditions for firing the transitions. This is related to the problem of supervisory control [12], where a supervisor is

imposed on a system, restricting transitions from being fired at some of the states. We can map the states of the transformed version N' into the states of the original version N by projecting out additional variables that N' may have on top of the places of N. In this way, we will be able to related the sets of states of the original and transformed version.

The supporting process policy can be classified as having a *disjunctive architecture for decentralized control* [15]. Although the details of the transformation are not given here, they should be clear from the theoretical explanation.

> At a state s, a transition t is *supported by a process* $\pi \in proc(t)$ only if π knows in s about t having a maximal priority (among all the currently enabled transitions of the system), i.e., $s \models K_\pi \varphi_{max(t)}$; a transition t can be fired (is enabled) in a state only if, in addition to its original enabledness condition, at least one of the processes in $proc(t)$ supports it.

Based on the definition of knowledge, we have the following monotonicity property of knowledge:

Theorem 1. *Given that $s \models K_\Pi \varphi$ in the original program N, (when not taking the priorities into account) then $s \models K_\Pi \varphi$ also in the transformed version N'.*

This property is important to ensure the maximality of the priority of a transition after the transformation. The knowledge about maximality will be calculated *before* the transformation, and will be used to control the execution of the transitions. Then, we can conclude that the maximality remains also *after* the transformation.

We consider three levels of knowledge of processes related to having a maximal enabled transition:

φ_1 Each process knows about all of its enabled transitions that have maximal priorities (among all enabled transitions).
That is, $\varphi_1 = \bigwedge_{\pi \in \mathcal{S}} \bigwedge_{t \in \pi} (\varphi_{max(t)} \rightarrow K_\pi \varphi_{\max(t)})$.

φ_2 For each process π, when one of its transitions has a maximal priority, the process knows about at least *one* such transition.
$\varphi_2 = \bigwedge_{\pi \in \mathcal{S}} ((\bigvee_{t \in \pi} \varphi_{max(t)}) \rightarrow (\bigvee_{t \in \pi} K_\pi \varphi_{\max(t)}))$.
Note that when all the transitions of each process π are totally ordered, then $\varphi_1 = \varphi_2$.

φ_3 For each state where the system is not in a deadlock, *at least one process* can identify *one* of its transitions that has maximal priority.
$\varphi_3 = \varphi_{df} \rightarrow (\bigvee_{\pi \in \mathcal{S}} \bigvee_{t \in \pi} K_\pi \varphi_{\max(t)})$.

We Denote the fact that φ is an invariant (i.e., holds in every reachable state) by using the usual temporal logic notation $\Box\varphi$ (see [9]). Notice that $\varphi_1 \rightarrow \varphi_2$ and $\varphi_2 \rightarrow \varphi_3$ hold, hence also $\Box\varphi_1 \rightarrow \Box\varphi_2$ and $\Box\varphi_2 \rightarrow \Box\varphi_3$. Processes have less knowledge according to φ_2 than according to φ_1, and then even less knowledge if only φ_3 holds.

Definition 15. *Let priorS(N, φ_i) be the set of executions of N when transitions are fired according to the supporting process policy when $\Box\varphi_i$ holds.*

That is, when $\Box\varphi_1$ holds, the processes support all of their maximal enabled transitions. When $\Box\varphi_2$ holds, the processes support at least one of their maximal enabled transitions, but not necessarily all of them. When $\Box\varphi_3$ holds, at least one enabled transition will be supported by some process, at each state, preventing deadlocks that did not exist in the prioritized net.

Lemma 5. *priorS(N, φ_1) = priorE$(N, \ll)$. Furthermore, for $i = 2$ or $i = 3$, priorS$(N, \varphi_i) \subseteq$ priorE$(N, \ll)$.*

This is because when $\Box\varphi_2$ or $\Box\varphi_3$ hold, but $\Box\varphi_1$ does not hold, then some maximally enabled transitions are supported, but some others may not. On the other hand, if $\Box\varphi_1$ holds, the supporting process policy does not limit the firing of maximal enabled transitions.

Implementing the Local Support Policy: The Support Table

We first create a *support table* as follows. We check for each process π, reachable state $s \in reach(N)$ and transition $t \in \pi$, whether $s \models K_\pi \varphi_{\max(t)}$. If it holds, we put in the support table at the entry $s|_\pi$ the transition t. In fact, according to Lemma 3, it is sufficient to check this for a single representative state containing $s|_\pi$ out of each equivalence class of '$\equiv_\pi$'. There can be multiple (or zero) transitions in an entry $s|_\pi$.

Let $\varphi_{support(\pi)}$ denote the disjunction of the formulas $\varphi_{s|_\pi}$ such that the entry $s|_\pi$ is nonempty in the support table. It is easy to see from the definition of φ_3 that checking $\Box\varphi_3$ is equivalent to checking the validity of the following Boolean implication:

$$\varphi_{df} \rightarrow \bigvee_{\pi \in \mathcal{S}} \varphi_{support(\pi)} \tag{1}$$

This means that at every reachable and non deadlock state, at least one process knows (and hence supports) at least one of its maximal enabled transitions.

Now, if at least $\Box\varphi_3$ holds, the support table we constructed for checking it can be consulted by the transformed program for implementing the supporting process policy. Each process π is equipped with the entries of this table of the form $s|_\pi$ for reachable s. Before making a transition, a process π consults the entry $s|_\pi$ that corresponds to its current local information, and supports only the transitions that appear in that entry. The transformed program can be represented as an Extended Petri Net. The construction is simple. The size of the support table is limited to the number of different local informations of the process and not to the (sometimes exponentially bigger) size of the state space.

Priority Approximation

It is typical that there will be many states where φ_3 does not hold. In the Petri Net in Figure 3, when s includes p_3 but neither p_6 nor p_7 (which are both in the

neighborhood of π_l because of the joint transition c), in the above construction π_l does not support e: e has a lower priority than j, and π_l does not know whether j is currently enabled, has terminated, or even if the nondeterministic selection at p_6 has picked up transition d, and j is not executing thereafter. Similarly, π_l does not support f because of the possibility that transition h, with the higher priority, might be enabled simultaneously.

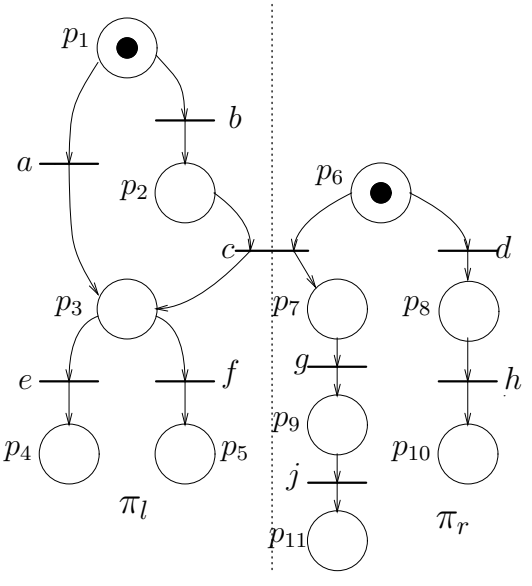


Fig. 3. Petri Net with priorities $e \ll j$ and $f \ll h$

When $\Box\varphi_3$ does not hold, one can provide various suboptimal solutions, which try to approximate the priority selection, meaning that not at all times the executed transition will be maximal. Consider a state s that does not result in a deadlock in the original net, where $s \not\models \varphi_3$. In this case, the entries $s|_\pi$ are empty for each process π, and thus in state s, no transition will be supported. Hence none will be fired, resulting in a deadlock.

A *pessimistic approach* to fix this situation, without guaranteeing completely prioritized behavior, is to add to each empty entry $s|_\pi$ *at least one* of the transitions that are maximal among the enabled transitions of π. Another possibility, which adds less arbitrary transitions to the support table, but requires more intensive computation, is based on an iterative approach. Select an empty entry $s|_\pi$ in the support table where some transition $t \in \pi$ is enabled and is maximal among the enabled transitions of π. Put t into entry $s|_\pi$ of the support table. Update the formula (1), by adding the disjunct $\varphi_{s|_\pi}$ to $\varphi_{support(\pi)}$. Then recheck Formula (1). Repeat adding transitions to empty entries in the support table until (1) holds. When it holds, it means that for each reachable state, there is a supported enabled transition, preventing new deadlocks.

Synchronizing Processes Approach

When Formula (1) does not hold, and thus also $\Box\varphi_3$, we can combine the knowledge of several processes to make decisions. This can be done by putting a supervisor that checks the combined local information of multiple processes. We then arrange a support table based on the joint local information of several processes $s|_{\Pi}$ rather than the local information of single processes $s|_{\pi}$. This corresponds to replacing π with Π in the formulas φ_1, φ_2 and φ_3. Such supervisors may reduce concurrency. However, this is not a problem if the controlled processes are threads, residing already in the same processor. It is not clear apriory on which sets of processes we want to put a supervisor in order to make their combined knowledge help in deciding the highest priority transition. Model checking under different groupings of processes, controlled and observed together, is then repeated until $\Box\varphi_1$ (or $\Box\varphi_2$ or $\Box\varphi_3$) holds.

Another possibility is the transfer of additional information via messages from one process to another. This also reduces concurrency and increases overhead. An approach that temporarily synchronize processes together in order to achieve combined knowledge, using a coordinator algorithm, is described in [8].

Using Knowledge with Perfect Recall

Knowledge with perfect recall [10] assumes that a process π may keep its own local history, i.e., the sequence of local information sequence (sequence of local states) occurred so far. This may separate different occurrences of the same local information, when they appear at the end of different local histories. This allows the processes to decide on supporting a transition even in some cases where it was not possible under the previous knowledge definition.

Knowledge with perfect recall is defined so that a process *knows some property φ at some state s and given some local history σ*, if φ holds for each execution when reaching a state with the same local history σ. In our case, since the system is asynchronous, the processes are not always aware of other processes making moves, unless these moves can affect their own neighborhood (hence their local information). Hence the local history includes only moves by transitions that have some common input or output place with $ngb(\pi)$.

Definition 16. *Let ρ be a sequence of transitions of a Petri Net N and π a process of N. Then $\rho|_{\pi}$ is obtained from ρ by erasing the transitions that are independent of all the transitions in π.*

Observe that $\rho|_{\pi}$ includes exactly the transitions of ρ that change the neighborhood of π, including the transitions of π itself.

Definition 17. *Let $\sigma = s_1 t_1 s_2 t_2 \dots t_n s_{n+1}$ be a prefix of an execution of a Petri Net N and π a process. Then $\sigma|_{\pi}$, the* local history of π according to σ, *is an alternating sequence of local informations and transitions $l_{i_1} t_{i_1} l_{i_2} t_{i_2} \dots l_{i_k}$, where $t_{i_1} t_{i_2} \dots t_{i_{k-1}} = t_1 t_2 \dots t_n|_{\pi}$, $i_k = i_{k-1} + 1$, and for each index i_j we have that $l_{i_j} = s_{i_j}|_{\pi}$.*

Thus, $\sigma|_\pi$ keeps from σ the transitions that change the neighborhood of σ, according to their order of appearance, and the local information of π just before and after each such transition. Since the system is asynchronous, π is not aware of the occurrence of any number of transitions that do not change its history. Recall that if a transition t that does not change the neighborhood of π is executed from state s, resulting in state s', then $s|_\pi = s'|_\pi$.

Now we can extend the definition of $\models$ in order to define knowledge with perfect recall.

Definition 18. *If σ is a finite prefix of an execution of a Petri Net N ending with a state s, then $\sigma \models \varphi$ exactly when $s \models \varphi$.*

We can also define an equivalence relation between finite prefixes:

Definition 19. *Let σ, σ' be two finite prefixes of a Petri Net N. Then $\sigma \equiv_\pi \sigma'$ when $\sigma|_\pi = \sigma'|_\pi$.*

This means that π observes the same alternating sequence of transitions and local informations in both σ and σ'. We are ready now to define knowledge with perfect recall.

Definition 20. *Let σ be a finite prefix of a Petri Net N. Then a process π* knows with perfect recall *ψ after σ, if for each σ' such that $\sigma \equiv_\pi \sigma'$, $\sigma' \models \psi$.*

These definitions can be generalized to sets of processes by replacing π with Π. The properties φ_1, φ_2 and φ_3 can be checked where the knowledge operators refer to knowledge with perfect recall.

An algorithm for model checking knowledge with perfect recall was shown in [10], and our algorithm can be seen as a simplified version of it.

Definition 21. *Let $indseq_\pi$ be the set of finite sequences of transitions that do not change the neighborhood of π.*

Definition 22. *Let $\mathcal{A} = (S, s_0, T)$ be a finite automaton representing the global states S of a Petri Net N, including the initial state $s_0 \in S$ and the transitions T between them. For each process π, we construct a* support automaton *$\mathcal{A}_\pi$ representing the set of states of $\mathcal{A}$ where the Petri Net N can be after a given local history. The automaton $_\pi$ has the following components:*

- *The states are 2^S.*
- *The initial state is the set of states $\{s|\exists \mu \in indseq_\pi \, s.t. \, s_0[\mu\rangle s\}$. That is, the initial state of this automaton contains all the states obtained from s_0 by executing a finite number of transitions independent of (i.e., invisible of) π.*
- *The transition relation is $\Gamma \xrightarrow{t} \Gamma'$ between two states Γ, $\Gamma' \in 2^S$ and a transition $t \in T$ as follows: $\Gamma' = \{s'|\exists s \in \Gamma \, \exists \mu \in indseq_\pi \, s.t., s[t\mu\rangle s'\}$. That is, a move from Γ to Γ' corresponds to the execution of a transition t that change the neighborhood of π followed by transitions independent of π.*

Model checking is possible even though the local histories may be unbounded because the number of such subsets Γ is bounded, and the successor relation between such different subsets, upon firing a transition t, as described above, is fixed.

Instead of the support table, for each process π we have a support automaton, representing the determinization of the above automaton. At runtime, the execution of each transition visible by π, i.e., one that can change π's neighborhood, will cause a move of this automaton (this means access to the support automaton of π with the execution of these transitions, even when they are not in π). If currently the state of the support automaton corresponds to a set of states Γ where in *all of them* the transition $t \in \pi$ is maximally enabled (checking this for the states in Γ was precalculated by model checking before performing the transformation), then π currently supports t.

Unfortunately, the size of the support automaton, for each process, can be exponential in the size of the global state space (corresponding to the possible number of subsets of states where the current execution can be given the local history). This gives quite a high overhead to such a transformation. Note that the local histories of the transformed net is a subset of the local histories of the original, priorityless net. Thus, Theorem 1 still holds when relativized to knowledge with perfect recall.

Returning to the example in Figure 3, knowledge with perfect recall can separate at a state where p_3 has the token but neither p_6 nor p_7 have a token, the case where a was executed from the case where c was executed. If a was executed, then π_l can safely support e, whose priority is comparable only with j, which is never enabled. Conversely, if c was executed, process π_l can support f, which is comparable only with h.

5 Discussion

We will now put our knowledge-based approach in the context of supervisor control synthesis. First, consider the general case where we have a concurrent system, described as a Petri Net (or a finite state transition system), on top of which we want to impose some property. The property imposed restricts the system to a particular set of states, and possibly a set of transitions allowed from any given state. This kind of restriction covers invariants and priorities. The transitions of the Petri Net are partitioned into processes, as in Definition 6, and each process can be controlled by a supervisor with local memory. Such a supervisor can observe the transitions that change the neighborhood of a process π. Indeed, in control theory, such transitions are said to be *observable* by π (and the rest are thus unobservable by π). A supervisor for π can allow or block the execution of enabled transitions that belong to π. In a conjunctive controller, a transition t must be supported by *all* the supervisors of the processes $proc(t)$, while in a disjunctive controller, it is enough that *at least one* of the controllers for this set of processes supports t in order for it to fire. In this work, instead of creating supervisors for the processes, we transform the Petri Net to achieve the same effect.

The knowledge based approach may fail to provide a distributed controller, even if one exists. Consider, for example, the Net in Figure 4. According to the knowledge approach, when the local information of π_l includes p_1 and neither

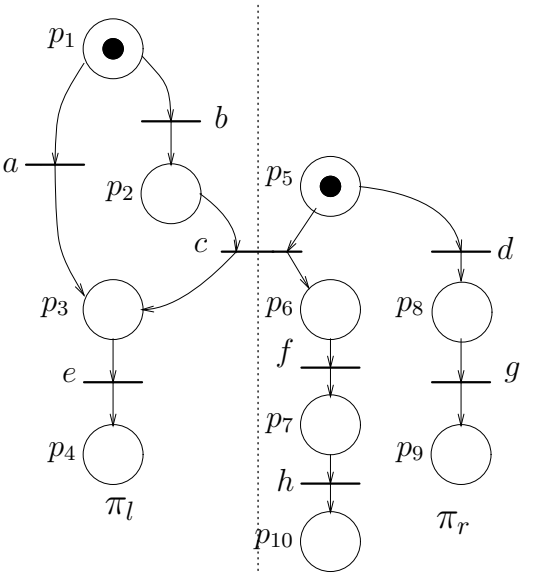


Fig. 4. Net with Priorities $e \ll h$ and $b \ll g$

p_5 nor p_6 (both of these latter places are in the neighborhood of π_l), process π_l does not know if g is enabled or not (the right process can have a token at p_7, p_8, p_9 or p_{10}). So π_l cannot support the transition b. Furthermore, when the left process has the local information that includes p_3 but neither p_5 nor p_6, we cannot support transition e, since our knowledge does not distinguish between the case where h is enabled or not. If we use knowledge of perfect recall, then when we arrive at state p_3, we know whether previously a was executed or b. If a was executed, then it is safe to execute e. But if b is executed (and subsequently c), then executing e may not be safe and adding the perfect recall does not help. Thus, our knowledge approach does not help us to construct a distributed supervisor.

Still, abandoning the knowledge approach, one can construct a distributed supervisor. This is done by having the left process π_l deciding to support only a from any local information with token in p_1. In this case, it will be safe to execute e later. In the right process π_r, the interaction c is not possible, hence d will be executed and thereafter g. In this case, there is no problem with priorities, as e is only ordered with respect to h. Abandoning b in favor of a can be based on a lookahead, like in a game strategy: although both a and b have the same priority, blocking b is done to prevent reaching into a troubled state later. A search for a game strategy is possible in the sequential case; however, the undecidability of the controller synthesis for the distributed case means that it cannot be uniformally applied here.

We have shown an example where the knowledge approach fails to construct a distributed controller for priorities even when a controller exists. However, it

is shown in [8] that the problem of deciding whether such a distributed controller exists is, in general, an undecidable problem. Thus, we propose that the knowledge based approach for constructing distributed controlers is a practical approach.

Knowledge was suggested as a tool for constructing a distributed supervisor in [13]. The problem there is different; the system needs to be controlled to behave *exactly* according to some regular language. In that paper, knowledge-controllability (termed *Kripke observability*) is studied as a basis for constructing a distributed supervisor. The requirement there is that *for each* transition, if it is enabled by the controlled system but must be blocked according to the additional constraint, then at least one process knows that fact and is thus able to prevent its execution. The construction here is different. We require that *at least* one process knows that the occurrence of *some* of its enabled transitions preserve the correctness of the imposed constraint, hence supporting its execution. The approach of [13] requires sufficient knowledge to allow *any* enabled transition that preserves the imposed constraint. Our approach preserves the correctness of the supervisor even when knowledge about other such transitions is limited, at the expense of restricting the choice of transitions.

6 Conclusions

Developing concurrent systems is an intricate task. One methodology, which lies behind the BIP system, is to define first the architecture and transitions, and at a later stage add priorities among the transitions. This methodology allows a convenient separation of the design effort. We presented in this paper the idea of using model checking analysis to calculate the local knowledge of the concurrent processes of the system about currently having a maximal priority transition. Model checking is used to transform the system into a priorityless version that implements the priorities. There are different versions of knowledge, related to the different ways we are allowed to transform the system. For example, the knowledge of each process, at a given time, may depend on including information about the history of computation.

After the analysis, we sometimes identify states where no process has enough information about having a maximal priority transition. In such cases, synchronizing between different processes, reducing the concurrency, is possible; semiglobal observers can coordinate and observe together several processes, obtaining joint knowledge of several processes. Another possible solution (not further elaborated here) involves adding coordination messages.

More generally, we suggest a programming methodology, based on a basic design (in this case, the architecture and the transitions) with added constraints (in this case, priorities). Model checking of knowledge properties is used to lift these added constraints by means of a program transformation. The resulted program behaves in an equivalent way, or approximates the behavior of the basic design with the constraints.

References

1. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed Semantics and Implementation for Systems with Interaction and Priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
2. Emerson, E.A., Clarke, E.M.: Characterizing Correctness Properties of Parallel Programs using Fixpoints. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 169–181. Springer, Heidelberg (1980)
3. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
4. Halpern, J.Y., Zuck, L.D.: A little knowledge goes a long way: knowledge based derivation and correctness proof for a family of protocols. Journal of the ACM 39(3), 449–478 (1992)
5. Hoare, C.A.R.: Communicating Sequential Processes. Communication of the ACM 21, 666–677 (1978)
6. Gößler, G., Sifakis, J.: Priority Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 443–466. Springer, Heidelberg (2004)
7. Genrich, H.J., Lautenbach, K.: System Modeling with High-level Petri Nets. Theoretical computer Science 13, 109–135 (1981)
8. Graf, S., Peled, D., Quinton, S.: Achieving Distributed Control Through Model Checking. In: CAV 2010. Springer, Heidelberg (to appear, 2010)
9. Manna, Z., Pnueli, A.: How to Cook a Temporal Proof System for Your Pet Language. In: POPL 1983, Austin, TX, pp. 141–154 (1983)
10. van der Meyden, R.: Common Knowledge and Update in Finite Environment. Information and Computation 140, 115–157 (1980)
11. Quielle, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems. In: CESAR, 5th International Symposium on Programming, pp. 337–350 (1981)
12. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM journal on control and optimization 25(1), 206–230 (1987)
13. Rudie, K., Ricker, S.L.: Know means no: Incorporating knowledge into discrete-event control systems. IEEE Transactions on Automatic Control 45(9), 1656–1668 (2000)
14. Rudie, K., Wonham, W.M.: Think globally, act locally: decentralized supervisory control. IEEE Transactions on Automatic Control 37(11), 1692–1708 (1992)
15. Yoo, T.S., Lafortune, S.: A general architecture for decentralized supervisory control of discrete-event systems. Discrete event dynamic systems, theory & applications 12(3), 335–377 (2002)

Quantitative Simulation Games*

Pavol Černý, Thomas A. Henzinger, and Arjun Radhakrishna

IST Austria (Institute of Science and Technology, Austria)

Abstract. While a boolean notion of correctness is given by a preorder on systems and properties, a quantitative notion of correctness is defined by a distance function on systems and properties, where the distance between a system and a property provides a measure of "fit" or "desirability." In this article, we explore several ways how the simulation preorder can be generalized to a distance function. This is done by equipping the classical simulation game between a system and a property with quantitative objectives. In particular, for systems that satisfy a property, a quantitative simulation game can measure the "robustness" of the satisfaction, that is, how much the system can deviate from its nominal behavior while still satisfying the property. For systems that violate a property, a quantitative simulation game can measure the "seriousness" of the violation, that is, how much the property has to be modified so that it is satisfied by the system. These distances can be computed in polynomial time, since the computation reduces to the value problem in limit average games with constant weights. Finally, we demonstrate how the robustness distance can be used to measure how many transmission errors are tolerated by error correcting codes.

1 Introduction

Classical formalizations of systems and properties are boolean: given a system and a property, the property is either true or false of the system. The classical view partitions the world into "correct" and "incorrect" systems, offering few nuances. In reality, of several systems that satisfy a property in the boolean sense, often some are more desirable than others, and of the many systems that violate a property, usually some are less objectionable than others. For instance, among the systems that satisfy the response property that every request be granted, we may prefer systems that grant requests quickly (the quicker, the better), or we may prefer systems that issue few unnecessary grants (the fewer, the better); and among the systems that violate the response property, we may prefer systems that serve many initial requests (the more, the better), or we may prefer systems that serve many requests in the long run (the greater the fraction of served to unserved requests, the better).

There is thus a natural question whether it is possible to extend the standard specification frameworks and verification algorithms to capture a finer and more

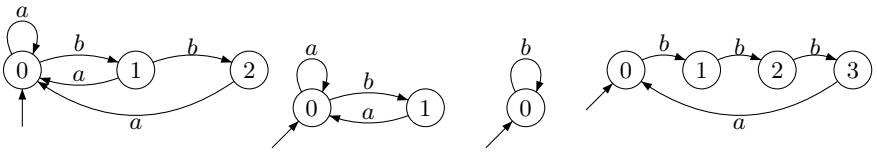
* This work was partially supported by the European Union project COMBEST and the European Network of Excellence ArtistDesign.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 42–60, 2010.

quantitative view of the relationship between specifications and systems. We focus on extending the notion of simulation to the quantitative setting. For reactive systems, the standard correctness requirement is that all executions of an implementation have to be allowed by the specification. Requiring that the specification simulates the implementation is a stricter condition, but it is computationally less expensive to check. The simulation relation defines a preorder on systems. We extend the simulation preorder to a distance function, that is, a function that given two systems returns the distance between them.

Let us consider the definition of simulation of an implementation I by a specification S as a two-player game, where Player 1 (the implementation) chooses moves (transitions) and Player 2 (the specification) tries to match each move. The goal of Player 1 is to prove that simulation does not hold, by driving the game into a state from which Player 2 cannot match the chosen move; the goal of Player 2 is to prove that there exists a simulation relation, by playing the game forever. In order to extend this definition to capture how "good" (or how "bad") the simulation is, we make the players pay a certain price for their choices. The goal of Player 1 is then to maximize the cost of the game, and the goal of Player 2 is to minimize it. The cost is given by an objective function. In this article, the limit average objective function is considered. For example, for incorrect implementations, that is those for which the specification S does not simulate the implementation I, we might be interested in how often the specification (Player 2) cannot match an implementation move. We formalize this using a game with a limit-average objective between modified systems. The specification is allowed to "cheat," by following a non-existing transition, while the implementation is left unmodified. More precisely, the specification is modified by giving the transitions from the original system a weight of 0, and adding new "cheating" transitions with a non-zero positive weight. As Player 2 is trying to minimize the value of the game, she is motivated not to cheat. The value of the game measures how often the specification can be forced to cheat by the implementation, that is, how often the implementation violates the specification (i.e., commits an error) in the worst case. We call this distance function *correctness*.

Consider now the examples in Figure 1. We take the system S_1 as the specification. The specification allows at most two symbols b to be output in the row. Now let us consider the two incorrect implementations I_3 and I_4. The implementation I_3 outputs an unbounded number of b's in a row, while the implementation I_4 can output three b's in a row. The specification S_1 will thus not be able to simulate either I_3 or I_4, but I_4 is a "better" implementation in the sense that it violates the requirement to a smaller degree. We capture this by allowing S_1 to cheat in the simulation game by taking an existing edge while outputting a different symbol. When simulating the system I_3, the specification S_1 will have to output a b when taking the edge from state 2 to state 0. This cheating transition will be taken every third move while simulating I_3. The correctness distance from S_1 to I_3 will therefore be $1/3$. When simulating I_4, the specification S_1 needs to cheat only one in four times—this is when I_4 takes a transition from its state 2 to state 3. The distance from S_1 to I_4 will therefore be $1/4$.



(a) S_1 (b) I_2 (c) I_3 (d) I_4

Fig. 1. Example Systems

Considering the implementation I_2 from the Figure 1, it is easy to see that it is correct with respect to the specification S_1. The correctness distance would thus be 0. However, it is also easy to see that I_2 does not include all behaviors allowed by S_1. Our second distance function, *coverage*, is the dual of the correctness distance. It measures how many of the behaviors allowed by the specification are actually implemented by the implementation. This distance is obtained as the value for the implementation in a game in which I is required to simulate the specification S, with the implementation being allowed to cheat. Our third distance function is called *robustness*. It measures how robust the implementation I is with respect to the specification S in the following sense: we measure how often the implementation can make an unexpected error (i.e., it performs a transition not present in its transition relation), with the resulting behavior still being accepted by the specification. Unexpected errors could be caused, for example, by a hardware problem, by a wrong environment assumption, or by a malicious attack. Robustness measures how many such unexpected errors are tolerated.

The correctness, coverage, and robustness distances can be obtained by solving the value problem in the corresponding games. The distances we considered lead to limit average games with constant weights. The value of such games can be computed in polynomial time [21].

Finally, we present an application of the robustness distance as well as an application of the coverage distance. First, we consider error correction systems for transmitting data over noisy channels and show that the robustness distance measures how many transmission errors can be tolerated by an implementation. Three implementations are analyzed, one based on the Hamming code, one based on triple modular redundancy, and an implementation without any error correction. Second, a specification of a reactive system with inputs and outputs is considered, and we use the coverage metric to determine what part of the input words for which a specification defines an output is covered by different implementations.

Related work. Weighted automata [5,6,2,11] provide a way to assign values to words, and to languages defined by finite-state systems. In contrast, we propose measuring distances between systems, and our approach provides distances on systems which cannot be obtained by calculating a measure for the two systems in question separately.

There have been several attempts to give a mathematical semantics to reactive processes which is based on quantitative metrics rather than boolean preorders [19,8]. In particular for probabilistic processes, it is natural to generalize bisimulation relations to bisimulation metrics [10,20], and similar generalizations can be pursued if quantities enter not through probabilities but through discounting [9] or continuous variables [4] (this work uses the Skorohod metric on continuous behaviors to measure the distance between hybrid systems). We consider distances between purely discrete (nonprobabilistic, untimed) systems, and our distances are directed rather than symmetric (based on simulation rather than bisimulation).

Software metrics measure properties such as lines of code, depth of inheritance (in an object-oriented language), number of bugs in a module or the time it took to discover the bugs (see for example [7,13,15]). These metrics measure syntactic properties of the source code, and are fundamentally different from our distance functions that capture the difference in the behavior (semantics) of programs.

2 Quantitative Simulation Games

2.1 Transition Systems and Games

Transition Systems. A *transition system* is a tuple $\langle S, \Sigma, E, s_0 \rangle$ where Σ is a finite alphabet, S is a finite set of states, $E \subseteq S \times \Sigma \times S$ is a set of labeled transitions between the states, and s_0 is the initial state. We require that for every $s \in S$, there exists a transition from s. The set of all transition systems is denoted by $\mathcal{S}$. A *weighted transition system* is a tuple $\langle S, \Sigma, E, s_0, v \rangle$ where S, Σ, E, and s_0 are as before, and v is a function from E to $\mathbb{Q}$. The set of all weighted transition systems is denoted by $\mathcal{S}_Q$.

A run in a transition system T is an infinite path $\rho = \rho_0 \rho_1 \rho_2 \ldots \in S^\omega$ where for all $i \geq 0$, $(\rho_i, \sigma, \rho_{i+1}) \in E$ for some $\sigma \in \Sigma$.

Game Graphs. A *game graph* G is a tuple $\langle S, S_1, S_2, \Sigma, E, s_0 \rangle$ where S, Σ, E and s_0 are as in transition systems and (S_1, S_2) is a partition of S. The choice of the next state is made by Player 1 (Player 2) when the current state is in S_1 (respectively, S_2). A *weighted game graph* is a game graph along with a weight function v from E to $\mathbb{Q}$. A run in the game graph G is called a *play*. The set of all plays is denoted by Ω.

When the two players represent the choices internal to a system, we call the game graph an *alternating transition system*. We only consider alternating transition systems where the states of Player 1 and Player 2 alternate. The set of all alternating transition systems is denoted by $\mathcal{A}$. The set of all weighted alternating transition systems is denoted by $\mathcal{A}_Q$.

Strategies. Given a game graph G, a *strategy* for Player 1 is a function $\pi : S^* S_1 \to S$ such that $\forall s_0 s_1 \ldots s_i \in S^* S_1$, we have that $(s_i, \pi(s_0 s_1 \ldots s_i)) \in E$. A strategy for Player 2 is defined in a similar way. The set of all strategies for Player i is denoted by Π_i. A play $\rho = \rho_0 \rho_1 \rho_2 \ldots$ *conforms* to a player p strategy π if $\forall i \geq 0 : (\rho_i \in S_p \implies \rho_{i+1} = \pi(\rho_0 \rho_1 \ldots \rho_i))$. The *outcome* of a Player

1 strategy π_1 and a Player 2 strategy π_2 is the unique play $out(\pi_1, \pi_2)$ that conforms to both π_1 and π_2.

Two restricted notions of a strategy are sufficient for many classes of games. A *memoryless strategy* is one where the value of the strategy function depends solely on the last state in the history, whereas a *finite-memory strategy* is one where the necessary information about the history can be summarized by a finite amount of information. Formally, a strategy π is called:

1. *Memoryless* if $\pi(w_1 s) = \pi(w_2 s)$ for all $w_1, w_2 \in S^*$ and $s \in S$.
2. *Finite-memory* if there exists a finite set M, an initial memory state $m_0 \in M$, a memory update function $\mu : S^* \times M \to M$ and move function $\nu : S \times M \to S$ such that
   (a) $\mu(ws, m_0) = \mu(s, \mu(w, m_0))$, and
   (b) $\pi(ws) = \nu(s, \mu(ws, m_0))$ for all $w \in S^*$ and $s \in S$.

Games and Objectives. A *game* consists of a game graph and a boolean or quantitative objective. A *boolean objective* is a function $\Phi : \Omega \to \{0, 1\}$. The goal of Player 1 in a game with boolean objective Φ is to choose a strategy so that, no matter what Player 2 does, the outcome maps to 1; and the goal of Player 2 is to ensure that the outcome maps to 0. A *quantitative objective* is a *value* function $f : \Omega \to \mathbb{R}$. The goal of Player 1 is to maximize the value f of the play, whereas the goal of Player 2 is to minimize it. Given a boolean objective Φ, a play ρ is *winning* for Player 1 if $\Phi(\rho) = 1$. Otherwise, it is winning for Player 2. A strategy π is a *winning strategy* for Player p if every play starting at the initial state and conforming to π is winning for Player p.

For a quantitative objective f, the value of the game for a Player 1 strategy π_1, denoted by $\nu_1(\pi_1)$, is defined as the minimum value of the outcome of the play resulting from a Player 2 strategy, i.e., $\nu_1(\pi_1) = \inf_{\pi_2 \in \Pi_2} f(out(\pi_1, \pi_2))$. The value of the game for Player 1 is defined as the supremum of the values of all Player 1 strategies, i.e., $\sup_{\pi_1 \in \Pi_1} \nu_1(\pi_1)$. The value of a Player 2 strategy π_2 and the value of the game for Player 2 are defined analogously as $\nu_2(\pi_2) = \sup_{\pi_1 \in \Pi_1} f(out(\pi_1, \pi_2))$ and $\inf_{\pi_2 \in \Pi_2} \nu_2(\pi_2)$. A strategy is an *optimal strategy* for a player if the value of the strategy for that player is equal to the value of the game.

We consider only the *limit-average* quantitative objective. Given a game graph with the weight function v and a play $\rho = \rho_0 \rho_1 \rho_2 \ldots$, for all $i \geq 0$, let $v_i = v(\langle \rho_i, \rho_{i+1} \rangle)$.

$$LimAvg(\rho) = \liminf_{n \to \infty} \frac{1}{n} \cdot \sum_{i=0}^{n-1} v_i$$

LimAvg is the long-run average of the weights occurring in a play.

Note that for *LimAvg* objectives, optimal memoryless strategies exist for both players [12].

2.2 Qualitative Simulation Games

The classical simulation preorder [17] is a useful and polynomially computable relation to compare two transition systems. In [1] this relation was extended to

alternating simulation between alternating transition systems. These relations can be cast in the form of 2-player games with qualitative objectives.

Simulation and Alternating Simulation. Consider two transition systems $A = \langle S, \Sigma, E, s_0 \rangle$ and $A' = \langle S', \Sigma, E', s'_0 \rangle$. The system A' *simulates* the system A if there exists a relation $H \subseteq S \times S'$ such that

1. $(s_0, s'_0) \in H$, and
2. $\forall s, t \in S, s' \in S' : (s, s') \in H \wedge (s, \sigma, t) \in E \Rightarrow (\exists t' : (s', \sigma, t') \in E' \wedge (s', t') \in H)$.

If A' simulates A, we write $A \leq A'$.

We define a restricted form of alternating simulation defined in [1], which is sufficient for the 2-player turn-based alternating systems we consider. For two alternating transition systems $A = \langle S, S_1, S_2, \Sigma, E, s_0 \rangle$ and $A' = \langle S', S'_1, S'_2, \Sigma, E', s'_0 \rangle$, *alternating simulation* of A by A' holds if there exists a relation $H \subseteq S \times S'$ such that:

1. $(s_0, s'_0) \in H$, and
2. $\forall s \in S, s' \in S' : (s, s') \in H \Rightarrow (s \in S_1 \Leftrightarrow s' \in S'_1)$
3. $\forall s, t \in S, s' \in S' : ((s, s') \in H \wedge s \in S_1) \Rightarrow \forall (s, \sigma, t) \in E : (\exists (s', \sigma, t') \in E' : (t, t') \in H)$.
4. $\forall s \in S, s', t' \in S' : ((s, s') \in H \wedge s \in S_2) \Rightarrow \exists (s', \sigma, t') \in E' : (\forall (s, \sigma, t) \in E : (t, t') \in H)$.

Simulation and Alternating Simulation Games. Given two (alternating) transition systems, A and A', we can construct a game $\mathcal{G}_{A,A'}$ ($\mathcal{H}_{A,A'}$) such that, (alternating) simulation of A by A' holds if and only if Player 2 has a winning strategy in $\mathcal{G}_{A,A'}$ ($\mathcal{H}_{A,A'}$). To construct simulation and alternating simulation games, we define quantitative simulation game graphs. The quantitative version of these game graphs are not necessary to define the classical simulation and alternating simulation games. However, they are introduced here as they will be used later to define quantitative simulation games.

Given two weighted transition systems A and A' with the same alphabet, we define the corresponding *quantitative simulation game graph* $G_{A,A'} = \langle S^G, S^G_1, S^G_2, \Sigma, E^G, s^G_0 \rangle$ as follows:

1. The alphabet Σ is the same as the alphabet of A and A'.
2. The state space $S^G = S \times (\Sigma \cup \{\#\}) \times S' \cup \{s_{\text{err}}\}$.
3. The states are partitioned into Player 1 and Player 2 states as follows: $(s, \#, s') \in S^G_1$, and $(s, \sigma, s') \in S^G_2$ for all $\sigma \in \Sigma$. Also, state $s_{\text{err}} \in S^G_1$.
4. The initial state is $s_0 = (s_0, \#, s'_0)$.
5. Each transition of the game graph corresponds to a transition in either A or A' as follows:
   (a) $((s, \#, s'), \sigma, (t, \sigma, s')) \in E^G \Leftrightarrow (s, \sigma, t) \in E$
   (b) $((s, \sigma, s'), \sigma, (s, \#, t')) \in E^G \Leftrightarrow (s', \sigma, t') \in E'$
   For each of the above transitions, the weight is the same as the weight of the corresponding transition in A or A'.

6. If there is no outgoing transition from a particular state, transitions to s_{err} are added with all symbols. s_{err} is a sink with transitions to itself on all symbols. Each of these transitions has weight 1.

For classical simulation games, we consider the same game graph without weights. Now, the boolean objective for the simulation game is as follows. If the play can proceed ad infinitum without reaching s_{err}, then Player 2 wins. If the play arrives at the s_{err} state, then Player 1 wins. We denote this classical simulation game as $\mathcal{G}_{A,A'}$. Intuitively, in every state, Player 1 chooses a transition of A and Player 2 has to match it by picking a transition of A'. If Player 2 cannot match at some point, Player 1 wins that play. It is easy to see that A' simulates A iff there is a winning strategy for Player 2 in $\mathcal{G}_{A,A'}$.

We can extend the simulation game to an alternating simulation game as follows. Given two quantitative alternating transition systems $A = \langle S, S_1, S_2, \Sigma, E, s_0, v\rangle$ and $A' = \langle S', S'_1, S'_2, \Sigma, E', s'_0, v'\rangle$ with the same alphabet, we define the corresponding *alternating simulation game graph* $H_{A,A'} = \langle S^H, S_1^H, S_2^H, \Sigma, E^H, s_0^H, v^H\rangle$ as:

1. The alphabet is the same as the alphabet of A and A'. The initial state is $(s_0, \#, s'_A, p)$ where p is 1 (2) if s_0 and s'_0 are both Player 1 (respectively, Player 2) states. Note that if one of them is a Player 1 state and the other is a Player 2 state, then alternating simulation of A by A' cannot hold and hence, we do not define the game graph for such cases.
2. Player 1 states of the graph are $S_1^H = \{(s, \#, s', 1) \mid s \in S_1 \wedge s_2 \in S'_1\} \cup \{(s, \sigma, s', 1) \mid s \in S_2 \wedge s' \in S'_1 \wedge \sigma \in \Sigma\} \cup \{s_{\text{err}}\}$. The first set of the union represents the states where Player 1 has to choose a transition for Player 2 to match and the second set represents the states where Player 2 has already chosen a transition with the symbol σ and Player 1 has to match it. State s_{err} is an error state.
3. Player 2 states of the graph are $S_2^H = \{(s, \#, s', 2) \mid s \in S_2 \wedge s' \in S'_2\} \cup \{(s, \sigma, s', 2) \mid s \in S_2 \wedge s' \in S'_1 \wedge \sigma \in \Sigma\}$. The sets in this union are analogous to the ones in Player 1 states.
4. The transitions correspond to A or A' transitions as follows:
   (a) If (s, σ, t) is a transition in A and $(s, \#, s', 1)$ is a Player 1 state, we have the corresponding transition $((s, \#, s', 1), \sigma, (t, \sigma, s', 2))$ in E^H, i.e., in states where Player 1 has to choose a transition of A, the A component of the state is changed to the destination of the A transition and the symbol is changed to the symbol of the A transition.
   (b) If (s', σ, t') is a transition in A' and $(s, \#, s', 2)$ is a Player 2 state, we have the corresponding transition $((s, \#, s', 2), \sigma, (s, \sigma, t', 1))$ in E^H. These transitions are similar to the previous case, but Player 2 has to choose a A' transition for Player 1 to match.
   (c) If (s, σ, t) is a transition in A and $(s, \sigma, s', 1)$ is a Player 1 state, we have the corresponding transition $((s, \sigma, s', 1), \sigma, (t, \#, s', 1))$ in E^H. Here, Player 1 chooses a transition to match the previous move of Player 2 which had the symbol σ. The A component of the state is changed accordingly and the symbol is reset to $\#$.

(d) If (s', σ, t') is a transition in A' and $(s, \sigma, s', 2)$ is a Player 2 state, we have the corresponding transition $((s, \sigma, s', 2), \sigma, (s, \#, t', 2))$ in E^H. This is the dual of the previous case.

The weight of each transition is equal to the weight of the corresponding A or A' transition.

5. If there is no outgoing transition from a particular state, a transition to s_{err} is added on all symbols. s_{err} is a sink with transitions to itself on all symbols. Each of these transitions has weight 1.

We consider the game graph without weights to define the alternating simulation game. As in the case of simulation games, the objective of Player 2 is to ensure that the play proceeds ad infinitum without reaching s_{err}, and the objective of Player 1 is to ensure that the play reaches s_{err}. We denote this qualitative alternating simulation game as $\mathcal{H}^{A,A'}$. Intuitively, as in the simulation game, whenever in a Player 1 state of A, Player 1 chooses a transition and Player 2 has to match it in A'. But, in a Player 2 state of A', Player 2 chooses a transition of A' and Player 1 matches it in A. Player 1 wins if a transition cannot be matched at some point, and Player 2 wins otherwise. Again, it can be seen that alternating simulation of A by A' holds iff there exists a winning strategy for Player 2 .

2.3 Quantitative Simulation Games

We now define a generalized notion of simulation games called quantitative simulation games. We replace the qualitative objectives of a simulation game by a *LimAvg* objective to measure distances between systems.

Quantitative Simulation Games. Given two quantitative transition systems A and A', the *quantitative simulation game* is played on the quantitative simulation game graph $G_{A,A'}$ with the objective of Player 1 being to maximize the *LimAvg* value of the play, while the objective of Player 2 being to minimize it. We denote this game as $\mathcal{Q}_{A,A'}$.

Quantitative Alternating Simulation Games. For two alternating transition systems A and A', we similarly define the *quantitative alternating simulation game* played on $H_{A,A'}$ with the same objectives as a quantitative simulation game. We denote this game as $\mathcal{P}_{A_1,A_2}$.

2.4 Modification Schemes

We will use quantitative simulation games to measure various properties of systems. For computing these properties, we need to use small modifications of the original systems. For example, when trying to compute the distance as the number of errors an implementation commits with respect to a specification, we add to the specification some recovery behavior to be used in case of error. To ensure that the specification does not use this recovery behavior when it is not necessary, there is an extra cost for using it. We encode these kind of modifications using the notion of modification schemes. However, to ensure that modifications schemes do not change the basic structure system, we impose a strict set of rules on these schemes.

A *modification scheme* is a function $m : \mathcal{S} \to \mathcal{S}_Q \cup \mathcal{A}_Q$ from transition systems to quantitative (alternating) transition systems, which can be computed using the following steps:

1. Edges may be added to the transition system.
2. Each state may be replaced by a local subgraph. The graph is the same for all states of the system. All edges of the graph, including those obtained from the previous step, have to be preserved.
3. Every edge of the system is associated with a weight from $\mathbb{Q}$.

The above rules ensure that the modified system retains the structure of the original system. We present two examples of modification schemes.

Output Modification. This scheme is used to add behavior to a system that allows it to output an arbitrary symbol while moving to a state specified by an already existing transition. For every transition (s, σ, s'), transitions with different symbols are added to the system i.e., $\{(s, \alpha, s') \mid \alpha \in \Sigma\}$. These transitions are given an weight of 2 to prohibit their free use. All other transitions have the weight zero. Given a transition system T, we denote the modified system as $\mathit{OutMod}(T)$.

Error Modification. In a perfectly functioning system, errors may occur due to unpredictable events. We model this with an alternating transition system with one player modeling the original system (Player 1) and the other modeling the controlled error (Player 2). At every state, Player 2 chooses whether or not a error occurs by choosing one of the two successors. From one of these states, Player 1 can choose the original successors of the state and from the other, she can choose either one of the original successors or one of the error transitions. Therefore, Player 2 controls the possibility of an error occurring, whereas Player 1 actually chooses the transition the system takes. We penalize Player 2 for the choice of not allowing errors to happen.

Given $T = \langle S, \Sigma, E, s_0 \rangle$ we define $\mathit{ErrMod}(T)$ to be the quantitative alternating transition system obtained after the following steps.

1. All transitions that differ from existing transitions only in the symbol are added to the system as in *OutMod*.
2. The graph used to replace each state s is shown in Figure 2.
3. Only the transitions labeled $\neg c$ are given the weight 2. The rest are given the weight 0.

In addition to these modification schemes, we define the trivial modification scheme where no changes are made to the transitions of the system and every edge is given the weight 0. We call this scheme *NoMod*.

3 Simulation Distances

We present here examples of distances that can be defined using quantitative simulation games.

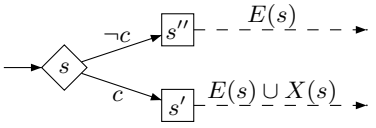


Fig. 2. Graph for *ErrMod*

3.1 Correctness

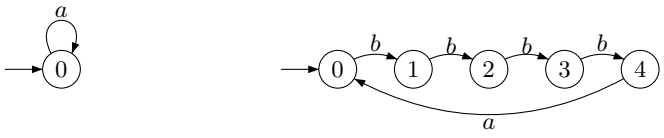
Given a specification T_2 and an implementation T_1, such that T_2 is incorrect with respect to T_1, the correctness distance measures the degree of incorrectness. The boolean simulation relation between systems can determine whether the behavior of one system can be simulated by another. However, this relation is very strict in a certain way. Even a single nonconformant symbol can destroy this relation. Here we present a game which is not as strict and measures the minimal number of required errors, i.e. the minimal number of times the specification has to use nonmatching symbols when simulating the implementation.

Definition 3.1 (Correctness Distance). *The* correctness distance $d_{cor}(T_1, T_2)$ *from transition system* T_1 *to transition system* T_2 *is the Player 1 value of the quantitative simulation game* $\mathcal{C}_{T_1,T_2} = \mathcal{Q}_{NoMod(T_1),OutMod(T_2)}$.

The game $\mathcal{C}$ can be intuitively understood as follows. Given two transition systems T_1 and T_2, we are trying to simulate the system T_1 by T_2, but the specification T_2 is allowed to make errors. The implementation T_1 tries to make the specification commit as many errors as possible. Every move by Player 1 chooses a transition of T_1. Every matching move of Player 2 is a zero weight transition of T_2. If Player 2 cannot match a move, she must still choose an existing transition an incurs a weight of 2. (Other error models are possible where Player 2 can use a completely new transition.) Player 2 tries to show that the number of errors of T_1 is as small as possible for all strategies of Player 1, i.e., all behaviors of the implementation. If the implementation is correct (T_2 simulates T_1), then the correctness distance is 0. Otherwise, the value of the game is the *LimAvg* of the number of errors, i.e., the long run average of the errors.

We present a few example systems and their distances here to demonstrate the fact that the above game measures distances that correspond to intuition. In Figure 3 and Figure 1, S_1 is the specification system against which we want to measure the rest of the systems. In this case, the specification says that there cannot be more than two b's in a row. The distances between these systems according to the *LimAvg* correctness game are summarized in Table 1.

Among the systems which do not satisfy the specification, i.e. I_3 and I_4, we can intuitively see that I_3 is worse than I_4 in the sense that I_3 violates the specification that there are no more than two b's in a row more often than I_4. This fact is reflected in the distances as I_3 is more distant from S_1 than I_4. However, surprisingly the distance to I_5 is less than the distance to I_4. In fact, the distances reflect on the long run the number of times the specification has to err to simulate the implementation.



(a) I_1 (b) I_5

Fig. 3. Example Systems

Table 1. Distances according to the Correctness game

T_1	T_2	$d_{\text{cor}}(T_1, T_2)$
S_1	S_1	0
S_1	I_1	0
S_1	I_2	0
S_1	I_3	1/3
S_1	I_4	1/4
S_1	I_5	1/5

3.2 Coverage

Now, we present the dual game of the one presented above. Here, we measure the behaviors that are present in one system but not in the other system. Given a specification T_2 and an implementation T_1, the coverage distance corresponds to the behavior of the specification which is farthest from any behaviour of the implementation. Hence, we have that the coverage distance from a system T_1 to a system T_2 is the correctness distance from T_2 to T_1.

Definition 3.2 (Coverage Distance). *The* coverage distance $d_{cov}(T_1, T_2)$ *from transition system* T_1 *to transition system* T_2 *is the Player 1 value of the quantitative simulation game* $\mathcal{V}_{T_1,T_2} = \mathcal{Q}_{NoMod(T_2),OutMod(T_1)}$.

$\mathcal{V}$ measures the distance from T_1 to T_2 as the minimal number of errors that have to be committed by T_1 to cover all the behaviors of T_2.

Again, we present examples of systems and their distances according to $\mathcal{V}$. We use the systems from the examples in Figure 3 and Figure 1. The distances are summarized in Table 2.

3.3 Robustness

Given a specification system and a correct implementation of the specification, the notion of robustness presented here is a measure of the number of errors by the implementation that makes it nonconformant to the specification. The more such errors tolerated by the specification, the more robust the implementation is with respect to the specification. In other words, the distance measures the number of critical points, or points where an error will lead to an unacceptable behavior. The lower the value of the robustness distance to a given specification,

Table 2. Distances according to the Coverage game

T_1	T_2	$d_{\text{cov}}(T_1, T_2)$
S_1	S_1	0
S_1	I_1	2/3
S_1	I_2	1/3
S_1	I_3	1
S_1	I_4	3/4
S_1	I_5	4/5

the more robust an implementation is. In case of an incorrect implementation, the simulation of the implementation does not hold irrespective of whether implementation commits errors. Hence, in that case, the robustness distance will be 1.

Definition 3.3 (Robustness Distance). *The* robustness distance $d_{rob}(T_1, T_2)$ *from transition system* T_1 *to transition system* T_2 *is the Player 1 value of the quantitative alternating simulation game* $\mathcal{R}_{T_1,T_2} = \mathcal{P}_{ErrMod(T_1),ErrMod_\emptyset(T_2)}$.

The game $\mathcal{R}_{ErrMod(T_1),ErrMod_\emptyset(T_2)}$ is simple and is played in the following steps:

1. The specification T_2 chooses whether the implementation T_1 is allowed to make an error.
2. The implementation chooses a transition on the implementation system. She is allowed to err based on the specification choice in the previous step.
3. Specification chooses a matching move to simulate the implementation.

The specification tries to minimize the number of moves where she prohibits the implementation to commit errors (without destroying the simulation relation), whereas the implementation tries to maximize it. Intuitively, the positions where the specification cannot allow errors are the critical points for the implementation.

Let us examine the examples from Figure 1 and Figure 3 in detail. In the game played between S_1 and S_1, every position is critical. At each position, if an error is allowed, the system can output three b's in a row by using the error transition to return to state 0 while outputting a b. The next two moves can be b's irrespective whether errors are allowed or not. This breaks the simulation. Now, consider I_1. This system can be allowed to err every two out of three times

Table 3. Distances according to the Robustness game

T_1	T_2	$d_{\text{rob}}(T_1, T_2)$
S_1	S_1	1
S_1	I_1	1/3
S_1	I_2	2/3
S_1	I_3	1
S_1	I_4	1
S_1	I_5	1

without violating the specification. This shows that I_1 is more robust than S_1 for implementing S_1. The list of distances is summarized in Table 3.

3.4 Computation of Simulation Distances

The computational complexity of computing the three distances defined here is the same as solving the value problem for the respective games.

The d_{cor}, d_{cov} and d_{rob} games are simple graph games with *LimAvg* objectives. The decision problem (deciding whether the value is greater than a given value) for these games is in NP $\cap$ co-NP [21], but no PTIME algorithm is known. However, for *LimAvg* objectives the existence of a pseudo-polynomial algorithm, i.e., polynomial for unary encoded weights, implies that the computation of the distances can be achieved in polynomial time. This is due to the fact that we use constant weights. Using the algorithm of [21] d_{cor}, d_{cov} and d_{rob} distances can be computed in time $O((|S||S'|)^3 \cdot (|E||S| + |E'||S|))$ where S and S' are state spaces of the two transition systems; and E and E' are the sets of transitions of the two systems.

4 Applications of Distances

In this section, we present two examples of application of the distances defined in Section 3 to measure interesting properties of larger systems. In Section 4.1, we show examine forward error correction systems for bit streams and show a relation between their robustness measured by $\mathcal{R}$ and the bit-error rate they can tolerate. In Section 4.2, we measure the coverage of a number of implementations of a request-grant system with respect to a specification and illustrate how d_{cov} measures the restriction placed on the environment by the implementations.

4.1 Forward Error Correction Systems

Forward Error Correction systems are a mechanism of error control for data transmission on noisy channels [18]. A very important characteristic of these error correction systems is the *maximum tolerable bit-error rate*, which is the maximum number of errors the system can tolerate while still being able to successfully decode the message. We show that this property can be measured as the d_{rob} distance between a system and an ideal system (specification).

We will examine three forward error correction systems: one with no error correction facilities, the Hamming(7,4) code [14], and triple modular redundancy [16]. Intuitively, each of these systems is at a different point in the trade-off between efficiency of the transmission and the tolerable bit-error rate. By design, the system with no error correction can tolerate no errors and the Hamming(7,4) system can tolerate one error in seven bits and the triple modular redundancy system can tolerate one error in three bits. However, the overhead incurred for transmission of messages increases with increasing error tolerance. The system with no error correction uses no extra bits while, the Hamming(7,4) system and

No error correction

proc *sender*$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$ $\equiv$
  call *send*$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$; **.**
proc *receiver*() $\equiv$
  call *receive*$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$;
  return$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$**.**

Hamming(7,4) error correction

proc *sender*$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$ $\equiv$
  $\mathbf{P}_0 := \mathbf{B}_0 \oplus \mathbf{B}_1 \oplus \mathbf{B}_3$
  $\mathbf{P}_1 := \mathbf{B}_0 \oplus \mathbf{B}_2 \oplus \mathbf{B}_3$
  $\mathbf{P}_2 := \mathbf{B}_1 \oplus \mathbf{B}_2 \oplus \mathbf{B}_3$
  call *send*$(\mathbf{P}_0, \mathbf{P}_1, \mathbf{B}_0, \mathbf{P}_2, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$; **.**
proc *receiver*() $\equiv$
  call *receive*$(\mathbf{P}_0, \mathbf{P}_1, \mathbf{B}_0, \mathbf{P}_2, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$;
  $\mathbf{P}_0 := \mathbf{P}_0 \oplus \mathbf{B}_0 \oplus \mathbf{B}_1 \oplus \mathbf{B}_3$;
  $\mathbf{P}_1 := \mathbf{P}_1 \oplus \mathbf{B}_0 \oplus \mathbf{B}_2 \oplus \mathbf{B}_3$;
  $\mathbf{P}_2 := \mathbf{P}_2 \oplus \mathbf{B}_1 \oplus \mathbf{B}_2 \oplus \mathbf{B}_3$;
  $\mathbf{B}_0 := \mathbf{B}_0 \oplus (\neg\mathbf{P}_0.\mathbf{P}_1.\neg\mathbf{P}_2)$;
  $\mathbf{B}_1 := \mathbf{B}_1 \oplus (\mathbf{P}_0.\neg\mathbf{P}_1.\mathbf{P}_2)$;
  $\mathbf{B}_2 := \mathbf{B}_2 \oplus (\mathbf{P}_0.\mathbf{P}_1.\neg\mathbf{P}_2)$;
  $\mathbf{B}_3 := \mathbf{B}_3 \oplus (\mathbf{P}_0.\mathbf{P}_1.\mathbf{P}_2)$;
  return$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$**.**

Triple modular redundancy

proc *sender*$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$ $\equiv$
  call *send*$(\mathbf{B}_0, \mathbf{B}_0, \mathbf{B}_0)$;
  call *send*$(\mathbf{B}_1, \mathbf{B}_1, \mathbf{B}_1)$;
  call *send*$(\mathbf{B}_2, \mathbf{B}_2, \mathbf{B}_2)$;
  call *send*$(\mathbf{B}_3, \mathbf{B}_3, \mathbf{B}_3)$; **.**
proc *receiver*() $\equiv$
  call *receive*$(\mathbf{B}_{01}, \mathbf{B}_{02}, \mathbf{B}_{03})$;
  call *receive*$(\mathbf{B}_{11}, \mathbf{B}_{12}, \mathbf{B}_{13})$;
  call *receive*$(\mathbf{B}_{21}, \mathbf{B}_{22}, \mathbf{B}_{23})$;
  call *receive*$(\mathbf{B}_{31}, \mathbf{B}_{32}, \mathbf{B}_{33})$;
  $\mathbf{B}_0 := \mathbf{B}_{01}.\mathbf{B}_{02} \vee \mathbf{B}_{02}.\mathbf{B}_{03} \vee \mathbf{B}_{03}.\mathbf{B}_{01}$;
  $\mathbf{B}_1 := \mathbf{B}_{11}.\mathbf{B}_{12} \vee \mathbf{B}_{12}.\mathbf{B}_{13} \vee \mathbf{B}_{13}.\mathbf{B}_{11}$;
  $\mathbf{B}_2 := \mathbf{B}_{21}.\mathbf{B}_{22} \vee \mathbf{B}_{22}.\mathbf{B}_{23} \vee \mathbf{B}_{23}.\mathbf{B}_{21}$;
  $\mathbf{B}_3 := \mathbf{B}_{31}.\mathbf{B}_{32} \vee \mathbf{B}_{32}.\mathbf{B}_{33} \vee \mathbf{B}_{33}.\mathbf{B}_{31}$;
  return$(\mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3)$**.**

Fig. 4. Forward Error Correction Algorithms

the triple modular redundancy system use 3 and 8 extra bits for transmitting a four bit message. We compute the values of the error tolerance by measuring robustness with respect to an ideal system which can tolerate an unbounded number of errors.

The pseudo-code for the three systems we are examining is presented in Figure 4. The basic architecture of each system is the same. Each system has a four bit input and an encoder which adds the error correction bits to these. Then, the bits are multiplexed and transmitted over a noisy channel. The bits received on the other side of the channel are de-multiplexed and decoded and output. Each system is split into the sender and receiver. The only errors we allow are bit flips during transmission.

The transition systems for these systems are constructed according to the following rules:

1. The state space of the system is $\{0, 1, \#\}^n \times \{0, 1, \#\}^m$ where n and m are constants specific to the system. The first component is the list of bits to be transmitted by the sender, and the second component is the list of bits already received by the receiver. The initial state is $(\#^n, \#^m)$.

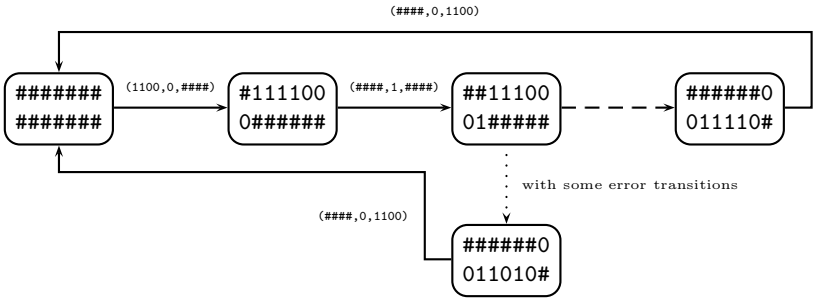


Fig. 5. Part of the transition graph for Hamming(7,4) system

Table 4. Robustness of FEC systems

T_1	T_2	$d_{\text{rob}}(T_1, T_2)$
No error correction	Ideal System	1
Hamming (7,4)	Ideal System	6/7
Triple modular redundancy	Ideal System	2/3

2. The alphabet for the transition systems consist of $\Sigma = \{0, 1, \#\}^4 \times \{0, 1\} \times \{0, 1, \#\}^4$. Here, the first part of the symbol is the input received at the sender, the second part is the bit that is transmitted and the third part is the output at the receiver.
3. All the actions except the transmission are considered to happen instantaneously, as they are local and have negligible error rates.
4. Bit flips can occur during the transmission and the state is changed according to the bit received. These transitions which have a bit flip are considered as erroneous transmissions. To measure the robustness of the system, we will be using the modification scheme *ErrMod*. transitions.

Example. Suppose we are working with the Hamming(7,4) system. Let us examine the transmission of the bit block 1100. The encoded bit string for this block is 0111100. Now, from the initial state $(\#^7, \#^7)$, on the input 1100, the transmitted bit is 0 (the first bit of the encoded string) and the state changes to $(\#111100, 0\#\#\#\#\#\#)$ (assuming no errors). From this state, we go on the symbol $(\#\#\#\#, 1, \#\#\#\#)$ to the state $(\#\#11100, 01\#\#\#\#\#)$ and so on. The outline of the set of states and transitions for this transmission is illustrated in Figure 5.

The values of $d_{\text{rob}}$of these systems measured against the ideal system are summarized in Table 4. As the table shows, the robustness measured are what one would expect from the systems. The system which uses no error correction is the least robust as it cannot tolerate even a single bit error. The Hamming(7,4) system does better as it can tolerate one error in seven bits on the long run, whereas the system which uses Triple modular redundancy can tolerate one error in three bits. The robustness values clearly mirror the error tolerance values as each robustness value is equal to $1 - e$ where e is the corresponding error tolerance value.

4.2 Environment Restriction for Reactive Systems

In reactive systems, the transitions of the system are controlled by two agents, the system itself and the environment. The system has no control over the actions of the environment. Hence, while considering the refinement of a specification for a reactive system, care has to be taken to ensure that apart from the fact that all behaviors of the implementation are simulated by the specification, but also that the behavior of the environment is not restricted more than in the specification. A number of refinements of the classical simulation relation have been suggested to include this requirement, such as ready simulation [3].

We propose here a method to measure the amount of restriction the implementation system places on the environment over and above the restriction in the specification. The measure proposed here not only takes into consideration the languages of the two systems (restricted to the environment actions), but also the distance of the farthest unimplemented behavior in the implementation. For example, consider a specification that allows the environment behavior r_1^ω and two implementations I_1 and I_2 that do not allow it. However, say I_1 allows the behavior $(r_1 r_2)^\omega$ whereas I_2 allows only r_2^ω, the implementation I_1 will be given a higher rating than the implementation I_2.

The way we will measure the amount of environment restriction is using the coverage distance (d_{cov}) introduced in Section 3. We model a reactive system with inputs and outputs as a transition systems with the alphabet $\Sigma^{\mathcal{I}} \cup \Sigma^{\mathcal{O}}$ (where

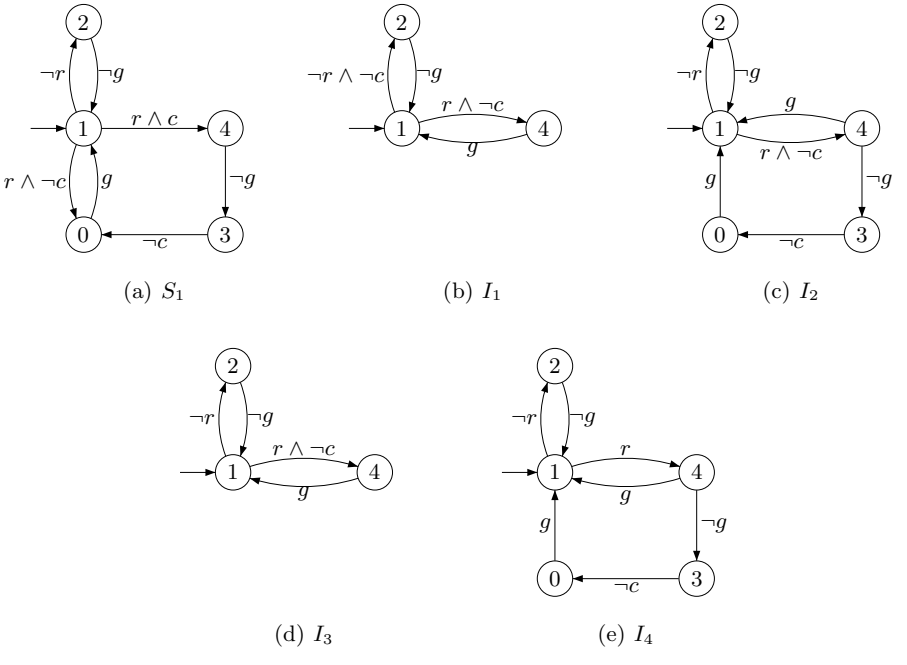


Fig. 6. Request-grant Systems

Table 5. Restrictiveness of request-grant systems

T_1	T_2	$d_{\text{cov}}(T_1, T_2)$
S_1	S_1	0
S_1	I_1	1/2
S_1	I_2	1/4
S_1	I_3	1/4
S_1	I_4	0

$\mathcal{I}$ and $\mathcal{O}$ are the environment actions (inputs) and system actions (outputs) respectively), and the transitions labeled with $\mathcal{I}$ and $\mathcal{O}$ alternate. To measure the excessive restriction on the environment, we project out the $\mathcal{O}$ symbols (as we are not interested in correctness) and then compute the d_{cov} distance between the system and the implementation. We demonstrate that this method of measuring environment restriction by computing the distances for a *request-grant* system.

Consider the specification S_1 and the implementations I_n in the Figure 6. All these systems are built so that every request r is granted by g in the same step or in the next step. However, if cancel c is high, there should be no grant in that step. These requirement mandatorily forbids some environment behaviors. For example, the input behavior with both r and c high all the time does not have any valid output behavior. The specification S_1 restricts the environment so that for every request r, cancel c is low in the current or the following step. This is the most permissive restriction possible. Implementations I_1, I_2, I_3 and I_4 restrict the environment to various amounts by allowing no cancels at all, allowing no cancels for the relevant two steps, allowing no cancel when there is a request, and allowing no cancel for the step following a request respectively. The restrictiveness as measured by the $\mathcal{V}$ is summarized in Table 5.

The values in Table 5 reflects the intuitive notion that I_1 is the most restrictive, followed by I_2 and I_3, which are equally restrictive and then by I_4 which allows all the input behaviors of the specification.

5 Conclusion

We have motivated the notion of distance between two systems or between a system and a specification, and introduced quantitative simulation games as a framework for measuring such distances. We presented three particular distances — two for quantifying aspects of correct systems, namely coverage and robustness; and one for measuring the degree of correctness of an incorrect system.

There are several possible directions for future work. The theoretical aspects of the quantitative simulation game framework need to be developed. In the boolean setting, the simulation relation establishes a preorder on systems. A preorder is a reflexive and transitive relation; a generalization to the quantitative setting would be a directed metric, that is, a distance function such that the distance of an object to itself is zero, and such that it conforms to the triangle inequality property. We will investigate whether these properties hold for the

distance function we defined. Furthermore, we plan to investigate how the distances between systems change under certain transformations, such as parallel composition or abstraction. Another interesting question is how to synthesize a system that minimizes a distance from a given specification — for example, given a specification, one might be interested in synthesizing the most robust system conforming to the specification. Further possibilities include building a tool for measuring the robustness distance for programs or protocols implementing various error recovery or error correction mechanisms.

References

1. Alur, R., Henzinger, T., Kupferman, O., Vardi, M.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
2. Bloem, R., Chatterjee, K., Henzinger, T., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
3. Bloom, B.: Ready simulation, bisimulation, and the semantics of CCS-like languages. PhD thesis, MIT (1989)
4. Caspi, P., Benveniste, A.: Toward an approximation theory for computerised control. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 294–304. Springer, Heidelberg (2002)
5. Chatterjee, K., Doyen, L., Henzinger, T.: Quantitative languages. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 385–400. Springer, Heidelberg (2008)
6. Chatterjee, K., Doyen, L., Henzinger, T.: Expressiveness and closure properties for quantitative languages. In: LICS, pp. 199–208 (2009)
7. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Trans. Software Eng. 20(6), 476–493 (1994)
8. de Alfaro, L., Faella, M., Stoelinga, M.: Linear and branching system metrics. IEEE Trans. Software Eng. 35(2), 258–273 (2009)
9. de Alfaro, L., Henzinger, T., Majumdar, R.: Discounting the future in systems theory. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1022–1037. Springer, Heidelberg (2003)
10. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled markov processes. Theor. Comput. Sci. 318(3), 323–354 (2004)
11. Droste, M., Gastin, P.: Weighted automata and weighted logics. Theor. Comput. Sci. 380(1-2), 69–86 (2007)
12. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. International Journal of Game Theory, 163–178 (1979)
13. Fenton, N.: Software Metrics: A Rigorous and Practical Approach, Revised (Paperback). Course Technology (1998)
14. Hamming, R.W.: Error detecting and error correcting codes. Bell System Tech. J. 29, 147–160 (1950)
15. Lincke, R., Lundberg, J., Löwe, W.: Comparing software metrics tools. In: ISSTA, pp. 131–142 (2008)
16. Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM J. Res. Dev. 6(2), 200–209 (1962)

17. Milner, R.: An algebraic definition of simulation between programs. In: IJCAI, pp. 481–489 (1971)
18. Shannon, C.E.: A Mathematical Theory of Communication. Bell System Technical Journal 27 (1948)
19. van Breugel, F.: An introduction to metric semantics: operational and denotational models for programming and specification languages. Theor. Comput. Sci. 258(1-2), 1–98 (2001)
20. van Breugel, F., Worrell, J.: Approximating and computing behavioural distances in probabilistic transition systems. Theor. Comput. Sci. 360(1-3), 373–385 (2006)
21. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theor. Comput. Sci. 158(1&2), 343–359 (1996)

The Localization Reduction and Counterexample-Guided Abstraction Refinement

Edmund M. Clarke[1], Robert P. Kurshan[2], and Helmut Veith[3]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
[2] Cadence Design Systems, Inc., New York, NY 10014
[3] Formal Methods in Systems Engineering, Vienna University of Technology, Austria

Abstract. Automated abstraction is widely recognized as a key method for computer-aided verification of hardware and software. In this paper, we describe the evolution of counterexample-guided refinement and other iterative abstraction refinement techniques.

1 Introduction

The state explosion problem is still the major disadvantage of Model Checking. One of the most successful ways of dealing with this problem is to use abstraction to create a conservative abstraction and check it instead of the original model. Finding the right abstraction is nontrivial. If the abstraction is too coarse, there may be false negatives. On the other hand, if the abstraction is too precise, the resulting model may still be too large. One often used approach for finding a satisfactory model is to combine abstraction and refinement in an iterative manner. The first such iterative procedure was the Localization Reduction for models with state variables proposed by R. Kurshan in his 1995 book Computer-Aided Verification of Coordinating Processes [Kur94]. At CAV 2000, E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith generalized the Localization Reduction in a paper entitled Counterexample-Guided Abstraction Refinement [CGJ$^+$00]. Their new method, called CEGAR, combined the iterative refinement principle of the localization reduction with ideas from abstraction-based model checking [CGL94] and predicate abstraction [GS97] into a model checking framework for arbitrary Kripke structures and universal CTL* formulas that is also applicable to infinite-state systems, and thus, to software.

In this paper, we describe the evolution of counterexample-guided refinement and other iterative abstraction refinement techniques with an emphasis on the early history of the method. We also discuss later developments such as CEGAR for software and alternative approaches for generating good abstractions that do not involve the generation of counterexamples.

Sections 2 and 4 are contributed by Bob Kurshan, Section 3 by Ed Clarke and Helmut Veith.

2 Localization Reduction

Localization reduction [Kur94, Kur00] evolved from an algorithm for the verification of timed automata, based on successive approximations [AIKY93] (inspired

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 61–71, 2010.

by Newton's method). It provides an iterative algorithm for design abstraction, relative to a property to be verified.

The localization reduction algorithm iterates over abstractions determined by counterexamples on successive refinements. Ed Clarke *et al* used the iterative refinement technique in CEGAR [CGJ+00] described in the following section. There are many interesting related procedures, *e.g.*, [DD01, JM05]. In a significant SAT-based improvement [MA03], the successive abstractions are determined not by the counterexamples but by the unSAT core used to refute falsification in the original model at a given depth. This "Abstraction-Refinement" loop has led to further improvements of this basic idea, driven by the strength of the SAT solver in finding efficient refutations [AM04, AM07]. Through the successive improvements, the SAT solver is brought into play more and more as a deductive reasoning engine. We may speculate about the future: next, perhaps, quantifiers will be supported. Eventually, automated theorem proving may reemerge through this thrust as truly automated deduction inspired by DPLL-style deductive procedures.

The original localization reduction algorithm was implemented in 1992 into the COSPAN model checker [HHK96], when the second author (Kurshan) worked in Bell Labs Research. However, he was barred from publishing the details on account of the utilization of COSPAN as the model checker of FormalCheck [For97]. FormalCheck, a design automation tool, was then under development by Lucent Technologies, the parent company of Bell Labs, for internal use and possible eventual commercial licensing. (FormalCheck was in fact released commercially by Lucent in 1997.) To comply with this restriction, the localization reduction algorithm was first published only in a very general form in [Kur94].

In November 25, 1997, the details of the algorithm were published as US patent no. 5,691,925 entitled "Deriving tractable sub-system for model of larger system". It was issued to R. P. Kurshan and R. H. Hardin, who had worked out many of the details that made the algorithm effective.

Once published as a patent, it might have been allowed to discuss the details publicly. However, there was yet some additional sensitivity surrounding this matter: it was planned to license FormalCheck to be marketed by an EDA company (to be chosen at an auction), and notwithstanding the issuance of the patent, the managers of this plan were against further publication of the algorithm at that time. Indeed, in 1998 FormalCheck was licensed by Lucent to be marketed by Cadence Design Systems. With that, the same reluctance to allow publication passed to Cadence.

A bit anti-climactically, in 2000 the second author finally was allowed to present the details publicly. This was done in a talk at the IBM FV'2000 Seminar in Haifa, Israel (August 15-17, 2000). In the same year, [CGJ+00] had been published. There is a cautionary message in this recounting concerning the relative freedom to publish in an academic institution, compared with even such an exemplary research institution as Bell Labs was at the time.

The overview of the localization reduction algorithm published in [Kur94] could be described as follows.

All properties and the design are described by ω-automata, defined in terms of program variables with their respective assignments. (These variables were implicit in [Kur94], which instead described the algorithm in terms of the underlying Boolean algebra of all variable assignments, in keeping with the presentation style of that book.)

The design can be structured in terms of its *variable dependency graph*: the directed graph whose nodes are the design variables and whose directed edges indicate a dependency of the one variable on the other. Of special interest are the connected components of this graph that consist of directed paths to a variable used to define a property with respect to which the design is to be checked. Together, these components comprise the *fan-in cone* of the property (in the design).

Any part of the design outside the fan-in cone of a property cannot influence the truth of the property on the design (at least for safety properties – for general properties, it is more complicated, but the same general idea applies, after tracing variable dependencies from automata acceptance structures).

Initially, any part of the design outside the fan-in cone of the property to be checked is removed, forming the *pruned design*.

The property is first checked in the *top* abstraction comprised of the null design with all design variables free (unconstrained) – *i.e.*, the property is checked with no design. If the property passes, it is a tautology. Otherwise, there is an error track (involving only the variables used to express the property). Using the greedy routine follow described below, a quick attempt is made to lift the error track to an error track of the full pruned design. This attempt will likely fail (in this initial iteration); the cause of the failure would be a reachable state of the pruned design from which no input can cause a transition to the next state of the error track.

There will be some design variables which, had they been free, could have been assigned values that would allow the next transition of the lifted error track. A heuristically small set of such variables is (somehow) identified. Let's call these *blocking variables*. In the variable dependency graph, they will comprise (connected) paths to the *active* variables – the variables of the current abstraction that carry their respective full assignments, including the variables that define the property being checked. (Initially, the set of active variables consists of just the set of variables that define the property to be checked.) This set of variables: the active variables plus the identified blocking variables, together with their full assignments, forms the next abstraction refinement. With respect to the variable dependency graph, all design variables on the boundary of this refinement are freed *i.e.*, assigned nondeterministically within their respective ranges). Let's call the set of these freed variables the *free fence* of the new abstraction. Although it was not discussed in [Kur94], in the COSPAN implementation, the blocking variables were chosen so as to minimize the size of the resulting free fence, in a manner described below.

The algorithm iterates monotonically through successive refinements until either an error track is successfully lifted to the full pruned design (by follow), or the property passes in some abstraction (or a computer memory limitation is hit).

Effectively, this was the description of the algorithm given in [Kur94].

That in [Kur94] this description was given in terms of the underlying Boolean algebra defined by all possible variable assignments and the ω-automata that define the design and property, may be the reason that even this level of generality was allowed to be published: from that description, the algorithm was not evident to the reviewers. While an elucidation certainly was called for, it could have prevented publication at any level of detail of this important algorithm.

In its initial implementation in COSPAN, the follow routine simply used random simulation to attempt to lift the error track to the full design. This was found to result in an over-all faster localization reduction than alternatives based on symbolic simulation and BDD-based constraint-solving (which often lacked the required capacity). Later, with the advent of more efficient SAT, a SAT-based constraint-solver would have provided a much more efficient and effective follow.

Additionally, in each iteration, the abstraction was further simplified through *variable resizing*, a very simple form of predicate abstraction that reduced variable ranges; and through constant propagation (especially effective when some variables had been resized to constants).

The key to the efficacy of the algorithm, however, was the heuristic used to identify the blocking variables of each iteration, due to R. H. Hardin. Based on Wagner's "Maximal Flow Algorithm" [Wag75], it selected a set of blocking variables that approximately minimized the sum of the logs of the number of possible values of the respective variables in the resulting free fence of the abstraction that resulted from a particular choice of blocking variables. This was solved as a minimal flow problem by dividing each variable into an input ivariable and an output ovariable with a single channel between them having capacity equal to the log of the number of free values for the variable. Each ovariable was joined to the foreign ivariables it depends on with an infinite capacity channel. The active variables are an infinite source of flow to the variables they depend on (so the old free fence is fed by an infinite source), and each state variable ovariable has an infinite-capacity channel to an infinite sink. Thus the flow will be limited by the capacity of certain ivariable-to-ovariable channels internal to original variables. These limiting variables form the new free fence.

The handling of automata acceptance conditions added another layer of complexity that is outside the scope of this paper, but their treatment merely entailed an adjustment to the above algorithm.

3 Counterexample-Guided Abstraction Refinement

The counterexample-guided abstraction refinement framework (CEGAR) was developed at Carnegie Mellon University in the summer of 1999 in the context

of Yuan Lu's PhD thesis [Lu00] and presented in mid July 2000 at CAV in Chicago [CGJ+00, CGJ+03], cf. Figure 1. In the same conference, recent Turing award winner Amir Pnueli emphasized the importance of abstraction for model checking in his keynote address *"Abstraction, Composition, Symmetry, and a Little Deduction: The Remedies to State Explosion"* [Pnu00]. Interestingly, Pnueli speculated about "useful additional measures of automation" for these hitherto manual techniques.

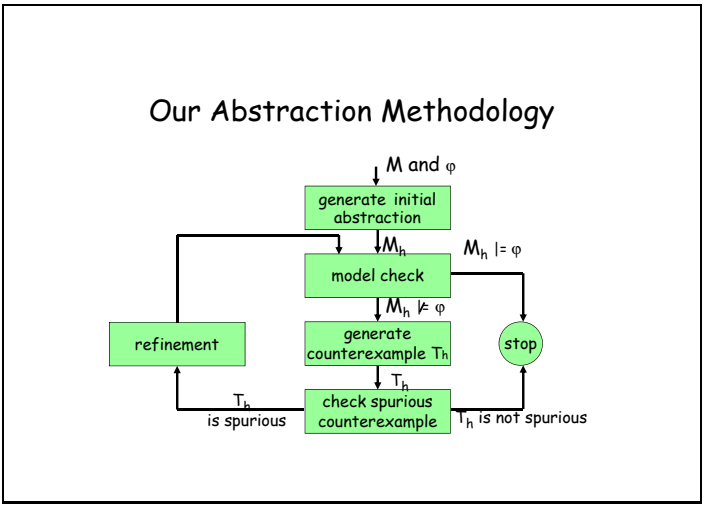


Fig. 1. Slide from CAV 2000

In their approach to the localization reduction, the Lucent management was more successful than in other matters: While the authors were aware of Kurshan's book [Kur94], the localization reduction was unclear to us. Nevertheless we tried to understand it from the book, and it served as an inspiration for CEGAR. The localization reduction was the most prominent paragraph of related work in [CGJ+00].

CEGAR leveraged several threads of research into a new concept for model checking:

– **Existential Abstraction.** CEGAR built on the existing framework of abstraction-based model checking [CGL94]. Given an abstraction function h and an ACTL* property φ, existential abstraction ensures a preservation

$$h(M) \models \varphi \qquad \Longrightarrow \qquad M \models \varphi,$$

i.e., a positive model checking result for property φ on the abstract (small) model $h(M)$ implies truth of the property in the original (larger) system

M. Similarly as in abstract interpretation, the specific abstractions h were manually chosen. The challenge in the manual construction of h is to find the right level of abstraction which reduces the size of the model enough to make $h(M) \models \varphi$ algorithmically tractable, but leaves sufficient precision in the model to avoid false negatives, i.e., situations where $h(M) \not\models \varphi$ and $M \models \varphi$. Note that the possibility of false negatives renders abstract model checking incomplete.

The abstract mapping used in the localization reduction is an instance of existential abstraction, where individual system variables are either unchanged or entirely abstracted away.

- **Predicate Abstraction.** CEGAR made use of predicate abstraction which was introduced by Graf and Saïdi in the context of the PVS theorem prover [GS97]. In predicate abstraction, an abstraction h maps a state s to a formula σ such that $s \models \sigma$. In the seminal paper [GS97], σ is given by $h(s) = \bigwedge_{s \models \psi, \psi \in P} \psi \wedge \bigwedge_{s \models \neg\psi, \psi \in P} \neg\psi$ where P is a finite set of atomic predicates over the program variables. While predicate abstraction nicely fits into abstract model checking [CGL94], it provides two important advantages: First, an abstract state σ can represent an infinite set of concrete states $h^{-1}(\sigma)$. Second, algorithmic reasoning about abstractions can be relegated to decision procedures. Both advantages became important for the development of software model checking.
- **Refinement.** CEGAR extended the idea of counterexample-guided refinement from localization reduction [Kur94] for COSPAN to this more general framework. Counterexample-guided refinement gives an algorithmic alternative to the manual construction of abstraction functions: The model checker starts with a coarse abstraction h. If a spurious counterexample occurs on $h(M)$, the abstraction function is refined as to eliminate the spurious counterexample, and model checking resumes with the refined abstract model, Thus, abstract model checking in combination with abstraction refinement yields an efficient and complete model checking procedure.
- **Symbolic Simulation of Abstract Counterexamples.** CEGAR used symbolic simulation of the abstract counterexample to decide if a counterexample is spurious, and to determine a refined abstraction function. The first implementation of CEGAR was based on the symbolic model checker NuSMV and thus geared to finite state models. Our symbolic simulation algorithm essentially performed a BDD-based forward simulation which either succeeded in a real counterexample, or got stuck in a "failure state" from which a refinement could be determined.

A significant amount of work was devoted to the simplification of the method and also to find generic terminology. The name of the method itself was a topic of repeated discussions. Thus, the paper went through *many* iterations until it reached its final form.

Methodologically, CEGAR has certainly changed our view of counterexamples [CV03]. Before CEGAR, counterexamples were considered important debugging information for the verification engineer, but not a central part of the verification tool chain, and unappealing from a theoretical perspective. With the arrival of CEGAR and bounded model checking [BCCZ99], counterexamples were treated as witnesses of existential temporal specifications, and thus they became a data structure of interest in their own right. Counterexamples are closely related to the path-sensitivity of a model checking analysis; a reconstruction of CEGAR in an abstract interpretation framework is mathematically feasible, but less natural [GQ01].

In fact, the method of [CGJ$^+$00] suffered from a flaw of beauty symptomatic of the situation before 2000. CEGAR was presented for the fragment of ACTL* which admits counterexamples of the form produced by SMV, i.e., finite paths or finite paths leading to a finite loop. It is of course easy to see that many ACTL* specifications such as $\mathbf{AFAX}p$ require more complex counterexamples, but for a long time this was a blind spot in the model checking literature. The first two decades of model checking research neither produced a precise definition of counterexamples, nor a systematic study of counterexamples for ACTL*.

Both questions were addressed in a follow-up paper at LICS 2002 [CJLV02]. The paper thoroughly investigated the notion of counterexamples – note that the whole system always is a counterexample, yet typically a useless one – and gave a semi-formal definition of counterexamples which is based on simulation but also requires "simplicity" (algorithmic or cognitive simplicity) of counterexamples.

Importantly, the 2002 paper demonstrated the completeness of CEGAR for full branching time logic: First, it was shown that each specification in a large class of logics including ACTL* has a tree-like counterexample. Tree-like counterexamples are structures obtained by recursively gluing together finite paths and finite loops. Then, the paper demonstrated how the CEGAR method of [CGJ$^+$00] and the BDD-based implementation can be extended to full ACTL*.

In two other CEGAR successor papers of 2002 at FMCAD and CAV [CCK$^+$02, CGKS02], CEGAR was studied in a bounded model checking framework. In both papers, a SAT solver is used for the symbolic simulation of a possibly spurious abstract counterexample. If the SAT instance is unsatisfiable, the abstract counterexample is spurious and needs to be refined. The papers use different techniques for determining the refinement: In [CGKS02], ILP (integer linear programming) and machine learning are used to identify important variables which are absent in the abstraction. The second paper [CCK$^+$02] monitors the SAT checking phase in order to analyze the impact of individual variables. Thus, the paper is a direct predecessor of the work by Amla and McMillan [MA03] discussed in Section 4.

CEGAR has found its most important applications in software model checking. Predicate abstraction facilitates the separation of concerns between decision procedures to extract a finite state model from an infinite-state software model, and model checkers to analyze this model. We briefly sketch the main lines of development. SLAM [BR02], BLAST [HJMS02], and MAGIC [CCG$^+$03,

Cha05] combined CEGAR with predicate abstraction and decision procedures. SATABS [CKSY05] made use of a bit-precise program presentation and a SAT solver as decision procedure. SLAM was the first tool to apply a CEGAR loop for software, and the first software model checker with significant industrial impact.

The most important development in refinement was the interpolation method by Ken McMillan [McM03] which was first applied to software model checking in collaboration with BLAST [HJMM04] and promises to be a foundation for a systematic approach to refinement in infinite-state systems.

More recently, software model checking has also turned to liveness properties, a topic of continued interest to Amir Pnueli. Amir collaborated with Rybalchenko and Podelski on a method [PR04, PPR05] which was implemented in the Terminator tool [CPR05] using a counterexample-guided abstraction refinement loop.

A good survey on software model checking can be found in [JM09].

4 Automatic Abstraction without Counterexamples

In a stunning advance of the essential *successive approximations* idea, McMillan and Amla presented in effect "counter-example guided refinement without counter-examples" [MA03]. In their algorithm, the successive abstractions are determined not by the counterexamples but by the unSAT proof used to refute falsification in the original model at a given sequential depth. Since this gives a proof of the absence of a counter-example at the given depth, the authors have termed this "proof-based abstraction". Rather than ".. without counter-examples", their method could also be said to be "counter-example guided refinement for *every* counter-example of a given length".

Their method employs a similar abstraction-refinement loop as the previous methods, but uses unSAT clauses in place of the counter-example. First, they perform SAT-based bounded model checking to some depth k in the pruned design. If this fails to generate a counter-example, then they extract the unSAT clauses from the SAT solver. These constitute a proof that no counter-example exists at depth k (or less).

The proof extraction process begins with conflict clause generation derived from the sequence of resolution steps that follow the implication graph. For each generated conflict clause, they record the sequence of clauses that were resolved to produce it. A proof of unsatisfiability follows through a depth-first search that begins at the empty clause and recursively deduces each successive clause in terms of the sequence of clauses that produced it.

Viewing the bounded model checking problem as a set of constraints (initial constraints, transition constraints and final or liveness constraints – for a safety or eventuality property, respectively), they consider the set of these constraints (represented in binary conjunctive normal form) that are used in the proof of unsatisfiability. Any constraint, all of whose clauses are not used in the proof of unsatisfiability, can be removed without affecting the proof. After such removal of constraints, the resulting model is a conservative abstraction of the original model that is also guaranteed to admit of no counter-example of length k (or less).

Next, they perform ordinary (unbounded) model checking on this conservative abstraction. If it passes, then the property is verified for the original model. If it is falsified at some depth k', then they know that $k' > k$, as a counter-example of length k or less has already been ruled out in both the original and abstract model.

Their algorithm now iterates by performing a new bounded model checking run to length k'. This is guaranteed to terminate (or run out of memory), as k is strictly increasing, but cannot exceed the depth of the generated abstract model. When it is equal, the check of the abstract model will verify.

Note that falsification occurs only in the bounded model checking step on the full model, so there can be no issue of a bogus counter-example coming from a check of an abstract model. Conversely, verification occurs only during a check of the abstract model.

While they use SAT-based bounded model checking for the bounded check, they have found that BDD-based model checking is often the most effective type of model checker on the abstract model, as the abstractions are often small.

Note also that unlike the previous methods for which the abstraction refinements grow monotonically, in their procedure, there may be no relationship between successive abstractions. In their procedure, though, the *length* of the generated counter-examples is strictly increasing.

They report that in practice, they have found that when the procedure terminates, k is roughly half the depth of the abstract model.

Acknowledgements. We are thankful to Yuan Lu for comments on Section 3.

References

[AIKY93] Alur, R., Itai, A., Kurshan, R.P., Yannakakis, M.: Timing Verification by Successive Approximation. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 137–150. Springer, Heidelberg (1993); Also Inf. Comput. 118(1), 142–157 (1995)

[AM04] Amla, N., McMillan, K.L.: A Hybrid of Counterexample-Based and Proof-Based Abstraction. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 260–274. Springer, Heidelberg (2004)

[AM07] Amla, N., McMillan, K.L.: Combining Abstraction Refinement and SAT-Based Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 405–419. Springer, Heidelberg (2007)

[BCCZ99] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[BR02] Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)

[CCG+03] Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: ICSE, pp. 385–395. IEEE Computer Society, Los Alamitos (2003)

[CCK+02] Chauhan, P., Clarke, E.M., Kukula, J.H., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces

using sat based conflict analysis. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 33–51. Springer, Heidelberg (2002)

[CGJ+00] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)

[CGJ+03] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)

[CGKS02] Clarke, E.M., Gupta, A., Kukula, J.H., Strichman, O.: Sat based abstraction-refinement using ilp and machine learning techniques. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 265–279. Springer, Heidelberg (2002)

[CGL94] Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)

[Cha05] Chaki, S.: A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs. PhD Thesis, Carnegie Mellon University (2005)

[CJLV02] Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: LICS, pp. 19–29. IEEE Computer Society Press, Los Alamitos (2002)

[CKSY05] Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Satabs: Sat-based predicate abstraction for ansi-c. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)

[CPR05] Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)

[CV03] Clarke, E.M., Veith, H.: Counterexamples revisited: Principles, algorithms, applications. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 208–224. Springer, Heidelberg (2004)

[DD01] Das, S., Dill, D.L.: Successive Approximation of Abstract Transition Relations. In: LICS 2001, pp. 51–58. IEEE Computer Society Press, Los Alamitos (2001)

[For97] Lucent's Bell Introduces FormalCheck. Electronic News (April 1997)

[GQ01] Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, p. 356. Springer, Heidelberg (2001)

[GS97] Graf, S., Saıdi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

[HHK96] Hardin, R.H., Har'El, Z., Kurshan, R.P.: COSPAN. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 423–427. Springer, Heidelberg (1996)

[HJMM04] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. ACM, New York (2004)

[HJMS02] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)

[JM05] Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)

[JM09] Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4) (2009)

[Kur94] Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)

[Kur00] Kurshan, R.P.: Program Verification. Notices of the AMS 47(5), 534–545 (2000)

[Lu00] Lu, Y.: Automated Abstraction in Model Checking. PhD Thesis, Carnegie Mellon University (2000)

[MA03] McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)

[McM03] McMillan, K.L.: Interpolation and sat-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)

[Pnu00] Pnueli, A.: Keynote address: Abstraction, composition, symmetry, and a little deduction: The remedies to state explosion. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, p. 1. Springer, Heidelberg (2000)

[PPR05] Pnueli, A., Podelski, A., Rybalchenko, A.: Separating fairness and well-foundedness for the analysis of fair discrete systems. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 124–139. Springer, Heidelberg (2005)

[PR04] Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society, Los Alamitos (2004)

[Wag75] Wagner, H.M.: Principles of Operations Research, pp. 953–958. Prentice Hall, Englewood Cliffs (1975); see Appendix I: Advanced Topics in Network Algorithms

A Scalable Segmented Decision Tree Abstract Domain

Patrick Cousot[2,3], Radhia Cousot[1,3], and Laurent Mauborgne[3,4]

[1] Centre National de la Recherche Scientifique
[2] Courant Institute of Mathematical Sciences, New York University
[3] École Normale Supérieure, Paris
[4] Instituto Madrileño de Estudios Avanzados, Madrid

Dedicated to the memory of Amir Pnueli

1 Introduction

The key to precision and scalability in all formal methods for static program analysis and verification is the handling of disjunctions arising in relational analyses, the flow-sensitive traversal of conditionals and loops, the context-sensitive inter-procedural calls, the interleaving of concurrent threads, etc. Explicit case enumeration immediately yields to combinatorial explosion. The art of scalable static analysis is therefore to abstract disjunctions to minimize cost while preserving weak forms of disjunctions for expressivity.

Building upon packed binary decision trees to handle disjunction in tests, loops and procedure/function calls and array segmentation to handle disjunctions in array content analysis, we introduce *segmented decision trees* to allow for more expressivity while mastering costs via widenings.

2 Semantic Disjunctions in Abstract Interpretation

The main problem in applying abstract interpretation [2,5,6] to static analysis is to abstract a non-computable fixpoint collecting semantics $\mathbf{lfp}^{\sqsubseteq}_{\bot} F$ for a concrete transformer $F \in \mathcal{C} \mapsto \mathcal{C}$, partial order $\sqsubseteq$, and infimum $\bot$ into an abstract semantics $\mathbf{lfp}^{\sqsubseteq^\sharp}_{\bot^\sharp} F^\sharp$ for an abstract transformer $F^\sharp \in \mathcal{A} \mapsto \mathcal{A}$, abstract order $\sqsubseteq^\sharp$, and abstract infimum $\bot^\sharp$ where the existence of fixpoints is guaranteed by Tarski's theorem [21] on complete lattices or its extension to complete partial orders (cpos). The collecting semantics is the specification of the undecidable properties we want to collect about programs. The abstract semantics is an effective approximation of the collecting semantics. For soundness, $\mathbf{lfp}^{\sqsubseteq}_{\bot} F \sqsubseteq \gamma(\mathbf{lfp}^{\sqsubseteq^\sharp}_{\bot^\sharp} F^\sharp)$ where $\gamma \in \mathcal{A} \mapsto \mathcal{C}$ is the concretization function. Particular cases involve a Galois connection $\langle \mathcal{C}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq^\sharp \rangle$ such that $\forall x \in \mathcal{C} : \forall y \in \mathcal{A} : \alpha(x) \sqsubseteq^\sharp y \iff x \sqsubseteq \gamma(y)$ and the case of completeness requiring $\alpha(\mathbf{lfp}^{\sqsubseteq}_{\bot} F) = \mathbf{lfp}^{\sqsubseteq^\sharp}_{\bot^\sharp} F^\sharp$.

In general the concrete domain $\langle \mathcal{C}, \sqsubseteq, \bot, \sqcup \rangle$ is a complete lattice or cpo and the concrete transformer F is in disjunctive form $F \triangleq \bigsqcup_{i \in \Delta} F_i$ and often

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 72–95, 2010.

completely distributive ($\forall i \in \Delta : F_i(\bigsqcup_{\mathtt{j}\in\Delta'} X_j) = \bigsqcup_{\mathtt{j}\in\Delta'} F_i(X_j)$) or continuous (completely distributive on increasing chains).

In that most usual case, the iterative fixpoint computation $X^0 = \bot$, ..., $X^{n+1} = F(X^n)$, ..., $X^\omega = \bigsqcup_{n\geqslant 0} X^n = \mathbf{lfp}^{\sqsubseteq}_{\bot} F$ is $X^1 = F(\bot) = \bigsqcup_{i\in\Delta} F_i(\bot)$, $X^2 = F(X^1) = \bigsqcup_{i\in\Delta} F_i(\bigsqcup_{\mathtt{j}\in\Delta} F_j(\bot)) = \bigsqcup_{i,\mathtt{j}\in\Delta^2} F_i \circ F_j(\bot)$, ..., $X^{n+1} = F(X^n) = \bigsqcup_{i\in\Delta} F_i(\bigsqcup_{i_1,\ldots,i_n\in\Delta^n} F_{i_1} \circ \ldots \circ F_{i_n}(\bot)) = \bigsqcup_{i_1,\ldots,i_n,i_{n+1}\in\Delta^{n+1}} F_{i_1} \circ \ldots \circ F_{i_n} \circ F_{i_{n+1}}(\bot)$, ..., so that passing to the limit $\mathbf{lfp}^{\sqsubseteq}_{\bot} F = X^\omega = \bigsqcup_{n\geqslant 0} X^n = \bigsqcup_{n\geqslant 0} \bigsqcup_{i_1,\ldots,i_n\in\Delta^n} F_{i_1} \circ \ldots \circ F_{i_n}(\bot)$. This shows that the disjunctive explosion problem appears in the concrete iterative fixpoint definition.

If the abstraction is a Galois connection, the abstraction preserves existing joins. It follows that $\alpha(\mathbf{lfp}^{\sqsubseteq}_{\bot} F) = \alpha(\bigsqcup_{n\geqslant 0} \bigsqcup_{i_1,\ldots,i_n\in\Delta^n} F_{i_1} \circ \ldots \circ F_{i_n}(\bot)) = \bigsqcup^\sharp_{n\geqslant 0} \bigsqcup^\sharp_{i_1,\ldots,i_n\in\Delta^n} \alpha(F_{i_1} \circ \ldots \circ F_{i_n}(\bot))$ which is most often over approximated as $\bigsqcup^\sharp_{n\geqslant 0} \bigsqcup^\sharp_{i_1,\ldots,i_n\in\Delta^n} \alpha \circ F_{i_1} \circ \gamma \circ \alpha \circ F_{i_2} \circ \gamma \circ \ldots \circ \alpha \circ F_{i_n} \circ \gamma(\alpha(\bot))) \sqsubseteq^\sharp \bigsqcup^\sharp_{n\geqslant 0} \bigsqcup^\sharp_{i_1,\ldots,i_n\in\Delta^n} F^\sharp_{i_1} \circ F^\sharp_{i_2} \circ \ldots \circ F^\sharp_{i_n}(\bot^\sharp)) = \mathbf{lfp}^{\sqsubseteq^\sharp}_{\bot^\sharp} F^\sharp$ where $\forall i \in \Delta : \alpha \circ F_i \circ \gamma \sqsubseteq^\sharp F^\sharp_i$, $F^\sharp \triangleq \bigsqcup^\sharp_{i\in\Delta} F^\sharp_i$ and $\bot^\sharp \triangleq \alpha(\bot)$. This shows that the disjunctive explosion problem does also exist in the abstract.

The situation is even worst in absence of best abstraction, that is of a Galois connection, since the concrete transformers F_i may have many, possibly non-comparable, abstractions $F^\sharp_i$. In absence of minimal abstractions (as shown by the abstraction of a disk by polyhedra [12]), infinitely many potential abstractions may exist. Choosing which abstraction should better be used during the analysis is another source of potential combinatorial explosion.

3 Handling Disjunctions in Abstract Interpretation

Contrary to purely enumerative or symbolic encodings of program properties, abstract interpretation offers solutions to the combinatorial explosion of disjunctions so as to minimize computational costs. The key idea is to abstract away irrelevant properties of the collecting semantics.

The abstract domain $\langle \mathcal{A}, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp \rangle$ can be chosen as finite (e.g. predicate abstraction [3,13]) or better of finite height (e.g. constant propagation [15]) to bound n in $\mathbf{lfp}^{\sqsubseteq^\sharp}_{\bot^\sharp} F^\sharp = \bigsqcup^\sharp_{n\geqslant 0} {F^\sharp}^n(\bot^\sharp)$).

However this solution has been shown to have intrinsic limitations [8] that can be eliminated thanks to infinite abstract domains not satisfying the ascending chain condition together with widenings ∇ and narrowings Δ [2,4,5] (including the common practice of including the widening in the transformers $F^\sharp_i$, $i \in \Delta$ mentioned in [9], by choosing $\boldsymbol{\lambda} X \bullet X \nabla (\alpha \circ F_i \circ \gamma(X)) \sqsubseteq^\sharp F^\sharp_i$, $i \in \Delta$, which is not such a good idea since it precludes the later use of a narrowing).

Moreover, in absence of a best abstraction, that is of a Galois connection, a choice is usually made among the possible non-comparable abstractions $F^\sharp_i$ of the concrete transformers F_i to minimize costs [7].

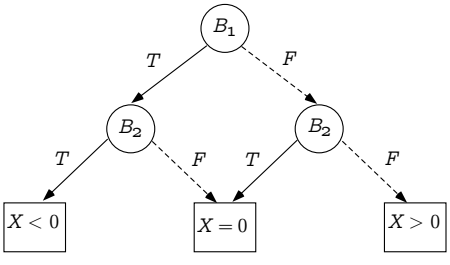
The objective of minimizing computational costs is antagonist to that of precision necessary for making proofs, so that a lot of work in abstract interpretation is on the design of abstract domains offering cost effective yet precise enough ways of abstracting infinite disjunctions.

In this line of work, we consider the examples of binary decision trees in Sect. 4 and segmented arrays in Sect. 6 which common generalization is segmented decision trees as introduced in Sect. 7, a generalization which is more precise while remaining scalable.

4 Binary Decision Trees

Inspired by the use of Binary Decision Diagrams in abstract interpretation [17,18], binary decision trees can express disjunctive properties depending on the values of binary variables with opportunistic sharing at the leaves [16,19]. They are an instance of the reduced cardinal power of abstract domains [6, Sect. 10.2] mapping the values of boolean variables (represented in decision nodes) to an abstraction of the other variables (represented in the leaf nodes) for the values of the boolean variables along the path leading to the leave.

Example 1. The following binary decision tree



written $[\![B_1 : [\![B_2 : (\!| X < 0 |\!), (\!| X = 0 |\!)]\!], [\![B_2 : (\!| X = 0 |\!), (\!| X > 0 |\!)]\!]]\!]$ encodes

$(B_1 \wedge B_2 \wedge X < 0) \vee (((B_1 \wedge \neg B_2) \vee (\neg B_1 \wedge B_2)) \wedge X = 0) \vee (\neg B_1 \wedge \neg B_2 \wedge X > 0)$. □

The parenthesized representation of trees uses $(\!| \ldots |\!)$ for leaves and $[\![x : \ldots]\!]$ for left to right decision nodes on variable x.

Example 2. In the following example, ASTRÉE [10] discovers a relation between the boolean variable `B` and the unsigned integer variable `X`.

```
% cat -n decisiontree.c
     1  typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
     2  BOOLEAN B;
     3  void main () {
     4    unsigned int X, Y;
     5    while (TRUE) {
     6      __ASTREE_analysis_log(());
```

```
     7      B = (X == 0);
     8      if (!B) {
     9        Y = 1 / X;
    10      };
    11    };
    12  }
astree --exec-fn main --print-packs decisiontree.c
...
  <boolean relations:decisiontree.c@9@5=
  if B then <integers (intv+cong+bitfield+set): X in {0} >
  else <integers (intv+cong+bitfield+set): X in [1, 4294967295] >
  >
...
%
```

At line 9, the relation between `B` and `X` is expressed by a binary decision tree with `B` as only decision node and `X` in the leave nodes. This binary decision tree states that if `B` is `TRUE` then `X` is zero else `X` is positive. Since `B` is checked to be false there is no division by zero at line `9`. □

5 Variable Packing

Relational abstractions such as octagons [20], polyhedra [12] or binary decision trees in Sect. 4 can express relations between values of variables hence complex disjunctions. However their cost may grow polynomially or even exponentially in size and time. Even for abstract domains such as the octagon domain which has a very light cubic cost compared to many relational domains (such as exponential costs in the number of variables for polyhedra [12]), this would still be too high for very large embedded programs with tens of thousands variables as found in aeronautics.

Variable packing is a way to control memory and computation costs by limiting the number of variables that can be related at a given program point. The decision of which variables can be considered in such abstract relations at each program control point can be taken during the analysis e.g. by widenings (deciding which variables to eliminate by projection from the relation). Variable packing is an even more radical solution which consists in controlling costs a priori, by statically restricting the variables which can be considered in such abstract relations at each program control point. Such a more radical solution is necessary when the cost of even one iteration with all variables in the relation is prohibitive.

The idea is to make small packs of variables that are related while no attempt is made to relate variables appearing in different packs. A simple and cheap preanalysis groups variables that are interdependent (used together in expressions, loop indices, etc.) in a way that can be represented by the relation. Relations are established among variables associated to each pack, but no relation is kept between variables of distinct packs. The cost thus becomes linear, as it is linear in the number of packs (which is linear in the code size, and so, in the number of variables) and, e.g. for octagons, cubic in the size of packs (which depends on the size of the considered packs, and is a constant).

Example 3. In *Ex. 2*, ASTRÉE has packed the boolean variable `B` and the unsigned integer variable `X` together (but the pack does not contain variable `Y`). The abstract domain used in the leaves is the reduced product [6] of several abstract domains which can be specified as an option of the analysis and by default are the following

```
/* Domains: ... Packed (Boolean relations (based on Absolute value
equality relations, and Symbolic constant propagation (max_depth=
20), and Linearization, and Integer intervals, and Integer
congruences, and Integer finite sets, and Integer bitfields, and
Float intervals)), and ... */
...
Boolean relations :
List of packs
  decisiontree.c@9@5 { B X }
...
  <boolean relations:decisiontree.c@9@5=
  if B then <integers (intv+cong+bitfield+set): X in {0} >
  else <integers (intv+cong+bitfield+set): X in [1, 4294967295] >
  >
...
```

The output in *Ex. 2* only prints informative results, specifically intervals [4,5], simple congruences [14], bit fields (recording an invariant bit pattern in the binary representation of the variable, if any) and sets of small values (recording the set of possible values of the variable in a small range of values near zero) for that example. □

Candidates for packing in a binary decision tree are the boolean variables to which a boolean expression is assigned or which are involved in a test as well as the variables which depend directly or indirectly on such a boolean variable, with a maximum number of boolean variables which can be set by an option `--max-bool-var` (3 by default). The option `--print-packs` allows printing packs of variables determined by the pre-analysis for the binary decision trees. In case of insufficient precision, ASTRÉE can be asked to create binary decision tree packs containing given variables by the `__ASTREE_boolean_pack` directive inserted in the program code. Of course putting all boolean program variables in a single pack would certainly achieve great precision but is also the best way to get a combinatorial explosion.

6 Array Segmentation

Array segmentation was introduced in [11] to abstract array content (as opposed to array bounds [4]). The array is divided into consecutive segments. The content of each segment is abstracted uniformly but different segments can have different abstractions. Starting from a single initial segment with uninitialized content, segments are split and filled by assignments to arrays elements and joined when merging control flows. A widening may be necessary to merge segments so as to avoid the degenerescence to the case of one element per segment.

The bounds of the segments are specified by a set of side-effect free expressions which all have the same concrete value (maybe unknown in the abstract). Segments can be empty thus allowing an array segmentation to encode a disjunction of cases in a way that avoids the explosion of cases.

Example 4. The segmentation {0}⊤{1}0{i}?⊤{n} of an array A states that $0 < 1 \leqslant \mathtt{i} < \mathtt{n}$, that the values of the array elements A[0], A[i], A[i+1], ..., A[n-1] are all unknown, while the values of A[1], A[2], ..., A[i-1], if any, are all initialized to zero. It is possible that $\mathtt{i} = 1$ in which case the segment A[1], ..., A[i-1] is empty, and $\mathtt{i} < \mathtt{n}$ so that the segment A[i], ..., A[n-1] cannot be empty (it contains at least one element).

The segmentation {0}⊤{1}0{i}⊤{n} is similar but for the fact that $\mathtt{i} > 1$ so that the segment A[1], ..., A[i-1] cannot be empty. So {0}⊤{1}0{i}?⊤{n} is a compact encoding of the disjunctive information ({0}⊤{1,i}⊤{n}) ∨ ({0}⊤{1}0{i}⊤{n}) distinguishing the first case $\mathtt{i} = 1$ from the second $\mathtt{i} > 1$.

Similarly {0}⊤{1}0{i}?⊤{n}? is the disjunction ({0}⊤{1,i,n}) ∨ ({0}⊤{1,i}⊤{n}) ∨ ({0}⊤{1}0{i,n}) ∨ ({0}⊤{1}0{i}⊤{n}). Of course, expressivity is limited since it is not possible to express that either $\mathtt{i} = 1$ or $\mathtt{i} = \mathtt{n}$ but not both (although this might be separately expressible by the abstraction of the simple variables i and n). □

Note that there are no holes in the segmentation since any such hole is just a segment which content is unknown.

An enriched semantics of arrays is used viewing an array value as the pair of an index and the value of the corresponding array element. In this way the uniform abstraction used in a segment can relate array values within each segment to their index included between the segment bounds.

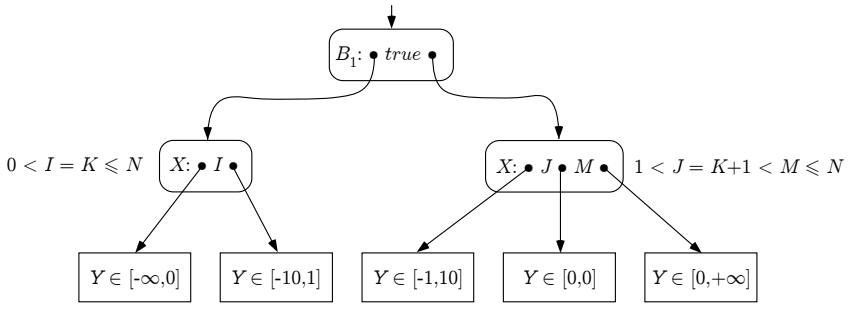
The array segmentation is implemented as a functor which means that the abstract domains representing sets of expressions and array index-elements abstractions should be passed as parameters to the functor to create a particular instance. The advantage of this approach is that the abstract domain parameters can be changed without having to rewrite the static analyzer.

Moreover the static analyzer can optionally perform a reduced product [6] between an instance of the array segmentation functor and the abstract domains that are used for variables appearing in the expressions of the segment bounds. It follows that the array segmentation takes into account both the operations on arrays (array element access and assignment) and operations on variables related to array indexes.

7 Segmented Decision Trees

Segmented decision trees are decision trees where the choices are made on the values of variables according to ranges specified by a symbolic segmentation.

Example 5. The segmented decision tree (where *false* < *true* for booleans)



can be written in the parenthesized form

$$\begin{array}{l}[\![B_1 : \quad [\![X\,\{0 < I = K \leqslant N\} : \quad (\!|Y \in [-\infty, 0]\,|\!) \quad I \quad (\!|Y \in [-10, 1]\,|\!) \quad]\!] \quad true \\ \qquad [\![X\,\{1 < J = K + 1 < M \leqslant N\} : \\ \qquad\qquad (\!|Y \in [-1, 10]\,|\!) \quad J \quad (\!|Y \in [0, 0]\,|\!) \quad M \quad (\!|Y \in [0, +\infty]\,|\!) \quad]\!] \quad]\!]\end{array}$$

This segmented decision tree encodes the fact that if B_1 is false (i.e. $B_1 < true$) then if $X < I$ then Y is non-positive while if $X \geqslant I$ then $-10 \leqslant Y \leqslant 1$. Similarly, if B_1 is true (i.e. $B_1 \geqslant true$) then either $X < J$ and $-1 \leqslant Y \leqslant 10$, or $J \leqslant X < M$ and Y is null, or $X > M$ and Y is non-negative. So the leaf nodes specify abstract properties of Y while the decision nodes on B_1 and X specifying conditions for these properties to hold. Attached to each decision node, is a side relation on expressions that holds in the concrete under the condition that this node is reached. For example $(B_1 \wedge 1 < J = K + 1 < M \leqslant N) \vee (\neg B_1 \wedge 0 < I = K \leqslant N)$. These expressions are usually in a restricted normal form. In this example the normal form of expressions is an integer constant, a variable, or a variable plus an integer constant and the side relations are those expressible with the octagon abstract domain [20]. The segment bounds are any representative of the equivalent class of expressions which have equal concrete values (so that we could have chosen K for I and $K + 1$ for J). The abstract domain of side relations is assumed to be expressive enough to maintain such equality information between expressions in normal form (i.e. $I = K$ and $J = K + 1$). □

As for boolean decision trees, an ordering is imposed on all decision variables. That allows binary operations on decisions trees to operate on the same variable [1]. But unlike binary decision trees, the number of choices for a given variable is not bounded *a priori* and the choices may be on different criteria (the bounds in the symbolic segmentations) at each node, even if they have the same decision variables.

As for simple array segmentation, the ordering of the bounds of each segment describes an order on expressions. That means that segments could describe two kinds of informations: a serie of tests deciding what choice to make and a pre-order on some expressions. Unlike for array segmentation, that information

[1] In addition it may allow to eliminate the nodes with only one child, an optimization we will not apply in this paper.

would now be relative to the choices above a node in a tree. So we decided to separate the two notions, such that at each node, we have a decision variable, a pre-order and a segmentation that respects the pre-order (instead of prescribing it) and leads to subtrees. A direct consequence is that segments will be much more simple, as that removes the necessity of a `?` tag for emptyness or of the lowest and highest bounds of the segmentation. Also, it allows the use of single expressions as bounds instead of sets of expressions.

Separating the pre-order information from the decision process allows us to use much more precise domains for the pre-order and leads to more precise unifications of segmentations. But storing a pre-order on all possible expressions that can be used at a given node in the tree might lead to very expensive representations. So we chose instead to rely on reduction with other abstract domains for pre-order information valid at every node in the tree and store at each node the ordering information that we can add to what holds for its father. In that way, if the abstraction of an instruction implies an ordering that is not relevant to the tree, it will not increase the complexity of the tree, but further operations might still use that information from reduction with the other abstract domains. In the drawings and examples, we will represent that information as a set of inequations between expressions, but in practice, depending on the canonical expressions we use, we can implement it in a more efficient way, using for example small octagons if the canonical expressions are of the form $\pm X + c$ with X a variable and c a constant. Also, in order to simplify the presentation, in the schemata, we put all global pre-order information at the root[2].

Definition 1. A segmented decision tree $t \in \mathbb{T}((\mathbb{D}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ over decision variables in the totally ordered set $(\mathbb{D}, <_{\mathbb{D}})$, canonical expressions in $\mathbb{E}$, ordering abstract domain D_c (with concretization γ_c) and leaf abstract domain D_ℓ (with concretization γ_ℓ) is either $(\!|\, p \,|\!)$ with p an element of D_ℓ or $[\![\, \mathtt{x}\,\{C\} : t_0 b_1 t_1 \ldots b_n t_n \,]\!]$ such that $\mathtt{x}$ is the smallest variable in $\mathbb{D}$, each b_i $(1 \leqslant i \leqslant n)$ is an element of $\mathbb{E}$, C is an element of D_c and each $t_i \in \mathbb{T}((\mathbb{D}\backslash\{x\}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ $(0 \leqslant i \leqslant n)$. □

To define the concretization of a segmented decision tree, we will write ρ for concrete environments assigning concrete values $\rho(\mathtt{x})$ to variables $\mathtt{x}$ and $[\![e]\!]\rho$ for the concrete value of the expression e in the concrete environment ρ. The concretization of a segmented decision tree reduced to a leave is

$$\gamma_t(\ (\!|\, p \,|\!)\ ) \triangleq \gamma_\ell(p)$$

and the concretisation of a segmented decision tree rooted at a decision node is

$$\begin{aligned}
&\gamma_t.(\ [\![\, \mathtt{x}\,\{C\} : t_0 b_1 t_1 \ldots b_n t_n \,]\!]\ ) \triangleq \\
&\quad \{\rho \in \gamma_c(C) \mid \forall i \in [1,n) : [\![b_i]\!]\rho \leqslant [\![b_{i+1}]\!]\rho \wedge \\
&\qquad\qquad (n = 0 \vee \rho(\mathtt{x}) < [\![b_1]\!]\rho) \Longrightarrow \rho \in \gamma_t(t_0) \wedge \\
&\qquad\qquad \forall i \in [1,n) : ([\![b_i]\!]\rho \leqslant \rho(\mathtt{x}) < [\![b_{i+1}]\!]\rho) \Longrightarrow \rho \in \gamma_t(t_i) \wedge \\
&\qquad\qquad (n > 0 \wedge \rho(\mathtt{x}) \geqslant [\![b_n]\!]\rho) \Longrightarrow \rho \in \gamma_t(t_n)\}
\end{aligned}$$

[2] For clarity, some redundancy is sometimes preserved in segmented decision trees while, for brevity, some repetitive information is omitted in fixpoint computations.

We introduce also the notation $\bot_{\mathbb{D}}$ for the decision tree in $\mathbb{T}((\mathbb{D}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ such that each node is of the form $[\![\mathtt{x}\{\top_C\} : t]\!]$ and the only leaf is $(\!| \bot_\ell |\!)$, where $\top_C$ is the top element of D_c and $\bot_\ell$ is the bottom element of D_ℓ. When $\mathbb{D}$ is clear from the context, we simply write $\bot$.

7.1 Segmented Decision Tree Abstract Functor

The segmented decision tree abstract functor $\mathbb{T}((\mathbb{D}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ is a parameterized abstract domain taking as a parameter a totally ordered set $(\mathbb{D}, <_{\mathbb{D}})$ of decision variables, a set $\mathbb{E}$ of canonical expressions, an ordering abstract domain D_c and a leaf abstract domain D_ℓ for the leaves.

The abstract domain D_ℓ for the leaves is usually the reduced product [6] of several abstract domains, as was the case for binary decision trees in Sect. 4. The list of abstract domains appearing in this reduced product at the leaves is assumed to be an option of the static analyzer constructor. Therefore this option specifies a particular instance of the segmented decision tree abstract functor used to build a particular instance of the static analyzer for that option. The advantage of this modular approach is that the static analyzer can be changed by changing the options, without any re-programming.

The maximal height of the segmented decision trees is a parameter of the static analysis which can therefore be changed before each run of the static analyzer. A variable packing pre-analysis is used to determine which variables $\mathbb{D}$ are chosen to appear in the decision and leave nodes. The number of variables in the decision nodes is bounded by this maximal height. Following [11], the choice of which expressions $b_1, \ldots, b_n \in \mathbb{E}$, $n \geqslant 0$ do appear in decision nodes is made during the static analysis.

7.2 Reduction of an Abstract Property by a Segmented Decision Tree

Given a segmented decision tree t and an abstract property $p \in D$ of the variables in abstract domain $\langle D, \sqsubseteq, \bot, \sqcup, \sqcap \rangle$ with concretization γ, $t \sqcap_D p$ is the abstraction of the conjunction $\gamma_t(t) \cap \gamma(p)$ in the abstract domain D. It is the intersection of p with the join of the abstract properties obtained along paths of t feasible for p.

Example 6. In *Ex. 5*, the hypotheses that B_1 is *true* and $X < M$ imply that $Y \in [-1, 10] \sqcup [0, 0] = [-1, 10]$. The implied condition collects information along the path and at the leaves and is therefore $B_1 \wedge (X < M) \wedge (1 < J = K + 1 < M \leqslant N) \wedge Y \in [-1, 10]$. □

The operation $t \sqcap_D p$ is used in the definition of the reduced product of the abstract domain $\langle D, \sqsubseteq \rangle$ by the abstract domain of segmented decision trees.

A path of t is feasible for p if it corresponds, at each level in the tree, to a segment of the node which, according to p and the conjunction of side conditions collected from the root to this node, is not empty. For each decision variable, the conjunction of the hypothesis p and the collected side conditions is used to

determine to which segments the value of the variable does belong. The information available on this variable is thus the join of the information available at the leaves for each segment plus the fact that the variable value is between the extreme bounds of these segments along this path. More formally ($\sqsubseteq$ is the abstract implication, $(\, \mathit{true} \ ?\ a : b\,) \triangleq a$, $(\, \mathit{false} \ ?\ a : b\,) \triangleq b$ is the conditional, $D(p)$ is the best approximation of $p \in D_c \cup D_\ell$ in D (in absence of best abstraction, an over-approximation must be used)).

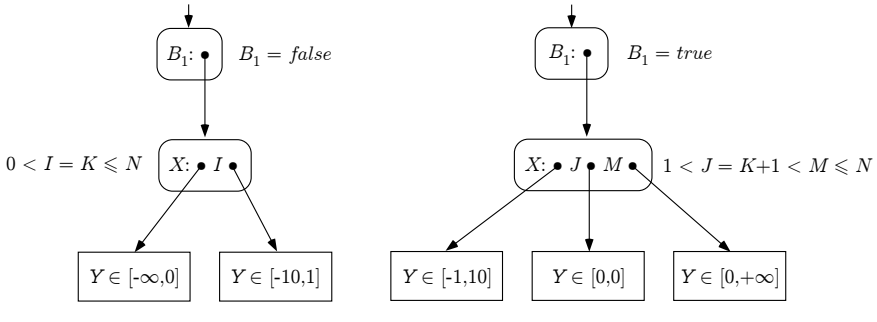
$$
\begin{aligned}
\bot_{\mathbb{D}} \sqcap_D p &\triangleq \bot \\
t \sqcap_D \bot &\triangleq \bot \\
(\!|\, p' \,|\!) \ \sqcap_D p &\triangleq D(p') \sqcap p \\
[\![\, \mathtt{x}\,\{C\} : \ t_0 b_1 t_1 \ldots b_n t_n \,]\!] \ \sqcap_D p &\triangleq \ t_0 \sqcap_D (p \sqcap D(C) \sqcap (\, n = 0 \ ?\ \top : D(\mathtt{x} < b_1))\,) \\
&\quad \sqcup \bigsqcup_{i=1}^{n-1} t_i \sqcap_D (p \sqcap D(C) \sqcap D(b_i \leqslant \mathtt{x} < b_{i+1})) \\
&\quad \sqcup (\, n > 0 \ ? t_n \sqcap_D (p \sqcap D(C) \sqcap D(\mathtt{x} \geqslant b_n)) : \bot)
\end{aligned}
$$

Observe that $p \sqsubseteq q$ implies $\gamma(p) \subseteq \gamma(q)$ whereas $p \not\sqsubseteq q$ does not implies that $\gamma(p) \not\subseteq \gamma(q)$ whereas the sufficient condition $p \sqcap q = \bot$ implies $\gamma(p) \cap \gamma(q) = \emptyset$ and so $\gamma(p) \not\subseteq \gamma(q)$.

7.3 Reduction of a Segmented Decision Tree by an Abstract Property, Tests

In a test, all paths that are feasible in the segmented decision tree are preserved while all the paths that, for sure, can never be followed according to the tested condition are disregarded.

Example 7. Assume that in *Ex. 5*, the test is on B_1. On the *true* branch of the test, the *false* subtree is disregarded while on the *false* branch of the test the *true* subtree is disregarded. We get:



□

Formally, we define the restriction $t \sqcap_t p$ of the segmented decision tree t by condition $p \in D$ as

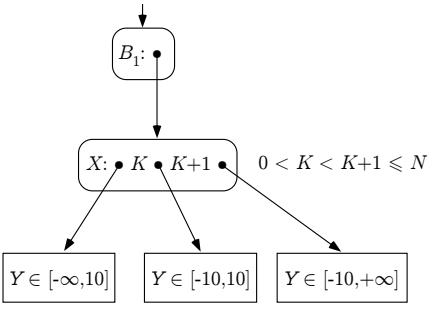
$$
\begin{array}{l}
\bot_{\mathbb{D}} \sqcap_t p \triangleq \bot_{\mathbb{D}} \\
t \sqcap_t \bot \triangleq \bot_{\mathbb{D}} \\
\quad (\!|p'|\!) \quad \sqcap_t p \triangleq \quad (\!|p' \sqcap_{D_\ell} D_\ell(p)|\!) \\
\quad [\![\mathtt{x}\,\{C\} : t_0 b_1 t_1 \ldots b_n t_n]\!] \quad \sqcap_t p \triangleq \\
\quad \text{let} \quad t'_0 \triangleq t_0 \sqcap_t (p \sqcap D(C) \sqcap (\, n = 0 \;?\; \top : D(\mathtt{x} < b_1)\,)) : \\
\quad \text{and for } i = 1, \ldots, n-1 : \; t'_i \triangleq t_i \sqcap_t (p \sqcap D(C) \sqcap D(b_i \leqslant \mathtt{x} < b_{i+1})), \\
\quad \text{and if } n > 0, \; t'_n \triangleq t_n \sqcap_t (p \sqcap D(C) \sqcap D(\mathtt{x} \geqslant b_n)) \\
\quad \text{in } (\, (\forall i \in [0,n] : t'_i = \bot_{\mathbb{D}}) \;?\; \bot_{\mathbb{D}} : \\
\qquad \text{let } l = \min\{i \in [0,n] \mid t'_i \neq \bot_{\mathbb{D}}\} \text{ and } m = \max\{i \in [0,n] \mid t'_i \neq \bot_{\mathbb{D}}\} \text{ in} \\
\qquad\quad [\![\mathtt{x}\,\{\mathrm{relax}_C\,(C,p)\} : L'_l b_{l+1} t'_{l+1} \ldots b_m t'_m]\!] \quad)
\end{array}
$$

Note that the information of p that changes the ordering is kept global. On the contrary, it is possible to relax the incremental information on pre-order at each node: each constraint implied by both C and p can safely be removed from C, leading to a more compact representation. The choice of computing that relaxation depends on the domain for C. This choice is noted $\mathrm{relax}_C\,(C,p)$ in the algorithm. In addition, we can perform another optimization when a node is left with only one child. In that case, we can join the pre-order of the node and its child and put top as the incremental pre-order for the child.

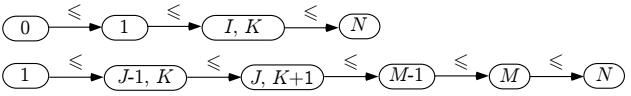
7.4 Segments Unification, Tree Merges and Binary Operations

When performing a binary operation on two decision trees (joins, widening, ordering...), we have to go through the two trees at the same time. Because the variables are ordered, for each pair of subtree during the traversal, the root is on the same decision variable, but the segments may have no bound in common. So, in order to push the binary operation to subtrees, we need to find a common refinement for the two segments (as given by the refinement algorithm in [11]), which we call segments unification.

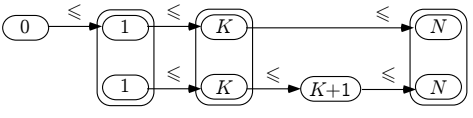
Example 8. Consider the random assignment $B_1 =?$ to the boolean variable B_1 in the context of *Ex. 5*. The subtrees of the two segments of B_1 must be merged, as follows



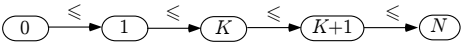
The pre-orders on the decision variable X in *Ex. 5* involve the bounds of the decision variable and the equivalence classes of the expressions appearing in the segmentation.



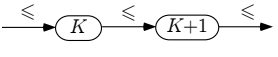
The union of these pre-orders eliminates the variables I, J, and M since they are not comparable in both pre-orders. For example the first pre-order may correspond to a program context where $I = K$ but this might not hold in the program context corresponding to the second pre-order. However although K and N might have different values in these two program contexts, the relation $K \leqslant N$ is valid in both program contexts and so is preserved in the union of the pre-orders.



This union contains only one maximal chain



which yields a relation between (classes of equal) expressions which is valid in both pre-orders and can therefore be attached to the merged node for X. The segmentation for X in the merged tree is the subchain obtained by considering classes of expressions with representatives appearing in either of the original segmentations (that is $K = I$ and $K + 1 = J$ while 0, 1 and N did not appear).



- The subtree in the refined first segment $X < K$ is the merge of the subtrees of the corresponding segment $Y \in [-\infty, 0]$ on the left ($X < I = K$) and $Y \in [-1, 10]$ on the right ($X < J = K + 1$), that is $Y \in [-\infty, 0] \sqcup [-1, 10] = [-\infty, 10]$.
- The subtree in the refined second segment $K \leqslant X < K + 1$ is the merge of the subtrees of the corresponding segments on the left ($Y \in [-10, 1]$ when $X \geqslant I = K$) and on the right ($Y \in [-1, 10]$ when $X < J = K + 1$), that is $Y \in [-10, 1] \sqcup [-1, 10] = [-10, 10]$.
- The subtree in the refined third and last segment $X \geqslant K + 1$ is the merge of the subtrees of the corresponding segments on the left ($Y \in [-10, 1]$ for $X \geqslant I = K$) and on the right ($Y \in [0, 0] \sqcup [0, +\infty]$ for $J = K + 1 \leqslant X < M$ or $M \leqslant X$), that is $Y \in [-10, 1] \sqcup [0, 0] \sqcup [0, +\infty] = [-10, +\infty]$). □

In contrast with simple array segmentation, we may have richer ordering informations, and it will be useful to provide precise segments unification. Given

two nodes $[\![\mathtt{x}\,\{C^0\} : t^0_0 b^0_1 \ldots b^0_{n^0} t^0_{n^0}]\!]$ and $[\![\mathtt{x}\,\{C^1\} : t^1_0 b^1_1 \ldots b^1_{n^1} t^1_{n^1}]\!]$, assuming that we collected all the preordering informations in C^0 and C^1, we must compute two nodes $[\![\mathtt{x}\,\{C^0\} : t'^0_0 b_1 \ldots b_n t'^0_n]\!]$ and $[\![\mathtt{x}\,\{C^1\} : t'^1_0 b_1 \ldots b_n t'^1_n]\!]$ which do share the same bounds and are sound (and precise) over-approximations of the input nodes. For these nodes to be valid, the bounds $b_1 \ldots b_n$ must respect the ordering in both C^0 and C^1. So, the first step is to compute $C^0 \cup_C C^1$. Then the resulting nodes will be more precise if the new bounds can all be compared to the previous bounds. That is, for all $k \in \{0,1\}$, for all $i \leq n^k$ and all $j \leq n$, either $\forall \rho \in \gamma_c(C^k)[\![b^k_i]\!]\rho \leq [\![b_j]\!]\rho$ or $\forall \rho \in \gamma_c(C^k)[\![b^k_i]\!]\rho \geq [\![b_j]\!]\rho$. Then solving the segment unification problem consists in finding a chain in the pre-order defined by $C^0 \cup_C C^1$ such that all elements of the chain respect that property. In general there is no best chain, but the longer the chain the better, and it is certainly best to maximize the number of bounds b^k_i such that there is a b_j such that $\forall \rho \in \gamma_c(C^k)[\![b^k_i]\!]\rho = [\![b_j]\!]\rho$, meaning that a maximum number of bounds are preserved.

The algorithm to find the bounds $b_1 \ldots b_n$ will thus starts by building a graph whose vertices are elements of each segments and the expressions in E that are equal to a bound in each segment according to the C^k's (including extrema for the decision variable, if any). Then on that graph, we merge the strongly connected components. Each vertex of this graph can be colored by a couple of bounds (b^0_i, b^1_j) or by one bound of a segment. Then we find the path in the graph with maximal number of colors. A couple of optimizations can be used to compute that path, and any path is a correct, although maybe imprecise, answer, so we can also stop that algorithm at any point based on a time limit.

Once we have computed the bounds $b_1 \ldots b_n$, we compute the new subtrees

$$[\![\mathtt{x}\,\{C^0\} : t^0_0 b^0_1 \ldots b^0_{n^0} t^0_{n^0} \bowtie_{C^0} b_1 \ldots b_n]\!]$$
and
$$[\![\mathtt{x}\,\{C^1\} : t^1_0 b^1_1 \ldots b^1_{n^1} t^1_{n^1} \bowtie_{C^1} b_1 \ldots b_n]\!]$$

where:

$t_0 a_1 t_1 \ldots a_m t_m \bowtie_C b_1 \ldots b_n =$

— if $m = 0$ then
  – if one of the b_i is such that $\mathtt{x} < b_i$ in C, let l be the smallest such i. The result is $t_0 b_1 \ldots b_{l-1} t_0 b_l \bot b_{l+1} \ldots b_m \bot$
  – else $t_0 b_1 \ldots b_n t_0$
— else if $n = 0$ then $t_0 \cup \ldots \cup t_m$
— else if $b_1 = a_1$ in C, we must look at b_2, if any, in case of segment creation:
  – if $n > 1$ and $b_2 = b_1$ in C, then we create a segment. The value of that segment depends on the binary operation (as in [11]). If the operation is a join, a widening or the segment is the first argument of inclusion testing then the value is $\bot$. If the operation is a meet, or a narrowing, or the second argument of an inclusion testing then the value is $\top$. Let us call $\mathbb{I}$ that value. the result is then $t_0 b_1 (\mathbb{I} a_1 t_1 \ldots a_m t_m \bowtie_C b_2 \ldots b_n)$
  – else the result is $t_0 b_1 (t_1 a_2 t_2 \ldots a_m t_m \bowtie_C b_2 \ldots b_n)$
— else if $a_1 \leq b_1$ in C then

  - if $m > 1$ and $b_1 \leq a_2$ in C then $(t_0 \cup t_1)b_1t_1a_2 \dots a_mt_m \bowtie_C b_1 \dots b_n$
  - else $(t_0 \cup t_1)a_2t_2 \dots a_mt_m \bowtie_C b_1 \dots b_n$
- else $(b_1 \leq a_1)$,
  - if $b_1 \leq \mathtt{x}$ in C then $\bot b_1(t_0a_1t_1 \dots a_mt_m \bowtie_C b_2 \dots b_n)$
  - else $t_0b_1(t_0a_1t_1 \dots a_mt_m \bowtie_C b_2 \dots b_n)$

Once we have computed two new trees which agree on the bounds, we can perform the binary operation. The binary operation must be carried on the incremental pre-orders and on each pair of subtrees recursively, until reaching the leaves where we can compute the binary operation on D_ℓ.

Join. There is a special case when joining two trees: in case we create new segments, there is an opportunity to add incremental pre-order informations. If we merge $[\![\mathtt{x} \{C\} : \dots]\!]$ in a context of pre-order information C_1, and $\bot$ in the context of C_2, then the result is $[\![\mathtt{x} \{C \sqcap_c (\text{relax}_C (C_1, C_2))\} : \dots]\!]$ because we know that on that segment only the pre-order in C_1 is true. For that mechanism to be precise, we need to keep track of the pair of possible pre-orders during the computation of the joins.

Widening. Because segments are split, their number in a decision node can grow indefinitely so that the segmented decision tree can explode in breadth. This must be prevented by a segment widening. We are in a situation where one abstract domain (the segments of a node) is a basis for another one. We can use a mechanism similar to [22] and widen on the segmentations, then on that common segmentation apply the widening child by child.

A first possibility for the segmentation widening is the segments unification of [11] which is based on the use of the common expressions in the two segmentations. In that way, the number of expressions that appear in segments can only decrease. We can achieve the same property by keeping all expressions that can occur as bounds of the first argument of the widening and only those expressions. This is easily implemented by keeping only those expressions in the graph where we look for a best chain of expressions.

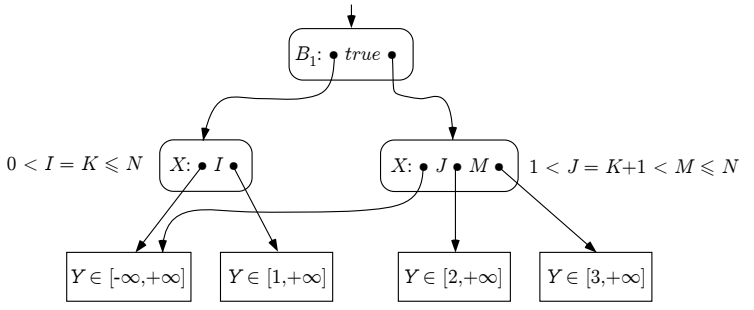
7.5 Assignments

As variables may occur in leaves, expression bounds or as decision variables, we have to consider all cases, keeping in mind that those cases are not exclusive.

Assignments to leaf variables. An assignment to a variable appearing in the leaf nodes only will determine the feasible paths to the leaves where it appears and perform the assignments in each of these leaves (in the abstract domain of the leaves).

Example 9. Assuming in *Ex. 5* that nothing is known on the upper bound of I, J, K, M, and N in the variable environment, the assignment $Y = X$ will determine that either $\neg B_1$ in which case if $X < I$ then else $X \geqslant I > 0$ so

$Y \in [1, +\infty]$ or B_1 holds and so either $X < J$ in which case $Y \in [-\infty, +\infty]$, or $1 < J \leqslant X$ so $Y \in [2, +\infty]$, or else $1 < J < M \leqslant X$ and so $Y \in [2, +\infty]$. We get



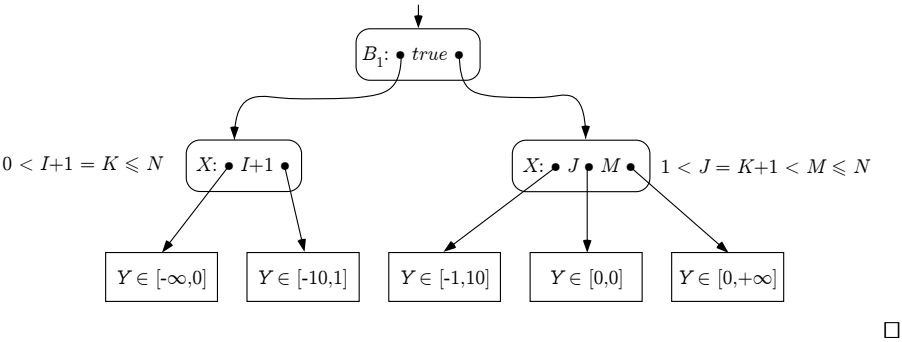
□

In general the assignment of an expression to a variable involves some conditions (such as absence of overflow, positiveness, non-nullity, etc) that have to be taken into account by pruning the tree as in Sect. 7.3. In case where we have to do such pruning, we can follow the same algorithm, but performing the assignment at the leaves in addition to imposing the test.

Assignments to segment bound variables. An assignment to a variable appearing in segment bounds may be invertible, in which case segments which were based on the old value of the variable can be expressed based on the new value, or not invertible, in which case it is not possible to keep the segments bounds when they are only expressible in terms of the old values of the assigned variable.

More precisely, if the assignment can be expressed as $b = f(b)$ and f invertible, we can replace the variable b by $f^{-1}(b)$ in each expression appearing in bounds of the decision tree, and that encodes the same property. To complete the assignment, we must also carry it to incremental pre-orders at each node.

Example 10. Consider the assignment $I = I - 1$ in the context of *Ex. 5*. After this assignment the old value I_o of I (to which *Ex. 5* is referring to) can be expressed in terms of the new value I_n as $I_n = I_o - 1$ so $I_o = I_n + 1$ by inversion. So, we get the post-condition of the assignment $I = I - 1$ by replacing I by $I + 1$ in the segmented decision tree of *Ex. 5*.



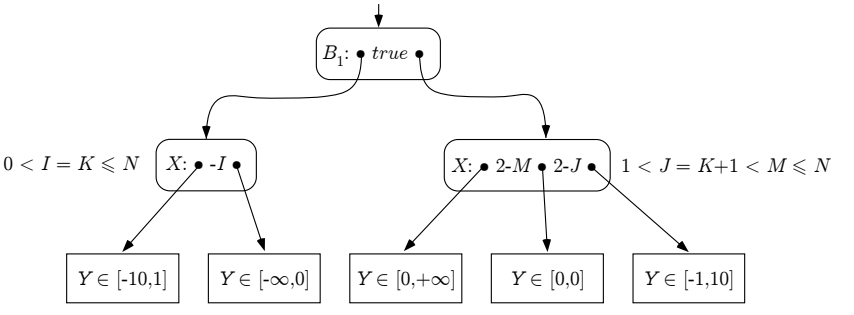
□

In case of non-invertible assignment to a variable b appearing in a bound, we look at all bounds were b appears. If at that bound the pre-order information can provide another expression that is known to be equal to the bound but that does not contain b, we can replace the bound by that expression. Otherwise, we drop the bound from the segmentation and merge the two consecutive subtrees that were separated by that bound. As for the tests, if that results in only one child for a node, we can push up the incremental pre-order information of that child.

In addition, the assignment must be carried also in the incremental pre-orders.

Assignments to decision variables. For an invertible assignment $X = f(X)$ to a decision variable X where X_o (resp. X_n) is the value of X before (resp. after) the assignment, we have $X_n = f(X_o)$ such that $X_o = f^{-1}(X_n)$. The segment conditions $b \leqslant X_o$ are transformed into $b \leqslant f^{-1}(X_n)$ that is $f(b) \leqslant X_n$ when f is increasing and $f(b) \geqslant X_n$ when f is decreasing. Similarly $X_o < b$ are transformed into $f^{-1}(X_n) < b$ that is $X_n < f(b)$ when f is strictly increasing and $x_n > f(b)$ when f is strictly decreasing. If f also depends on other decision variables, their abstract value must be evaluated along the path to the nodes for X.

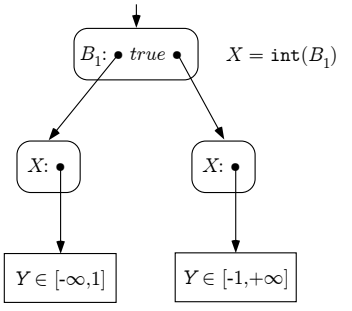
Example 11. Consider the invertible assignment is $X = \texttt{int}(B_1) - X$ in *Ex. 5* where $\texttt{int}(\mathit{false}) = 0$ and $\texttt{int}(\mathit{true}) = 1$ that is $X = -X$ when B_1 is *false* and $X = 1 - X$ when B_1 is *true* , which are both invertible assignments.



□

In case of non-invertible assignment, we cannot keep the segments related to the assigned variable. So we merge the children of that variable. In case where the assignment `x=e` can be represented exactly in the pre-order domain, then we are as precise as possible. Otherwise, it is still possible to add some information in the tree. For example if the expression is a monotone function over another variable `y` smaller than `x` (for $<_{\mathbb{D}}$) then we can store the inequalities implied by the segmentation over `y` into the incremental pre-orders associated with `y`. If the variable is greater than `x`, we can do the same if all the nodes on `y` share a bound.

Example 12. Consider the non-invertible assignment is $X = \texttt{int}(B_1)$. The post-condition is preserved while selectively merging the children. Assuming $\texttt{int}(b)$ to be a canonical integer expression for canonical Boolean expressions b, we get:



□

8 Abstracting Functions (and Array Contents)

Binary Decision Diagrams were originaly developped to represent boolean functions [1]. In the same way, segmented decision trees can be used to approximate functions over totaly ordered domains: we make decisions for each parameter of the function, and the leaves of the tree represent the possible values of the function for that constraint on the parameters.

Example 13. The function $\sin x$, $x \in [0, 2\pi]$ could be approximated by the segmented decision tree $[\![x \{0 \leqslant x \leqslant 2\pi\} : (\!| \sin x : [0,1] |\!) \ \pi \ (\!| \sin x : [-1,0] |\!)]\!]$. □

Formaly, a function $f(x_0, \ldots, x_n)$ can be seen as a set of vectors of size $n+1$ of the form $< v_0, \ldots, v_n, f(v_0, \ldots, v_n) >$. Then a property over functions is a set of sets of vectors. The first abstraction we perform is to go back to sets of vectors, by taking the union of the sets, then we are in a setting where we can use segmented decision trees directly, with decision variables the first n variables ordered by the order on parameters of the function. Such abstraction could be very powerful to summarize functions and perform modular analyzes.

Multi-dimensional arrays can be seen as functions from index values to array content. So we can use the same combination of abstractions and obtain precise representations. Because arrays don't have formal parameters, we just need a convention to name to variables which will correspond to the array dimensions and to the array content. One possibility is to subscript the array name with the number of the dimension for the indexes and with v for the content.

One more important difference between functions and arrays is the assignment. It is easily implemented as an assignment to the content variable under the appropriate condition for the dimension variables. May-assign can arize when such conditions cannot be represented exactly in the expressions allowed as bounds, but in general this mechanism will be very precise.

If we specialize that abstraction to arrays of dimension 1, we have an abstraction that is equivalent to [11] where the leaves consisted of abstractions of couples index-content.

9 Examples

9.1 Conditional Computation

```
           int x1, x2, y, z;
           struct s;
/*  0: */  x1 = 0;
           y = z = 1;
           s = INIT;
/*  1: */  while /* 2: */( z < 100 ) {
/*  3: */    if (x1 < y) s = INIT; /* 4: */
             else { /* 5: */
                if (x2 > y) s = SPECIAL; /* 6: */
                else s = computation(s, x2); /* 7: */
/*  8: */    }
/*  9: */    z++;
/* 10: */    if (?) y++; /*11: */
/* 12: */    if (?) x1++;/*13: */
/* 14: */  }
/* 15: */  if (x1>y && x2>y) /* 16: */ assert(s==SPECIAL);
```

In that abstract program, the structure `s` is the output and the variable `x2` is the input. On some variation, `x2` may change at each loop iteration but that does not change the invariants here, so we just consider that its value is fixed.

A pre-analysis can fix the decision variables to be `x1` and `x2` as they are in guards to compute values and then in a guard to test those values. The structure `s` will be at the leaves (the result of `computation` is abstracted by `COMP`).

The fixpoint iteration with widening will go as follows (we write $[\![\mathtt{v} : \ldots]\!]$ for $[\![\mathtt{v}\,\{true\} : \ldots]\!]$):

`1-9:` $[\![x1\,\{x1 = 0,\, y = z = 1\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]]\!]$ ¿Initialization with input $x2$ʔ

`14:` $[\![x1\,\{x1 \in [0,1],\, y \in [1,2],\, z = 2\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]]\!]$

`2:` $[\![x1\,\{0 \leq x1 \leq y \leq z,\, x1 < z,\, 0 < y\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]]\!]$

`3:` $[\![x1\,\{0 \leq x1 \leq y \leq z,\, x1 < z,\, 0 < y\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]]\!]$

`4:` $[\![x1\,\{0 \leq x1 < y \leq z < 100\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]]\!]$

`5:` $[\![x1\,\{0 < y = x1 < z < 100\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]]\!]$

`6:` $[\![x1\,\{0 < y = x1 < z < 100,\, y < x2\} : [\![x2 :\ (\!|\,\mathtt{SPECIAL}\,|\!)]\!]]\!]$

`7:` $[\![x1\,\{0 < y = x1 < z < 100,\, x2 \leq y\} : [\![x2 :\ (\!|\,\mathtt{COMP}\,|\!)]\!]]\!]$

`8:` $[\![x1\,\{0 < y = x1 < z < 100\} : [\![x2 :\ (\!|\,\mathtt{COMP}\,|\!)]\!]\ y + 1\ (\!|\,\mathtt{SPECIAL}\,|\!)]\!]$

`9:` $[\![x1\,\{0 \leq x1 \leq y \leq z < 100,\, x1 < z,\, 0 < y\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]\ y\ [\![x2 :\ (\!|\,\mathtt{COMP}\,|\!)]\!]\ y + 1\ (\!|\,\mathtt{SPECIAL}\,|\!)]\!]$

`10:` $[\![x1\,\{0 \leq x1 \leq y < z < 101,\, x1 < z - 1,\, 0 < y\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]\ y\ [\![x2 :\ (\!|\,\mathtt{COMP}\,|\!)]\!]\ y + 1\ (\!|\,\mathtt{SPECIAL}\,|\!)]\!]$

`11:` $[\![x1\,\{0 \leq x1 < y \leq z < 101,\, x1 < z - 1,\, 1 < y\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]\ y - 1\ [\![x2 :\ (\!|\,\mathtt{COMP}\,|\!)]\!]\ y\ (\!|\,\mathtt{SPECIAL}\,|\!)]\!]$

`12:` $[\![x1\,\{0 \leq x1 \leq y \leq z < 101,\, x1 < z - 1,\, 0 < y\} : [\![x2 :\ (\!|\,\mathtt{INIT}\,|\!)]\!]\ y - 1$

$[\![x2 : (\!|\texttt{COMP} \cup \texttt{INIT}|\!)\ y\ (\!|\texttt{COMP} \cup \texttt{SPECIAL}|\!)]\!]\ y$

$[\![x2 : (\!|\texttt{COMP}|\!)\ y\ (\!|\texttt{COMP} \cup \texttt{SPECIAL}|\!)\ y+1\ (\!|\texttt{SPECIAL}|\!)]\!]]\!]$

`13:` $[\![x1\,\{1 < x1 < z < 101,\ 0 < y \leq z\} : [\![x2 : (\!|\texttt{INIT}|\!)]\!]\ y$

$[\![x2 : (\!|\texttt{COMP} \cup \texttt{INIT}|\!)\ y\ (\!|\texttt{COMP} \cup \texttt{SPECIAL}|\!)]\!]\ y+1$

$[\![x2 : (\!|\texttt{COMP}|\!)\ y\ (\!|\texttt{COMP} \cup \texttt{SPECIAL}|\!)\ y+1\ (\!|\texttt{SPECIAL}|\!)]\!]]\!]$

`14:` $[\![x1\,\{0 \leq x1 < z < 101,\ 0 < y \leq z\} : [\![x2 : (\!|\texttt{INIT}|\!)]\!]\ y-1$

$[\![x2 : (\!|\texttt{COMP} \cup \texttt{INIT}|\!)\ y\ (\!|\texttt{COMP} \cup \texttt{SPECIAL} \cup \texttt{INIT}|\!)]\!]\ y$

$[\![x2 : (\!|\texttt{COMP} \cup \texttt{INIT}|\!)\ y\ (\!|\texttt{COMP} \cup \texttt{SPECIAL}|\!)]\!]\ y+1$

$[\![x2 : (\!|\texttt{COMP}|\!)\ y\ (\!|\texttt{COMP} \cup \texttt{SPECIAL}|\!)\ y+1\ (\!|\texttt{SPECIAL}|\!)]\!]]\!]$

`2:` $=$ `14:` without $z < 101$

⟅as the union of `14:` and `2:` is `14:` here and this is the abstract loop invariant⟆

`15:` $=$ `2:`

`16:` $[\![x1\,\{0 < y < x1 < z,\ y < x2,\ 99 < z\} : [\![x2 : (\!|\texttt{SPECIAL}|\!)]\!]]\!]$

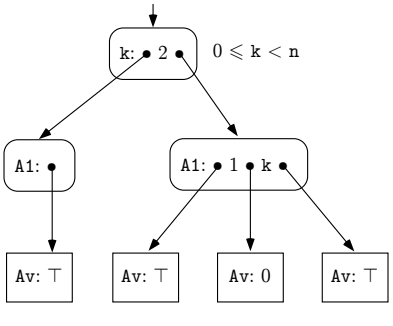
⟅The assertion in `16:` is proved correct.⟆

9.2 Partial Array Initialization

The program below partially initializes an array `A`.

```
         int n; /* n > 0 */
         int k, A[n];
/* 0: */ k = 0;
/* 1: */  while /* 2: */ (k < n) {
/* 3: */     if (k > 0) {
/* 4: */        A[k] = 0;
/* 5: */      };
/* 6: */      k = k+1;
/* 7: */  };
/* 8: */
```

The ordering abstract domain D_c is assumed to be the octagon abstract domain [20]). Following Sect. 8, an array `A` is abstracted by two fresh variables $\texttt{A1} \in \mathbb{D}$ to segment indices `A1` of array `A`, $\texttt{A1} \in [\texttt{A}.\mathit{low}, \texttt{A}.\mathit{high}]$ and a variable $\texttt{Av} \in \mathbb{D}$ standing for any value of the array in a given segment such that $\texttt{Av} <_{\mathbb{D}} \texttt{A1}$ and `Av` is a leave. For leaves we use constant propagation [15]. The loop invariant found at point `3` is



The fixpoint iteration with widening is the following:

$\texttt{0:}\quad [\![\, \texttt{k}\,\{0 < \texttt{n}, 0 \leqslant \texttt{A1} < \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅k and A uninitialized⟆

$\ell\texttt{:}\quad \bot$ ⟅$\ell = \texttt{1}, \ldots, \texttt{8}$, infimum⟆

$\texttt{1:,2:,3:,6:}\quad [\![\, \texttt{k}\,\{\texttt{k} = 0 < \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅0: where $\texttt{k} = 0$, $\texttt{k} < \texttt{n}$, $\texttt{k} \leqslant 0$⟆

$\texttt{7:}\quad [\![\, \texttt{k}\,\{\texttt{k} = 1 \leqslant \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅6: where $\texttt{k} = \texttt{k} + 1$⟆

$\texttt{2:,3:}\ [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} \leqslant 1,\ \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅joining 1: and 7:, test $\texttt{k} < \texttt{n}$⟆

$\texttt{4:}\quad [\![\, \texttt{k}\,\{1 = \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅3: with $\texttt{k} > 0$⟆

$\texttt{5:}\quad [\![\, \texttt{k}\,\{1 = \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ 2\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$

⟅4: with $\texttt{A[k]} = 0$ where $\texttt{k} = 1$⟆

$\texttt{6:}\quad [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} \leqslant 1,\ \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1}\,\{\texttt{k} = 0\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 1$

$[\![\, \texttt{A1}\,\{\texttt{k} = 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ 2\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$

⟅joining 3: and $\texttt{k} \leqslant 0$ so $\texttt{k} = 0$ together with 5: where $\texttt{k} = 1$⟆

$\texttt{7:}\quad [\![\, \texttt{k}\,\{1 \leqslant \texttt{k} \leqslant 2,\ \texttt{k} \leqslant \texttt{n}\} : [\![\, \texttt{A1}\,\{\texttt{k} = 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1}\,\{\texttt{k} = 2\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ 2\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅6: where $\texttt{k} = \texttt{k} + 1$⟆

$\texttt{1:} \sqcup_t \texttt{7:}\ [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} \leqslant 2,\ \texttt{k} \leqslant \texttt{n}\} : [\![\, \texttt{A1}\,\{0 \leqslant \texttt{k} \leqslant 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1}\,\{\texttt{k} = 2\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ 2\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅join of 1: and 7:⟆

$\texttt{2:,3:}\quad [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1}\,\{0 \leqslant \texttt{k} \leqslant 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ \texttt{k}\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅2: ∇ (1: $\sqcup_t$ 7) [3], test $\texttt{k} < \texttt{n}$⟆

$\texttt{4:}\quad [\![\, \texttt{k}\,\{0 < \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1}\,\{\texttt{k} = 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ \texttt{k}\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅3: with $\texttt{k} > 0$⟆

$\texttt{5:}\quad [\![\, \texttt{k}\,\{0 < \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1}\,\{\texttt{k} = 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ 2\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ \texttt{k} + 1\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅4: with $\texttt{A[k]} = 0$⟆

$\texttt{6:}\quad [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1}\,\{\texttt{k} = 0\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 1$

$[\![\, \texttt{A1}\,\{\texttt{k} = 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ 2\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ \texttt{k} + 1\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$

⟅joining 3: and $\texttt{k} \leqslant 0$ with 5:⟆

$\texttt{7:}\quad [\![\, \texttt{k}\,\{0 < \texttt{k} \leqslant \texttt{n}\} : [\![\, \texttt{A1}\,\{\texttt{k} = 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1}\,\{\texttt{k} = 2\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ 2\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 3$

$[\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ \texttt{k}\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅6: where $\texttt{k} = \texttt{k} + 1$⟆

$\texttt{1:} \sqcup_t \texttt{7:}\ [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} \leqslant \texttt{n}\} : [\![\, \texttt{A1}\,\{0 \leqslant \texttt{k} \leqslant 1\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ \texttt{k}\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$ ⟅join of 1: and 7:⟆

$\texttt{2:,3:}\quad [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} < \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ \texttt{k}\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ ]\!]$

⟅2: ∇ (1: $\sqcup_t$ 7), test $\texttt{k} < \texttt{n}$, convergence, 3: is the abstract loop invariant⟆

$\texttt{8:}\quad [\![\, \texttt{k}\,\{0 \leqslant \texttt{k} = \texttt{n}, 0 \leqslant \texttt{A1} < \texttt{n}\} : [\![\, \texttt{A1} :\ (\!|\, \texttt{Av} : \top \,|\!)\ ]\!]\ 2$

$[\![\, \texttt{A1}\,\{0 \leqslant \texttt{k} < \texttt{n}\} :\ (\!|\, \texttt{Av} : \top \,|\!)\ 1\ (\!|\, \texttt{Av} : 0 \,|\!)\ ]\!]\ ]\!]$

⟅2: and $\texttt{k} \geqslant \texttt{n}$, program postcondition⟆

[3] When a new branch is taken in a test within a loop the widening is usually delayed, which we avoid to shorten the example.

Observe that the segmented decision tree automatically discovers a partition of the loop body as given by the condition $\mathtt{k} > 0$ while the segmented array partitions the values of the array elements according to variable `k`.

9.3 Multidimentional Arrays

The program below partially initializes a matrix `M`.

```
           int m, n; /* m, n > 0 */
           int i, j, M[m,n];
/*  0: */  i = 0;
/*  1: */  while /* 2: */ (i < m) {
/*  3: */     j = i+1;
/*  4: */     while /* 5: */ (j < n) {
/*  6: */        M[i,j] = 0;
/*  7: */        j = j+1;
/*  8: */     };
/*  9: */     i = i+1;
/* 10: */  };
/* 11: */
```

A global invariant is $0 \leq \mathtt{M1} < \mathtt{m}$ and $0 \leq \mathtt{M2} < \mathtt{n}$, so we keep it implicit in the following fixpoint iteration:

$\mathtt{0}$: $[\![\mathtt{M1} : [\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)]\!]]\!]$ ⟅program precondition: `i`, `j`, and `A` uninitialized⟆

ℓ: $\bot$ ⟅$\ell = 1, \ldots, 11$, infimum⟆

$\mathtt{1}$:,$\mathtt{2}$:,$\mathtt{3}$: $[\![\mathtt{M1}\,\{\mathtt{i} = 0\} : [\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)]\!]]\!]$ ⟅`0:` with $\mathtt{i} = 0, \mathtt{i} < \mathtt{m}$ [4]⟆

$\mathtt{4}$:,$\mathtt{5}$:,$\mathtt{6}$: $[\![\mathtt{M1}\,\{\mathtt{i} = 0, \mathtt{j} = \mathtt{i} + 1 = 1 < \mathtt{n}\} : [\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)]\!]]\!]$
⟅`3:` with `j = i+1;`, $\mathtt{j} < \mathtt{n}$⟆

$\mathtt{7}$: $[\![\mathtt{M1}\,\{\mathtt{i} = 0, \mathtt{j} = \mathtt{i} + 1 = 1 < \mathtt{n}\} :$
$[\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)\ \mathtt{j}\ (\!| \mathtt{Mv} : 0 |\!)\ \mathtt{j} + 1\ (\!| \mathtt{Mv} : \top |\!)]\!]\ \mathtt{i} + 1\ [\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)]\!]$
$]\!]$ ⟅`6:` with `M[i,j] = 0;`⟆

$\mathtt{8}$: $[\![\mathtt{M1}\,\{\mathtt{i} = 0, \mathtt{j} = \mathtt{i} + 2 = 2 \leqslant \mathtt{n}\} :$
$[\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)\ \mathtt{j} - 1\ (\!| \mathtt{Mv} : 0 |\!)\ \mathtt{j}\ (\!| \mathtt{Mv} : \top |\!)]\!]\ \mathtt{i} + 1\ [\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)]\!]$
$]\!]$ ⟅`7:` with `j = j+1;`⟆

$\mathtt{4} : \sqcup_t \mathtt{8}$: $[\![\mathtt{M1}\,\{\mathtt{i} = 0, \mathtt{i} + 1 \leqslant \mathtt{j} \leqslant \mathtt{i} + 2 \leqslant \mathtt{n}\} :$
$[\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)\ 1\ (\!| \mathtt{Mv} : 0 |\!)\ \mathtt{j}\ (\!| \mathtt{Mv} : \top |\!)]\!]\ \mathtt{i} + 1\ [\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)]\!]$
$]\!]$ ⟅join of `4:` and `8:`⟆

$\mathtt{5}$: $[\![\mathtt{M1}\,\{\mathtt{i} = 0, \mathtt{i} + 1 \leqslant \mathtt{j} \leqslant \mathtt{n}\} :$
$[\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)\ 1\ (\!| \mathtt{Mv} : 0 |\!)\ \mathtt{j}\ (\!| \mathtt{Mv} : \top |\!)]\!]\ \mathtt{i} + 1\ [\![\mathtt{M2} : (\!| \mathtt{Mv} : \top |\!)]\!]$
$]\!]$ ⟅`5:` ∇ (`4:` $\sqcup_t$ `8:`) [4]⟆

[4] When a new branch is taken in a test within a loop the widening is usually delayed, which we avoid to shorten the example.

9: $\llbracket$ M1 {i = 0, i + 1 ⩽ j = n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ i + 1
$\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket\rrbracket$
⟅5: and j ⩾ n⟆

10: $\llbracket$ M1 {i = 1, i ⩽ j = n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket\rrbracket$
⟅9: and i = i+1;⟆

1: ⊔$_t$ 10: $\llbracket$ M1 {i = 1, i ⩽ j = n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket\rrbracket$
⟅join of 1: and 10:⟆

2: $\llbracket$ M1 {0 ⩽ i} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket\rrbracket$
⟅2: ∇ (1: ⊔$_t$ 10:)⟆

3: $\llbracket$ M1 {0 ⩽ i < m} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket\rrbracket$
⟅2: and j < n⟆

4:,5:,6: $\llbracket$ M1 {0 ⩽ i < m, j = i + 1 < n} :
$\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket$
$\rrbracket$
⟅3:, j = i+1; and j < n⟆

7: $\llbracket$ M1 {0 ⩽ i < m, j = i + 1 < n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i
$\llbracket$ M2 : ⟮Mv : ⊤⟯ j ⟮Mv : 0⟯ j + 1 ⟮Mv : ⊤⟯ $\rrbracket$ i + 1 $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket$
$\rrbracket$
⟅6: and M[i, j] = 0;⟆

8: $\llbracket$ M1 {0 ⩽ i < m, j = i + 2 ⩽ n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ n$\rrbracket$ i
$\llbracket$ M2 : ⟮Mv : ⊤⟯ j − 1 ⟮Mv : 0⟯ j ⟮Mv : ⊤⟯ $\rrbracket$ i + 1 $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket$
$\rrbracket$
⟅7: with j = j+1;⟆

4: ⊔$_t$ 8: $\llbracket$ M1 {0 ⩽ i < m, i + 1 ⩽ j ⩽ i + 2 ⩽ n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i
$\llbracket$ M2 : ⟮Mv : ⊤⟯ i + 1 ⟮Mv : 0⟯ j ⟮Mv : ⊤⟯ $\rrbracket$ i + 1 $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket$
$\rrbracket$
⟅join of 4: and 8:⟆

5: $\llbracket$ M1 {0 ⩽ i < m, i + 1 ⩽ j ⩽ n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i
$\llbracket$ M2 : ⟮Mv : ⊤⟯ i + 1 ⟮Mv : 0⟯ j ⟮Mv : ⊤⟯ $\rrbracket$ i + 1 $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket$
$\rrbracket$
⟅5: ∇ (4: ⊔$_t$ 8:)⟆

9: $\llbracket$ M1 {0 ⩽ i < m, i + 1 ⩽ j = n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i
$\llbracket$ M2 : ⟮Mv : ⊤⟯ i + 1 ⟮Mv : 0⟯ $\rrbracket$ i + 1 $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket$
$\rrbracket$
⟅5: and j ⩾ n⟆

10: $\llbracket$ M1 {0 < i ⩽ m, i ⩽ j = n} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ 1 ⟮Mv : 0⟯ $\rrbracket$ i − 1
$\llbracket$ M2 : ⟮Mv : ⊤⟯ i ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket$
$\rrbracket$
⟅9: and i = i+1;⟆

1: ⊔$_t$ 10: $\llbracket$ M1 {0 ⩽ i ⩽ m} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ M1 + 1 ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket\rrbracket$
⟅join of 1: and 10:(segments unification yields $1 \leq$ M1 $+ 1 \leq i$ for subtree merges)⟆

2: $\llbracket$ M1 {0 ⩽ i} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ M1 + 1 ⟮Mv : 0⟯ $\rrbracket$ i $\llbracket$ M2 : ⟮Mv : ⊤⟯ $\rrbracket\rrbracket$
⟅2: ⊑ (1: ⊔$_t$ 10:), stabilization at a fixpoint⟆

11: $\llbracket$ M1 {0 < m = i} : $\llbracket$ M2 : ⟮Mv : ⊤⟯ M1 + 1 ⟮Mv : 0⟯ $\rrbracket\rrbracket$
⟅2: and i ⩾ m, program postcondition.⟆

10 Conclusion

Many static analyses are very impressive on small examples but fail to scale up. The problem mainly originates from the explosion of possibles cases in handling disjunctions. Mastering the exponential growth is the key to scalability, while enabling weak forms of disjunction is essential to the precision which is necessary to avoid false alarms. Based on two abstract domain functors that have shown experimentally to scale up, we have proposed a new combination which expressivity is better than each of them taken separately and which complexity can be mastered by imposing both static restrictions (like maximal depth or variable packing) and dynamic restrictions (by widening to control the breath of the tree).

References

1. Bryant, R.E.: Graph based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (1986)
2. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, March 21 (1978) (in French)
3. Cousot, P.: Verification by abstract interpretation. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 243–268. Springer, Heidelberg (2004)
4. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. 2nd Int. Symp. on Programming, pp. 106–130. Dunod, Paris (1976)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th POPL, Los Angeles, pp. 238–252. ACM Press, New York (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6th POPL, San Antonio, pp. 269–282. ACM Press, New York (1979)
7. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
8. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
9. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Proc. of the Seventh ACM Conf. on Functional Programming Languages and Computer Architecture, La Jolla, June 25-28, pp. 170–181. ACM Press, New York (1995)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: The Astrée static analyzer, `www.astree.ens.fr`, `www.absint.com/astree/`
11. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation abstract domain function for the fully automatic and scalable inference of array properties. Research report, Microsoft Research, Redmond (September 2009)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th POPL, Tucson, pp. 84–97. ACM Press, New York (1978)
13. Graf, S., Saïdi, H.: Verifying invariants using theorem proving. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 196–207. Springer, Heidelberg (1996)

14. Granger, P.: Static analysis of arithmetical congruences. Int. J. Comput. Math. 30(3&4), 165–190 (1989)
15. Kildall, G.: A unified approach to global program optimization. In: 1st POPL, Boston, October 1973, pp. 194–206. ACM press, New York (1973)
16. Mauborgne, L.: Analyse statique et domaines abstraits symboliques. Mémoire d'habilitation à diriger les recherches en informatique, Université de Paris Dauphine, February 12 (2007)
17. Mauborgne, L.: Abstract interpretation using TDGs. In: Le Charlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 363–379. Springer, Heidelberg (1994)
18. Mauborgne, L.: Abstract interpretation using typed decision graphs. Science of Computer Programming 31(1), 91–112 (1998)
19. Mauborgne, L.: Binary decision graphs. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 101–116. Springer, Heidelberg (1999)
20. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
21. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–310 (1955)
22. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145. Springer, Heidelberg (1996)

Towards Component Based Design of Hybrid Systems: Safety and Stability*

Werner Damm[1], Henning Dierks[2], Jens Oehlerking[1], and Amir Pnueli[3]

[1] Department for Computer Science
University of Oldenburg, Germany
{damm,jens.oehlerking}@informatik.uni-oldenburg.de
[2] Department of Electrical and Information Engineering
Hamburg University of Applied Sciences, Germany
[3] Computer Science Department
Courant Institute of Mathematical Sciences
New York University

Abstract. We propose a library based incremental design methodology for constructing hybrid controllers from a component library of models of hybrid controllers, such that global safety and stability properties are preserved. To this end, we propose hybrid interface specifications of components characterizing plant regions for which safety and stability properties are guaranteed, as well as exception mechanisms allowing safe and stability-preserving transfer of control whenever the plant evolves towards the boundary of controllable dynamics. We then propose a composition operator for constructing hybrid automata from a library of such pre-characterized components supported by compositional and automatable proofs of hybrid interface specifications.

1 Introduction

The use of component-based design approaches for embedded automotive applications has received strong momentum from the establishment of a de-facto standard for automotive component development by the Autosar consortium (see http://www.autosar.org). While the current notion of component interfaces in Autosar is rather weak, the overall strategic direction of achieving cost reductions by boosting re-use also of application components is expected to lead to an elaboration of the standard towards richer component interfaces (see e.g. [JMM08], [DMO+07]) providing information required for all phases of automotive embedded design flows, notably for establishing safety and real-time requirements. Related projects such as the Integrated Projects Speeds[1] have provided formal contract based component interface specifications (see [JMM08]) for real-time

* This paper reporting on joint research with Amir Pnueli is dedicated to the memory of Amir Pnueli. It has been partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Centre "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

[1] IST Project 03347, see www.speeds.eu.com

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 96–143, 2010.

and safety requirements (as in [DMO+07], [DPJ09]). Reflecting the significant share of control applications in embedded automotive development, this paper strives to extend this research by enabling component-based design for hybrid control applications. In particular, we propose a notion of *hybrid interface specifications* which is sufficiently expressive to cover the bridge between specification and implementation models of control, as elaborated below, and provide a framework for hierarchical construction of hybrid controllers supported by automatic verification tools enabling compositional verification of such interface specifications, which subsume both safety and stability properties.

It is in particular

(1) the need to support not only specification but as well the design phase of such systems,
(2) and the need to cater for both safety and stability requirements

which require extensions of previous work such as [Fre05, Fre06, Fre08] and [TPL04, HMP01] on compositional verification of hybrid systems. It follows from (1), that models which assume instantaneous reactions on mode-switching conditions, as is typically done in hybrid automata, are inadequate, since mode switching typically entails task-switching and thus comes with significant time penalties. This as well as delays in sensing and actuating the plant caused [JBS07] to propose lazy linear hybrid automata as a more realistic model of hybrid control; this model, however, is not supported by a compositional verification approach. Stauner [Sta01, Sta02] addresses the bridge between specification and design models by proposing a relaxed non-standard semantics of hybrid automata serving as specification models and proposes a systematic approach to derive discrete time design models from specification models such that this robustly refines the specification model. This approach assures that the implementation maintains safety properties, but lacks compositionality. A decompositional approach for verification of stability properties has been developed in [OT09] which constructs Lyapunov functions [Lya07] for individual modes so that there combination yields a Lyapunov function for the complete systems thus establishing stability. This approach is, however, not compositional, in that it assumes the complete system to be given as a basis for decomposition, and does not take implementation aspects into account.

The key achievement of this paper is thus the development of a compositional framework for component based design of hybrid controllers taking into account realistic assumptions about reaction times. This entails the need for what we call *alarms* in hybrid interface specifications, which alert the environment of the component about an encountered plant dynamics, for which stability or safety are endangered, such as through trajectories violating the components assumptions on plant dynamics. Such alarms come with an escape interval, during which safety and stability are still guaranteed by the component itself, thus providing sufficient margin for task-switching. It is this paradigm shift from centralized control of mode-switching to a decentralized setting, where modes take responsibility for creating awareness about the need to perform mode-switching, which is fundamental for enabling a distributed component based design of

control systems. We provide a simple language in the style of Massaccio [HMP01] for the hierarchical construction of hybrid controllers, focusing in this paper on sequential composition: this allows to connect (alarm raising) outports of components through guarded transitions with inports of components, whose interface specification explicates assumptions on plant states at entry time so that safety and stability can be guaranteed, or to propagate alarms upward in the hierarchy by connecting these to outports of the enclosing system. Such port connections are annotated both with guards and jumps. We support distributed implementations of such composed components and give a distributed algorithm for resolving multiple helpful offers in alarm situations. This algorithm identifies a single helpful component, to which control is transferred, supporting in particular the delegation of control to yet unknown helpers in the environment of the composed system. Additional ingredients of interface specifications are plant safety and stability requirements, where we support both asymptotic stability as well as time bounded reachability of plant regions. We describe an approach of turning hybrid automata into a basic component supporting distributed helper identification, and employ fully automatic verification procedures [OT09] for verifying the compliance of such a component realization against its interface specification, for dynamics given as linear differential systems of equations, linear convex guards, and linear jumps, in particular relying on automatic synthesis of parameterized families of Lyapunov functions. For composed components, we give automatically verifiable verification conditions jointly ensuring, that local interface specifications augmented with auxiliary invariants such as local parameterized Lyapunov functions imply the interface specification of the composed system. These verification conditions can be seen as generating a constraint systems on parameters of local Lyapunov functions – if this constraint system is solvable, the combined system will meet its stability specification.

This paper is organized as follows. The subsequent section introduces our version of hybrid automata allowing super-dense discrete time switches as well as their parallel composition. Section 3 gives syntax and semantics of hybrid interface specifications, shows how to build basic components from hybrid automata, and provides syntax and semantics for sequential composition of components, incorporating the distributed protocol for helper election. Section 4 shows how to establish the correctness of interface specifications automatically, first for basic components, and then for composed systems. We use as running example a simple automatic cruise control (ACC) system to illustrate our approach. The conclusion summarizes the key achievements and explains directions of further research.

2 Basic Definitions

In the following, we will define a hybrid automaton formalism as required for the compositional design methodology, differentiating between input variables that are assumed to be outside the control of the automaton, and controllable variables that are either local, or outputs variables of the automaton.

Definition 1 (Hybrid Automaton with Inputs). *A* hybrid automaton *is a tuple*

$$H = (\mathbb{M}, Var^{loc}, Var^{in}, Var^{out}, R^D, R^I, R^C, \Phi, \Theta)$$

where

1. $\mathbb{M}$ *is a finite set of* modes*,*
2. Var^{loc}, Var^{in} *and* Var^{out} *are disjoint sets of* local, input and output variables *over* $\mathbb{R}$*, denote* $Var = Var^{loc} \cup Var^{in} \cup Var^{out}$*,*
3. Φ *is a first-order predicate over* Var *and a variable* M*, which takes values in* $\mathbb{M}$*, describing all combinations of* initial states and modes*,*
4. Θ *is a mapping that associates with each mode* $m \in \mathbb{M}$ *a* local invariant $\Theta(m)$*, which is a quantifier-free formula over* Var*,*
5. R^D *is the* discrete transition relation *with elements* $(m, \Phi, \mathcal{A}, m')$ *called* transitions*, which are graphically represented as* $m \xrightarrow{\phi,\mathcal{A}} m'$*, where*
   - $m, m' \in \mathbb{M}$*,*
   - ϕ *is a first-order predicate over* Var*,*
   - $\mathcal{A}$ *is a first-order predicate over* $Var \cup Var^{loc'} \cup Var^{out'}$ *specifying discrete updates, where* $Var^{loc'}$ *and* $Var^{out'}$ *are the primed variants of the variables in* Var^{loc} *and* Var^{out}*,*

   R^D *consists of two disjoint sets* R^D_U *and* R^D_L*. The subset* R^D_U *contains the* urgent *transitions, and the subset* R^D_L *the* lazy *transitions.*
6. R^I *is a first-order predicate over* Var^{in} *and* $(Var^{in})^\bullet$ *of the form* $\bigwedge v^\bullet \bowtie r$*, where* $v \in Var^{in}, \bowtie \in \{\leq, =, \geq\}, r \in \mathbb{R}$*. It defines the* differential inclusion modeling the evolution of the input signals*.*
7. R^C *is a first-order predicate over* Var *and* $(Var^{loc})^\bullet \cup (Var^{out})^\bullet$ *of the form* $\bigwedge v^\bullet \bowtie e$*, where* $v \in Var^{loc} \cup Var^{out}, \bowtie \in \{\leq, =, \geq\}$*, and* e *is a real linear arithmetic expression over* Var*. It defines the* differential inclusion *modeling the continuous transition relation.*

The discrete update predicate will also often be implicitly defined by a sequence of assignments of the form $v := e$, with $v \in Var^{loc} \cup Var^{out}$ and e an expression over Var. We identify such a sequence of assignments with the predicate relating the pre- and post-states of its sequential execution. For graphical representation, lazy transitions will be labeled in the form $\phi/\mathcal{A}$, and urgent transitions $\uparrow \phi/\mathcal{A}$, with $\mathcal{A}$ either in predicate or assignment notation. If a set of assignments is empty, it will be left out, meaning that all variables in Var remain unchanged upon switching. For a predicate ϕ, $\phi[\mathcal{A}]$ is the result of the substitution induced by $\mathcal{A}$ on ϕ. Discrete variables may be included into hybrid automata according to our definition via an embedding of their value domain into the reals, and associating a derivative of constantly zero to them (locals and outputs). Timeouts are easily coded via explicit local timer variables with a derivative taken from $\{-1, 0, 1\}$.

2.1 Behavior

We now give the formal definition of runs of a hybrid automaton H capturing the evolution of modes and the real-valued variables over time. To this end, we consider the continuous time domain $\mathit{Time} = \mathbb{R}_{\geq 0}$ of non-negative reals, for the mode observable a function $M : \mathit{Time} \to \mathbb{M}$, and for the vector of variables in Var a corresponding function $X : \mathit{Time} \to \mathbb{R}^{|\mathit{Var}|}$, with $X(t) = [X^C(t)^T, X^I(t)^T]^T$, describing for each time point $t \in \mathit{Time}$ the current mode m and the current value of all variables in Var, respectively. Here, X^C covers all variables in $\mathit{Var}^{loc} \cup \mathit{Var}^{out}$ and X^I all variables in Var^{in}. Furthermore, the vector concatenation $\pi(t) = [M(t), X^C(t)^T, X^I(t)^T]^T$ describes the overall (hybrid) state of H. The order of variables in each of the two sub-vectors is fixed, but arbitrary. We identify the state vector $\pi(t)$ with the corresponding valuation of the variables in $M \cup \mathit{Var}$. For simplicity, for a predicate Ψ, we use the notation $\pi(t) \models \Psi$, if the valuation associated with $\pi(t)$ fulfills Ψ. We also define a substitution based on vectors, such that the predicate $\Psi[\mathit{Var}/X(t)]$ is the predicate Ψ with the variable values given by the valuation $X(t)$ substitute the corresponding variables in Var. The vectors $X^C(t)$ and $X^I(t)$ are handled in the same manner. The *time derivative* of $X(t)$ is denoted $dX/dt(t)$ or $X^\bullet(t)$.

Definition 2 (Runs of a Hybrid Automaton). *A* run *of a hybrid automaton*

$$H = (\mathbb{M}, \mathit{Var}^{loc}, \mathit{Var}^{in}, \mathit{Var}^{out}, R^D, R^I, R^C, \Phi, \Theta)$$

corresponding to a sequence of switching times $(\tau_i)_{i\in\mathbb{N}} \in \mathit{Time}^{\mathbb{N}}$ *with*

$$\tau_0 = 0 \wedge \forall i \in \mathbb{N} : \tau_i \leq \tau_{i+1}$$

is a sequence of tuples (π_i)*,*

$$\pi_i = \begin{bmatrix} M_i \\ X_i \end{bmatrix}, \text{ with } X_i = \begin{bmatrix} X_i^C \\ X_i^I \end{bmatrix},$$

where $M_i : [\tau_i, \tau_{i+1}] \to \mathbb{M}$*,* $X_i^C : [\tau_i, \tau_{i+1}] \to \mathbb{R}^n$*, and* $X_i^I : [\tau_i, \tau_{i+1}] \to \mathbb{R}^k$ *are continuously differentiable functions such that*

(1) initial state: $\pi_0(0) \models \Phi$
(2) non-Zeno: $\forall t \in \mathit{Time} \exists i \in \mathbb{N} : t \leq \tau_i$
(3) mode switching times: $\forall i \in \mathbb{N} \forall t \in [\tau_i, \tau_{i+1}) : M_i(t) = M(\tau_i)$
(4) continuous evolution: $\forall i \in \mathbb{N} \forall t \in (\tau_i, \tau_{i+1}) :$
$(dX_i^C/dt(t), X_i(t)) \models R^C(M_i(\tau_i))$
(5) input evolution: $\forall i \in \mathbb{N} \forall t \in \mathit{Time} : (dX_i^I/dt(t), X_i^I(t)) \models R^I$
(6) invariants: $\forall i \in \mathbb{N} \forall t \in \mathit{Time} : X_i(t) \models \Theta(M_i(t))$
(7) urgency: $\forall i \in \mathbb{N} \forall t \in [\tau_i, \tau_{i+1}) \forall (M_i(t), \phi, \mathcal{A}, m') \in R_U^D$ *we have that* $X_i(t) \not\models \phi$*.*

(8) discrete transition firing: $\forall i \in \mathbb{N}:$

$$\begin{aligned}&(M_i(\tau_{i+1}) = M_{i+1}(\tau_{i+1}) \wedge X_i(\tau_{i+1}) = X_{i+1}(\tau_{i+1}))\\ \vee(&\exists(m, \phi, \mathcal{A}, m') \in R^D : M_i(\tau_{i+1}) = m \wedge M_{i+1}(\tau_{i+1}) = m'\\ &\wedge X_i(\tau_{i+1}) \models \phi\\ &\wedge \models \mathcal{A}[Var^{loc\prime} \cup Var^{out\prime}/X^C_{i+1}(\tau_{i+1}), Var/X_i(\tau_{i+1})]\\ &\wedge X^I_i(\tau_{i+1}) = X^I_{i+1}(\tau_{i+1}).\end{aligned}$$

Define $\pi(t)$ *as the* $\pi_i(t)$*, such that* $\forall j > i : \tau_j > t$*, i.e.,* $\pi(t)$ *is the system state after all (possibly superdense) switches that occur at time* t*. Define* $X(t), M(t), X^C(t)$ *and* $X^I(t)$ *in the same manner.*

Clause (1) stipulates that a run must start with an allowed initial state. The time sequence $(\tau_i)_{i\in\mathbb{N}}$ identifies the points in time, at which mode-switches may occur, which is expressed in Clause (3). Only at those points discrete transitions (having a noticeable effect on the state) may be taken. On the other hand, it is not required that any transition fires at some point τ_i, which permits to cover behaviors with a finite number of discrete switches within the framework above. Our simple plant models with only one mode provide examples. As usual, we exclude non-Zeno behavior (in Clause (2)). Clauses (4) and (5) force all variables to actually obey their respective differential inclusions. Clause (6) requires, for each mode, the valuation of continuous variables to meet both local and global invariants while staying in this mode. Clause (7) forces an urgent discrete transition to fire when its trigger condition becomes true. The effect of a discrete transition is described by Clause (8). Whenever a discrete transition is taken, local and output variables may be assigned new values, obtained by evaluating the right-hand side of the respective assignment using the previous value of locals and outputs and the current values of the input. If there is no such assignment, the variable maintains its previous value, which is determined by taking the limit of the trajectory of the variable as t converges to the switching time τ_{i+1}.

Definition 3 (Reach Set). *For some* $t \geq 0$*, define a* time bounded reach set *reach*(H, Φ, t) *of hybrid automaton* H *from predicate* Φ *as the closure of*

$$\{X | \exists \text{ trajectory } X(\cdot) \text{ of } H, t \geq t' \geq 0 : X(0) \models \Phi \wedge X(t') = X\}.$$

Analogously, define the unbounded reach set *reach*(H, Φ) *of hybrid automaton* H *from predicate* Φ *as the closure of*

$$\{X | \exists \text{ trajectory } X(\cdot) \text{ of } H, t' \geq 0 : X(0) \models \Phi \wedge X(t') = X\}.$$

2.2 Parallel Composition

Output variables of H_1 which are at the same time input variables of H_2, and vice versa, establish communication channels with instantaneous communication. Those variables establishing communication channels remain output variables of $H_1 \parallel H_2$. Modes of $H_1 \parallel H_2$ are the pairs of modes of the component automata.

Definition 4 (Parallel Composition). *Let two hybrid automata*

$$H_i = (\mathbb{M}_i, Var_i^{loc}, Var_i^{in}, Var_i^{out}, R_i^D, R_i^I, R_i^C, \Phi_i, \Theta_i),$$

$i = 1, 2$ *be given with* $Var_1^{loc} \cap Var_2^{loc} = \varnothing$. *Then the parallel composition*

$$H_1 \,\|\, H_2 = (\mathbb{M}, Var^{loc}, Var^{in}, Var^{out}, R^D, R^I, R^C, \Phi, \Theta)$$

is given by:

- $\mathbb{M} = \mathbb{M}_1 \times \mathbb{M}_2$,
- $Var^{loc} = Var_1^{loc} \cup Var_2^{loc}$,
- $Var^{out} = Var_1^{out} \cup Var_2^{out}$,
- $Var^{in} = (Var_1^{in} \cup Var_2^{in}) - Var^{out}$,
- $R^C((m_1, m_2)) = R_1^C(m_1) \wedge R_2^C(m_2)$
- $R^I = R_1^I \wedge R_2^I$,
- R_U^D *consists of the following transitions:*
  (1) $((m_1, m_2), \Phi_1, \mathcal{A}_1, (m_1', m_2))$ *for each* $(m_1, \Phi_1, \mathcal{A}_1, m_1') \in R_{U,1}^D$, *and*
  (2) transitions of the form (1) with the role of H_1 *and* H_2 *interchanged,*
- R_L^D *consists of the following transitions:*
  (1) $((m_1, m_2), \Phi_1, \mathcal{A}_1, (m_1', m_2))$ *for each* $(m_1, \Phi_1, \mathcal{A}_1, m_1') \in R_{L,1}^D$, *and*
  (2) transitions of the form (1) with the role of H_1 *and* H_2 *interchanged,*
- $\Phi = \Phi_1 \wedge \Phi_2$,
- $\Theta((m_1, m_2)) = \Theta_1(m_1) \wedge \Theta_2(m_2)$.

3 Hierarchical Controller Design

In this chapter we elaborate our approach towards component based design of hybrid controllers. We use a simplified automatic cruise control application to illustrate our design methodology and the supporting formal definitions. To simplify the exposition, we restrict ourselves to controller design for a fixed, given plant – a generalization of the approach would simply take the reference plant specification as an additional parameter of hybrid interface specifications.

Formally, a plant specification is just a single-mode hybrid automaton extended with specifications of safety and stability conditions of the plant. Its input variables subsume the set of actuators, whose valuations are determined by the controller based on the current observable state of the plant as given by a valuation of its sensors. Note that we allow additional input variables to the plant – these correspond to what is typically called *disturbances*. The rate of change of these can be restricted by associated differential inclusions, while bounds on these can be expressed as invariance properties of the plant model. Stability requirements are expressed in terms of the interface variable of the plant and allow to specify a (convex) stability region subsuming the intended point of equilibrium.

Definition 5 (Plant). *A* plant *is a hybrid automaton*

$$P = (\mathbb{M}_P, \varnothing, Var_P^{in}, Var_P^{out}, R_P^D, R_P^C, R_P^I, \Phi_P, \Theta_P)$$

with

- *an open first-order predicate* φ_P^{safe} *over* $Var_P^{in} \cup Var_P^{out}$ *describing the safety constraints of the plant,*
- *a convex, open first-order predicate* φ_P^{stable} *over* $Var_P^{in} \cup Var_P^{out}$ *describing the system stability condition.*

We also define a set of sensor variables $S \subseteq Var_P^{Out}$ *and a set of actuator variables* $A \subseteq Var_P^{in}$.

Example. For the simple ACC system, we restrict ourselves to a simple plant allowing to directly influence the acceleration a of the car, which is only perturbed through a disturbance s. The velocity v is observable by the controller. To simplify the discussion, we assume a fixed set point $v_{desired}$, and let v denote the difference between the actual velocity v_{actual} and the desired velocity $v_{desired}$. The stability requirement thus specifies, that v should be close to zero, while the plant is considered to be in an unsafe state if the deviation from desired actual velocity exceeds $35m/sec$. Define

$$P = (\{m_p\}, \varnothing, \{a, s\}, \{v\}, \varnothing, R_P^C, R_P^I, true, \Theta_P)$$

with

- $R_P^C(m_P) = (v^\bullet = sa)$
- $R_P^I(m_P) = true$
- $\Theta_P(m_P) = 0.975 \leq s \leq 1.025$,

$\varphi_P^{safe} = (-35 < v < 35)$, and $\varphi_P^{stable} = (-2 < v < 2)$.

Example (cont.) To motivate our approach to component based design of hybrid controllers, consider the design of an ACC controller C controlling the above plant, given by the following specification:

Comp.	Var^{in}	Var^{out}	φ^{assm}	φ^{prom}	Δ^{stable}
C	$\{v\}$	$\{a\}$	$-30 \leq v \leq 30$	$-2 \leq v^\bullet \leq 1.5$	300

The controller is employed in the design setting shown in Figure 1. It is required to drive the plant into its stability region within 300 seconds, for all plants states which deviate from the desired velocity $v_{desired}$ by at most $30m/sec.$, and as long as disturbances are bounded by the plant invariant. We are also looking for an implementation which meets comfort requirements, here simplified to a maximal deceleration of $2m/sec^2$, and a maximal acceleration of $1.5m/sec^2$. Entry conditions shown in Figure 1 will be discussed below.

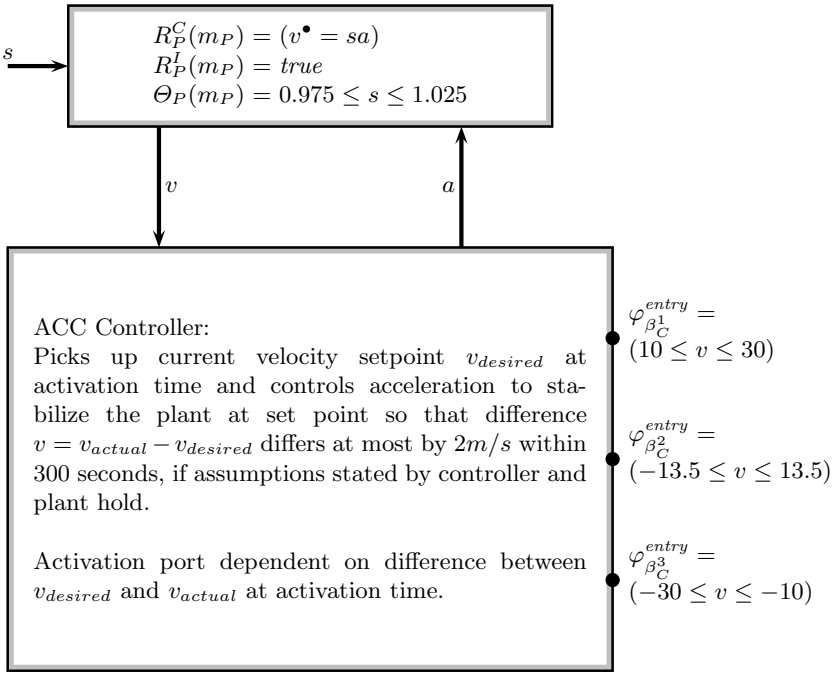


Fig. 1. Controller Design Setting

We envision a future design process for such embedded control applications which is supported by design libraries, whose components encapsulate "standard" control solutions. A designer would then check the hybrid interface specifications for a possible control component supporting the ACC requirement specification shown in Figure 1.

Let us assume that he finds the following component specification of a component named PI.

Comp.	Var^{in}	Var^{out}	φ^{assm}	φ^{prom}	Δ^{stable}
PI	$\{v\}$	$\{a\}$	$-15 \leq v \leq 15$	$-1.4 \leq v^\bullet \leq 1.4$	300

This component is offering a convergence time to the stability region which matches the specifications, although only for a restricted subspace of the plant. It employs an acceleration which matches those of the specification, and thus altogether looks like a good starting point for the controller design, if we can extend the scope of the controllable plant region with some "glue" control.

This takes us to a central point of our design methodology. Since we are addressing, as in AUTOSAR, distributed control applications, where different subcomponents of controllers are running on different electronic control units, we can no longer as in hybrid automata rely on a centralized control structure ensuring mode switching. Instead, in this distributed setting, it becomes the duty of

components to raise an alarm if the dynamics of the plant is evolving in an unforeseen way (such as through differences between the idealized plant model and the actual controlled plant, e.g., unmodeled disturbances). This must happen in time to allow a control component capable of addressing the critical dynamics to take control.

We thus extend our interface concept by allowing the declaration of possibly multiple outports encapsulating possibly different types of dynamics raising *alarms*. Such outport specifications signal the plant state causing the alarm, and provide time-guarantees for maintaining stability and safety for a rescue period, thus providing a time-window in which the switch of control can be initiated. To support a distributed agreement protocol for the selection of the as it were "helper component", there is a persistency requirement on such alarm signals. They also exhibit the plant state at the latest allowable time for control-switch, as determined by the duration of the rescue period and the cost for control switching, which we assume to be given as a design parameter τ.

Example (cont.) For the PI control component, we have two sources of endangering the component's promises in maintaining safety and stability, in that the vehicle could become either significantly slower or faster than the desired speed, as catered for in the following two outport specifications:

Outport	Comp.	$\varphi_\alpha^{alarmOn}$	μ_α	Δ_α	φ_α^{exit}
α_{PI}^1	PI	$v \geq 14$	0.006	0.01	$13.9 \leq v \leq 14.1$
α_{PI}^2	PI	$v \leq -14$	0.006	0.01	$-14.1 \leq v \leq -13.9$

By way of returning to our design scenario, let us assume that the design library offers a component which addresses traffic situations, where the actual velocity is significantly below the desired velocity, a simple acceleration component *ACCELERATE* with a still acceptable constant acceleration, which could recover from such plant states and force the plant into regions allowing a more fine-grained control, as exemplified below.

Comp.	Var^{in}	Var^{out}	φ^{assm}	φ^{prom}	Δ^{stable}
$ACCELERATE$	$\{v\}$	$\{a\}$	$-30 \leq v \leq -5$	$v^\bullet = 1.5$	300

The *ACCELERATE* component raises an alarm if its assumptions are endangered:

Outport	Comp.	$\varphi_\alpha^{alarmOn}$	μ_α	Δ_α	φ_α^{exit}
$\alpha_{ACCELERATE}$	$ACCELERATE$	$v \geq -6$	0.005	0.01	$v \geq -6$

Intuitively, the combination of the *ACCELERATE* component with the *PI* component yields a more robust system, in that safety and stability are now guaranteed for a larger plant region.

We have now motivated most concepts occurring in a hybrid interface specification. The one remaining concept is that of *inports*: these serve to activate

components resp. resume execution of components, under specified entry conditions, which jointly cover the assumptions on plant states for which this component guarantees safety and stability. Each inport specification defines as well a time-window in which it promises to respond to rescue requests arriving at this port.

Example (cont.) The *PI* component offers a single inport Alarm requests are answered within a time window of 0.0025 seconds.

Inport	Comp.	λ_β	φ_β^{entry}
β_{PI}	*PI*	0.0025	$-13.5 \leq v \leq 13.5$

To summarize, a hybrid interface specification for a given plant model P consists of

- a static interface definition of real-valued data interface variables and boolean control variables,
- specification of inports and outports, which in addition to the concepts elaborated in the example also define control signals used for distributed agreement in context-switching, as elaborated below,
- a specification of the plant states for which this component guarantees safety, stability, and promises,
- promises on the rate of change of out-variables,
- a maximal time to convergence to the plant stability region.

Definition 6 (Component Interface Specification). *A* component C *associated with a plant* P *is described by an externally visible interface* $SPEC_C$ *consisting of:*

- Var_C^{in}*, a set of real valued input variables with* $S \subseteq Var_C^{in}$*,*
- Var_C^{out}*, a set of real valued output variables which is disjoint from* Var_C^{in} *and with* $A \subseteq Var_C^{out}$*,*
- $C_C^{in} = \{suspend_C\} \cup \{c_\beta, start_\beta \mid \beta \in A_C^{in}\} \cup \{take_\alpha \mid \alpha \in A_C^{out}\}$*, a set of binary control inputs,*
- $C_C^{out} = \{active_C, fail_C\} \cup \{take_\beta \mid \beta \in A_C^{in}\} \cup \{b_\alpha, switch_\alpha \mid \alpha \in A_C^{out}\}$*, a set of binary control outputs,*
- *a set* A_C^{in} *of incoming ports ("inports") given as tuples*

$$\beta = (c_\beta, \lambda_\beta, take_\beta, start_\beta, \varphi_\beta^{entry}),$$

  where $c_\beta \in C_C^{in}$*,* $\lambda_\beta > 0$*,* $take_\beta \in C_C^{out}$*,* $start_\beta \in C_C^{in}$*, and* φ_β^{entry} *is a first-order predicate over* $Var_C^{in} \cup Var_C^{out}$*,*
- *a set* A_C^{out} *of outgoing ports ("outports") given as tuples*

$$\alpha = (b_\alpha, \varphi_\alpha^{alarm}, \mu_\alpha, \Delta_\alpha, take_\alpha, switch_\alpha, \varphi_\alpha^{exit}),$$

  where $b_\alpha \in C_C^{out}$*,* φ_α^{alarm} *is a closed first-order predicate over* $Var_C^{in} \cup Var_C^{out}$*,* $\mu_\alpha > 0$*,* $\Delta_\alpha > 0$*,* $take_\alpha \in C_C^{in}$*,* $switch_\alpha \in C_C^{out}$*, and* φ_α^{exit} *is a first-order predicate over* $Var_C^{in} \cup Var_C^{out}$*,*

- φ_C^{prom}, *a first-order predicate over* $Var_C^{in\bullet} \cup Var_C^{out\bullet} \cup Var_C^{in} \cup Var_C^{out}$,
- φ_C^{assm}, *a first-order predicate over* $Var_C^{in} \cup Var_C^{out}$,
- $\Delta_C^{stable} > 0$ *is a time after which the system is required to converge to* φ_P^{stable}.

In the rest of this paper we use φ_C^{entry} to abbreviate $\bigvee_{\beta \in A_C^{in}} \varphi_\beta^{entry}$.

Definition 7 (Switch Time). *Define a global variable* $0 < \tau \in \mathbb{R}$*, representing the maximum time needed for a component switch.*

We will now discuss how to build controllers hierarchically, and use the running ACC controller design to illustrate the key underlying concepts and issues.

Example (cont.)

Comp.	Var^{in}	Var^{out}	φ^{assm}	φ^{prom}	Δ^{stable}
PI	$\{v\}$	$\{a\}$	$-15 \leq v \leq 15$	$-1.4 \leq v^\bullet \leq 1.4$	300
$ACCELERATE$	$\{v\}$	$\{a\}$	$-30 \leq v \leq -5$	$v^\bullet = 1.5$	300

Outport	Comp.	$\varphi_\alpha^{alarmOn}$	μ_α	Δ_α	φ_α^{exit}
α_{PI}^1	PI	$v \geq 14$	0.006	0.01	$13.9 \leq v \leq 14.1$
α_{PI}^2	PI	$v \leq -14$	0.006	0.01	$-14.1 \leq v \leq -13.9$
$\alpha_{ACCELERATE}$	$ACCELERATE$	$v \geq -6$	0.005	0.01	$v \geq -6$

Inport	Comp.	λ_β	φ_β^{entry}
β_{PI}	PI	0.0025	$-13.5 \leq v \leq 13.5$
$\beta_{ACCELERATE}$	$ACCELERATE$	0.0025	$-30 \leq v \leq -10$

We compose the PI and $ACCELERATE$ components as follows:

- an alarm raised by $ACCELERATE$ is forwarded to the inport of the PI component,
- an alarm raised by the PI component of type "actual speed is too slow to be handled by PI component" is forwarded to the inport of the PI component.

Note that such a composition leaves unspecified, how to handle plant dynamics, in which the current speed is much too fast in relation to the desired speed for the PI component to maintain stability. In general, the composition of two components then yields a composed component, whose outports must cater for alarms which are not handled internally. Figure 2 gives an informal description of the composition of the PI component and the $ACCELERATE$ component.

We can now perform a what in industrial design processes is typically referred to as a *virtual integration test*:

- Is the plant state at context switch time as given by the exit predicate of the outport compatible with the plant state required by the connected inport?
- Does the inport have sufficient time to take a decision on whether it is willing to accept the alarm? To this end we compare the promised minimal stability period μ_α of outport, as expressed by the λ_β which provides an upper bound for the inport to reply to incoming alarms.

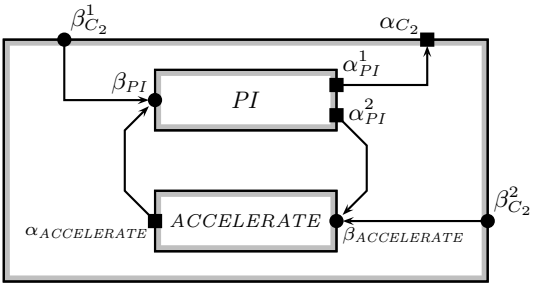


Fig. 2. Interconnection of *ACCELERATE* and *PI*

Having successfully carried out the virtual integration test, we can now either derive a component interface specification for the composed system C_2, by propagating information derived from local specifications towards the boundary of the system, or check, whether an a priori given specification of the composed system C_2 is derivable from local specifications. This reasoning will be formalized in Section 4 on verification of hierarchical component based controller designs. The key property conspicuous already in our simple example is, that this reasoning is completely independent of the actual implementation of the subsystems. Indeed, the *PI* component might itself be composed of several subsystems, or be given as a what we call *basic* component: both the virtual integration test as well as compliance of the composed system to an interface specification of the composed system is purely based on component interface specifications. Industrial jargon uses the term *grey box view* to refer to schematics of a composed system as in Figure 2, where only the interface specifications of subsystem as well as their interconnection via ports are known. In contrast, a *white box view* of a composed system would also make visible the internal realization of the subsystems, across all levels of hierarchy. We will use a white box view on composed systems to define their semantics, and a grey box view in Section 4 for compositional verification of such systems.

As in the ACC example, components of a composed system offer different capabilities in establishing safety and stability requirements of one and the same plant. The formal definition of what we call the transition composition of components exhibits the following aspects.

- Alarms may be either be handled locally of forwarded to a yet unknown environment.
- We allow for multiple helpers, and offer guards on port connections to filter propagation of alarms.
- On the path to local helpers, interface variables of the entered component may be updated (thus motivating the usage of term *transition composition*).
- Statically it must be possible to have at least one feasible path to ask for help: the disjunction of all guard conditions related to a single outport must be a tautology.

In Def. 6 we defined the externally visible interface of a component C. Components may also use *local* variables internally and we denote the set of these variables with Var_C^{loc}. In the following definition we use $C(\alpha)$ (resp. $C(\beta)$) to denote the component C the port α (resp. β) belongs to, i.e. $\alpha \in A_C^{out}$ (resp. $\beta \in A_C^{in}$).

Definition 8 (Transition Composition of Components). *Let C be a component and let $\{C_1, \dots C_n\}$ be a finite set of basic or composed components with*

- $Var_C^{in} = Var_{C_i}^{in}$ *for all* i,
- $Var_C^{out} = Var_{C_i}^{out}$ *for all* i,
- *all* $A_{C_i}^{out}, A_{C_i}^{in}$, *as well as* A_C^{out} *and* A_C^{in} *are disjoint.*

We define $Var_C^{loc} := \bigcup_i Var_{C_i}^{loc}$ *and call* C *a* transition composition *of the components* $C_1, \dots C_n$ *with* port connection $(\mathcal{P}, \mathcal{Q})$ *(we use* $\mathcal{S}_{(\mathcal{P},\mathcal{Q})}(C_1, \dots, C_n)$ *to denote it) iff*

(a) $\mathcal{P}$ *is given as a set of tuples describing* transitions *of the form*

$$(\alpha, \{(\beta_1, g_1, \mathcal{A}_1), \dots, (\beta_k, g_k, \mathcal{A}_k)\}, \{(\alpha_1, g_1'), \dots, (\alpha_l, g_l')\}),$$

with
- $\alpha \in \bigcup_i A_{C_i}^{out}$,
- $k, l \geq 0, k + l > 0$,
- $\beta_i \in \bigcup_i A_{C_i}^{in}$ *with* $\beta_i \neq \beta_j$ *for all* $i \neq j \in \{1, \dots, k\}$,
- $\alpha_i \in A_C^{out}$ *with* $\alpha_i \neq \alpha_j$ *for all* $i \neq j \in \{1, \dots, l\}$,
- g_i *and* g_i' *are first-order predicates over* $Var_C^{in} \cup Var_C^{loc} \cup Var_C^{out}$, *such that for each tuple* $\bigvee_i g_i \vee \bigvee_i g_i'$ *holds,*
- *for all* $(\alpha, \{(\beta_1, g_1, \mathcal{A}_1), \dots, (\beta_k, g_k, \mathcal{A}_k)\}, \{(\alpha_1, g_1'), \dots, (\alpha_l, g_l')\}) \in \mathcal{P}$, α *belongs to a different component than all* β_i *(no loops,* $C(\alpha) \neq C(\beta_i)$*),*
- $\mathcal{A}_i$ *is a set of assignments for* $Var_{C(\alpha_i)}^{in} \setminus S$ *depending on* $Var_{C(\alpha)}^{in} \cup Var_{C(\alpha)}^{out}$,
- *for all* α, *there exists exactly one tuple* $(\alpha, S_1, S_2) \in \mathcal{P}$ *(each outgoing alarm is connected to exactly one family of incoming alarms).*

(b) The second component $\mathcal{Q}$ *of the port connection is a totally defined mapping from* A_C^{in} *to* $\bigcup_i A_{C_i}^{in}$.

This definition connects the components $C_1, \dots, C_n$ that have equal Var^{in} and Var^{out} and the result is a component C. The composition C and its components $C_1, \dots, C_n$ carry control in- and outputs that are connected appropriately in the above definition. A tuple

$$p = (\alpha, \{(\beta_1, g_1, \mathcal{A}_1), \dots, (\beta_k, g_k, \mathcal{A}_k)\}, \{(\alpha_1, g_1'), \dots, (\alpha_l, g_l')\}),$$

connects an outport α of a component C_i with some inports β_i of the other components and with some outports α_j of the composition C. The idea is that the outgoing signal of α is forwarded to the β_i's and α_j's such that each receiver of this signal is able to respond appropriately and to take over. However, this action

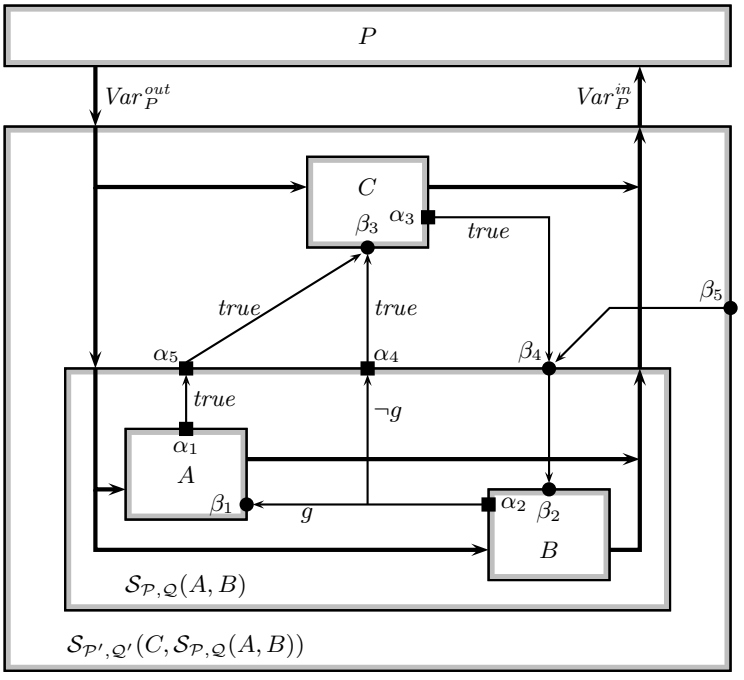
requires that the corresponding entry condition of the activated component is met. Hence, the composition must be able to ensure that and therefore p adds a guard for each receiving port. Moreover, it adds assignments to each connection between α and an inport β_i of another component. This allows for setting non-sensor variables as required. The above definition lists sanity conditions for a port connection p. There must be at least one receiver, no inport is used twice as receiver, no outport is used twice as receiver, there is always at least one of the guards satisfied, and there are no loops, i.e., no component receives its own outgoing signal. All these tuples are collected in the set $\mathcal{P}$ and for each outport that appears in $C_1, \ldots, C_n$ we have exactly one $p \in \mathcal{P}$ describing the receivers of this signal.

The inports of the composition are receiving signals and therefore this information must be forwarded to some inports of its components. In the above definition this is done by an one-to-one mapping $\mathcal{Q}$. In Fig. 3 an example for transition composition is given. It consists of a composition $\mathcal{S}_{\mathcal{P},\mathcal{Q}}(A, B)$ of A and B which is then composed with C.

We now return to the role of the control signals in component interface specifications. Consider the transition composition of components A, B, and C in Figure 3, and assume a distributed implementation, where A and B are allocated on ecu_1, and C is allocated on ecu_2, and consider an alarm raised by component B at outport α_2. We will now informally describe a distributed agreement protocol, ensuring that all ECUs agree on the component to handle the alarm, called *distributed identification of helpers*. The challenge arises from the distributed execution, and the possibility of multiple helpers. Much as in distributed cache coherency protocols, we need to enforce a serialization point so as to avoid race conditions leading to inconsistent states, where say both A and C believe to be the chosen helper. We describe the protocol informally using the sequence chart given in Figure 4. Formally, we will define with each component wrapper automata jointly implementing the protocol. Such wrapper automata are constructed both for outports – such as $H_3(B)$ – , for port connections – such as $H_{\mathcal{P}}$ and $H_{\mathcal{P}'}$ – , and for inports, such as $H_{\beta_1}(A)$ and $H_{\beta_2}(C)$ in Figure 4.

The protocol is initiated from the basic component B: it raises an alarm at its outport α_2, which causes the control signal b_{α_2} associated with this outport to be generated. There are two levels of hierarchy enclosing component B, each specifying through port connections potential helper components. From the inner hierarchy, we see that either the alarm can be handled locally – this is reflected by requesting help from import β_1 of component A through generation of the control signal c_{β_1} associated with this inport, or externally, which is represented by generating an alarm at the outport α_4 of the composed system, represented by setting its control signal to 1. This is handled by the wrapper automaton $H_{\mathcal{P}}$ for this port connection running on ecu_1.

We now have two independent message flows – in which the generated control signals are propagated either locally within ecu_1 or externally to ecu_2 (typically with different arrival times). The local flow will lead to a response of component A through its inport β_1 to be ready for taking over, as indicated by setting the

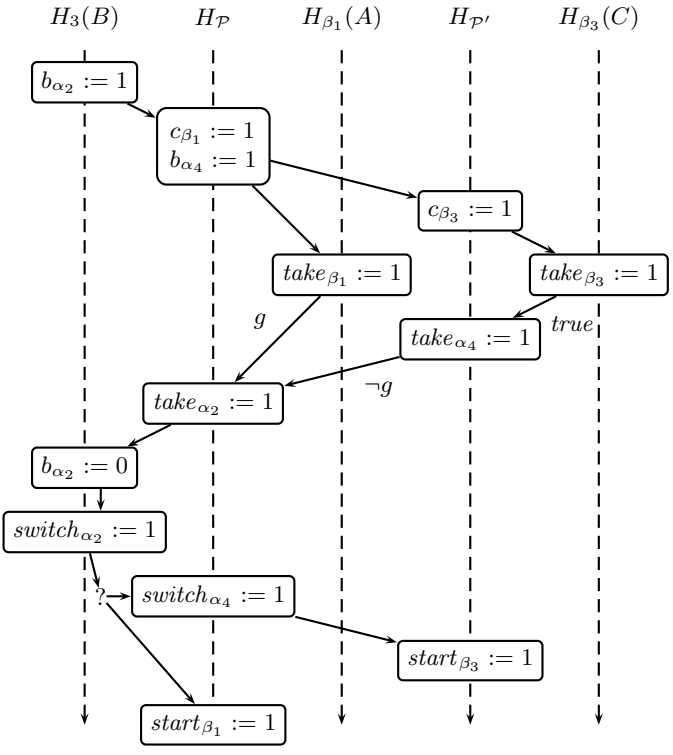


Bottom: $\mathcal{S}_{\mathcal{P},\mathcal{Q}}(A,B)$ with $\mathcal{P} = \left\{ \begin{array}{l} (\alpha_1, \varnothing, \{(\alpha_5, true)\}), \\ (\alpha_2, \{(\beta_1, g)\}, \{(\alpha_4, \neg g)\}) \end{array} \right\}$ and $\mathcal{Q}(\beta_4) = \beta_2$.

Overall: $\mathcal{S}_{\mathcal{P}',\mathcal{Q}'}(C,\mathcal{S}_{\mathcal{P},\mathcal{Q}}(A,B))$ with $\mathcal{P}' = \left\{ \begin{array}{l} (\alpha_3, \{(\beta_4, true, \varnothing)\}, \varnothing), \\ (\alpha_4, \{(\beta_3, true, \varnothing)\}, \varnothing), \\ (\alpha_5, \{(\beta_3, true, \varnothing)\}, \varnothing) \end{array} \right\}$ and $\mathcal{Q}'(\beta_5) = \beta_4$.

Fig. 3. An example for a composition

associated control signal $take_{\beta_1}$. The external flow will lead to the generation of a help request (control signal c_{β_3}) to the inport of component C, generated by a wrapper automaton interpreting the port connection $\mathcal{P}'$ of the outer hierarchy level. Component C will indicate its readiness to accept the alarm by setting the control variable $take_{\beta_3}$. Outports of composed components act as a proxy for the environment, in that the existence of at least one helper component in the environment of the sequential composition of A and B, such as component C, is then communicated internally, with the outport behaving on behalf of an inport of such an environment component. Thus outport α_4 generates $take_{\alpha_4}$ in response to receiving $take_{\beta_3}$. The protocol automaton $H_{\mathcal{P}}$ associated with port connection $\mathcal{P}$ will for each offer for help consult guards of port connections, and only register this offer for help, if the guard condition is true. Note that this condition might change dynamically; $H_{\mathcal{P}}$ thus constantly monitors guard conditions associated with registered helpers, and removes registration if associated guards become false. The protocol will ensure that helpers maintain their readiness for help, until finally a helper is selected. This selection occurs as follows. As soon as



The diagram represents the sequence of signals that are set as a consequence of setting b_{α_2}. In this example there is a race between the inport β_1 and β_3. As soon as a *take* is observed by $H_{\mathcal{P}}$, the automaton forwards this to B. When the *switch* signal is set, the automaton $H_{\mathcal{P}}$ decides by considering the guards who takes over.

Fig. 4. Sequence of communications when b_{α_2} is set

there is at least one registered helper, $H_{\mathcal{P}}$ signals this information to the outport α_2 which raised the alarm, by setting its in-control signal $take_{\alpha_2}$. If this signal arrives prior to the expiration of the rescue period, taking into account the time needed for context switching, the context switch is initiated by the declaration on part of B to now delegate control to *some* helper, through setting $switch_{\alpha_2}$. $H_{\mathcal{P}}$, the single point of serialization in this protocol, consults upon receiving $switch_{\alpha_2}$ the list of registered helpers, and nondeterministically picks one of these. In the example above, depending on the guard condition g at helper selection time, this can either be the local helper A, in which case it passes control to helper A by setting its in-control-signal $start_{\beta_1}$, or some external helper, in which case $H_{\mathcal{P}}$ delegates control to the helper in the environment of the sequential composition of A and B by setting $switch_{\alpha_4}$. In the former case, the environment would be informed, that a helper has been found by resetting b_{α_4}, which in turn would cause component C to withdraw its offer. In the latter case, b_{α_2} is withdrawn,

and any local helper such as B will withdraw its offer. We note that watchdogs are employed in various places in the protocol to monitor the deadline for context switching, as defined by the rescue period of the alarm raising outport and the time needed for context switching. In case such a watchdog expires, a failure state is reached.

We now complement the definition of hybrid component interfaces and transition composition of components with three steps addressing in particular their semantic foundation. We will first show how to systematically derive basic components from hybrid automata and define the induced semantics of basic components. We then turn to the semantics of transition composition, which is derived inductively from the semantics of its components and automata implementing the distributed helper agreement protocol. Finally, we define the satisfaction relation between component implementations and hybrid interface specifications. The section is wrapped up by completing the construction of the automatic cruise controller.

Definition 9 (Hybrid Automaton for a Basic Component). *Let*

$$H = (\mathbb{M}, Var^{loc}, Var^{in}, Var^{out}, R^D, R^C, R^I, \Phi, \Theta)$$

be a hybrid automaton with disjoint Var^{loc}, Var^{in}, and Var^{out}. Define

- *for each incoming port β in A^{in}, a first-order predicate Φ_β on Var^{loc}, describing admissible initial values for the local variables when $start_\beta$ is received,*
- *for each incoming port β in A^{in}, an initial mode $m_\beta \in \mathbb{M}$.*

We call H an admissible hybrid automaton for a basic component C *associated with a plant P if and only if*

- $Var_C^{in} = Var^{in}$,
- $Var_C^{out} = Var^{out}$,
- $\Phi \implies \bigvee_{\beta \in A_C^{in}} \Phi_\beta$, *and*
- $\varphi_C^{assm} \implies \bigwedge_{m \in \mathbb{M}} \Theta_C(m)$

In this case we use H_C to denote the hybrid automaton H of the basic component C.

Now consider an admissible hybrid automaton for a given interface specification. From an implementation perspective, we now wrap the code generated from this hybrid automaton with code supporting suspension and activation, an automaton providing a single point of serialization for all outports, and automata implementing the distributed agreement protocol requirements for inports. Cast as a formal definition of the semantics of basic components, this can be formalized as a parallel composition of a relaxation of the hybrid automata providing the component implementation, and as timed automata implementing the distributed agreement protocol. This relaxation of the admissible hybrid automaton enforces no constraints on the input and output variables whenever the component is inactive.

Definition 10 (Semantics of a Basic Component). *Let*

$$H = (\mathbb{M}, Var^{loc}, Var^{in}, Var^{out}, R^D, R^C, R^I, \Phi, \Theta)$$

be a hybrid automaton, and C a basic component such that H is admissible for C. The semantics $[\![C]\!]$ of C is the parallel composition of hybrid automata

$$I_C = H_1 \parallel H_2 \parallel H_3 \parallel \left(\|_{\beta \in A_C^{in}} H_\beta \right),$$

which are defined as follows.

- *H_1 is a hybrid automaton that is basically H but augmented with a mode m_{inact} for inactivity and a mode m_{failed} for failures:*

$$H_1 = (\mathbb{M} \cup \{m_{inact}, m_{failed}\}, Var^{loc}, Var^{in}_{H_1}, Var^{out} \cup \{fail_C\}, R^D_{H_1}, R^C_{H_1}, true, \Phi \wedge M = m_{inact}, \Theta_{H_1}),$$

  where

  - $m_{inact}, m_{failed} \notin \mathbb{M}$,
  - $Var^{in}_{H_1} = Var^{in} \cup \{active_C\} \cup \{start_\beta \mid \beta \in A^{in}_C\}$,
  -

$$R^D_{U,H_1} = R^D_U \cup \{(m, active_C = 0, \varnothing, m_{inact}) \mid m \in \mathbb{M}\} \quad (1)$$

$$\cup \{(m_{inact}, active_C = 1 \wedge start_\beta \wedge \varphi^{entry}_\beta, \Phi'_\beta \wedge \bigwedge_{v \in Var^{out}} (v' = v), m_\beta) \mid \beta \in A^{in}_C\}, \quad (2)$$

$$\cup \{(m_{inact}, active_C = 1 \wedge start_\beta \wedge \neg\varphi^{entry}_\beta, fail_C := 1, m_{failed}) \mid \beta \in A^{in}_C\}, \quad (3)$$

    where Φ'_β is Φ_β with every variable primed,

  - $R^D_{L,H_1} = R^D_L$
  - $R^C_{H_1}(m) = R^C(m)$, *if* $m \in \mathbb{M}$, *and* $R^C_{H_1}(m_{inact}) = true$,
  - $\Theta_{H_1}(m) = \Theta(m)$, *if* $m \in \mathbb{M}$, *and* $\Theta_{H_1}(m_{inact}) = true$,

- *Automaton H_2 ensures that the variables $active_C$ and $fail_C$ are only changed by discrete transitions (Fig. 5). It also reacts to $suspend_C$ signals by deactivating the component.*
- *The hybrid automaton H_3 handles all outgoing ports. It is defined as follows:*

$$H_3 = (\mathbb{M}_3, V_3^{loc}, V_3^{in}, V_3^{out}, R_3^D, R_3^C, R_3^I, \Phi_3, \Theta_3)$$

with

$$\mathbb{M}_3 = \left(2^{A_C^{out}} \times 2^{A_C^{out}}\right) \cup \{Fail\}, \tag{4}$$

$$V_3^{loc} = \{t_\alpha \mid \alpha \in A_C^{out}\},$$

$$V_3^{in} = \{active_C\} \cup Var_C^{in} \cup Var_C^{out} \cup \{take_\alpha \mid \alpha \in A_C^{out}\},$$

$$V_3^{out} = \{fail_C\} \cup \{b_\alpha, switch_\alpha \mid \alpha \in A_C^{out}\},$$

$$R_{U,3}^D = \{((X,Y), \varphi_\alpha^{alarm} \wedge active_C, b_\alpha := 1, t_\alpha := 0, \tag{5}$$
$$(X \cup \{\alpha\}, Y)) \mid \alpha \notin X \cup Y\}$$
$$//Alarm\ \alpha\ is\ set$$
$$\cup \{((X,Y), t_\alpha \geq \alpha, fail_C := 1, Fail) \mid \alpha \in X\} \tag{6}$$
$$// Failure:\ alarm\ \alpha\ has\ exceeded\ the\ maximum\ duration$$
$$\cup \{((X,Y), take_\alpha \wedge t_\alpha < \Delta_\alpha - \tau, b_\alpha := 0, t_\alpha := 0, \tag{7}$$
$$(X \setminus \{\alpha\}, Y \cup \{\alpha\})) \mid \alpha \in X \setminus Y\}$$
$$//Alarm\ \alpha\ is\ reset\ because\ of\ a\ take$$
$$\cup \{((X,Y), t_\alpha = 0, \tag{8}$$
$$switch_\alpha := 1, \forall v \in V_3^{out} \cup V_3^{loc} \setminus \{switch_\alpha\} : v := 0,$$
$$(\varnothing, \varnothing)) \mid \alpha \in Y\}$$
$$//A\ switch_\alpha\ is\ set\ because\ of\ a\ previous\ take_\alpha$$

$$R_{L,3}^D = \{((X,Y), \mu_\alpha \leq t_\alpha < \Delta_\alpha \wedge \neg\varphi_\alpha^{alarm}, b_\alpha := 0, t_\alpha := 0, \tag{9}$$
$$(X \setminus \{\alpha\}, Y)) \mid \alpha \in X\}$$
$$//Alarm\ \alpha\ can\ be\ reset$$

$$R_3^C(v) = 0\ for\ all\ v \in V_3^{out}$$

$$R_3^C(v) = 1\ for\ all\ v \in V_3^{loc}$$

$$R_3^I = true$$

$$\Phi_3 = (\mathbb{M} = (\varnothing, \varnothing) \wedge \bigwedge_{v \in V_3^{out} \cup V_3^{loc}} v = 0)$$

$$\Theta_3 = true \tag{10}$$

– *The automaton H_β for an incoming port $\beta = (c_\beta, \lambda_\beta, take_\beta, start_\beta) \in A_C^{in}$ is given in Fig. 6.*

On H_1 of the semantics of a basic component: The purpose of H_1 is to add the intended control strategy of H into the semantics. However, we have to augment this by additional locations and some transitions in order to satisfy $SPEC_C$. The automaton H_1 behaves like H except for the case that $active_C$ becomes false. In this case H_1 switches by (1) from an arbitrary location to the additional location m_{inact} which stands for inactivity. As soon as H_1 discovers a rising edge of $active_C$ together with $start_\beta$ the automaton fires transition (2) because it knows that is activated again via the incoming port β. The assignment that

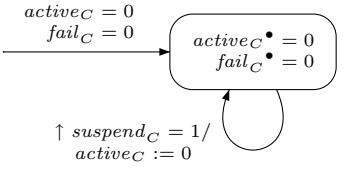


Fig. 5. Automaton H_2

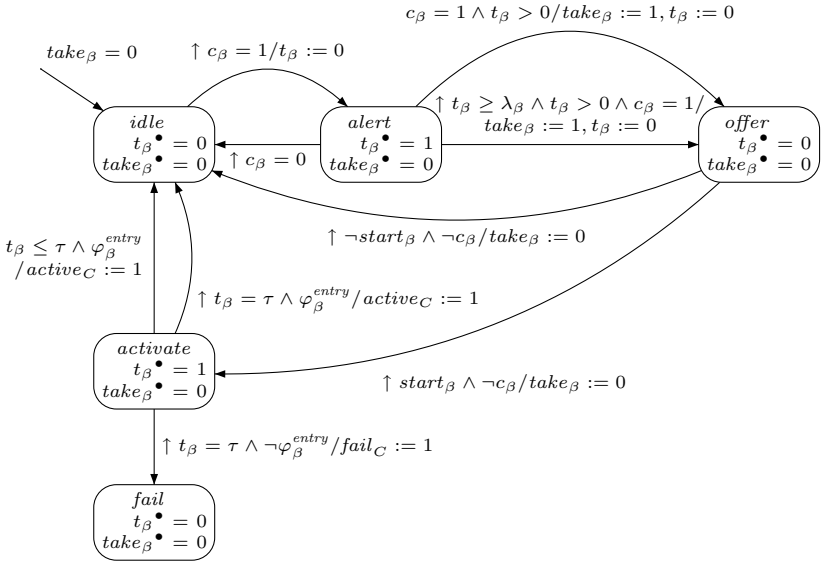


Fig. 6. Automaton H_β

belongs to this transition keeps all variables of Var_C^{out} stable but executes the assignments Φ_β given by the definition of β when switching to m_β. Note that both Φ_β and m_β were given in Def. 9. However, if the activation takes place while the entry condition φ_β^{entry} is *not* given, then H_1 switches to a location m_{failed} and sets $fail_C$.

On H_3 of the semantics of a basic component: The purpose of H_3 is to manage the outgoing alarm signals b_α. To this end H_3 has locations defined in (4) of the form (X, Y) with $X, Y \in 2^{A_C^{out}}$ with only one exception needed for failures. The idea of a location (X, Y) is that for all $\alpha \in X$ the corresponding signal b_α is currently set. If $\alpha \in Y$ then this means that the component has observed a rising edge of $take_\alpha$. The transitions are defined to keep these invariants. In (5) the signal b_α is set as soon as φ_α^{alarm} is satisfied while C is active. With this transition a timer t_α is reset to observe whether a $take_\alpha$ is set in time or φ_α^{alarm} becomes false in time. If this is not the case, then the transitions in (6)

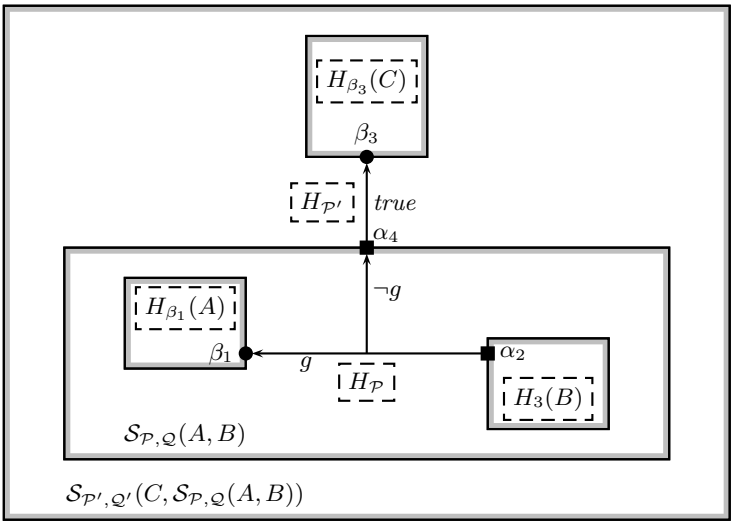


Fig. 7. Semantics: Automata involved for α_2

will switch to the exceptional failure state and will set $fail_C$. The transitions (7) are fired provided that a $take_\alpha$ signal arrives in time, i.e. it arrives τ time units before the component is signaling b_α for Δ_α time units. This transition moves α from the X set into the Y set. For an α being in Y the automaton has to react within 0 time units by setting $switch_\alpha$ as given in (8). If b_α is set for at least μ_α time units the automaton is able to reset this signal provided that φ_α^{alarm} is not satisfied anymore (9). Note that these transitions are not urgent, hence the automaton is not forced to do this if the transition is enabled.

On H_β of the semantics of a basic component: The purpose of H_β is to handle incoming requests at inport β. Initially it waits in *idle* until a rising edge of c_β is observed. This enforces a transition to *alert* where the automaton might remain for a while. Due to the timer t_β and the urgent transition with guard $t_\beta \geq \lambda_\beta$ the automaton sets $take_\beta$ after at most λ_β time units and switches to state *offer*. A prerequisite is that the c_β is still set, otherwise H_β moves back to *idle*. In *offer* it waits for a falling edge of c_β. If this happens with $start_\beta$ being set, the automaton switches to *activate* and activates the component after at most τ seconds. Otherwise, i.e. $start_\beta$ is not set, the automaton proceeds to *idle* without activating the component. In case that the activation must take place (τ seconds elapsed) and the entry condition φ_C^{entry} is not true, the automaton sets *fail*.

We can now define the semantics of the transition composition of components $C_1, \ldots, C_n$, where in taking a white box view we assume as given the semantics $[\![C_j]\!]$ of the components. The semantics is defined in terms of the parallel composition of the hybrid automata representing the semantics of its components, and three timed automata which define activation and failure of the composed sys-

tems in terms of the status of its components, interpret the connection of inports of the composed system to local inports in propagating control signals outside-in, and interpret all port connections for all local outports to implement the distributed helper identification protocol. Note that control signals ensure that there is always at most one active component inside a transition composition of components.

Definition 11 (Semantics of a Transition Composition). *Let C be a component obtained by a transition composition of the components $C_1, \ldots C_n$ with port connection $(\mathcal{P}, \mathcal{Q})$. The semantics $[\![C]\!]$ of C is the parallel composition of hybrid automata*

$$I_C = \left(\Big|\Big|_i [\![C_i]\!] \right) \parallel H_T \parallel H_{\mathcal{P}} \parallel H_{\mathcal{Q}}$$

with the following components.

- *H_T is a hybrid automaton that handles the reactions to $suspend_C$, $start_\beta$, and $Fail_{C_i}$ appropriately:*

$$\begin{aligned}
\mathbb{M}_T =& \{inactive, active, failed\} \\
V_T^{loc} =& \varnothing, \\
V_T^{in} =& \{start_\beta \mid \beta \in A_C^{in}\} \cup \{active_{C_i}, fail_{C_i} \mid i \in \{1, \ldots, n\}\} \\
& \cup \{suspend_C\} \\
V_T^{out} =& \{suspend_{C_i} \mid i \in \{1, \ldots, n\}\} \cup \{active_C, fail_C\} \\
R_{U,T}^D =& \{(m, fail_{C_i}, fail_C := 1, failed) \mid i \in \{1, \ldots, n\}, m \in \mathbb{M}_T\} \\
& \quad // \text{ A component failed, hence the composition fails.} \\
& \cup \{(m, suspend_C, \\
& \qquad active_C := 0, suspend_{C_1} := 1, \ldots, suspend_{C_n} := 1, \\
& \qquad inactive) \mid m \in \mathbb{M}_T \setminus \{failed\}\} \\
& \quad // \text{ Suspend is forwarded to all components.} \\
& \cup \{(inactive, \neg suspend_C, \\
& \qquad suspend_{C_1} := 0, \ldots, suspend_{C_n} := 0, inactive)\} \\
& \quad // \text{ Reset of suspend is forwarded to all components.} \\
& \cup \{(inactive, start_\beta \wedge \varphi_\beta^{entry}, active_C := 1, active)\} \\
& \quad // \text{ A } start_\beta \text{ activates the composition } C. \\
& \cup \{(inactive, start_\beta \wedge \neg\varphi_\beta^{entry}, fail_C := 1, failed)\} \\
& \quad // \text{ A } start_\beta \text{ without meeting the entry condition.} \\
& \cup \{(active, active_{C_i}, \\
& \qquad suspend_{C_1} := 0, \ldots, suspend_{C_{i-1}} := 0, suspend_{C_{i+1}} := 0, \\
& \qquad \ldots, suspendC_n := 0, active)\} \\
& \quad // \text{ When a } C_i \text{ becomes active, all others are suspended.}
\end{aligned}$$

$$
\begin{aligned}
R^D_{L,T} =& \varnothing \\
R^C_T(v) =& 0 \text{ for all } v \in V^{out}_T \cup V^{loc}_T \\
R^I_T =& true \\
\Phi_T =& (\mathbb{M} = inactive \wedge \bigwedge_{v \in V^{out}_T \cup V^{loc}_T} v = 0) \\
\Theta_T =& true
\end{aligned}
$$

– $H_{\mathcal{Q}}$ *is a hybrid automaton that implements the invariant*

$$
\begin{aligned}
\forall \beta \in A^{in}_C : (c_\beta &\longleftrightarrow c_{\mathcal{Q}(\beta)}) \\
\wedge (take_{\mathcal{Q}(\beta)} &\longleftrightarrow take_\beta) \\
\wedge (start_\beta &\longleftrightarrow start_{\mathcal{Q}(\beta)})
\end{aligned}
$$

– *Let* $A^{src} = \bigcup_i A^{out}_{C_i}$ *and* $A^{dst} = \bigcup_i A^{in}_{C_i}$. *We set*

$$H_{\mathcal{P}} = (\mathbb{M}_{\mathcal{P}}, V^{loc}_{\mathcal{P}}, V^{in}_{\mathcal{P}}, V^{out}_{\mathcal{P}}, R^D_{\mathcal{P}}, R^C_{\mathcal{P}}, R^I_{\mathcal{P}}, \Phi_{\mathcal{P}}, \Theta_{\mathcal{P}})$$

with

$$
\begin{aligned}
\mathbb{M}_{\mathcal{P}} =& 2^{A^{src}} \times 2^{A^{src} \times (A^{dst} \cup A^{out}_C)}, \\
V^{loc}_{\mathcal{P}} =& \varnothing,
\end{aligned}
$$

$$
\begin{aligned}
V^{in}_{\mathcal{P}} =& Var^{in}_C \cup Var^{out}_C \cup \{b_\alpha, switch_\alpha \mid \alpha \in A^{src}\} \\
& \cup \{take_\beta \mid \beta \in A^{dst}\} \cup \{take_\alpha \mid \alpha \in A^{out}_C\}, \\
V^{out}_{\mathcal{P}} =& \{c_\beta, start_\beta \mid \beta \in A^{dst}\} \cup \{b_\alpha, switch_\alpha \mid \alpha \in A^{out}_C\} \\
& \cup \{take_\alpha \mid \alpha \in A^{src}\}, \\
R^D_{U,\mathcal{P}} =& \bigcup_{p \in \mathcal{P}} R^D_{U,\mathcal{P}}(p) \quad \text{where } R^D_{U,\mathcal{P}}(p) \text{ is defined in Fig. 8} \\
R^D_{L,\mathcal{P}} =& \varnothing \\
R^C_{\mathcal{P}}(v) =& 0 \text{ for all } v \in V^{out}_{\mathcal{P}} \cup V^{loc}_{\mathcal{P}} \\
R^I_{\mathcal{P}} =& true \\
\Phi_{\mathcal{P}} =& (\mathbb{M} = (\varnothing, \varnothing) \wedge \bigwedge_{v \in V^{out}_{\mathcal{P}} \cup V^{loc}_{\mathcal{P}}} v = 0) \\
\Theta_{\mathcal{P}} =& true
\end{aligned}
$$

In summary, the semantics of a transition composition is the parallel composition of semantics of its components together with three additional hybrid automata H_T, $H_{\mathcal{P}}$ and $H_{\mathcal{Q}}$. The automaton H_T implements the following properties:

– Whenever a $fail_{C_i}$ of a component occurs this leads to a $fail_C$ signal. There is no way to reset $fail_C$.

- Whenever a $start_\beta$ signal is given then the composition is activated provided that the entry condition φ_β^{entry} is met. Otherwise H_C produces a $fail_C$ signal.
- It observes $active_{C_i}$ signals of the components and provide $suspend_{C_j}$ signals to all $j \neq i$. This is needed in case of an internal take over situation.

In the case of $H_{\mathcal{Q}}$ this automaton just implements invariants over the variables. As the details of this automaton is straightforward they are not given here. The

For a tuple

$$p = (\alpha, \{(\beta_1, g_1, \mathcal{A}_1), \ldots, (\beta_k, g_k, \mathcal{A}_k)\}, \{(\alpha_1, g_1'), \ldots, (\alpha_l, g_l')\}) \in \mathcal{P}$$

we set $R^D_{U,\mathcal{P}}(p)$ to

$$\{((X,Y), b_\alpha, c_{\beta_1} := 1, \ldots, c_{\beta_k} := 1, b_{\alpha_1} := 1, \ldots, b_{\alpha_l} := 1, \quad (11)$$
$$(X \cup \{\alpha\}, Y)) \mid \alpha \notin X\}$$
$$//\text{Alarm } \alpha \text{ is set, all connected } \beta_i, \alpha_j \text{ are set}$$
$$\cup\{((X,Y), take_{\beta_i} \wedge g_i, take_\alpha := 1, (X, Y \cup \{(\alpha, \beta_i)\})) \mid \alpha \in X, (\alpha, \beta_i) \notin Y\} \quad (12)$$
$$//\text{There is a } take \text{ from a connected } \beta_i \text{ which is forwarded}$$
$$\cup\{((X,Y), take_{\alpha_j} \wedge g_j', take_\alpha := 1, (X, Y \cup \{(\alpha, \alpha_j)\})) \mid \alpha \in X, (\alpha, \alpha_j) \notin Y\} \quad (13)$$
$$//\text{There is a } take \text{ from a connected } \alpha_j \text{ which is forwarded}$$
$$\cup\{((X,Y), \neg(take_{\beta_i} \wedge g_i), take_\alpha := 1, \quad (14)$$
$$(X, Y \setminus \{(\alpha, \beta_i)\})) \mid \alpha \in X, (\alpha, \beta_i) \in Y, \exists \gamma : (\alpha, \gamma) \in Y \setminus \{(\alpha, \beta_i)\}$$
$$//\beta_i \text{ is not able to take over anymore, but an alternative is left}$$
$$\cup\{((X,Y), \neg(take_{\beta_i} \wedge g_i), take_\alpha := 0, \quad (15)$$
$$(X, Y \setminus \{(\alpha, \beta_i)\})) \mid \alpha \in X, (\alpha, \beta_i) \in Y, \neg(\exists \gamma : (\alpha, \gamma) \in Y \setminus \{(\alpha, \beta_i)\})\}$$
$$//\beta_i \text{ is not able to take over anymore and no alternative is left}$$
$$\cup\{((X,Y), \neg(take_{\alpha_i} \wedge g_i'), take_\alpha := 1, \quad (16)$$
$$(X, Y \setminus \{(\alpha, \alpha_j)\})) \mid \alpha \in X, (\alpha, \alpha_j) \in Y, \exists \gamma : (\alpha, \gamma) \in Y \setminus \{(\alpha, \alpha_j)\}$$
$$//\alpha_j \text{ is not able to take over anymore, but an alternative is left}$$
$$\cup\{((X,Y), \neg(take_{\alpha_i} \wedge g_i'), take_\alpha := 0, \quad (17)$$
$$(X, Y \setminus \{(\alpha, \alpha_j)\})) \mid \alpha \in X, (\alpha, \alpha_j) \in Y, \neg(\exists \gamma : (\alpha, \gamma) \in Y \setminus \{(\alpha, \alpha_j)\})\}$$
$$//\alpha_j \text{ is not able to take over anymore and no alternative is left}$$
$$\cup\{((X,Y), switch_\alpha, start_{\beta_i} := 1, \mathcal{A}_i, RESET(Var_{\mathcal{P}}^{out} \setminus \{\beta_i\}), \quad (18)$$
$$(\varnothing, \varnothing)) \mid \alpha \in X, (\alpha, \beta_i) \in Y))\}$$
$$//\text{There is a } switch \text{ which is forwarded to a connected } \beta$$
$$\cup\{((X,Y), switch_\alpha, switch_{\alpha_j} := 1, RESET(Var_{\mathcal{P}}^{out} \setminus \{\alpha_j\}), \quad (19)$$
$$(\varnothing, \varnothing)) \mid \alpha \in X, (\alpha, \alpha_j) \in Y))\}$$
$$//\text{There is a } switch \text{ which is forwarded to a connected } \alpha$$

where $RESET(\{v_1, \ldots, v_M\})$ stands for $v_1 := 0, \ldots, v_M := 0$.

Fig. 8. Urgent transitions for a $p \in \mathcal{P}$

invariants implemented are also fairly obvious. Not trivial is the construction of $H_{\mathcal{P}}$. Its purpose is to implement the correct handling of the outgoing signals of the components. A location of $H_{\mathcal{P}}$ is a pair (X, Y), where

- X contains all $\alpha \in A^{src}$ which are currently set. The set A^{src} is the union of all outport of the components $C_1, \ldots, C_n$.
- Y is set containing pairs (α, β_i) or (α, α_j) where $\alpha \in A^{src}$, β_i is an inport of a component and α_j is an outport of the composition C. A pair being in Y stands for the information that α is set and a *take* from β_i resp. α_j has been set.

The transitions of $H_{\mathcal{P}}$ are given in Fig. 8 and they are implementing the invariants for the locations given above. Whenever b_α becomes true, then α is added to the X set (11). Moreover, this transition forwards this signal to all connected inport and outport. As soon as a *take* is received from there the corresponding pair is added to the Y set (12–13). To do so, it is required that the given guard g_i resp. g'_j are satisfied. The transitions (14–17) handle the various cases that either a guard becomes not satisfied or the corresponding *take* signal was reset. They have to distinguish the case whether the $take_\alpha$ signal has to be reset or not. If a (α, β_i) or (α, α_j) must be removed from the Y set, then the question is whether an alternative is left. If it is, the $take_\alpha$ can remain set, otherwise it must be reset.

As soon as a $switch_\alpha$ occurs the $H_{\mathcal{P}}$ automaton has to react without delay due to the transitions (18–19). They select a $switch_{\alpha_j}$ or $start_{\beta_i}$ signal where (α, α_j) resp. (α, β_i) must be in the current Y set. This ensures that the corresponding guard is currently satisfied. In case of $start_{\beta_i}$ the assignments are also executed. Note that the selection is done nondeterministically if more than one option is available. However, the reaction to a $switch_\alpha$ happens without delay because the transitions are defined as being urgent.

Definition 12 (Semantics of a Closed Loop). *The semantics of a closed loop consisting of a component C and a plant P is defined as $[\![C \parallel P]\!] := [\![C]\!] \parallel P$.*

We now complete the design of the ACC controller.

Example (cont.) We complete the design by adding a *BRAKE* component to cater for the so far unserved alarm of the PI controller for plant situations where the actual velocity is much higher than the desired velocity, and arrive at the hierarchical composition depicted in Figure 9.

All specifications of components are summarized in Tables 1, 2, and 3. We provide admissible hybrid automata for the basic components *BRAKE*, *PI* and *ACCELERATE* as follows.

$$H_{BRAKE} = (\{m_{BRAKE}\}, \varnothing, \{v\}, \{a\}, \varnothing, true, true, true, \Theta_{BRAKE})$$

with

- $\Theta_{BRAKE}(m_{BRAKE}) = (a = -2)$.

$$H_{PI} = (\{m_{C_{PI}}\}, \{x\}, \{v\}, \{a\}, \varnothing, R^C_{PI}, true, \Phi_{PI}, \Theta_{PI})$$

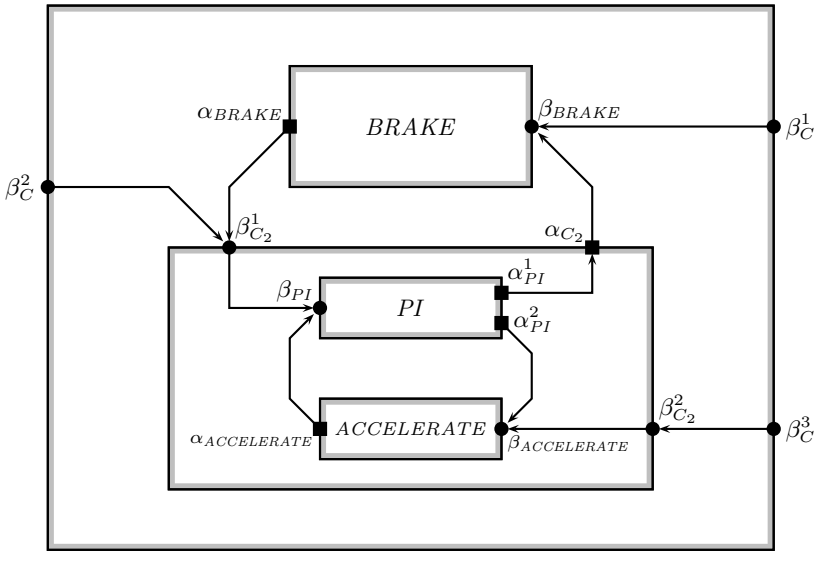


Fig. 9. Interconnection Structure for the ACC Example

Table 1. Component Interfaces (and local variables) for ACC Example

Comp.	Var^{in}	Var^{out}	Var^{loc}	φ^{assm}	φ^{prom}	Δ^{stable}
C	$\{v\}$	$\{a\}$	$\{x\}$	$-30 \leq v \leq 30$	$-2 \leq v^{\bullet} \leq 1.5$	300
$BRAKE$	$\{v\}$	$\{a\}$	$\varnothing$	$5 \leq v \leq 30$	$v^{\bullet} = -2$	300
C_2	$\{v\}$	$\{a\}$	$\{x\}$	$-30 \leq v \leq 15$	$-1.4 \leq v^{\bullet} \leq 1.5$	300
PI	$\{v\}$	$\{a\}$	$\{x\}$	$-15 \leq v \leq 15$	$-1.4 \leq v^{\bullet} \leq 1.4$	300
$ACCELERATE$	$\{v\}$	$\{a\}$	$\varnothing$	$-30 \leq v \leq -5$	$v^{\bullet} = 1.5$	300

Table 2. Outports for ACC Example

Outport	Comp.	$\varphi_{\alpha}^{alarmOn}$	μ_{α}	Δ_{α}	φ_{α}^{exit}
α_{BRAKE}	$BRAKE$	$v \leq 6$	0.005	0.01	$v \leq 6$
α_{C_2}	C_2	$v \geq 14$	0.006	0.01	$13.9 \leq v \leq 14.1$
α_{PI}^1	PI	$v \geq 14$	0.006	0.01	$13.9 \leq v \leq 14.1$
α_{PI}^2	PI	$v \leq -14$	0.006	0.01	$-14.1 \leq v \leq -13.9$
$\alpha_{ACCELERATE}$	$ACCELERATE$	$v \geq -6$	0.005	0.01	$v \geq -6$

with

- $\Theta_{PI}(m_{PI}) = (a = -0.001x - 0.052v)$.
- $R_{PI}^{C}(m_{C_{PI}}) = (x^{\bullet} = v)$
- $\Phi_{PI} = (x = 0)$

$$H_{ACCELERATE} = (\{m_{ACCELERATE}\}, \varnothing, \{v\}, \{a\}, \varnothing, true, true, true, \Theta_{ACCELERATE})$$

Table 3. Inports (and initialization of local variables) for ACC Example

Inport	Comp.	λ_β	φ_β^{entry}	Φ_β
β_C^1	C	0.004	$10 \leq v \leq 30$	n/a
β_C^2	C	0.004	$-13.5 \leq v \leq 13.5$	n/a
β_C^3	C	0.004	$-30 \leq v \leq -10$	n/a
β_{BRAKE}	$BRAKE$	0.003	$10 \leq v \leq 30$	n/a
$\beta_{C_2}^1$	C_2	0.004	$-13.5 \leq v \leq 13.5$	n/a
$\beta_{C_2}^2$	C_2	0.004	$-30 \leq v \leq -10$	n/a
β_{PI}	PI	0.0025	$-13.5 \leq v \leq 13.5$	$x = 0$
$\beta_{ACCELERATE}$	$ACCELERATE$	0.0025	$-30 \leq v \leq -10$	n/a

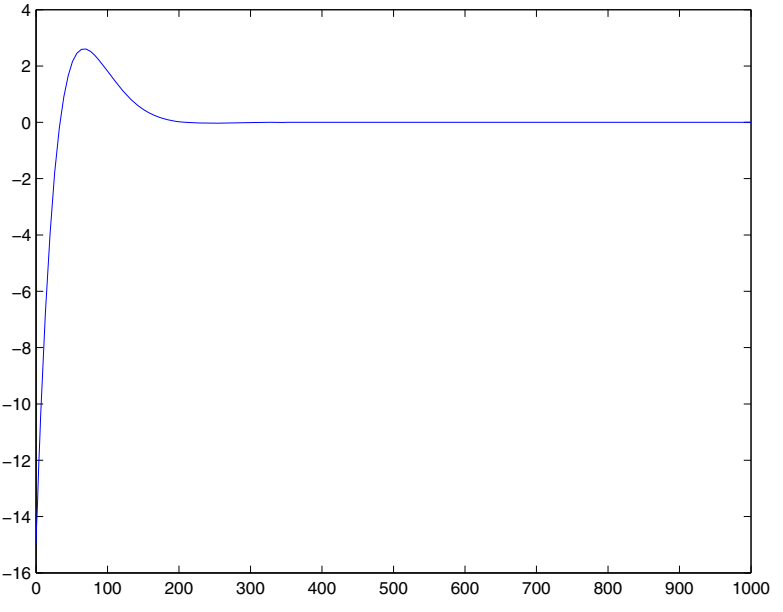


Fig. 10. Example trajectory of $H_{PI} \parallel P$: time (horizontal axis) vs. velocity differential v (vertical axis)

with

– $\Theta_{ACCELERATE}(m_{ACCELERATE}) = (a = 1.5)$.

Figure 10 gives a sample trajectory with the closed loop of the PI controller and the plant model.

We conclude this section by formally defining the satisfaction relation for hybrid interface specifications.

Definition 13 (Satisfaction of Component Interface Specification). *An implementation I (in terms of a hybrid automaton) satisfies the interface specification $SPEC_C$ of a component C associated with a plant P (denoted by $I \parallel P \models SPEC_C$) iff*

(A) $Var_I^{out} \supseteq Var_C^{out} \cup C_C^{out}$,
(B) $Var_I^{in} \supseteq Var_C^{in} \cup C_C^{in}$,
(C) $\models \varphi_C^{entry} \implies \neg\varphi_\alpha^{alarm}$ *for all* $\alpha \in A_C^{out}$,
(D) $\models \varphi_C^{entry} \implies \varphi_P^{safe} \wedge \varphi_C^{assm}$

and for all runs that are failure-free (i.e, $\Box\neg fail_C$*) satisfy the following requirements:*

(E) $\neg active_C$ UNLESS $(\bigvee_{\beta\in A_C^{in}}(start_\beta \wedge \varphi_\beta^{entry}))$ *(a component is inactive initially unless it is started via an inport* β*.)*
(F) $\forall\alpha \in A_C^{out} : \neg\Diamond(\neg\varphi_\alpha^{alarm} \,\mathsf{U}\, (\varphi_\alpha^{alarm} \wedge active_C \wedge \neg b_\alpha))$ *(when* φ_α^{alarm} *becomes true while the component is active,* b_α *holds)*
(G) $\forall\alpha \in A_C^{out} : \neg\Diamond(\neg b_\alpha \,\mathsf{U}\, (b_\alpha \,\mathsf{U}_{<\mu_\alpha} \neg b_\alpha \wedge \neg take_\alpha))$ *(when* b_α *becomes false without receiving a take,* b_α *was on for at least* μ_α *time units)*
(H) $\forall\beta \in A_C^{in} : \neg\Diamond(\neg start_\beta \,\mathsf{U}\, start_\beta \wedge \neg active_C) \wedge$
$\neg\Diamond(\neg active_C \,\mathsf{U}\, active_C \wedge \neg \bigvee_{\beta\in A_C^{in}} start_\beta)$ *(a component is activated exactly when a* $start_\beta$ *has a rising edge)*
(I) $\neg\Diamond(\neg suspend_C \,\mathsf{U}\, suspend_C \wedge active_C) \wedge$
$\neg\Diamond(active_C \,\mathsf{U}\, \neg active_C \wedge \neg suspend_C)$ *(a component is deactivated exactly when* $suspend_C$ *has a rising edge)*
(J) $\forall\beta \in A_C^{in} : \neg\Diamond(c_\beta \wedge \neg take_\beta \,\mathsf{U}_{>\lambda_\beta} true)$ *(an incoming alarm is answered by* $take_\beta$ *after at most* λ_β *time units)*
(K) $\forall\beta \in A_C^{in} : \Box(take_\beta \wedge \neg c_\beta \,\mathsf{U}_{=0} \neg take_\beta)$ *(*$take_\beta$ *is only set when an alarm is incoming)*
(L) $\forall\beta \in A_C^{in} : \neg\Diamond(take_\beta \wedge c_\beta \,\mathsf{U}\, (\neg take_\beta \wedge c_\beta))$ *(*$take_\beta$ *is not withdrawn if the incoming alarm remains)*
(M) $\neg\Diamond(\neg active_C \,\mathsf{U}\, (active_C \wedge \neg\varphi_C^{entry}))$ *(*φ^{entry} *holds whenever a component is activated)*
(N) $\Box(active_C \implies \varphi_P^{safe} \wedge \varphi_C^{assm} \wedge \varphi_C^{prom})$ *(active components guarantee* $\varphi_P^{safe} \wedge \varphi_C^{assm} \wedge \varphi_C^{prom}$*)*
(O) $\Box(active_C \implies ((\Diamond_{\Delta_C^{stable}}(\Box\varphi_P^{stable})) \text{ UNLESS } (\neg active_C))$ *(active components guarantee convergence to* φ_P^{stable} *within* Δ_C^{stable} *time units)*
(P) $\forall\alpha,\alpha' \in A_C^{out} : \neg\Diamond(\neg switch_\alpha \,\mathsf{U}\, (switch_\alpha \wedge b_{\alpha'}))$ *(when* $switch_\alpha$ *has a rising edge, then all outgoing alarms are reset)*
(Q) $\forall\beta \in A_C^{in}, \alpha \in A_C^{in} : \neg\Diamond(\neg switch_\alpha \,\mathsf{U}\, (switch_\alpha \wedge take_\beta))$ *(when any* $switch_\alpha$ *has a rising edge, then all* $take_\beta$ *are reset)*
(R) $\forall\alpha \in A_C^{out} : \neg\Diamond(\neg switch_\alpha \,\mathsf{U}\, (switch_\alpha \wedge \neg\varphi_\alpha^{exit}))$ *(when* $switch_\alpha$ *becomes true,* φ_α^{exit} *is guaranteed)*

and for all trajectories that initially fulfill φ^{entry} *and fail for the first time at time* t *(i.e.,* $\neg fail_C \,\mathsf{U}_{=t} fail_C$*):*

(S) $(\exists\alpha \in A_C^{out}, t_\alpha \leq t - \Delta_\alpha : \forall t' \in [t_\alpha, t] : b_\alpha(t')) \vee (\exists\beta \in A_C^{in} : start_\beta(t) \wedge \neg\varphi_\beta^{entry}(t))$ *(a failure is preceded by an alarm that becomes and stays active at least* Δ_α *time units earlier* or *the failure is due to an entry condition for an inport* β *that is not met during the activation.)*
(T) $\Box(fail_C \implies (\Box fail_C))$ *(once a failure occurs,* $fail_C$ *remains true)*

4 Hierarchical Verification of Robust Safety and Stability

In the following we give verification conditions for basic components and transition compositions thereof. These verification conditions imply that the component fulfills its interface specification, so that both safety and stability properties are guaranteed while the component is active. The verification conditions are of three types:

- inequalities on scalars: these include the different timing constraints involving the λ_β, μ_α, or Δ_α,
- implications on first order predicates: these relate the predicates that are part of the interface specification (e.g., φ_C^{entry}, $\varphi_\alpha^{alarmOn}$, φ_β^{entry}) to one another, and
- Lyapunov function conditions: these are used to prove stability of the system, using parameterized Lyapunov functions.

Lyapunov functions are abstract energy functions of the closed loop. Intuitively, a Lyapunov function maps each system state onto a nonnegative energy value, with the restriction that the function must always decrease along every trajectory, unless a designated equilibrium point is reached. Since the Lyapunov function also has its minimum at this equilibrium, it serves as a tool to prove convergence. In the following, we define Lyapunov functions for single-mode systems. Lyapunov functions will be used in the verification conditions for stability. In particular, there will be constraints on the existence of Lyapunov functions with a particular parameterization. Specialized software (SDP solvers [Bor99, RPS99]) can be used to carry out the automatic computation of these functions.

Definition 14 (Lyapunov Functions). *Let*

$$H = (\{m\}, Var^{loc}, Var^{in}, Var^{out}, \varnothing, R^C, R^I, \Phi, \Theta)$$

be a hybrid automaton and $Var = Var^{loc} \cup Var^{in} \cup Var^{out}$.

Let $X = [(X^O)^T, (X^L)^T, (X^I)^T]^T$ *be a vector of valuations of* Var*, with subvectors* X^O*,* X^L *and* X^I *pertaining to the variables in* Var^{out}*,* Var^{loc}*, and* Var^{in}*, respectively. Furthermore, define the vector of controlled variables as* $X^C := [(X^O)^T, (X^L)^T]^T$*. Assume that* $R^C(m)$ *is given by a differential inclusion*

$$X^{C\bullet} \in F(X), F : Var \to 2^{\mathbb{R}^{|Var^{loc}|+|Var^{out}|}}$$

and R^I *by a differential inclusion*

$$X^{I\bullet} \in G(X^I), G : Var^{in} \to 2^{\mathbb{R}^{|Var^{in}|}}$$

A Lyapunov function *wrt.*

- *an equilibrium state* $X_e^O \in \mathbb{R}^{|Var^{out}|}, X_e^O \models \varphi_P^{stable}$ *and*
- *a nonnegative function* $f : Var \to \mathbb{R}$ *such that* $f(X) = 0$ *if* $X^O = X_e^O$

is a function $\mathcal{V} : \mathbb{R}^{|Var|} \to \mathbb{R}$ *for which there exist* $k_1, k_2, k_3 > 0$ *such that for all* $X \models \Theta(m)$*:*

(1) $k_1 ||X^O - X_e^O||^2 \leq \mathcal{V}(X) \leq k_2 f(X)$

(2) $\mathcal{V}^\bullet(X) := \sup_{\bar{X}^I \in G(X^I), \bar{X}^C \in F(X)} \left(\frac{d\mathcal{V}}{dX}(X) \cdot \begin{bmatrix} \bar{X}^C \\ \bar{X}^I \end{bmatrix} \right) \leq -k_3 f(X)$

The values of k_2 and k_3 will later be used to estimate convergence time toward a φ_P^{stable}. Note that the set of Lyapunov functions for a given system is closed under conic combination, i.e., if the $\mathcal{V}_i$ are Lyapunov functions for a system, then, for all $\lambda_i \geq 0$ such that at least one $\lambda_i > 0$, $\sum_i \lambda_i \mathcal{V}_i$ is also a Lyapunov function. Moreover, the constants k_1, k_2 and k_3 also add up in the same manner, as the Lyapunov function $\sum_i \lambda_i \mathcal{V}_i$ will have constants k_1, k_2, and k_3, that are the weighted sums over the λ_i of the corresponding constants of the $\mathcal{V}_i$. This property will be exploited heavily in the verification conditions. The function f can, in general, be chosen arbitrarily. However, some choices will result in better convergence time estimates than others. Ideally, the function $f(X)$ should be of similar form as $\mathcal{V}(X)$ and $\mathcal{V}^\bullet(X)$, since this provides the tightest overapproximations.

Note that, while the equilibrium state X_e^O is only defined in terms of output variables of a system, the function $\mathcal{V}$ can potentially talk about all variables in *Var*, but is not required to do so. This is sufficient, because we are only interested in convergence of variables that are a) externally visible, and b) actually under the control of the system. However, it is sometimes helpful to define a Lyapunov also in terms of non-output variables, especially if the output variables show non-monotonic behavior. The ACC provided as a running example is such a case.

For the stability analysis, we will apply Lyapunov functions to the closed loop only, i.e., the input variables of the system to be analyzed will be viewed as external disturbances, and the only local variables will be those of the controller, as the local variable set of the plant is per definition empty. Any sensor or actuator variables will also appear as output variables in the closed loop, as per Def. 4.

For a vector $v \in \mathbb{R}^n$, defined $v^{(i)}$ as the i-th component of v. Next, we give the verification conditions for basic components, given a plant P.

Theorem 1 (Verification Conditions for Basic Components). *Let* C *be a basic component with implementation semantics* I *as defined in Def. 10, with an admissible hybrid automaton* H_C*, which just consists of a single mode and no discrete transitions. Define* $H = (\{m\}, Var^{loc}, Var^{in}, Var^{out}, \varnothing, R^C, R^I, \Phi, \Theta)$ *as the hybrid automaton obtained by the parallel composition* $H_C \parallel P$*. For a valuation of the variables* $Var^{in} \cup Var^{out}$*, define the vector* $X^{IO} \in \mathbb{R}^{|Var^{in}|+|Var^{out}|}$*. Define the* parameter space *for the Lyapunov functions as* $\mathbb{R}^{r_C}$ *for an arbitrary* r_C*. Define the predicate* P_C^{Lyap} *on the elements* $\theta_C^{(i)}$ *of a* $\theta_C \in \mathbb{R}^{r_C}$ *as*

$$P_C^{Lyap} := \forall 1 \leq i \leq r_C : \theta_C^{(i)} \geq 0 \wedge \exists 1 \leq i \leq r_C : \theta_C^{(i)} > 0.$$

If

1. $\bigvee_\beta(reach(H, \varphi_\beta^{entry} \wedge \Phi_\beta)) \wedge \bigwedge_\alpha reach(H, \neg\varphi_\alpha^{alarm}, \Delta_\alpha) \implies \varphi_P^{safe} \wedge \varphi_C^{assm}$
2. $\forall\alpha \in A_C^{out}, \beta \in A_C^{in} : \varphi_\beta^{entry} \implies \neg\varphi_\alpha^{alarm}$
3. $\bigvee_\beta(reach(H, \varphi_\beta^{entry} \wedge \Phi_\beta)) \wedge R^C(m) \wedge \varphi_C^{assm} \implies \varphi_C^{prom}$
4. $\forall\alpha \in C_C^{out} : reach(H, \partial\varphi_\alpha^{alarmOn}, \Delta_\alpha) \implies \varphi_\alpha^{exit}$, *where* $\partial\Phi$ *is the border of a closed predicate* Φ,
5. *there exist* r_C *Lyapunov functions* $\mathcal{V}_C^i : \mathbb{R}^{|Var_C|} \to \mathbb{R}, 1 \leq i \leq r_C$ *for* H *wrt. the same equilibrium* $X_e^O \models \varphi_P^{stable}$ *and function* $f(X)$, *with constants* $k_1(C,i)$, $k_2(C,i)$, $k_3(C,i)$
6. *for all* i , $1 \leq i \leq r_C$, *there exist two constants* $c_{entry}(C,i)$ *and* $c_{stab}(C,i)$ *such that* $(X \models \left(\bigvee_{\beta \in C_C^{in}} \varphi_\beta^{entry} \wedge \Phi_\beta\right) \implies \mathcal{V}_C^i(X) \leq c_{entry}(C,i))$ *and* $(\mathcal{V}_C^i(X) \leq c_{stab}(C,i) \implies X \models \varphi_P^{stable})$
7. *there exists a* $\theta_C \models P_C^{Lyap}$ *such that*

$$\Delta_C^{stable} \geq \frac{1}{rate_C(\theta_C)} \ln\left(\frac{\sum_i \theta_C^{(i)} c_{entry}(C,i)}{\sum_i \theta_C^{(i)} c_{stab}(C,i)}\right),$$

where

$$rate_C(\theta_C) = \frac{\sum \theta_C^{(i)} k_3(C,i)}{\sum \theta_C^{(i)} k_2(C,i)}$$

then $I \parallel P \models SPEC_C$.

The Lyapunov function conditions require a detailed explanation. First, assume that r_C is set to 1. Note that, in condition (O) of Definition 13, we are interested not only in convergence to φ_P^{stable}, but in convergence in bounded time. If this were not the case, then stability of the parallel composition between component and plant would already be proven if we could find a Lyapunov function for the system. To verify such a time bound, we additionally need to keep track of the decrease rate of the Lyapunov function value (i.e., the "energy"). This rate is characterized by the ratio $k_3(C,1)/k_2(C,1)$, which gives an exponential decrease rate of the Lyapunov function value over time. To bound the convergence time, we also need to know a bound on the initial Lyapunov function value upon activation ($c_{entry}(C,1)$) and a Lyapunov function value that is low enough for φ_P^{stable} to be fulfilled ($c_{stab}(C,1)$). Together, these scalar values can be used to conservatively estimate a time bound for convergence, exploiting the exponential convergence of $\mathcal{V}_C^1$.

If we do not intend to further compose C, then setting $r_C = 1$ is indeed enough, as a single Lyapunov function already guarantees the stability property. However, if we want to support further composition, we need to provide means to show that the newly composed component again guarantees φ_P^{stable} in bounded time, i.e., that there also exists a Lyapunov function for this transition composition. However, our function $\mathcal{V}_C^1$ can also talk about local variables of C, which are invisible in the transition composition. Nevertheless, we have to

provide verification conditions for transition compositions that guarantee that the "energy" does not increase when a switch to a new component takes place. For this reason, a component has to communicate the Lyapunov function values it can have at two time instants: when it is activated and when it is deactivated. Since we cannot talk about local variables at the interface, we associate a *projection* of the Lyapunov function on the externally visible variables with each in- and outport of the component. For the composition, we then compare the two projected functions for all in- and outports to be connected to ensure a decrease of the energy upon switching.

Here lies also the motivation for setting r_C to a number larger than 1. In general, systems can have infinitely many possible Lyapunov functions. By picking just one function, we risk that, for the transition composition, the stability proof will not be possible. Usually only some Lyapunov functions for the subcomponents will be suitable for verifying the non-increasingness conditions. It is therefore helpful for the subcomponents to communicate as many Lyapunov function projections to the outside as reasonably possible. We can then exploit the conic closure property of Lyapunov functions to construct *infinitely many* different Lyapunov functions for the subcomponent, while trying to satisfy the non-increasingness conditions for all port connections of the composed component.

Proof (of Theorem 1). We have to show that $I_C \parallel P \models SPEC_C$ holds when the assumption of the theorem are given for I_C. To prove that we have to show that all of the requirements given in Def. 13 hold.

(A):

$$\begin{aligned} Var^{out}_{I_C} &= Var^{out}_{H_C} \cup \{fail_C, active_C\} && \text{// Def. 10} \\ &\cup \{take_\beta \mid \beta \in A^{in}_C\} \\ &\cup \{b_\alpha, switch_\alpha \mid \alpha \in A^{out}_C\} \\ &= Var^{out}_{H_C} \cup C^{out}_C && \text{// } H_C \text{ admissible, Def. 9} \\ &= Var^{out}_C \cup C^{out}_C \end{aligned}$$

(B):

$$\begin{aligned} Var^{in}_{I_C} &= (Var^{in}_{H_C} \cup \{active_C, suspend_C\} && \text{// Def. 10} \\ &\cup \{start_\beta, c_\beta \mid \beta \in A^{in}_C\} \\ &\cup \{take_\alpha \mid \alpha \in A^{out}_C\}) \setminus Var^{out}_{I_C} \\ &= Var^{in}_{H_C} \cup C^{in}_C && \text{// } H_C \text{ admissible, Def. 9} \\ &= Var^{in}_C \cup C^{in}_C \end{aligned}$$

(C) follows directly from the assumption 2 of the theorem.

(D) follows from the assumptions 1 and 2:

$$\begin{aligned}
\varphi_C^{entry} &\Longrightarrow \bigvee_\beta \varphi_\beta^{entry} \\
&\Longrightarrow \bigvee_\beta \varphi_\beta^{entry} \wedge \bigwedge_\alpha \neg\varphi_\alpha^{alarm} && \text{// Assm. 2} \\
&\Longrightarrow \bigvee_\beta \varphi_\beta^{entry} \wedge \bigwedge_\alpha reach(H, \neg\varphi_\alpha^{alarm}, \Delta_\alpha) \\
&\Longrightarrow \bigvee_\beta reach(H, \varphi_\beta^{entry}) \wedge \bigwedge_\alpha reach(H, \neg\varphi_\alpha^{alarm}, \Delta_\alpha) \\
&\Longrightarrow \varphi_P^{safe} \wedge \varphi_C^{assm} && \text{// Assm. 1}
\end{aligned}$$

(E): This holds due to the construction of H_1. The initial location is m_{inact} and it requires a $start_\beta$ to activate the component. Moreover, whenever the activation takes place H_1 checks whether the entry condition φ_β^{entry} is given. If this is not the case, then H_1 sets $fail_C$.

(F): For a fixed α we consider the automaton H_3 of Def. 10 and a run in which the φ^{alarm} becomes true while b_α is not set (during all discrete steps at this time point).

If H_3 is in mode $Fail$ when the φ^{alarm} becomes true, then $fail_C$ is set and nothing is to show. Otherwise it is in a mode (X, Y). Due to the construction of H_3 we have the invariant $\alpha \in X \iff b_\alpha$. So, we have to consider two cases: $\alpha \notin X \cup Y$ and $\alpha \in Y \setminus X$. In the first case, there is an urgent transition that becomes enabled as soon as φ^{alarm} becomes true provided that the component is active. Hence, we can conclude that b_α will be set within that time point. In the second case we know that $Y \neq \varnothing$ and by the construction of the urgent transitions H_3 will switch to a mode with $\alpha \notin X \cup Y$ (first case) without delay.

(G): For a fixed α we consider the automaton H_3 of Def. 10 and a run of C that violates the given property. That means that we have two points in time $t_1 \leq t_2$ (with $t_2 - t_1 < \mu_\alpha$) where b_α was set and reset, respectively. Due to the construction of H_3 it is clear that setting b_α resets the clock t_α to 0. There are three kinds of transitions resetting b_α. They are guarded by $take_\alpha$ or guarded by $t_\alpha \geq \mu_\alpha$ or not enabled because of the invariant $b_\alpha \iff \alpha \notin Y$.

(H): Satisfied by the construction of H_1.

(I): The component can only reset *active* in the automaton H_2. The only transition there require to have a *suspend*. Since this transition is urgent, a rising edge of *suspend* with deactivate the component.

(J): This holds because H_β ensures that a rising edge of c_β starts a timer t_β and the $take_\beta$ signal is given at latest when this timer reaches the time bound λ_β.

(K): Satisfied by H_β because setting $take_\beta$ requires c_β to be true. If c_β is reset, then $take_\beta$ will follow within finitely many discrete steps because of urgent transitions in H_β.

(L): Satisfied by H_β because resetting $take_\beta$ requires $\neg c_\beta$ to hold.

(M): Satisfied by H_β because setting *active* requires φ_β^{entry} to hold.

(N): Since the system is initially in φ_C^{entry} or is activated in φ_C^{entry} (M) we can conclude that the system state belongs to

$$\bigvee_{\beta}(reach(H, \varphi_\beta^{entry})) \wedge \bigwedge_{\alpha} reach(H, \neg\varphi_\alpha^{alarm}, \Delta_\alpha)$$

because *fail* is not set. Moreover, we have

$$reach(H_C, \varphi^{entry}) \wedge R^C(m)$$

as conservative approximation of the evolution of the system's state. With the assumptions 1 and 3 we get the desired implication immediately.

(O): Let $X(t)$ be a the projection of a trajectory of $I||P$ onto $Var_C \cup Var_P$ with $X(0) \models \bigvee_{\beta \in C_C^{in}} \varphi_\beta^{entry} \wedge \Phi_\beta$. Assume, without loss of generality, that $active_C$ receives a rising edge at time 0. By condition (N) of Def. 13, we know that, as long as C is active, φ^{assm} holds, and therefore also $\Theta(m)$. Per verification condition 7, there exists a $\theta_C \in \mathcal{P}$ such that $\theta_C \models P_C^{Lyap}$ and the inequality on Δ_C^{stable} is fulfilled. Define $\mathcal{V}_C(\theta_C, X) := \sum_i \theta_C^{(i)} \mathcal{V}_C^i(X)$. Since for all i, $\mathcal{V}_C^{i\bullet}(X) \leq -k_3(C,i)||X^O - X_e^O||^2$, we obtain, by linear combination:

$$\mathcal{V}_C{}^\bullet(\theta_C, X) = \sum_i \theta_C^{(i)} \mathcal{V}_C^i{}^\bullet(X) \leq \left(-\sum_i \theta_C^{(i)} k_3(C,i)\right) f(X).$$

In the same manner, we can derive that

$$\mathcal{V}_C(\theta_C, X) \leq \left(\sum_i \theta_C^{(i)} k_2(C,i)\right) f(X).$$

Since $rate_C(\theta_C) = \sum_i \theta_C^{(i)} k_3(C,i) / \sum_i \theta_C^{(i)} k_2(C,i)$ this gives us

$$\mathcal{V}_C{}^\bullet(\theta_C, X) \leq \left(-\sum_i \theta_C^{(i)} k_3(C,i)\right) f(X) \leq -rate_C(\theta_C)\mathcal{V}_C(\theta_C, X)$$

and therefore

$$\mathcal{V}_C(\theta_C, X(t)) \leq e^{-rate_C(\theta_C)t} \mathcal{V}_C(\theta_C, X(0))$$

for all t such that C remains active on the interval $[0,t]$. Furthermore, since

$$\Delta_C^{stable} \geq \frac{\sum \theta_C^{(i)} k_2(C,i)}{\sum \theta_C^{(i)} k_3(C,i)} \ln\left(\frac{\sum_i \theta_C^{(i)} c_{entry}(C,i)}{\sum_i \theta_C^{(i)} c_{stab}(C,i)}\right),$$

we have that

$$\begin{aligned}\mathcal{V}_C(\theta_C, X(\Delta_C^{stable})) \leq & \exp\left(\frac{-rate_C(\theta_C)}{rate_C(\theta_C)} \ln\left(\frac{\sum_i \theta_C^{(i)} c_{entry}(C,i)}{\sum_i \theta_C^{(i)} c_{stab}(C,i)}\right)\right) \mathcal{V}_C(\theta_C, X(0)) \\ = & \exp\left(\ln\left(\frac{\sum_i \theta_C^{(i)} c_{stab}(C,i)}{\sum_i \theta_C^{(i)} c_{entry}(C,i)}\right)\right) \mathcal{V}_C(\theta, X(0)) \\ = & \frac{\sum_i \theta_C^{(i)} c_{stab}(C,i)}{\sum_i \theta_C^{(i)} c_{entry}(C,i)} \mathcal{V}_C(\theta_C, X(0)).\end{aligned}$$

Since $X(0) \models \left(\bigvee_{\beta \in C_C^{in}} \varphi_\beta^{entry} \wedge \Phi_\beta\right)$, this implies

$$\mathcal{V}_C(\theta_C, X(0)) \leq \sum_i \theta_C^{(i)} c_{entry}(C,i),$$

and we obtain

$$\mathcal{V}_C(\theta_C, X(\Delta_C^{stable})) = \sum_i \theta_C^{(i)} \mathcal{V}_C^i(X(\Delta_C^{stable})) \leq \sum_i \theta_C^{(i)} c_{stab}(C,i).$$

Note that this implies that there exists an i with $\mathcal{V}_C^i(X(\Delta_C^{stable})) \leq c_{stab}(C,i)$. This gives us $X(\Delta_C^{stable}) \models \varphi_P^{stable}$. Since $\mathcal{V}_C{}^\bullet(\theta_C, X) < 0$, we obtain that

$$\Box(active_C \implies ((\Diamond_{\Delta_C^{stable}}(\Box \varphi_C^{stable})) \,\mathsf{UNLESS}\, (\neg active_C)).$$

(P),(Q): Immediately clear by the construction of the transitions of H_3 setting $switch_\alpha$.

(R): By the construction of H_3 we know that whenever a $switch_\alpha$ becomes true this was preceded by setting b_α at most $\Delta_\alpha - \tau$ time units before. The rising edge of b_α was induced by an urgent transition of H_3 because φ_α^{alarm} became true. Hence, the time point in which $switch_\alpha$ becomes true belongs to the set $reach(H, \partial\varphi_\alpha^{alarm}, \Delta_\alpha)$. Because of assumption 4 we can conclude that φ_α^{exit} holds at this time point.

(S): The first reason for failure is managed by H_3. There is only one kind of transition that sets $fail_C$ in this automaton. These transitions are guarded by a constraint of the form $t_\alpha \geq \Delta_\alpha$ for an $\alpha \in A_C^{out}$. By the construction of H_3 it is clear that this transition is only enabled if there has been a preceding period of duration Δ_α where b_α was set.

The second reason for a failure is managed by H_β. If this automaton observes that φ_β^{entry} is not given when it tries to activate the component, then it sets $fail_C$. This is the only way for H_β to do this.

(T): Obvious by the construction of all automata in the semantics: There is no transition setting $fail_C$ to 0.

The following definition gives the induction invariants of a basic component C, that is, all additional information that has to be provided by the component, in order to allow stability proofs of transition compositions containing C.

Definition 15 (Induction Invariants for Basic Components). *For each basic component, the following* induction invariants *need to be provided, to facilitate further composition:*

1. *for all* $\alpha \in A_C^{out}$ *and* $1 \leq j \leq r_C$, *a function* $\mathcal{V}_\alpha^j : \mathbb{R}^{|Var_C^{in}|+|Var_C^{out}|} \to \mathbb{R}$ *such that* $(X^L, X^{IO}) \models reach(H, \varphi_\alpha^{exit}, \tau) \wedge \bigvee_\beta (reach(H, \varphi_\beta^{entry} \wedge \Phi_\beta)) \implies \mathcal{V}_\alpha^j(X^{IO}) \leq \mathcal{V}_C^j(X)$,
2. *for all* $\beta \in A_C^{in}$ *and* $1 \leq j \leq r_C$, *a function* $\mathcal{V}_\beta^j : \mathbb{R}^{|Var_C^{in}|+|Var_C^{out}|} \to \mathbb{R}$ *such that* $(X^L \models \Phi_\beta \wedge X^{IO} \models \varphi_\beta^{entry}) \implies \mathcal{V}_\beta^j(X^{IO}) \geq \mathcal{V}_C^j(X)$,
3. *for all* $1 \leq j \leq r_C$, *the constants* $k_2(C, j)$ *and* $k_3(C, j)$ *of the Lyapunov functions* $\mathcal{V}_C^j$,
4. *for all* $1 \leq j \leq r_C$, *the constants* $c_{entry}(C, j)$ *and* $c_{stab}(C, j)$.

Example (cont.) To verify that the example ACC system satisfies its component interface specification, we need to prove that all the verification conditions in Theorem 1 hold. Using component PI as an example, the closed-loop automaton $H = H_{PI}||P$ can be described by the following differential inclusion:

$$\begin{aligned} x(t)^\bullet &= v(t) \\ v(t)^\bullet &\in co\{-0.001025x(t) - 0.0533v(t), -0.0009075x(t) - 0.0507v(t)\} \end{aligned}$$

where *co* denotes the convex hull. We arrive at this description by eliminating the variables that do not appear in differential equations, but only in invariants, namely a and s.

First, we must compute Lyapunov functions for H, such that verification conditions 5, 6 and 7 are fulfilled. To compute quadratic Lyapunov functions for linear or affine dynamics, *semidefinite programming (SDP)* [BEFB94] tools can be used, for instance CSDP [Bor99] or SeDuMi [RPS99]. In a nutshell, the problem of finding a Lyapunov function according to some parameterized template is mapped onto a nonlinear, but still convex optimization problem that can be solved numerically [Pet99]. For polytopic differential inclusions like the one given above, it is sufficient to identify a Lyapunov function that works for the extremal dynamics (i.e., $v(t)^\bullet = -0.001025x(t) - 0.0533v(t)$ and $v(t)^\bullet = -0.0009075x(t) - 0.0507v(t)$). One such example Lyapunov function for H is

$$\mathcal{V}_{PI}^1(x, v) = 46.7455x^2 + 1101.9vx + 20729v^2,$$

with constants $k_2 = 1$ and $k_3 = 0.018$. Here, the function $f(X)$ was simply set to the Lyapunov function itself and the equilibrium point was $v = 0$. To compute associated constants $c_{stab}(PI, 1)$ and $c_{entry}(PI, 1)$, one needs to find a value c, such that the contour line $\mathcal{V}_{PI}^1(x, v) = c$ is entirely within φ_P^{stable}, and an upper bound on $\mathcal{V}_{PI}^1(x, v)$ for $x \models \bigvee_{\beta \in C_{PI}^{in}} (\varphi_\beta^{entry} \wedge \Phi_\beta)$. These are simple one-dimensional optimization problems. For $\mathcal{V}_{PI}^1$, possible values are $c_{entry}(PI, 1) = 4700000$ and $c_{stab}(PI, 1) = 50000$. From here, it is straightforward to show that the inequality on Δ_{PI}^{stable} is also satisfiable. Since we computed only one Lyapunov function, θ_{PI} is simply a scalar, and by setting $\theta_{PI} = 1$, the inequality is satisfied.

For several verification conditions, we require a reach set of H, starting from $\varphi^{entry}_{\beta_{PI}} \wedge \Phi_{\beta_{PI}}$. There are various ways of arriving at such a set, for example the tool PHAVer [Fre08], or barrier certificates computed from Lyapunov functions [PJ04]. The latter method is especially suitable in this case, as we already have a barrier certificate: we know that $\mathcal{V}^1_{PI}(x,v) \leq c_{entry}(PI,1)$, as long as PI is active. This sublevel set forms an ellipsoid in the state space, which serves as an over-approximation of the reach set. For convenience, we again over-approximate this ellipsoid by a box and obtain

$$-500 \leq x \leq 500 \wedge -30 \leq v \leq 30.$$

Now we are ready to show the remaining verification conditions, which are linear arithmetic constraints that can be proven with tools like iSat[2] (including the time bounded reachability computation in condition 4) of Theorem 1.

To allow for further composition, we also need to provide the remaining induction invariants, namely projections of $\mathcal{V}^1_{PI}(x,v)$ onto the inports and outports of PI.

For the inport $\beta := \beta_{PI}$, this is a function $\mathcal{V}^1_\beta(v)$ that is larger than all possible values of $\mathcal{V}^1_{PI}(x,v)$ when PI is activated. Since $\Phi_\beta = (x = 0)$, we can just substitute $x = 0$ and obtain $\mathcal{V}^1_\beta(v) = 20729v^2$.

Similarly, for the outports $\alpha_1 := \alpha^1_{PI}$ and $\alpha_2 := \alpha^2_{PI}$, we must provide functions $\mathcal{V}^1_{\alpha_1}(v)$ and $\mathcal{V}^1_{\alpha_2}(v)$, bounding the values of $\mathcal{V}^1_{PI}(x,v)$ from below when PI is deactivated via the corresponding outport. Here, we can use the unbounded reach set approximation we already computed to bound the values of x and a τ-bounded reach set computation on $\varphi^{exit}_{\alpha_i}$ to bound the values of v. For simplicity, we use constant functions $\mathcal{V}^1_{\alpha_1}(v) = \mathcal{V}^1_{\alpha_2}(v) = 2400000$ here.

The same procedure can then be applied to the other basic components *BRAKE* and *ACCELERATE*. Here, possible Lyapunov functions are the linear functions

$$\mathcal{V}^1_{BRAKE}(v) = v$$

and

$$\mathcal{V}^1_{ACCELERATE}(v) = -v.$$

Next, we give the verification conditions for composed components, which are the result of a transition composition of either basic or composed components. For all subcomponents involved in such a transition composition, the principle of information hiding stipulates that we do not have access to its internals, but only an external view of the subcomponent. This view consists of the information in a subcomponent's external interface specification, of which we assume that it is fulfilled, and the induction invariants listed above. For the Lyapunov function projections given in the induction invariant, the verification conditions imply the satisfaction of a nonincreasingness condition upon a component switch. To

[2] `http://isat.gforge.avacs.org/index.html`

guarantee safety, it is sufficient to require that the system states at which a switch can occur always fulfill the entry condition of the component we are switching to. Furthermore, we need to enforce constraints that make sure the composed component does not guarantee a behavior that the constituent subcomponents cannot guarantee themselves. This involves the different timing variables and the system's promises on the dynamics.

In the following, we assume that each outport of a composed components is only connected to a single outport of a subcomponent. This is done purely to simplify the formal description of the verification conditions. If two subcomponent outports α_i are connected to one outport α of a composed component, one can, for the purpose of the analysis that follows, simply duplicate the outport α. Therefore, this is not a real limitation in the analysis.

Theorem 2 (Verification Conditions for Composed Components). *Let C be a composed component with $C = \mathcal{S}_{(\mathcal{P},\mathcal{Q})}(C_1, \ldots, C_n)$ with semantics I_C. Define the* parameter space *for the Lyapunov functions as $\mathcal{R} = \mathbb{R}^{r_{C_1}} \times \ldots \times \mathbb{R}^{r_{C_n}}$ and let $r_C > 0$. For a vector $\theta_C \in \mathcal{R}$, we refer to the subvector containing the entries pertaining to C_i as $\theta_{C|C_i} \in \mathbb{R}^{r_{C_i}}$. Define the predicate P_C^{Lyap} on the elements $\theta_{C|C_i}^{(j)}$ of a $\theta_C \in \mathcal{R}$ as*

$$
\begin{aligned}
P_C^{Lyap} := \forall i : (\forall 1 \leq j \leq r_C : \theta_{C|C_i}^{(j)} \geq 0 \wedge \exists 1 \leq j \leq r_{C_i} : \theta_{C|C_i}^{(j)} > 0) \wedge \\
\bigwedge \forall (\alpha, \{(\beta_1, g_1, \mathcal{A}_1), \ldots, (\beta_k, g_k, \mathcal{A}_k)\}, \cdot) \in \mathcal{P}, \forall 1 \leq i \leq k : \\
\forall X^{IO} \models reach(\varphi_{C(\alpha)}^{prom}, \varphi_\alpha^{exit} \wedge g_i, \tau) : \\
\sum_j \theta_{C|C(\alpha)}^{(j)} \mathcal{V}_\alpha^j(X^{IO}) \geq \sum_j \theta_{C|C(\beta_i)}^{(j)} \mathcal{V}_{\beta_i}^j(\mathcal{A}_i(X^{IO})).
\end{aligned}
$$

If:

1. $\forall i : I_{C_i} \parallel P \models SPEC_{C_i}$, *where* I_{C_i} *is the semantics of* C_i,
2. $\forall \alpha \in A_C^{out}, \beta \in A_C^{in} : \varphi_\beta^{entry} \implies \neg \varphi_\alpha^{alarm}$,
3. $\forall (\alpha, \{(\beta_1, g_1, \mathcal{A}_1), \ldots, (\beta_k, g_k, \mathcal{A}_k)\}, \{(\alpha_1, g_1'), \ldots, (\alpha_l, g_l')\}) \in \mathcal{P}$:
   (a) $\forall i : \lambda_{\beta_i} \leq \mu_\alpha$
   (b) $\forall i : \mu_{\alpha_i} \leq \mu_\alpha \wedge \Delta_\alpha \leq \Delta_{\alpha_i}$
   (c) $\forall i : reach(\varphi_{C(\alpha)}^{prom}, \varphi_\alpha^{exit} \wedge g_i, \tau) \implies \varphi_{\beta_i}^{entry}[\mathcal{A}_i]$
   (d) $\forall i : \varphi_\alpha^{exit} \wedge g_i' \implies \varphi_{\alpha_i}^{exit}$
   (e) $\forall i : \varphi_\alpha^{alarmOn} \implies \varphi_{\alpha_i}^{alarmOn}$
4. $\forall \beta \in A_C^{in} : \lambda_\beta \geq \lambda_{\mathcal{Q}(\beta)}$
5. $\bigvee_i \varphi_{C_i}^{assm} \implies \varphi_C^{assm}$
6. $\bigvee_i \varphi_{C_i}^{prom} \implies \varphi_C^{prom}$
7. $\forall \beta \in A_C^{in} : \varphi_\beta^{entry} \implies \varphi_{\mathcal{Q}(\beta)}^{entry}$
8. *require that* P_C^{Lyap} *is satisfiable and that the solution, given as a valuation of* θ_C, *satisfies*

$$
\Delta_C^{stable} \geq \frac{1}{rate_C(\theta_C)} \ln \left(\frac{\max_i \sum_j \theta_{C|C_i}^{(j)} c_{entry}(C_i, j)}{\min_i \sum_j \theta_{C|C_i}^{(j)} c_{stab}(C_i, j)} \right),
$$

where

$$rate_C(\theta_C) = \min_i \frac{\sum_{j=1}^{r_{C_i}} \theta^{(j)}_{C|C_i} k_3(C_i, j)}{\sum_{j=1}^{r_{C_i}} \theta^{(j)}_{C|C_i} k_2(C_i, j)},$$

then $I_C \parallel P \models SPEC_C$.

Again, the same induction invariants as for basic components need to be provided. The Lyapunov function projections on in- and outports are derived from some solution θ_C of the constraint system P_C^{Lyap}, each of which represents a valid (but not explicitly specified) continuous energy function of the system, which cannot increase at its discontinuities. Remember that the θ_C serve as multipliers in a conic combination of Lyapunov functions. To prove stability of C, it is again sufficient to find a single θ_C solving P_C^{Lyap} that fulfills the timing constraint. With the same argument as for basic components, we however allow for the computation of r_C such solutions $\tilde{\theta}_{C,j}$. The Lyapunov function projections attached to the inports (outports) are the Lyapunov function projections of the connected inports (outports) of subcomponents C_i, but weighted by the solutions $\tilde{\theta}_{C,j}$. Remember that we assumed that each outport of C is only connected to one outport of a C_i – in this case, there is a one-to-one mapping of Lyapunov function projections. Due to the conic property of Lyapunov functions, any conic combination of the $\tilde{\theta}_{C,j}$ will again represent a solution of P_C^{Lyap}, which in turn implies the existence of a discontinuous, but decreasing energy function so that the propagation of information to the next higher level again works correctly. The rates and constants of the composed component can also be computed from the rates of the subcomponents C_i. Again, they are weighted by θ_C for each component, and then the worst case over all subcomponents is taken for the transition composition C.

Definition 16. *For each composed component, the following* induction invariants *need to be provided, to facilitate further composition:*

- *for each* $1 \leq j \leq r_C$ *where* r_C *is a positive integer:*
  - *for all outports* α *a function* $\mathcal{V}^j_\alpha : \mathbb{R}^{|Var^{in}_C|+|Var^{out}_C|} \to \mathbb{R}$
  - *for all inports* β *a function* $\mathcal{V}^j_\beta : \mathbb{R}^{|Var^{in}_C|+|Var^{out}_C|} \to \mathbb{R}$

  such that there exists a $\tilde{\theta}_{C,j} \models P_C^{Lyap}$ *with subvectors* $\tilde{\theta}_{C|C_i,j}$ *for each subcomponent* C_i*, and*
  - $\mathcal{V}^j_\alpha = \sum_{k=1}^{r_{C(\alpha')}} \tilde{\theta}^{(k)}_{C|C(\alpha'),j} \mathcal{V}^j_{\alpha'}$*, where* α' *is the unique outport connected to* α*,*
  - $\mathcal{V}^j_\beta = \sum_{k=1}^{r_{C(\mathcal{Q}(\beta))}} \tilde{\theta}^{(k)}_{C|C(\mathcal{Q}(\beta)),j} \mathcal{V}^j_{\mathcal{Q}(\beta)}$.
- *for each* $1 \leq j \leq r_C$
  - $k_2(C, j) = \max_i \sum_{k=1}^{r_{C_i}} \tilde{\theta}^{(k)}_{C|C_i,j} k_2(C_i, k)$,
    and $k_3(C, j) = \min_i \sum_{k=1}^{r_{C_i}} \tilde{\theta}^{(k)}_{C|C_i,j} k_3(C_i, k)$
  - $c_{entry}(C, j) = \max_i \sum_{k=1}^{r_{C_i}} \tilde{\theta}^{(k)}_{C|C_i,j} c_{entry}(C_i, k)$,
    and $c_{stab}(C, j) = \min_i \sum_{k=1}^{r_{C_i}} \tilde{\theta}^{(k)}_{C|C_i,j} c_{stab}(C_i, k)$

Proof (of Theorem 2). We have to show that $I_C \parallel P \models SPEC_C$ holds when the assumption of the theorem are given for I_C. To prove that we have to show that all of the requirements given in Def. 13 hold.

(A):

$$\begin{aligned} Var_{I_C}^{out} &= \bigcup_i Var_{C_i}^{out} \cup Var_{H_C}^{out} \cup Var_{H_\mathcal{P}}^{out} \cup Var_{H_\mathcal{Q}}^{out} && \text{// Def. 11} \\ &\supseteq Var_{C_1}^{out} \cup \{fail_C, active_C\} \\ &\cup \{take_\beta \mid \beta \in A_C^{in}\} \cup \{b_\alpha, switch_\alpha \mid \alpha \in A_C^{out}\} \\ &= Var_C^{out} \cup C_C^{out}. \end{aligned}$$

(B):

$$\begin{aligned} Var_{I_C}^{in} &= \left(\bigcup_i Var_{C_i}^{in} \cup Var_{H_C}^{in} \cup Var_{H_\mathcal{P}}^{in} \cup Var_{H_\mathcal{Q}}^{in} \right) \setminus Var_{I_C}^{out} \\ &\supseteq \left(\begin{array}{c} Var_{C_1}^{in} \cup \{active_C, suspend_C\} \\ \cup \{start_\beta, c_\beta \mid \beta \in A_C^{in}\} \cup \{take_\alpha \mid \alpha \in A_C^{out}\} \end{array} \right) \setminus Var_{I_C}^{out} \\ &= Var_{C_1}^{in} \cup C_C^{in} \\ &= Var_C^{in} \cup C_C^{in}. \end{aligned}$$

(C) follows directly from the assumption 2 of the theorem.

(D) follows from the assumptions 1, 6 and 8 in conjunction with the properties of the component's interface specifications:

$$\begin{aligned} \varphi_C^{entry} &\Longrightarrow \bigvee_\beta \varphi_\beta^{entry} \\ &\Longrightarrow \bigvee_\beta \varphi_{\mathcal{Q}(\beta)}^{entry} && \text{// Assm. 8} \\ &\Longrightarrow \bigvee_\beta \varphi_{C(\mathcal{Q}(\beta))}^{assm} && \text{// Assm. 1, (D) for } C_i \\ &\Longrightarrow \bigvee_i \varphi_{C_i}^{assm} \\ &\Longrightarrow \varphi_C^{assm} && \text{// Assm. 6} \end{aligned}$$

$$\begin{aligned} \varphi_C^{entry} &\Longrightarrow \bigvee_\beta \varphi_{\beta,C}^{entry} \\ &\Longrightarrow \bigvee_\beta \varphi_{\mathcal{Q}(\beta)}^{entry} && \text{// Assm. 8} \\ &\Longrightarrow \bigvee_i \varphi_{C_i}^{entry} \\ &\Longrightarrow \varphi_P^{safe} && \text{// (D) for } C_i \end{aligned}$$

(E): This holds due to the construction of H_T. All transitions setting $active_C$ require that there is a β such that $start_\beta \wedge \varphi_\beta^{entry}$ holds.

(F): When φ_α^{alarm} becomes true, a connected $\alpha_i \in A_{C_i}^{out}$ exists. For this outport we have this property already, hence we know that b_{α_i} is set because $\varphi_\alpha^{alarm} \Longrightarrow \varphi_{\alpha_i}^{alarm}$ is given in 3. As $H_\mathcal{P}$ forwards b_{α_i} to b_α we know that this holds.

(G): This holds again due to $H_\mathcal{P}$ which forwards both the b_{α_i} from a component C_i to b_α and the corresponding $take_\alpha$ to $take_{\alpha_i}$. Since C_i satisfies (G) we know that b_{α_i} holds for at least μ_{α_i} seconds before it can be reset without a $take_{\alpha_i}$ signal. As $\mu_{\alpha_i} \leq \mu_\alpha$ is given in assumption 3 we can conclude (G) for the outport α.

(H), (I): Both properties are satisfied because of H_T's construction.

(J): This holds because $H_\mathcal{Q}$ ensures that a rising edge of c_β produces a rising edge of $c_{\mathcal{Q}(\beta)}$ and with (J) for this component we can conclude that there is a $take_{\mathcal{Q}(\beta)}$ after at most $\lambda_{\mathcal{Q}(\beta)}$ seconds. This is forwarded by $H_\mathcal{Q}$. With $\lambda_{\mathcal{Q}(\beta)} \leq \lambda_\beta$ of assumption 4 this property holds.

(K): Again this holds because $H_\mathcal{Q}$ forwards the signals c_β and $take_{\mathcal{Q}(\beta)}$ to $c_{\mathcal{Q}(\beta)}$ resp. $take_\beta$ appropriately and the component itself satisfies this requirement.

(L): This given because $H_\mathcal{Q}$ just forwards both c_β (to $c_{\mathcal{Q}(\beta)}$) and $take_{\mathcal{Q}(\beta)}$ (to $take_\beta$). As (L) holds for the C_i the inport $\mathcal{Q}(\beta)$ belongs to, this property can be transferred directly to C.

(M): This is satisfied because the composition cannot be activated by a $start_\beta$ while $\neg\varphi_\beta^{entry}$ holds. The reason is that in this case H_T produces a $fail_C$ signal.

(N): Whenever C is active then exactly one of its components C_i is active. Exploiting (N) for C_i we get $\varphi_P^{safe} \wedge \varphi_{C_i}^{assm} \wedge \varphi_{C_i}^{prom}$. With the assumptions 6 and 7 we get the $\varphi_P^{safe} \wedge \varphi_C^{assm} \wedge \varphi_C^{prom}$.

(O): Let $X(t)$ be a the projection of a trajectory of $I_C \parallel P$ onto $Var_C \cup Var_P$. Assume, without loss of generality, that $active_C$ receives a rising edge at time 0. Let (C_l) be the finite or infinite sequence of active subcomponents of C along trajectory $X(t)$, in order of activation, and let (t_l) be the sequence of activation times. If (t_l) is finite, set $t_{l+1} = \infty$. Since $\tau > 0$, we know that (t_l) is strictly increasing. For $l > 1$, define $prev(X(t_l)) = \lim_{t \uparrow t_l} X(t)$. For a time t, let $C(t)$ be the unique subcomponent that is active at time t.

Inductively define the functions $\mathcal{V}_{C'}(\theta_{C'}, X(t))$ and $\mathcal{V}_{C'}^j(X(t))$ as follows:

- for a basic component C', $\mathcal{V}_{C'}(\theta_{C'}, X(t)) := \sum_k \theta_{C'}^{(k)} \mathcal{V}_{C'}^k(X(t))$, where $\mathcal{V}_{C'}^k(X(t))$ is the function as given in the verification conditions for basic components.
- for a composed component C', $\mathcal{V}_{C'}(\theta_{C'}, X(t)) := \sum_k \theta_{C'|C'(t)}^{(k)} \mathcal{V}_{C'(t)}^k(X(t))$, and $\mathcal{V}_{C'}^j(X(t)) = \sum_k \tilde{\theta}_{C'|C'(t),j}^{(k)} \mathcal{V}_{C'(t)}^k(X(t))$

Now, show that for all $\theta_C \models P_C^{Lyap}$, the following hold:

1) for all $t \in \mathbb{R} \cup \{\infty\}$ with $\forall t' \in [0, t) : active_C(t')$:

$$\mathcal{V}_C(\theta_C, X(t)) \leq e^{-rate_C(\theta_C)t} \mathcal{V}_C(\theta_C, X(0)).$$

2) $\mathcal{V}_C(\theta_C, X) \leq \min_l \sum_j \theta^{(j)}_{C|C_l} c_{stab}(C_l, j) \implies \varphi^{stable}_P$

3) $\mathcal{V}_C(\theta_{C|C_l}, X(0)) \leq \max_l \sum_j \theta^{(j)}_{C|C_l} c_{entry}(C_l, j)$

First, show that 1) holds for C. The proof is by induction. For basic components C_l, 1) holds, which was shown in the proof of Theorem 1, For composed components C_l, assume per induction that 1) holds. In particular, that gives us

$$\mathcal{V}^j_{C_l}(X(t)) \leq e^{-rate_{C_l}(\tilde{\theta}_{C,j})t} \mathcal{V}^j_{C_l}(X(0)).$$

Define

$$\mathcal{V}_{C_l}(\theta_{C|C_l}, X) := \sum_j \theta^{(j)}_{C|C_l} \mathcal{V}^j_{C_l}(X)$$

Then, per linear combination of the Lyapunov constraints, $\forall \theta_C \models P^{Lyap}_C, \forall t' \in [t_l, t_{l+1})$:

$$\begin{aligned}
\mathcal{V}_{C_l}(\theta_{C|C_l}, X(t')) \leq & \quad e^{-\sum_j \theta^{(j)}_{C|C_l} rate_{C_l}(\tilde{\theta}_{C,j})t'} \mathcal{V}_{C_l}(\theta_{C|C_l}, X(t_l)) \\
= & \quad e^{-\sum_j \theta^{(j)}_{C|C_l} \frac{k_3(C_l,j)}{k_2(C_l,j)}t'} \mathcal{V}_{C_l}(\theta_{C|C_l}, X(t_l)) \\
\leq & \quad e^{-rate_C(\theta_C)t'} \mathcal{V}_{C_l}(\theta_{C|C_l}, X(t_l))
\end{aligned}$$

Therefore, we know that $\mathcal{V}_C(\theta_C, X(t))$ will decrease according to $rate_C(\theta_C)$, whenever no component switch occurs. Now we need to make sure that the switches will never result in an increase of $\mathcal{V}_C(\theta_C, X(t))$ at any t_l. Observe that there exists a transition $(\alpha, \{(\beta_1, g_1, \mathcal{A}_1), \ldots, (\beta_k, g_k, \mathcal{A}_k)\}, \cdot) \in \mathcal{P}$, such that, by closedness of $reach(\varphi^{prom}_{C(\alpha)}, \varphi^{exit}_\alpha \wedge g_i, \tau)$, for all t_l, we have $prev(X(t_l)) \in reach(\varphi^{prom}_{C(\alpha)}, \varphi^{exit}_\alpha, \tau)$. Satisfaction of P^{Lyap}_C implies that, for all transitions:

$$\forall 1 \leq j \leq k : \forall X^{IO} \models reach(\varphi^{prom}_{C(\alpha)}, \varphi^{exit}_\alpha \wedge g_j, \tau) :$$

$$\sum_i \theta^{(i)}_{C|C(\alpha)} \mathcal{V}^i_\alpha(X^{IO}) \geq \sum_i \theta^{(i)}_{C|C(\beta_j)} \mathcal{V}^i_{\beta_j}(\mathcal{A}_j(X^{IO})),$$

which gives us

$$\forall t_l, l > 1 : \mathcal{V}_{C_{l-1}}(\theta_{C|C_{l-1}}, prev(X(t_l))) \geq \sum_j \theta^{(j)}_{C|C_{l-1}} \mathcal{V}_\alpha(prev(X^{IO}(t_l)))$$

$$\geq \sum_j \theta^{(j)}_{C|C_l} \mathcal{V}_\beta(\mathcal{A}_j(X^{IO}(t_l))) \geq \mathcal{V}_{C_l}(\theta_{C|C_l}, X(t_l)).$$

Together, this results in the desired inequality:

$$\forall t' \in [0, t) : active_C(t') \implies \mathcal{V}_C(\theta_C, X(t)) \leq e^{-rate_C(\theta_C)t} \mathcal{V}_C(\theta_C, X(0))$$

Now show 2), again by induction. For basic components C_l, we know that $\mathcal{V}^j_{C_l}(X) \leq c_{stab}(C_l, j) \implies \varphi^{stable}_P$. For composed components C_l, assume that 2) holds, and this follows directly, by setting $\theta_{C_l} = \tilde{\theta}_{C_l,j}$. If

$$\mathcal{V}_C(\theta_C, X(t)) \leq \min_l \sum_j \theta^{(j)}_{C|C_l} c_{stab}(C_l, j),$$

then this implies

$$\sum_j \theta^{(j)}_{C|C(t)} \mathcal{V}^j_{C(t)}(X(t)) \leq \sum_j \theta^{(j)}_{C|C(t)} c_{stab}(C(t), j)$$

For this inequality to hold, there must exists at least one j with

$$\mathcal{V}^j_{C'(t)}(X(t)) \leq c_{stab}(C(t), j),$$

which implies that $X(t) \models \varphi^{stable}_P$.

The proof of 3) is also done inductively. Again, for basic components C_l we know that $\mathcal{V}^j_{C_l}(X_l) \leq c_{entry}(C_l, j)$. For composed components, this property again follows, if we assume that 3) holds for C_l. Then

$$\mathcal{V}_C(\theta_C, X(0)) = \sum_j \theta^{(j)}_{C|C(0)} \mathcal{V}^j_{C(0)}(X(0))$$

$$\leq \sum_j \theta^{(j)}_{C|C(0)} c_{entry}(C(0), j) \leq \max_l \sum_j \theta^{(j)}_{C|C_l} c_{entry}(C_l, j)$$

Finally, we will use 1), 2) and 3) to prove the desired property. We know there exists a θ_C with:

$$\Delta^{stable}_C \geq \frac{1}{rate_C(\theta_C)} \ln \left(\frac{\max_i \sum_j \theta^{(j)}_{C|C_i} c_{entry}(C_i, j)}{\min_i \sum_j \theta^{(j)}_{C|C_i} c_{stab}(C_i, j)} \right),$$

$$\begin{aligned}
&\mathcal{V}_C(\theta_C, X(\Delta^{stable}_C)) \\
&\leq \quad \exp\left(\frac{-rate_C(\theta_C)}{rate_C(\theta_C)} \ln \left(\frac{\max_i \sum_j \theta^{(j)}_{C|C_i} c_{entry}(C_i, j)}{\min_i \sum_j \theta^{(j)}_{C|C_i} c_{stab}(C_i, j)} \right) \right) \mathcal{V}_C(\theta_C, X(0)) \\
&= \quad \exp\left(\ln \left(\frac{\min_i \sum_j \theta^{(j)}_{C|C_i} c_{stab}(C_i, j)}{\max_i \sum_j \theta^{(j)}_{C|C_i} c_{entry}(C_i, j)} \right) \right) \mathcal{V}_C(\theta_C, X(0)) \\
&= \quad \frac{\min_i \sum_j \theta^{(j)}_{C|C_i} c_{stab}(C_i, j)}{\max_i \sum_j \theta^{(j)}_{C|C_i} c_{entry}(C_i, j)} \mathcal{V}_C(\theta_C, X(0))
\end{aligned}$$

This implies that $\mathcal{V}_C(\theta_C, X(0)) \leq \max_i \sum_j \theta^{(j)}_{C|C_i} c_{entry}(C_i, j)$, and we obtain

$$\mathcal{V}_C(\theta_C, X(\Delta^{stable}_C)) \leq \min_i \sum_j \theta^{(j)}_{C|C_i} c_{stab}(C_i, j).$$

which implies that $X(\Delta^{stable}_C) \models \varphi^{stable}_P$. Since $\mathcal{V}_C(\theta_C, X(t))$ is nonincreasing, we obtain

$$\Box(active_C \implies ((\Diamond_{\Delta^{stable}_C}(\Box \varphi^{stable}_C)) \,\mathsf{UNLESS}\, (\neg active_C)).$$

(P),(Q): Immediately clear by the construction of the transitions of $H_{\mathcal{P}}$ setting $switch_\alpha$.

(R): By the construction of $H_{\mathcal{P}}$ we know that whenever a $switch_\alpha$ becomes true, this happens at the same time point as a $switch_{\alpha'}$ of a component C_i becomes true with α' being connected to α with guard g. Since (R) holds for this α' we know that $\varphi^{exit}_{\alpha'}$ must holds. The guard g must also be satisfied because otherwise the $switch_\alpha$ would not be set by $H_{\mathcal{P}}$. In sum, we have $\varphi^{exit}_{\alpha'} \wedge g$ which allows us to conclude that φ^{exit}_{α} must hold because of assumption 3.

(S): If C fails, then this can be due to a component C_i that failed. In this case we know that (S) holds for C_i and we can conclude that there is an $\alpha_i \in A^{out}_{C_i}$ such that b_{α_i} was set at least Δ_{α_i} time units before the failure or there was an unsatisfied entry condition $\varphi^{entry}_{\beta_i}$ of an inport β_i of C_i. In case of a time-out we know that the automaton $H_{\mathcal{P}}$ forwards b_{α_i} to an outport $\alpha' \in A^{out}_{C}$ the whole time by its construction. With assumption 3 we known that $b_{\alpha'}$ was set at least $\Delta_{\alpha'}$ before $fail_C$ is set. In case of an unsatisfied entry condition $\varphi^{entry}_{\beta_i}$ we know by 7 that $\varphi^{entry}_{\beta'}$ with $\mathcal{Q}(\beta') = \beta_i$ is a stronger condition. Hence, if this $\varphi^{entry}_{\beta'}$ is not met when $start_{\beta'}$ is set, then H_T will set $fail_C$. If it is satisfied, then follows that $\varphi^{entry}_{\mathcal{Q}(\beta')}$ is satisfied, too. Hence, the component C_i cannot set $fail_{C_i}$.

(T): Clear as $fail_C$ is the disjunction of all $fail_{C_i}$ and since all C_i satisfy (T) this property holds for $fail_C$, because there is no transition in H_T, $H_{\mathcal{P}}$, and $H_{\mathcal{Q}}$ resetting $fail_C$.

Example (cont.) To verify that the component C_2 fulfills its interface specification, we have to prove the verification conditions in Theorem 2. We have already shown verification condition 1. For verification condition 8, we again need to find Lyapunov function parameters. Since we just computed single Lyapunov functions for PI and $ACCELERATE$, the vector θ_{C_2} is of dimension 2, with one coefficient $\theta^1_{C_2}$ serving as a multiplier for the Lyapunov function pertaining to PI and the other coefficient $\theta^2_{C_2}$ for the Lyapunov function pertaining to $ACCELERATE$. The constraint system calls for the identification of values for θ_{C_2}, such that, whenever a new component is activated, there is no increase in the Lyapunov function value.

In particular, the constraint system looks like this:

$$P^{Lyap}_{C_2} = (\theta^1_{C_2} \geq 0 \wedge \theta^2_{C_2} \geq 0) \vee (\theta^1_{C_2} > 0 \vee \theta^2_{C_2} > 0)$$

$$\wedge \forall v \models reach(\varphi^{prom}_{PI}, \varphi^{exit}_{\alpha^2_{PI}}, \tau) : \theta^1_{C_2} \mathcal{V}^1_{\alpha^2_{PI}}(v) \geq \theta^2_{C_2} \mathcal{V}^1_{\beta_{ACCELERATE}}(v)$$

$$\wedge \forall v \models reach(\varphi^{prom}_{ACCELERATE}, \varphi^{exit}_{\alpha_{ACCELERATE}}, \tau) : \theta^2_{C_2} \mathcal{V}^1_{\alpha_{ACCELERATE}}(v) \geq \theta^1_{C_2} \mathcal{V}^1_{\beta_{PI}}(v)$$

Once the reach sets have been computed or overapproximated, this is again a constraint system that can be solved with SDP methods. A possible solution is $\theta^1_{C_2} = 1$ and $\theta^2_{C_2} = 150000$.

From this, we can now check whether the inequality on $\Delta_{C_2}^{stable}$ also holds, which it does, since

$$\Delta_{C_2}^{stable} = 300$$

$$\geq \frac{1}{rate_{C_2}(\theta_{C_2})} \ln \left(\frac{\max\{\theta^1_{C_2} c_{entry}(PI,1), \theta^2_{C_2} c_{entry}(ACCELERATE,1)\}}{\min\{\theta^1_{C_2} c_{stab}(PI,1), \theta^2_{C_2} c_{stab}(ACCELERATE,1)\}} \right)$$

$$= \frac{1}{rate_{C_2}(\theta_{C_2})} \ln \left(\frac{4700000}{50000} \right) = 252.4$$

where

$$rate_{C_2}(\theta_{C_2}) = \min \left\{ \frac{0.018}{1}, \frac{10000}{200000} \right\} = 0.018$$

In case this inequality would be violated, we could try to find a better solution to the above constraint system, since SDP based methods inherently support the specification of objective functions on the parameters.

The remaining verification conditions are either simple scalar inequalities or implications between non-parametric predicates and therefore easily verified.

5 Conclusion

We have presented a design methodology for hybrid systems, which supports component based incremental design processes and prepares the way for a fully distributed implementation of such controllers as relevant for Autosar based automotive development processes. In addressing this industrial need, we have developed a mathematical theory for incremental compositional verification of both safety and stability properties of hybrid controllers based on the key concept of hybrid interface specifications, and provided verification conditions both for basic components as well as hierarchically composed open systems. Concepts, methodology, and incremental verification have been illustrated using a simple automatic cruise control system as a running example.

While the presented methodology and verification approach is self-contained and of value in itself, we see several extensions which will be addressed in further work.

First, we would like to close the gap between idealized plant models and physical plants by a notion of robust plant refinement, which allows to measure degrees of deviation still tolerated without endangering the satisfaction of hybrid interface specifications. Much as safety analysis methods distinguish between nominal behavior and failure behaviors which nevertheless are still restricted by failure hypothesis, we expect to be able to characterize conditions under which we can then establish under non-nominal behavior within the failure hypothesis, that alarms are raised in time.

Secondly, we will extend the approach to handle parallel composition of components.

Acknowledgments

This research is carried out in the context of the transregional collaborative research project AVACS (www.avacs.org) – we thank our colleagues in the project area on Hybrid Systems for many inspiring discussions, and Uwe Waldmann for providing efficient methods for solving time bounded reachability problems appearing as verification conditions in Chapter 4.

In Memoriam

How can I express my thanks to Amir for all the inspiration he has given, for the sharing of concerns on many issues in life and politics, for the fun and the concerts, for his keen questioning, probing, for offering clean and concise perspectives on what seemed to be a jungle of problems?

He was to me like a scientific father, who was always open to new ideas, gently pushing in his humble and friendly style our thoughts in the right direction.

His impact on science is overwhelming, so is his impact on me.

Thanks, Amir

Werner

References

[BEFB94] Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear Matrix Inequalities in System and Control Theory. Society for Industrial and Applied Mathematics (SIAM) (1994)

[Bor99] Borchers, B.: CSDP, a C library for semidefinite programming. Optimization Methods and Software 10(1), 613–623 (1999), `https://projects.coin-or.org/Csdp/`

[DMO+07] Damm, W., Mikschl, A., Oehlerking, J., Olderog, E.-R., Pang, J., Platzer, A., Segelken, M., Wirtz, B.: Automating Verification of Cooperation, Control, and Design in Traffic Applications. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 115–169. Springer, Heidelberg (2007)

[DPJ09] Damm, W., Peikenkamp, T., Josko, B.: Contract Based ISO CD 26262 Safety Analysis. In: SAE World Congress – Session on Safety-Critical Systems (2009)

[Fre05] Frehse, G.: PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)

[Fre06] Frehse, G.: On Timed Simulation Relations for Hybrid Systems and Compositionality. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 200–214. Springer, Heidelberg (2006)

[Fre08] Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. STTT – International Journal on Software Tools for Technology Transfer 10(3), 263–279 (2008)

[HMP01] Henzinger, T., Minea, M., Prabhu, V.S.: Assume-Guarantee Reasoning for Hierarchical Hybrid Systems. In: di Benedetto, M., Sangiovanni-Vincentelli, A. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 275–290. Springer, Heidelberg (2001)

[JBS07] Jha, S., Brady, B.A., Seshia, S.A.: Symbolic Reachability Analysis of Lazy Linear Hybrid Automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 241–256. Springer, Heidelberg (2007)

[JMM08] Josko, B., Ma, Q., Metzner, A.: Designing Embedded Systems using Heterogeneous Rich Components. In: Proceedings of the INCOSE International Symposium (2008)

[Lya07] Lyapunov, M.A.: Problème général de la stabilité du movement. Ann. Fac. Sci. Toulouse 9, 203–474 (1907) (Translation of a paper published in Comm. Soc. Math. Kharkow, 1893, reprinted Ann. Math. Studies No. 17, Princeton Univ. Press, 1949)

[OT09] Oehlerking, J., Theel, O.: Decompositional construction of Lyapunov functions for hybrid systems. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 276–290. Springer, Heidelberg (2009)

[Pet99] Pettersson, S.: Analysis and Design of Hybrid Systems. PhD thesis, Chalmers University of Technology, Gothenburg (1999)

[PJ04] Prajna, S., Jadbabaie, A.: Safety Verification of Hybrid Systems Using Barrier Certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)

[RPS99] Romanko, O., Pólik, I., Sturm, J.F.: Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones (1999)

[Sta01] Stauner, T.: Systematic Development of Hybrid Systems. PhD thesis, Technische Universität München (2001)

[Sta02] Stauner, T.: Discrete-time refinement of hybrid automata. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 407–420. Springer, Heidelberg (2002)

[TPL04] Tabuada, P., Pappas, G.J., Lima, P.: Compositional abstractions of hybrid control systems. Discrete Event Dynamic Systems 14(2) (2004)

Mildly Context-Sensitive Languages via Buffer Augmented Pregroup Grammars

Daniel Genkin, Nissim Francez*, and Michael Kaminski

Department of Computer Science
Technion– Israel Institute of Technology
Haifa 32000
Israel
{danielg3,francez,kaminski}@cs.technion.ac.il

Abstract. A family of languages is called *mildly context-sensitive* if
- it includes the family of all ϵ-free context-free languages;
- it contains the languages
  - $\{a^n b^n c^n : n \geq 1\}$ – *multiple agreement*,
  - $\{a^m b^n c^m d^n : m, n \geq 1\}$ – *crossed dependencies*, and
  - $\{ww : w \in \Sigma^+\}$ – *reduplication*;
- all its languages are *semi-linear*; and
- their membership problem is decidable in polynomial time.

In our paper we introduce a new model of computation called *buffer augmented pregroup grammars* that defines a family of mildly context-sensitive languages. This model of computation is an extension of Lambek pregroup grammars with a variable symbol – the (*buffer*) and is allowed to make an arbitrary substitution from the original pregroup to the variable. This increases the pregroup grammar generation power, but still retains the desired properties of mildly context-sensitive languages such as semi-linearity and polynomial parsing. We establish a strict hierarchy within the family of mildly context-sensitive languages defined by buffer augmented pregroup grammars. In this hierarchy, the hierarchy level of the family language depends on the allowed number of occurrences of the variable in lexical category assignments.

Keywords: Formal language theory, mildly context-sensitive languages, pregroup grammars.

1 Introduction

Since their introduction in [7], *pregroup grammars* have attracted a lot of attention, giving rise to a *radically lexicalized* theory of formal (and, of course, natural) languages. The theory of formal languages partly developed from an abstraction originating in the syntax of natural languages, namely *constituency* (known also as *phrase-structure*). By this abstraction, *rewrite-rules* formed the basis of formal grammar, culminating in their classification by the well-known

* In memory of Amir Pnueli, a teacher and a friend.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 144–166, 2010.

Chomsky hierarchy. To their success in computer science contributed the realization of their suitability for specifying the syntax of programming languages, after they were abandoned as a tool for natural language syntax specification. The theory matured even more when the grammar classification was complemented by the classification of various classes of automata corresponding to the various classes of the Chomsky hierarchy of grammar formalisms, see [6], a standard reference to the area.

This standard approach to formal languages has certain characteristics, challenged by modern *computational linguistics*, summarized below.

- Terminals are *atomic* structureless entities, that can only be compared for equality.
- Similarly, *non-terminals* (better called *categories*) are also atomic, structureless entities, representing sets of strings (of terminals).
- Language variation (over some fixed set of terminals) is determined by *grammar variation*, which was taken to mean *variation in the rewrite rules.*
- String *concatenation* is the *only* admissible syntactic operation.

Modern computational linguistics is the source of a different abstraction, based on a different view of language theory known as *radical lexicalism.* There are several radically-lexicalized linguistic theories for natural language (we omit references, as the focus here is on *formal* languages), having the following characteristics.

- Terminals are *informative* entities, that have their own properties, determined by a *lexicon*, mapping terminals to "pieces of information" about them. The lexicon is the "heart" of a grammar. Most often, those pieces of information are taken as (finite) sets of complex categories.
- Similarly, *categories* are also structured entities, representing sets of strings (of terminals).
- Language variation (over some fixed set of terminals) is determined by *lexicon variation.* There is a *universal grammar* (common to all languages) that extends the lexicon by attributing categories to strings too, controlling the combinatorics of strings based on their categories.

There are variants that admit other syntactic operations besides concatenation. We will assume here that concatenation is maintained as the only operation.

Buszkowski [2] establishes the weak generative equivalence between pregroup grammars and context-free grammars. On the other hand, motivated by the syntactic structure of natural languages, computational linguists became interested in a family of languages that became to be known as *mildly context-sensitive* languages, that on the one hand transcend context-free languages in containing *multiple agreement* ($\{a^n b^n c^n : n \geq 1\}$), *crossed dependencies* ($\{a^m b^n c^m d^n : m, n \geq 1\}$), and *reduplication* ($\{ww : w \in \Sigma^+\}$), but on the other hand have *semi-linearity* [8] and their membership problem is decidable in polynomial time (in the length of the input word). Several formalisms for grammar specification are known to converge to the same class [9], namely to mildly context-sensitive languages.

In this paper, we explore a mild extension of pregroup grammars, obtained by adding to the underlying (free) pregroup a new element – the buffer, that is a lower bound on some set of elements of the underlying free pregroup, cf. [1]. We establish the main properties of this class of languages, namely semi-linearity and polynomial parsability.

The paper is organized as follows. In Section 2 we review the standard definition of pregroups and grammars based on them. Then, in Section 3 we define buffer augmented pregroup grammars and show that they are powerful enough to generate the characteristic mildly context-sensitive languages. In Section 4 we prove the pumping lemma for the languages generated by buffer augmented pregroup grammars. Sections 5 and 6 deal with complexity issues of languages generated by buffer augmented pregroup grammars. In Section 7 we establish a strict hierarchy in the class of these languages and in Section 8 we extend our model of computation to a number of buffers. Finally, Section 9 contains some concluding remarks.

2 Pregroups and Pregroup Grammars

In this section we recall the definition of pregroup grammars.

Definition 1. *A* pregroup *is a tuple* $\mathcal{P} = \langle \boldsymbol{G}, \leq, \circ, \ell, r, 1\rangle$*, such that* $\langle \boldsymbol{G}, \leq, \circ, 1\rangle$ *is a* partially-ordered monoid,[1] *i.e., satisfying*

(mon) *if* $A \leq B$*, then* $CA \leq CB$ *and* $AC \leq BC$

and ℓ*,*r *are unary operations (left/right inverses/adjoints) satisfying*

(pre) $A^\ell A \leq 1 \leq AA^\ell$ *and* $AA^r \leq 1 \leq A^r A$.

The following equalities can be shown to hold in any pregroup.

$$1^\ell = 1^r = 1,\ A^{\ell r} = A^{r\ell} = A,\ (AB)^\ell = B^\ell A^\ell,\ (AB)^r = B^r A^r.$$

Also, **(mon)** together with **(pre)** yield

$$A \leq B \text{ if and only if } B^\ell \leq A^\ell \text{ if and only if } B^r \leq A^r. \tag{1}$$

Let $\langle \mathcal{B}, \leq\rangle$ be a (finite) partially ordered set. *Terms* are of the form $A^{(n)}$, where $A \in \mathcal{B}$ and n is an integer. The set of all terms generated by $\mathcal{B}$ is denoted by $\tau(\mathcal{B})$.

Categories[2] are finite strings (products) of terms. The set of all categories generated by $\mathcal{B}$ is denoted by $\kappa(\mathcal{B})$.

Remark 1. By definition, $\kappa(\mathcal{B}) = (\tau(\mathcal{B}))^*$.

Extend '$\leq$' to $\kappa(\mathcal{B})$ by setting it to the smallest quasi-partial-order[3] satisfying

[1] '$\circ$' is usually omitted.

[2] They are also called *types*.

[3] That is, $\leq$ is not necessarily antisymmetrical.

(con) $\gamma A^{(n)} A^{(n+1)} \delta \leq \gamma\delta$ (*contraction*)
(exp) $\gamma\delta \leq \gamma A^{(n+1)} A^{(n)} \delta$ (*expansion*)
and
(ind) $\gamma A^{(n)}\delta \leq \gamma B^{(n)}\delta$ if $\begin{cases} A \leq B \text{ and } n \text{ is even, or} \\ B \leq A \text{ and } n \text{ is odd} \end{cases}$ (*induced steps*) .

The following two inequalities can be easily derived from **(con)**, **(exp)**, and **(ind)**.

(gcon) $\gamma A^{(n)} B^{(n+1)} \delta \leq \gamma\delta$, if $\begin{cases} A \leq B \text{ and } n \text{ is even, or} \\ B \leq A \text{ and } n \text{ is odd} \end{cases}$ (*generalized contraction*)

and

(gexp) $\gamma\delta \leq \gamma A^{(n+1)} B^{(n)} \delta$, if $\begin{cases} A \leq B \text{ and } n \text{ is even, or} \\ B \leq A \text{ and } n \text{ is odd} \end{cases}$ (*generalized expansion*) .

Obviously, **(con)** and **(exp)** are particular cases of **(gcon)** and **(gexp)**, respectively. Conversely, **(gcon)** can be obtained as **(ind)** followed by **(con)**, and **(gexp)** can be obtained as **(exp)** followed by **(ind)**. Consequently, if $\alpha' \leq \alpha''$, then there exists a *derivation*

$$\alpha' = \gamma_0 \leq \gamma_1 \leq \cdots \leq \gamma_m = \alpha'', \quad m \geq 0$$

such that for each $i = 1, 2, \ldots, m$, $\gamma_{i-1} \leq \gamma_i$ is **(gcon)**, **(gexp)**, or **(ind)**.

Proposition 1. ([7, Proposition 2]) *If $\alpha' \leq \alpha''$ has a derivation of length m, then there exist categories β and γ such that*

- *$\alpha' \leq \beta$ by* **(gcon)** *only;*
- *$\beta \leq \gamma$ by* **(ind)** *only;*
- *$\gamma \leq \alpha''$ by* **(gexp)** *only; and*
- *the sum of the lengths of the above three derivations is at most m.*

Corollary 1. *If $\alpha' \leq \alpha''$ where α'' is a term, then, effectively, this can be established without expansions.*

Let $\alpha' \equiv \alpha''$ if and only if $\alpha' \leq \alpha''$ and $\alpha'' \leq \alpha'$. The equivalence-classes of '$\equiv$' form the *free pregroup generated by* $\langle \mathcal{B}, \leq \rangle$, where $1 = [\epsilon]_\equiv$, $[\alpha']_\equiv \circ [\alpha'']_\equiv = [\alpha'\alpha'']_\equiv$. Also, $[\alpha']_\equiv \leq [\alpha'']_\equiv$ if and only if $\alpha' \leq \alpha''$. The adjoints are defined as follows.

$$[A_1^{(n_1)} \cdots A_l^{(n_l)}]^\ell = [A_l^{(n_l-1)} \cdots A_1^{(n_1-1)}]$$

and

$$[A_1^{(n_1)} \cdots A_l^{(n_l)}]^r = [A_l^{(n_l+1)} \cdots A_1^{(n_1+1)}].$$

Definition 2. *A* pregroup grammar *(PGG) is a tuple $G = \langle \Sigma, \mathcal{B}, \leq, I, \Delta \rangle$, where*

- *Σ is a finite set of terminals (the* alphabet*),*
- *$\langle \mathcal{B}, \leq \rangle$ is a finite partially ordered set of* atoms*,*

– I *is a finite-range mapping* $I : \Sigma \to 2^{\kappa(\mathcal{B})}$,[4] *and*
– $\Delta \subset \tau(\mathcal{B})$ *is a finite set of* distinguished categories.[5]

We extend I to Σ^+ by

$$I(w\sigma) = \{\tau\tau' : \tau \in I(w) \text{ and } \tau' \in I(\sigma)\}$$

and define the language $L(G)$ generated by G by

$$L(G) = \{w : \text{there exist } \tau \in I(w) \text{ and } \delta \in \Delta \text{ such that } \tau \leq \delta\}.$$

Example 1. Consider the PGG $G_a = \langle \Sigma, \mathcal{B}, =, I, \Delta \rangle$, where

– $\Sigma = \{a, b\}$,
– $\mathcal{B} = \{S, A\}$, and
– I is defined by
  • $I(a) = \{SA^\ell S^\ell, SA^\ell\}$, and
  • $I(b) = \{A\}$,
  and
– $\Delta = \{S\}$.

It can be readily seen that

$$L(G_a) = \{a^n b^n : n \geq 1\}.$$

Below is a derivation for $a^3b^3 \in L(G_a)$. The lexical category assignment chosen is

$$\overbrace{SA^\ell S^\ell}^{a}\overbrace{SA^\ell S^\ell}^{a}\overbrace{SA^\ell}^{a}\overbrace{A}^{b}\overbrace{A}^{b}\overbrace{A}^{b}$$

and cancellation is indicated by underline.

$$SA^\ell \underline{S^\ell S} A^\ell \underline{S^\ell S} A^\ell AAA \leq SA^\ell A^\ell \underline{A^\ell A} AA \leq SA^\ell \underline{A^\ell A} A \leq S\underline{A^\ell A} \leq S.$$

Theorem 1. ([2]) *An ϵ-free language L is $L(G)$ for some pregroup grammar G if and only if L is context-free.*

3 Buffer Augmented PGGs

In this section we introduce buffer augmented pregroup grammars and present some of their basic properties.

Definition 3. *A* buffer augmented *pregroup grammar (BAPGG) is a tuple $G = \langle \Sigma, \mathcal{B}, \leq, \mathcal{B}', I, \Delta \rangle$, where the components of G are as follows.*

– Σ *is a finite set of terminals (the* alphabet*).*
– $\langle \mathcal{B}, \leq \rangle$ *is a partially ordered finite set.*

[4] That is, $I(\sigma)$ is finite for all $\sigma \in \Sigma$.
[5] Cf. an equivalent definition in [2], where Δ consists of one term only.

- $\mathcal{B}' \subseteq \mathcal{B}$ *is the set of the* buffer *elements.*
- I *is a mapping that assigns to each element of* Σ *a finite set of categories from* $\kappa(\mathcal{B} \cup \{x\})$*, where* x *is a new (variable) symbol – the buffer, such that for all* $\sigma \in \Sigma$*, each* $\tau \in I(\sigma)$ *is of one of the following forms:*
  (*i*) $\tau \in \kappa(\mathcal{B} \setminus \mathcal{B}')$,
  (*ii*) $\tau = \alpha A^{(\pm 1)} \beta$*, where* $A \in \mathcal{B}'$, $\alpha, \beta \in \kappa(\mathcal{B} \setminus \mathcal{B}')$*, or*
  (*iii*) $\tau = \alpha x \beta$*, where* $\alpha, \beta \in \kappa(\mathcal{B} \setminus \mathcal{B}')$.
  In addition,
  - *for each* $\tau = \alpha A^{(\pm 1)} \beta \in I(\sigma)$ *there is* $\tau' = \alpha A^{(\pm 1)} \beta' \in I(\sigma)$ *such that* $\beta' \alpha \leq 1$ *or there is* $\tau' = \alpha' A^{(\pm 1)} \beta \in I(\sigma)$ *such that* $\beta \alpha' \leq 1$,[6] *and*
  - *if* $I(\sigma)$ *contains a category of the form* (*i*)*, then it contains no category of the form* (*ii*),[7] *and we shall say that* σ *is of type* (*i*) *or type* (*ii*)*, respectively.*
- $\Delta \subset \kappa(\mathcal{B} \setminus \mathcal{B}')$ *is a finite set of* distinguished *categories.*

The language generated by G is defined by

$$L(G) = \{w : \text{there exist } \tau \in I(w),\ \theta \in \mathcal{B}'^{+}, \text{ and } \delta \in \Delta \text{ such that } \tau[x := \theta] \leq \delta\},$$

where $\tau[x := \theta]$ is the result of simultaneous substitution of θ for x in τ.

We shall see in Section 5 that the membership problem for BAPGG languages is NP-complete. Therefore, for each positive integer K we associate with G the K-*restricted* language $L_K(G)$ generated by G that is defined as follows.

$$L_K(G) = \{w : \text{there exists } \tau \in I(w) \text{ having at most } K \text{ occurrences of } x, \text{ and there exist } \theta \in \mathcal{B}'^{+} \text{ and } \delta \in \Delta \text{ such that } \tau[x := \theta] \leq \delta\}.$$

That is, K is the number of times the BAPGG "may consult" its buffer. It is shown in Section 6 that the membership problem for restricted BAPGG languages can be solved in polynomial time.[8]

In what follows we establish some basic properties of the class of (restricted) BAPGG languages.

Theorem 2. *Buffer augmented pregroup grammars are at least as powerful as pregroup grammars.*

Proof. Let $G = \langle \Sigma, \mathcal{B}, \leq, I, \Delta \rangle$ be a PGG. Then $L(G) = L(G')$, where the BAPGG G' is defined by $G' = \langle \Sigma, \mathcal{B}, \leq, \emptyset, I, \Delta \rangle$. The proof is immediate by using the same assignment for both grammars..

Note that G' is indeed a BAPGG whose lexical category assignment satisfies clause (*i*) of the definition of I in Definition 3.

The same construction, obviously, works for the restricted languages.

[6] This condition is needed for the proof of the pumping lemma for BAPGG languages, but is not needed for for polynomial parsing of restricted BAPGG languages defined below.

[7] This constraint is need for polynomial parsing of restricted BAPGG languages, but is not needed for the proof of the pumping lemma for BAPGG languages.

[8] In other words, the membership problem for BAPGG languages is *fixed-parameter tractable.*

Remark 2. In fact, 1-restricted BAPGG languages are context-free. The proof is based on building the corresponding pushdown automaton. The automaton construction is similar to that in [3] (see also [5]) with the additional feature that, before reading x, the automaton can pop a number of symbols from $\{A^\ell : A \in \mathcal{B}'\}$ from the pushdown stack (using ϵ-moves) and then push there a number of symbols from $\mathcal{B}'$ (again using ϵ-moves).

We show next that characteristic mildly context-sensitive languages are generated by buffer augmented pregroup grammars. The grammar constructions are based on "push information" technique, where new terms are "pushed" into a number of positions in a category to cancel some of its other terms. Because the information being "pushed" is the same in all positions in the category, it can be used to compare the number of occurrences of a term in different positions. This can be thought of as a counterpart of *commutations* introduced in [4].

Multiple Agreement

Let $L_{ma} = \{a^n b^n c^n : n \geq 1\}$ and let $G_{ma} = \langle \Sigma, \mathcal{B}, =, \mathcal{B}', I, \Delta \rangle$, where

- $\Sigma = \{a, b, c\}$,
- $\mathcal{B} = \{A, P, T, U\}$,
- $\mathcal{B}' = \{A\}$,
- I is defined by
  - $I(a) = \{A^\ell, xT\}$,
  - $I(b) = \{T^r A^\ell T, T^r x U\}$, and
  - $I(c) = \{U^r A^\ell U, U^r x P\}$,

  and
- $\Delta = \{P\}$.

Then

$$L(G_{ma}) = L_3(G_{ma}) = L_{ma}.^9$$

For example, $aaabbbccc \in L_{ma}$ can be derived as follows. The lexical category assignment is

$$\overbrace{A^\ell}^{a}\ \overbrace{A^\ell}^{a}\ \overbrace{xT}^{a}\ \overbrace{T^r A^\ell T}^{b}\ \overbrace{T^r A^\ell T}^{b}\ \overbrace{T^r x U}^{b}\ \overbrace{U^r A^\ell U}^{c}\ \overbrace{U^r A^\ell U}^{c}\ \overbrace{U^r x P}^{c} \tag{2}$$

and, substituting $\theta = AA\,(\in \mathcal{B}'^+)$ for x, we obtain

$$\begin{aligned} A^\ell A^\ell \theta \underline{TT^r} A^\ell \underline{TT^r} A^\ell \underline{TT^r} \theta \underline{UU^r} A^\ell \underline{UU^r} A^\ell \underline{UU^r} \theta P & \\ &\leq A^\ell A^\ell \theta A^\ell A^\ell \theta A^\ell A^\ell \theta P \\ &= A^\ell A^\ell AAA^\ell A^\ell AAA^\ell A^\ell AAP \\ &\leq \underline{A^\ell A^\ell AAA^\ell A^\ell AAA^\ell A^\ell AA} P \\ &\leq P. \end{aligned}$$

The lexical category assignment (2) naturally extends on all elements of L_{ma}, implying $L_{ma} \subseteq L(G_{ma})$.

[9] Example 2 in the next section shows that L_{ma} is not a 2-restricted BAPGG language.

For the proof of the converse inclusion $L(G_{ma}) \subseteq L_{ma}$, let $w \in \Sigma^+$ be such that for some $\tau \in I(w)$[10] there exists a substitution $\theta \in \mathcal{B}'^+$ for which $\tau[x := \theta] \leq P$. It follows from the definition of I (and Corollary 1, of course) that $w = a^i b^j c^k$ and τ is of the form $\alpha x T T^r \beta x U U^r \beta x P$, where $\alpha = (A^\ell)^{i-1}$, $\beta = (T^r A^\ell T)^{j-1}$, and $\beta = (U^r A^\ell U)^{k-1}$. Thus $\theta = A^{i-1}$ $(= A^{j-1} = A^{k-1})$ and the desired inclusion follows.

Crossed Dependencies

Let $L_{cd} = \{a^n b^m c^n d^m : m, n \geq 1\}$ and let $G_{cd} = \langle \Sigma, \mathcal{B}, =, \mathcal{B}', I, \Delta \rangle$, where

- $\Sigma = \{a, b, c, d\}$,
- $\mathcal{B} = \{A, B, P, T, U, V\}$,
- $\mathcal{B}' = \{B\}$,
- I is defined by
  - $I(a) = \{A^\ell, A^\ell T\}$,
  - $I(b) = \{T^r B^\ell T, T^r x U\}$,
  - $I(c) = \{U^r A U, U^r A V\}$, and
  - $I(d) = \{V^r B^\ell V, V^r x P\}$,

  and
- $\Delta = \{P\}$.

Then

$$L(G_{cd}) = L_2(G_{cd}) = L_{cd}.$$

For example, $aabbbccddd \in L_{cd}$ can be derived as follows. The lexical category assignment is

$$\overbrace{A^\ell}^{a}\ \overbrace{A^\ell T}^{a}\ \overbrace{T^r B^\ell T}^{b}\ \overbrace{T^r B^\ell T}^{b}\ \overbrace{T^r x U}^{b}\ \overbrace{U^r A U}^{c}\ \overbrace{U A V^\ell}^{c}\ \overbrace{V^r B^\ell V}^{d}\ \overbrace{V B^\ell V}^{d}\ \overbrace{V^r x P}^{d}$$

and, substituting $\theta = BB$ $(\in \mathcal{B}'^+)$ for x, we obtain

$$\begin{aligned} A^\ell A^\ell \underline{TT^r} B^\ell \underline{TT^r} B^\ell \underline{TT^r} \theta \underline{UU^r} A \underline{UU^r} A \underline{VV^r} B^\ell \underline{VV^r} B^\ell \underline{VV^r} \theta P & \\ &\leq A^\ell A^\ell B^\ell B^\ell \theta A A B^\ell B^\ell \theta P \\ &= A^\ell A^\ell \underline{B^\ell B^\ell BB} A A \underline{B^\ell B^\ell BB} P \\ &\leq \underline{A^\ell A^\ell AA} P \\ &\leq P. \end{aligned}$$

The proof of the equality $L(G_{cd}) = L_{cd}$ is similar to that of the equality $L(G_{ma}) = L_{ma}$ and is omitted.

Reduplication

Let $\Sigma = \{a, b\}$, $L_{rd} = \{ww : w \in \Sigma^+\}$, and let $G_{rd} = \langle \Sigma, \mathcal{B}, =, \mathcal{B}', I, \Delta \rangle$, where

- $\Sigma = \{a, b\}$,
- $\mathcal{B} = \{A, B, P, T\}$,
- $\mathcal{B}' = \{A, B\}$,
- I is defined by

[10] Of course, by $I(\sigma_1 \cdots \sigma_n)$ we mean $\{\tau_1 \cdots \tau_n : \tau_i \in I(\sigma_i),\ i = 1, \ldots, n\}$.

- $I(a) = \{A^\ell, A^\ell xT, T^r A^\ell T, T^r A^\ell xP\}$ and
- $I(b) = \{B^\ell, B^\ell xT, T^r B^\ell T, T^r B^\ell xP\}$,

and

– $\Delta = \{P\}$.

Then

$$L(G_{rd}) = L_2(G_{rd}) = L_{rd}.$$

For example, $abbbabb \in L_{rd}$ can be derived as follows. The lexical category assignment is

$$\overbrace{A^\ell}^{a}\ \overbrace{B^\ell}^{b}\ \overbrace{B^\ell xT}^{b}\ \overbrace{T^r A^\ell T}^{a}\ \overbrace{T^r B^\ell T}^{b}\ \overbrace{T^r B^\ell xP}^{b}$$

and, substituting $\theta = BBA\,(\in {\mathcal{B}'}^+)$ for x, we obtain

$$\begin{aligned} A^\ell B^\ell B^\ell \theta \underline{TT^r} A^\ell \underline{TT^r} B^\ell \underline{TT^r} B^\ell \theta P &\leq A^\ell B^\ell B^\ell \theta A^\ell B^\ell B^\ell \theta P \\ &= \underline{A^\ell B^\ell B^\ell BBAA^\ell B^\ell B^\ell BBA} P \\ &\leq P. \end{aligned}$$

We omit the proof of the equality $L(G_{rd}) = L_{rd}$.

4 Pumping Lemma for (Restricted) BAPGG Languages

In this section we present the following version of pumping lemma for (restricted) BAPGG languages.

Theorem 3. *For each BAPGG language L there exist a positive integer N such that every $w \in L$, $|w| \geq N$, can be partitioned as $w = u_1 v_1 u_2 v_2 \cdots u_m v_m u_{m+1}$, where*

– $m \geq 1$,
– $|v_1|, |v_2| \geq 1$, *if* $m \leq 2$, *and* $|v_1| = \cdots = |v_m| = 1$, *if* $m \geq 3$, *and*
– *for all* $i \geq 1$

$$u_1 v_1^i u_2 v_2^i \cdots u_m v_m^i u_{m+1} \in L.^{11}$$

Proof. Let $L = L(G)$ for a BAPGG $G = \langle \Sigma, \mathcal{B}, \leq, \mathcal{B}', I, \Delta \rangle$ and let $G' = \langle \Sigma, \mathcal{B}, \leq, \emptyset, I, \Delta \rangle$. Then $L(G')$ is a context-free language, because, in this case, we may restrict I to the categories of the form (i), only. We choose N to be a pumping lemma constant for $L(G')$.

Let $w = \sigma_1 \cdots \sigma_n \in L$ be such that $|w| \geq N$. If $w \in L(G')$, then the theorem follows from the ordinary pumping lemma for context-free languages.

Otherwise, i.e., $w \notin G'$, every $\tau \in I(w)$ such that $\tau[x := \theta] \leq \delta$, for some $\theta \in {\mathcal{B}'}^+$ and some $\delta \in \Delta$, must contain an occurrence of x.

Given such τ, let m be the number of occurrences of x in it and let $\theta = A\theta'$, where $A \in \mathcal{B}'$ and $\theta' \in {\mathcal{B}'}^*$. Since $\tau[x := \theta] \leq \delta$ and $\delta \in \kappa(\mathcal{B} \setminus \mathcal{B}')$, the first occurrence of A in the jth θ in $\tau[x := A\theta']$, $j = 1, \ldots, m$, (from left to right) is cancelled by $A^{(\pm 1)}$ that comes from some $\alpha_j t_j \beta_j \in I(\sigma_{k_j})$ of type (ii), where $t_j = A^{(\pm 1)}$.

[11] Note the difference with the ordinary pumping lemma, where $i \geq 0$.

We let

- $v_j = \sigma_{k_j}$, $j = 1, \ldots, m$,
- $u_1 = \sigma_1 \cdots \sigma_{k_1 - 1}$,
- $u_j = \sigma_{k_{j-1}+1} \cdots \sigma_{k_j - 1}$, $j = 2, \ldots, m$, and
- $u_{m+1} = \sigma_{k_m + 1} \cdots \sigma_n$,

and the lexical category assignment to the symbols in $u_1 v_1^i u_2 v_2^i \cdots u_m v_m^i u_{m+1}$ and the substitution θ for x are as follows.

- The lexical category assignment to the elements of Σ occurring in the u_js is the original one.
- The i copies of $v_j = \sigma_{k_j}$, $j = 1, \ldots, m$, are assigned

$$\underbrace{\overbrace{\alpha_j t_j \beta'_j}^{\sigma_{k_j}} \cdots \overbrace{\alpha_j t_j \beta'_j}^{\sigma_{k_j}}}_{i-1 \text{ times}} \overbrace{\alpha_j t_j \beta_j}^{\sigma_{k_j}},$$

  if there is $\alpha_j t_j \beta'_j \in I(\sigma_{k_j})$ such that $\beta'_j \alpha \leq 1$, or are assigned

$$\overbrace{\alpha_j t_j \beta_j}^{\sigma_{k_j}} \underbrace{\overbrace{\alpha'_j t_j \beta_j}^{\sigma_{k_j}} \cdots \overbrace{\alpha'_j t_j \beta_j}^{\sigma_{k_j}}}_{i-1 \text{ times}},$$

  if there is $\alpha'_j t_j \beta_j \in I(\sigma_{k_j})$ such that $\beta_j \alpha' \leq 1$.
- The substitution for x is $A^i \theta'$.

Then, in the former case,

$$\cdots (\alpha_j t_j \beta'_j)^{i-1} \alpha_j t_j \beta_j \cdots \leq \cdots \alpha_j t_j^i \beta_j \cdots,$$

and, in the latter case,

$$\cdots \alpha_j t_j \beta_j (\alpha'_j t_j \beta_j)^{i-1} \cdots \leq \cdots \alpha_j t_j^i \beta_j \cdots.$$

That is, t_j^i cancels A^i in the substitution $A^i \theta'$, whereas all other cancellations are as in τ. Therefore,

$$u_1 v_1^i u_2 v_2^i \cdots u_m v_m^i u_{m+1} \in L.$$

Example 2. It immediately follows from Theorem 3 that the multiple agreement language L_{ma} is not a 2-restricted BAPGG language.

We conclude this section with the following corollary to Theorem 3.

Corollary 2. *BAPGG languages are semi-linear.*

5 Complexity of BAPGG Languages

In this section we show that the membership problem for BAPGG languages is NP-complete.

Theorem 4. *The membership problem for BAPGG languages is in NP.*

Proof. Let $G = \langle \Sigma, \mathcal{B}, \leq, \mathcal{B}', I, \Delta \rangle$ be a BAPGG and let

$$M = \max\{|I(\sigma)| : \sigma \in \Sigma\}.$$

Let $w \in L(G)$ and let $\tau \in I(w)$, $\theta \in {\mathcal{B}'}^+$, and $\delta \in \Delta$ be such that

$$\tau[x := \theta] \leq \delta.$$

Since θ is in ${\mathcal{B}'}^+$, no term occurring in a copy of it can be cancelled by a term occurring in another copy. Therefore,

$$|\theta| \leq M|w| + \max\{|\delta| : \delta \in \Delta\}.$$

That is, an appropriate lexical category assignment $\tau \in I(w)$, the substitution θ, and an appropriate $\delta \in \Delta$ can be "guessed" in an $O(|w|)$ time.

Theorem 5. *The membership problem for BAPGG languages is NP-hard.*

The proof of Theorem 5 is by a polynomial reduction from the 3-SAT problem. Namely, we shall construct a BAPGG G_{3-SAT} and define a polynomial time encoding of 3-CNF formulas (i.e., conjunctions of disjunctions of three literals) such that the encoding $[\varphi]$ of a 3-CNF formula φ is in $L(G_{3-SAT})$ if and only if φ is satisfiable.

The language of G_{3-SAT} is over the alphabet

$$\Sigma = \{b, l, r, \#, \$, @, \%, \} \cup \{t_m, t'_m, t''_m\}_{m=0,\ldots,7}$$

and 3-CNF formulas are encoded as follows.

Let x_i, x_j, and x_k be pairwise distinct variables and let $L_i \in \{x_i, \overline{x_i}\}$, $L_j \in \{x_j, \overline{x_j}\}$, and $L_k \in \{x_k, \overline{x_k}\}$ be literals. With the clause $\boldsymbol{c} = L_i \vee L_j \vee L_k$ we associate its *type* $t(\boldsymbol{c}) = t_m t'_m t''_m$, $m = 0, \ldots, 7$, that is defined as follows.

- $t(x_i \vee x_j \vee x_k) = t_0 t'_0 t''_0$.
- $t(x_i \vee x_j \vee \overline{x_k}) = t_1 t'_1 t''_1$.
- $t(x_i \vee \overline{x_j} \vee x_k) = t_2 t'_2 t''_2$.
- $t(x_i \vee \overline{x_j} \vee \overline{x_k}) = t_3 t'_3 t''_3$.
- $t(\overline{x_i} \vee x_j \vee x_k) = t_4 t'_4 t''_4$.
- $t(\overline{x_i} \vee x_j \vee \overline{x_k}) = t_5 t'_5 t''_5$.
- $t(\overline{x_i} \vee \overline{x_j} \vee x_k) = t_6 t'_6 t''_6$.
- $t(\overline{x_i} \vee \overline{x_j} \vee \overline{x_k}) = t_7 t'_7 t''_7$.

Let $\varphi = \bigwedge_{q=1}^{p} \boldsymbol{c}_q$ and let $x_1, \ldots, x_n$ be all variables that occur in ϕ. Then the encoding $[\boldsymbol{c}_q]$ of clause $\boldsymbol{c}_q = L_i \vee L_j \vee L_k$ that occurs in φ is

$$[\boldsymbol{c}_q] = \#l^{i-1}br^{n-i}l^{j-1}br^{n-j}l^{k-1}br^{n-k}\$t''_m t'_m t_m @\%,$$

where $t(\boldsymbol{c}_q) = t_m t'_m t''_m$.

Remark 3. In the above encoding, b is the "buffer symbol" to be substituted with the content of the buffer; the pairs of words (l^{i-1}, r^{n-i}), (l^{j-1}, r^{n-j}), and (l^{k-1}, r^{n-k}) indicate the literal variable (whose truth assignment will be cut from the "truth assignment word $v_1 \cdots v_n \in \{\bot, \top\}^n$ provided by the buffer); and the type $t(\boldsymbol{c}_q) = t_m t'_m t''_m$ determines the type of the literals in the clause $\boldsymbol{c}_q$. The delimiters #, \$, @, and % are needed for a technical (cancellation) purpose that will become clear in the sequel.

Now, the encoding $[\varphi]$ of φ over Σ is

$$[\phi] = [\boldsymbol{c}_1] \cdots [\boldsymbol{c}_p].$$

Let $L_{3-SAT} = \{[\phi] : \phi \in 3-SAT\}$ and let $G_{3-SAT} = \langle \Sigma, \mathcal{B}, =, \mathcal{B}', I, \{1\} \rangle$, where the components of G_{3-SAT} are defined below.

$$\mathcal{B} = \{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, S, \bot, \top\}.$$

Intuitively, A_ms, $m = 0, \ldots, 7$, correspond to truth assignments as follows.

$$\begin{aligned}
A_0 &\leftrightarrow (\bot, \bot, \bot).\\
A_1 &\leftrightarrow (\bot, \bot, \top).\\
A_2 &\leftrightarrow (\bot, \top, \bot).\\
A_3 &\leftrightarrow (\bot, \top, \top).\\
A_4 &\leftrightarrow (\top, \bot, \bot).\\
A_5 &\leftrightarrow (\top, \bot, \top).\\
A_6 &\leftrightarrow (\top, \top, \bot).\\
A_7 &\leftrightarrow (\top, \top, \top).
\end{aligned}$$

In particular, A_m corresponds to the only truth assignment that does not satisfy a clause of type $t_m t'_m t''_m$, $m = 0, \ldots, 7$.

Next, $\mathcal{B}' = \{\bot, \top\}$ and I is defined as follows.

- $I(b) = \{x\}$.
- $I(l) = \{\bot^\ell, \top^\ell\}$.
- $I(r) = \{\bot^r, \top^r\}$.
- $I(\#) = \{S\}$.
- $I(\$) = \{A_0^\ell, A_1^\ell, A_2^\ell, A_3^\ell, A_4^\ell, A_5^\ell, A_6^\ell, A_7^\ell\}$.
- $I(@) = \{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$. That is, the lexical category assignment to \$ is supposed to be canceled by the lexical category assignment to @, see Remark 3.
- $I(\%) = \{S^r\}$. That is, the lexical category assignment to # is supposed to be canceled by the lexical category assignment to %, see Remark 3.
- $I(t_0) = \{A_1 \bot^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \bot^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t'_0) = \{A_1 \bot^r A_1^\ell, A_2 \top^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \bot^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t''_0) = \{A_1 \top^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \bot^r A_3^\ell, A_4 \top^r A_4^\ell A_5 \top^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t_1) = \{A_0 \bot^r A_0^\ell, A_2 \bot^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \bot^r A_6^\ell, A_7 \top^r A_7^\ell\}$.

- $I(t'_1) = \{A_0 \bot^r A_0^\ell, A_2 \top^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \bot^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t''_1) = \{A_0 \bot^r A_0^\ell, A_2 \bot^r A_2^\ell, A_3 \bot^r A_3^\ell, A_4 \top^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t_2) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \bot^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t'_2) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \bot^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t''_2) = \{A_0 \bot^r A_0^\ell, A_1 \top^r A_1^\ell, A_3 \bot^r A_3^\ell, A_4 \top^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t_3) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \bot^r A_2^\ell, A_4 \bot^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \bot^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t'_3) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \top^r A_2^\ell, A_4 \bot^r A_4^\ell, A_5 \bot^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t''_3) = \{A_0 \bot^r A_0^\ell, A_1 \top^r A_1^\ell, A_2 \bot^r A_2^\ell, A_4 \top^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t_4) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \top^r A_3^\ell, A_5 \top^r A_5^\ell, A_6 \bot^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t'_4) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \top^r A_2^\ell, A_3 \top^r A_3^\ell, A_5 \bot^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t''_4) = \{A_0 \bot^r A_0^\ell, A_1 \top^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \bot^r A_3^\ell, A_5 \top^r A_5^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t_5) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_6 \bot^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t'_5) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \top^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t''_5) = \{A_0 \bot^r A_0^\ell, A_1 \top^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \bot^r A_3^\ell, A_4 \top^r A_4^\ell, A_6 \top^r A_6^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t_6) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \top^r A_5^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t'_6) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \top^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \bot^r A_5^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t''_6) = \{A_0 \bot^r A_0^\ell, A_1 \top^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \bot^r A_3^\ell, A_4 \top^r A_4^\ell, A_5 \top^r A_5^\ell, A_7 \top^r A_7^\ell\}$.
- $I(t_7) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \bot^r A_6^\ell\}$.
- $I(t'_7) = \{A_0 \bot^r A_0^\ell, A_1 \bot^r A_1^\ell, A_2 \top^r A_2^\ell, A_3 \top^r A_3^\ell, A_4 \bot^r A_4^\ell, A_5 \bot^r A_5^\ell, A_6 \top^r A_6^\ell\}$.
- $I(t''_7) = \{A_0 \bot^r A_0^\ell, A_1 \top^r A_1^\ell, A_2 \bot^r A_2^\ell, A_3 \bot^r A_3^\ell, A_4 \top^r A_4^\ell, A_5 \top^r A_5^\ell, A_6 \top^r A_6^\ell\}$.

Proof of the Inclusion $L_{3-SAT} \subseteq L(G_{3-SAT})$

The proof of the inclusion $L_{3-SAT} \subseteq L(G_{3-SAT})$ is based on Lemmas 1 and 2 below.

Lemma 1. *Let $\boldsymbol{c} = L_i \vee L_j \vee L_k$ be a 3-CNF clause of type m, $m = 0, \ldots, 7$, and let $v_i, v_j, v_k = \bot, \top$ be such that (v_i, v_j, v_k) satisfies $\boldsymbol{c}$. Then there exists*

$$A \in \{A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7\} \setminus \{A_m\}$$

such that $Av_i^r A^\ell \in I(t_m)$, $Av_j{}^r A^\ell \in I(t'_m)$, and $Av_k{}^r A^\ell \in I((t''_m)$.

Proof. The lemma follows immediately from the definition of $I(t_m)$, $I(t'_m)$, and $I(t''_m)$. For example, if $(v, v_j, v_k) = (\top, \top, \bot)$, then $A = A_6$.

Lemma 2. *Let $\boldsymbol{c} = L_i \vee L_j \vee L_k$ be a 3-CNF clause and let $V = (v_1, \cdots, v_n) \in \{\bot, \top\}^n$ be an "assignment vector" such that $\boldsymbol{c}|_V = \top$. Then there exists $\tau \in I([\boldsymbol{c}])$ such that*

$$\tau[x := v_1 \cdots v_n] \leq 1.$$

Proof. It follows from $\boldsymbol{c}|_V = \top$ that

$$\boldsymbol{c}(v_i, v_j, v_k) = \top. \tag{3}$$

Let $\boldsymbol{c}$ be of type m. Then,

$$[\boldsymbol{c}] \;=\; \#l^{i-1}br^{n-i}l^{j-1}br^{n-j}l^{k-1}br^{n-k}\$t''_m t'_m t_m @\% \;=$$

$$\#\underbrace{l\cdots l}_{i-1} b \underbrace{r\cdots r}_{n-i} \underbrace{l\cdots l}_{j-1} b \underbrace{r\cdots r}_{n-j} \underbrace{l\cdots l}_{k-1} b \underbrace{r\cdots r}_{n-k} \$t''_m t'_m t_m @\%,$$

and desired lexical category assingment $\tau \in I([\boldsymbol{c}])$ is defined by

$$\begin{aligned}\tau = \overbrace{S}^{\#} & \overbrace{v^{\ell}_{i-1}}^{l} \cdots \overbrace{v^{\ell}_{1}}^{l} \overbrace{x}^{b} \overbrace{v^{r}_{n}}^{r} \cdots \overbrace{v^{r}_{n-i}}^{r} \\ & \overbrace{v^{\ell}_{j-1}}^{l} \cdots \overbrace{v^{\ell}_{1}}^{l} \overbrace{x}^{b} \overbrace{v^{r}_{n}}^{r} \cdots \overbrace{v^{r}_{n-j}}^{r} \\ & \overbrace{v^{\ell}_{k-1}}^{l} \cdots \overbrace{v^{\ell}_{1}}^{l} \overbrace{x}^{b} \overbrace{v^{r}_{n}}^{r} \cdots \overbrace{v^{r}_{n-k}}^{r} \overbrace{A^{\ell}}^{\$} \overbrace{Av^{r}_{k}A^{\ell}}^{t''_m} \overbrace{Av^{r}_{j}A^{\ell}}^{t'_m} \overbrace{Av^{r}_{i}A^{\ell}}^{t_m} \overbrace{A}^{@} \overbrace{S^{r}}^{\%},\end{aligned}$$

where A is provided by (3) and Lemma 1. Therefore,

$$\begin{aligned}\tau[x := v_1 \cdots v_n] &= S v^{\ell}_{i-1} \cdots v^{\ell}_{1} v_1 \cdots v_n v^{r}_{n} \cdots v^{r}_{n-i} \\ &\quad\; v^{\ell}_{j-1} \cdots v^{\ell}_{1} v_1 \cdots v_n v^{r}_{n} \cdots v^{r}_{n-j} \\ &\quad\; v^{\ell}_{k-1} \cdots v^{\ell}_{1} v_1 \cdots v_n v^{r}_{n} \cdots v^{r}_{n-k} A^{\ell} A v^{r}_{k} A^{\ell} A v^{r}_{j} A^{\ell} A v^{r}_{i} A^{\ell} A S^{r} \qquad (4) \\ &\leq S v_i v_j v_k \; v^{r}_{k} v^{r}_{j} v^{r}_{i} S^{r} \\ &\leq 1.\end{aligned}$$

Now, let $\varphi = \bigwedge_{q=1}^{p} \boldsymbol{c}_q$ and let $V = (v_1, \cdots, v_n) \in \{\bot, \top\}^n$ be an "assignment vector" such that $\boldsymbol{c}|_V = \top$. By Lemma 2, for every $q = 1, \ldots, p$,

$$[\boldsymbol{c}_q][x := v_1 \cdots v_n] \leq 1.$$

Therefore,

$$\begin{aligned}[\phi][x := v_1 \cdots v_n] &= ([\boldsymbol{c}_1] \cdots [\boldsymbol{c}_p])[x := v_1 \cdots v_n] \\ &= ([\boldsymbol{c}_1][x := v_1 \cdots v_n]) \cdots ([\boldsymbol{c}_p][x := v_1 \cdots v_n]) \\ &\leq 1\end{aligned}$$

and the desired inclusion follows.

Proof of the Inclusion $L(G_{3-SAT}) \cap \{[\varphi] : \varphi \in \text{3-CNF}\} \subseteq L_{3-SAT}$

For the proof of the inclusion

$$L(G_{3-SAT}) \cap \{[\varphi] : \varphi \in \text{3-CNF}\} \subseteq L_{3-SAT} \qquad (5)$$

we shall need the following lemma.

Lemma 3. *Let $\boldsymbol{c}$ be a 3-CNF clause, $\tau \in I([\boldsymbol{c}])$, and let $V = (v_1, \ldots, v_n) \in \{\bot, \top\}^n$ be such that*

$$\tau[x := v_1 \cdots v_n] \leq 1.$$

then $\boldsymbol{c}|_V = \top$.

Proof. The lemma follows immediately from (4) and the definition of lexical category assignment I to t_m, t'_m, and t''_m, $m = 0, \ldots, 7$.

Now let $\varphi = \bigwedge_{q=1}^{p} \boldsymbol{c}_q$ be a 3-CNF formula and let

$$[\phi] = \underbrace{\#w_1\%}_{\boldsymbol{c}_1} \cdots \underbrace{\#w_p\%}_{\boldsymbol{c}_p},$$

where $[\boldsymbol{c}_q] = \#w_q\%$, $q = 1, \ldots, p$.

Let $V = (v_1, \ldots, v_n) \in \{\bot, \top\}^n$ and let $\boldsymbol{c} \in I([\phi])$ be such that

$$\boldsymbol{c}[x := v_1 \cdots v_n] \leq 1.$$

For the proof of the inclusion (5) we have to show that V satisfies ϕ.

We have

$$\tau[x := v_1 \cdots v_n] \;=\; S\tau_1'[x := v_1 \cdots v_n]S^r \;\cdots\; S\tau_p'[x := v_1 \cdots v_n]S^r$$

for appropriate τ_q's in $I(w_q)$, $q = 1, \ldots, p$. Then the qth S^r (from left to right) must be canceled from the left by the qth S, $q = 1, \ldots, p$. Therefore, $\tau_q'[x := v_1 \cdots v_n] \leq 1$, implying

$$\tau_q[x := v_1 \cdots v_n] = S\tau_q'[x := v_1 \cdots v_n]S^r \leq 1,$$

$q = 1, \ldots, p$. Since $\tau_q \in I([\boldsymbol{c}_q])$, by Lemma 3, V satisfies all clauses of φ and the proof is complete.

6 Complexity of Restricted BAPGG Languages

In this section we show that the membership problem for restricted BAPGG languages can be decided in polynomial time.

Theorem 6. *The membership problem for restricted BAPGG languages is in P.*

The proof of Theorem 6 is based on a sequence of reductions described below.

Let $G = \langle \Sigma, \mathcal{B}, \leq, \mathcal{B}', I, \Delta \rangle$ be a BAPGG, $K \geq 1$, and let $w = \sigma_1 \cdots \sigma_n \in \Sigma$. By definition, $w \in L_K(G)$ if and only if there exist $\tau_i \in I(\sigma_i)$, $i = 1, \ldots, n$, such that $\tau_1 \cdots \tau_n$ has at most K occurrences of x and there exist $\theta \in \mathcal{B}'^{+}$ and $\delta \in \Delta$ such that

$$(\tau_1 \cdots \tau_n)[x := \theta] \leq \delta.$$

Therefore, there exist positive integers $i_1', \ldots, i_k'$, $i_1, \ldots, i_k$, and $i_1'', \ldots, i_k''$, where $k \leq K$, such that

$$1 \leq i_1' \leq i_1 \leq i_1'' < \cdots < i_j' \leq i_j \leq i_j'' < \cdots < i_k \leq i_k'' \leq n; \tag{6}$$

for each $j = 1, \ldots, k$, $\tau_{i_j'} = \alpha_{i_j'} A^{\ell}_{i_j'} \beta_{i_j'}$ and $\tau_{i_j''} = \alpha_{i_j''} A^{r}_{i_j''} \beta_{i_j''}$ are categories of the form (ii), and $\tau_{i_j} = \alpha_{i_j} x \beta_{i_j}$ is a category of the form (iii);

$$(A^{\ell}_{i_j'} \beta_{i_j'} \tau_{i_j'+1} \cdots \tau_{i_j-1} \alpha_{i_j} x \beta_{i_j} \tau_{i_j+1} \cdots \tau_{i_j''-1} \alpha_{i_j''} A^{r}_{i_j''})[x := \theta] \leq 1; \text{[12]} \tag{7}$$

and

[12] That is, $A^{\ell}_{i_j'}$ and $A^{r}_{i_j''}$ $j = 1, \ldots, k$, cancel the rightmost and the leftmost symbols of θ, respectively.

$$\tau_1 \cdots \tau_{i'_1-1}\alpha_{i'_1}\beta_{i''_1}\tau_{i''_1+1} \cdots \tau_{i'_j-1}\alpha_{i'_j}\beta_{i''_j}\tau_{i''_j+1} \cdots \tau_{i'_k-1}\alpha_{i'_k}\beta_{i''_k}\tau_{i''_k+1} \cdots \tau_n \le \delta.^{13} \quad (8)$$

Thus, for all $k = 1, \ldots, K$, all sets of positive integers

$$\{i'_1, \ldots, i'_k\} \cup \{i_1, \ldots, i_k\} \cup \{i''_1, \ldots, i''_k\}$$

satisfying (6),[14] all assignments $\tau_{i'_j} = \alpha_{i'_j} A^{\ell}_{i'_j} \beta_{i'_j}$ and $\tau_{i''_j} = \alpha_{i''_j} A^{r}_{i''_j} \beta_{i''_j}$ of the from (ii), all assignments $\tau_{i_j} = \alpha^{r}_{i_j} x \beta_{i_j}$ of the from (iii), and all $\delta \in \Delta$ we shall look for $\theta \in \mathcal{B}'^{+}$ and categories $\tau_i \in I(\sigma_i)$,

$$i \in \{1, \ldots, n\} \setminus (\{i'_1, \ldots, i'_k\} \cup \{i_1, \ldots, i_k\} \cup \{i''_1, \ldots, i''_k\}),$$

such that (8) and

$$\begin{cases} (A^{\ell}_{i'_1}\beta_{i'_1}\tau_{i'_1+1} \cdots \tau_{i_1-1}\alpha_{i_1} x \beta_{i_1}\tau_{i_1+1} \cdots \tau_{i''_1-1}\alpha_{i''_1} A^{r}_{i''_1})[x := \theta] \le 1 \\ (A^{\ell}_{i'_2}\beta_{i'_2}\tau_{i'_2+1} \cdots \tau_{i_2-1}\alpha_{i_2} x \beta_{i_2}\tau_{i_2+1} \cdots \tau_{i''_2-1}\alpha_{i''_2} A^{r}_{i''_2})[x := \theta] \le 1 \\ \vdots \\ (A^{\ell}_{i'_k}\beta_{i'_k}\tau_{i'_k+1} \cdots \tau_{i_k-1}\alpha_{i_k} x \beta_{i_k}\tau_{i_k+1} \cdots \tau_{i''_k-1}\alpha_{i''_k} A^{r}_{i''_k})[x := \theta] \le 1 \end{cases} . \quad (9)$$

That is, (9) consists of k inequations (7): one inequation for each $j = 1, \ldots, k$.

Obviously, the number of such possible pairs (8) and (9) is bounded by a polynomial in n (whose degree is a function of K).

First we shall show that (8) can be solved in polynomial time. Let

$$\widehat{\Sigma} = \{\widehat{\sigma}_i : i \in \{i'_1, \ldots, i'_k\} \cup \{i''_1, \ldots, i''_k\}\}$$

be a disjoint copy of

$$\{\sigma_{i'_1}, \ldots, \sigma_{i'_k}\} \cup \{\sigma_{i''_1}, \ldots, \sigma_{i''_k}\}.$$

Consider the PGG $\widehat{G} = \langle \Sigma \cup \widehat{\Sigma}, \mathcal{B}, \le, \widehat{I}, \delta \rangle$, where $\widehat{I}$ is defined as follows.

$$\widehat{I}(\sigma) = \begin{cases} I(\sigma), \text{ if } \sigma \in \Sigma \\ \alpha_{i'_j}, \ \text{ if } \sigma = \widehat{\sigma_{i'_j}},\ j = 1, \ldots, k \\ \beta_{i''_j}, \ \text{ if } \sigma = \widehat{\sigma_{i''_j}},\ j = 1, \ldots, k \end{cases} .$$

Then there is a lexical category assignment for σ_i,

$$i \in \{1, \ldots, i'_1 - 1\} \cup \bigcup_{j=1}^{k-1} \{i''_j + 1, \ldots, i'_{j+1} - 1\} \cup \{i''_k + 1, \ldots, n\}$$

satisfying (8) if and only if

$$\sigma_1 \cdots \sigma_{i'_1-1}\widehat{\sigma_{i'_1}}\,\widehat{\sigma_{i''_1}}\sigma_{i''_1+1} \cdots \sigma_{i'_k-1}\widehat{\sigma_{i'_k}}\,\widehat{\sigma_{i''_k}} \cdots \sigma_n \in L(\widehat{G}).$$

---

[13] Note that all τ_is occurring in (8) are of the form (i).

[14] Actually, in (9) we assume that all inequalities in (6) are strict. It will be clear in the sequel how to treat the case of non-strict inequalities.

The latter membership can be tested in polynomial time, because, by Theorem 1, $L(\widehat{G})$ is context-free.

We shall show next how to solve (9) in polynomial time. First we observe that, by the definition of assignment I, for each solution of (9) and each inequation j, $j = 1, \ldots, k$, the following holds.

Each category of the from (ii) to the left of x is of the from $\alpha A^{\ell}\beta$ and each category of the from (ii) to the right of x is of the from $\alpha A^{r}\beta$.

For our next observation we shall need the following notation. For a category $\tau = \alpha A^{r}\beta$ of the form (ii) we denote by $\widetilde{\tau}$ the category $\tau = \alpha A^{\ell}\beta$ and for a category τ of the form (i), $\widetilde{\tau}$ is τ itself. Then, (9) if and only if

$$\begin{cases} \beta_{i_1}\widetilde{\tau_{i_1+1}}\cdots\widetilde{\tau_{i''_1-1}}\alpha_{i''_1}A^{\ell}_{i''_1}A^{\ell}_{i'_1}\beta_{i'_1}\tau_{i'_1+1}\cdots\tau_{i_1-1}\alpha_{i_1}\theta \leq 1 \\ \beta_{i_2}\widetilde{\tau_{i_2+1}}\cdots\widetilde{\tau_{i''_2-1}}\alpha_{i''_2}A^{\ell}_{i''_2}A^{\ell}_{i'_2}\beta_{i'_2}\tau_{i'_2+1}\cdots\tau_{i_2-1}\alpha_{i_2}\theta \leq 1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \beta_{i_k}\widetilde{\tau_{i_k+1}}\cdots\widetilde{\tau_{i''_k-1}}\alpha_{i''_k}A^{\ell}_{i''_k}A^{\ell}_{i'_k}\beta_{i'_k}\tau_{i'_k+1}\cdots\tau_{i_k-1}\alpha_{i_k}\theta \leq 1 \end{cases} . \tag{10}$$

That is, the left-hand side of the jth inequation in (10) can be thought of as a lexical assignment $\overline{I}$ over the alphabet

$$\overline{\Sigma} = \Sigma \cup \widetilde{\Sigma} \cup \Sigma' \cup \Sigma'',$$

where $\widetilde{\Sigma}, \Sigma', \Sigma''$, and $\overline{I}$ are defined as follows:

$$\widetilde{\Sigma} = \{\widetilde{\sigma} : \sigma \in \Sigma \text{ is of type } (ii)\}$$

is a disjoint copy of the subset of Σ consisting of all elements of Σ of type (ii),

$$\Sigma' = \{\sigma' : \sigma = \sigma_{i'_1}, \ldots, \sigma_{i'_k}\}$$

is a disjoint copy of $\{\sigma_{i'_1}, \ldots, \sigma_{i'_k}\}$,

$$\Sigma'' = \{\sigma'' : \sigma = \sigma_{i''_1}, \ldots, \sigma_{i''_k}\}$$

is a disjoint copy of $\{\sigma_{i''_1}, \ldots, \sigma_{i''_k}\}$, and

$$\overline{I}(\sigma) = \begin{cases} I(\sigma), & \text{if } \sigma \in \Sigma \\ \{\widetilde{\tau} : \tau \in I(\sigma)\}, & \text{if } \sigma \in \widetilde{\Sigma} \\ A^{\ell}_{i'_j}\beta_{i'_j}, & \text{if } \sigma = \sigma_{i'_j} \in \Sigma' \\ \alpha_{i''_j}A^{\ell}_{i''_j}, & \text{if } \sigma = \sigma_{i''_j} \in \Sigma'' \end{cases} .$$

The type of the elements of $\widetilde{\Sigma} \cup \Sigma' \cup \Sigma''$ is (ii).

Thus, we have reduced the membership problem for $L(G)$ to solving polynomially many systems of inequations described below.

Let $j = 1, \ldots, k$ and let $m_j = i''_j - i'_j$. We shall use the following renaming of the elements of Σ occurring in $w = \sigma_1 \cdots \sigma_n$:

$$\sigma_{j,i} = \begin{cases} \sigma_{i_j+i}, & \text{if } 1 \leq i < i''_j - i_j \text{ and } \sigma_{i_j+i} \text{ is of type } (i) \\ \widetilde{\sigma_{i_j+i}}, & \text{if } 1 \leq i < i''_j - i_j \text{ and } \sigma_{i_j+i} \text{ is of type } (ii) \\ \sigma_{i''_j}, & \text{if } i = i''_j - i_j \\ \sigma_{i'_j}, & \text{if } i = i''_j - i_j + 1 \\ \sigma_{i'_j+i-(i''_j-i_j+1)}, & \text{if } i''_j - i_j + 1 < i \leq m_j \end{cases} .$$

Then, we have to find assignment $\tau_{j,i} \in \overline{I}(\sigma_{j,i})$, $j = 1, \ldots, k$ and $i = 1, \ldots, m_j$, and $\theta \in {\mathcal{B}'}^+$ such that

$$\begin{cases} \beta_{i_1}\tau_{1,1}\cdots\tau_{1,m_1}\alpha_{i_1}\theta \leq 1 \\ \beta_{i_2}\tau_{2,1}\cdots\tau_{2,m_2}\alpha_{i_2}\theta \leq 1 \\ \qquad\qquad\qquad\qquad\quad \vdots \\ \beta_{i_k}\tau_{k,1}\cdots\tau_{k,m_k}\alpha_{i_k}\theta \leq 1 \end{cases} . \tag{11}$$

That is, the systems of inequations (10) and (11) are related as follows:

$$\tau_{j,1} = \widetilde{\tau_{i_j+1}}, \ldots, \tau_{j,i''_j-i_j} = \alpha_{i''_j}A^{\ell}_{i''_j}, \tau_{j,i''_j-i_j+1} = A^{\ell}_{i'_j}\beta_{i'_j} \ldots, \tau_{j,m_j} = \tau_{i_j-1},$$

$j = 1, \ldots, k$. In particular, m_j is the number of elements of Σ corresponding to the jth inequation in (10).

Since $\theta \in {\mathcal{B}'}^+$, the symbols occurring in θ can be cancelled only by the pregroup elements of the form A^{ℓ}, and such elements can occur only in assignments to the elements of Σ of type (ii). Therefore, the number of the elements of Σ of type (ii) occurring in each $\sigma_{j,1}\cdots\sigma_{j,m_j}$, $j = 1, \ldots, k$, is the same, say $m = |\theta|$.

Let $\theta = A_1 \cdots A_m$ and let $\sigma_{j,l_{j,i}}$ be the ith occurrence (from the left) of the elements of Σ of type (ii) in $\sigma_{j,1}\cdots\sigma_{j,m_j}$, $j = 1, \ldots, k$ and $i = 1, \ldots, m$. Then $\tau_{j,l_{j,i}}$ is of the form $\alpha_{j,l_{j,i}}A^{\ell}_i\beta_{j,l_{j,i}}$, where $\alpha_{j,l_{j,i}}, \beta_{j,l_{j,i}} \in \kappa(\mathcal{B} \setminus \mathcal{B}')$ and

$$\begin{cases} \beta_{i_1}\tau_{1,1}\cdots\tau_{1,l_{1,1}-1}\alpha_{1,l_{1,1}} \leq 1 \\ \beta_{i_2}\tau_{2,1}\cdots\tau_{2,l_{2,1}-1}\alpha_{2,l_{2,1}} \leq 1 \\ \qquad\qquad\qquad\qquad\quad \vdots \\ \beta_{i_k}\tau_{k,1}\cdots\tau_{k,l_{k,1}-1}\alpha_{k,l_{k,1}} \leq 1 \end{cases} , \tag{12}$$

i.e., the "before A_1" prefix in the left-hand side of each inequation reduces to 1;

$$\begin{cases} \beta_{1,l_{1,p}}\tau_{1,l_{1,p}+1}\cdots\tau_{1,l_{1,p+1}-1}\alpha_{1,l_{1,p+1}} \leq 1 \\ \beta_{2,l_{2,p}}\tau_{2,l_{2,p}+1}\cdots\tau_{2,l_{2,p+1}-1}\alpha_{2,l_{2,p+1}} \leq 1 \\ \qquad\qquad\qquad\qquad\qquad\quad \vdots \\ \beta_{k,l_{k,p}}\tau_{k,l_{k,p}+1}\cdots\tau_{k,l_{k,p+1}-1}\alpha_{k,l_{k,p+1}} \leq 1 \end{cases} , \tag{13}$$

i.e., the subword between A_p and A_{p+1} in the left-hand side of each inequation reduces to 1, $p = 1, \ldots, m-1$; and

$$\begin{cases} \beta_{1,l_{1,m}}\tau_{1,l_{1,m}+1}\cdots\tau_{1,m_1}\alpha_{i_1} \leq 1 \\ \beta_{2,l_{2,m}}\tau_{2,l_{2,m}+1}\cdots\tau_{2,m_2}\alpha_{i_2} \leq 1 \\ \qquad\qquad\qquad\qquad\quad \vdots \\ \beta_{k,l_{k,m}}\tau_{k,l_{k,m}+1}\cdots\tau_{k,m_k}\alpha_{i_k} \leq 1 \end{cases} , \tag{14}$$

i.e., the "after A_m" suffix in the left-hand side of each inequation reduces to 1.

To proceed from this point we shall need the following definition.

Definition 4. *Let $\tau_i = \alpha_i A^{\ell}_i \beta_i$, $\alpha_i, \beta_i \in \kappa(\mathcal{B} \setminus \mathcal{B}')$ and $A_i \in \mathcal{B}'$, $i = 1, \ldots, k$. We say that the set of categories $\{\tau_j\}_{j=1,\ldots,k}$ is* consistent, *if*

$$A_1 = \cdots = A_k.$$

Consider the directed graph $\mathcal{G}$ whose set of vertices consists of 0, $m+1$, and all $(k+1)$tuples of the form $(p, \tau_1, \ldots, \tau_k)$, $p = 1, \ldots, m$, where $\tau_j \in \overline{I}(\sigma_{j,l_{j,p}})$ and the set of categories $\{\tau_j\}_{j=1,\ldots,k}$ is consistent. Note that the number of vertices of $\mathcal{G}$ is linear in n.

The edges of $\mathcal{G}$ are as follows.

- There is no edge form vertex 0 to vertex $m+1$.
- There is an edge from vertex 0 to vertex

$$(p, \alpha_1 A_p \beta_1, \ldots, \alpha_k A_p \beta_k)$$

  if and only if $p = 1$ and there exist $\tau_{j,i} \in \overline{I}(\sigma_{j,i})$, $j = 1, \ldots, k$ and $i = 1, \ldots, l_{j,1} - 1$, such that

$$\begin{cases} \beta_{i_1} \tau_{1,1} \cdots \tau_{1,l_{1,1}-1} \alpha_1 & \leq 1 \\ \beta_{i_2} \tau_{2,1} \cdots \tau_{2,l_{2,1}-1} \alpha_2 & \leq 1 \\ & \vdots \\ \beta_{i_k} \tau_{k,1} \cdots \tau_{k,l_{k,1}-1} \alpha_k & \leq 1 \end{cases}, \tag{15}$$

  cf. (12).
- There is an edge from vertex

$$(p', \alpha'_1 A_{p'} \beta'_1, \ldots, \alpha'_k A_{p'} \beta'_k)$$

  to vertex

$$(p'', \alpha''_1 A_{p''} \beta''_1, \ldots, \alpha''_k A_{p''} \beta''_k)$$

  if and only if $p'' = p' + 1$ and there exist $\tau_{j,i} \in \overline{I}(\sigma_{j,i})$, $j = 1, \ldots, k$ and $i = l_{j,p'} + 1, \ldots, l_{j,p''} - 1$, such that

$$\begin{cases} \beta'_1 \tau_{1,l_{1,p'}+1} \cdots \tau_{1,l_{1,p''}-1} \alpha''_1 & \leq 1 \\ \beta'_2 \tau_{2,l_{2,p'}+1} \cdots \tau_{2,l_{2,p''}-1} \alpha''_2 & \leq 1 \\ & \vdots \\ \beta'_k \tau_{k,l_{k,p'}+1} \cdots \tau_{k,l_{k,p''}-1} \alpha''_k & \leq 1 \end{cases}, \tag{16}$$

  cf. (13).
- There is an edge from vertex

$$(p, \alpha_1 A_p \beta_1, \ldots, \alpha_k A_p \beta_k)$$

  to vertex $m+1$ if and only if $p = m$ and there exist $\tau_{j,i} \in \overline{I}(\sigma_{j,i})$, $j = 1, \ldots, k$ and $i = l_{j,m} + 1, \ldots, m_j$, such that

$$\begin{cases} \beta_1 \tau_{1,l_{1,m}+1} \cdots \tau_{1,m_1} \alpha_{i_1} & \leq 1 \\ \beta_2 \tau_{2,l_{2,m}+1} \cdots \tau_{2,m_2} \alpha_{i_2} & \leq 1 \\ & \vdots \\ \beta_k \tau_{k,l_{k,m}+1} \cdots \tau_{k,m_k} \alpha_{i_k} & \leq 1 \end{cases}, \tag{17}$$

  cf. (14).

Similarly to the case of (8), finding appropriate assignments, which satisfy the above (independent) inequations (15), (16), and (17) reduces to the membership

problems in corresponding context-free languages. Therefore, $\mathcal{G}$ can be constructed in polynomial time.

Since (9) if and only if there is a path from 0 to $m+1$ in $\mathcal{G}$, the proof of Theorem 6 is complete.

7 Hierarchy of Restricted BAPGG Languages

In this section we show that the class of K-restricted BAPGG languages is a proper subclass of the $(K+1)$-restricted languages. We shall show first that each K-restricted BAPGG language is also a $((K+1)$-restricted) BAPGG one.

Let $G = \langle \Sigma, \mathcal{B}, \leq, \mathcal{B}', I, \Delta \rangle$ be a BAPGG. Since, obviously, the class of (restricted) BAPGG languages is closed under union, we may assume that $\Delta = \{\delta\}$ consists of one category only. Consider the BAPGG $G' = \langle \Sigma, \mathcal{B} \cup \{Z\}, \leq', \mathcal{B}', I', \{\delta Z^k : k = 0, \ldots, K\} \rangle$, where $Z \notin \mathcal{B}$ and $\leq' = \leq \cup (Z, Z)$ and, for $\sigma \in \Sigma$, the lexical category assignment $I'(\sigma)$ is defined as follows.

– If σ is of type (i) or of type (ii), then

$$I'(\sigma) = \{(Z^r)^k \tau Z^k : \tau \in I(\sigma) \text{ and } k = 0, \ldots, K\}.$$

– If σ is of type (iii), then

$$I'(\sigma) = \{(Z^r)^k \tau Z^{k+1} : \tau \in I(\sigma) \text{ and } k = 0, \ldots, K-1\}.$$

We contend that $L(G') = L_K(G)$. We start with the proof of the inclusion $L(G') \subseteq L_K(G)$. Let $w \in L(G')$ and let $\tau \in I'(w)$, $\theta \in {\mathcal{B}'}^+$, and $k = 0, \ldots, K$ be such that

$$\tau[x := \theta] \leq \delta Z^k. \tag{18}$$

Then, τ has exactly k occurrences of x and, substituting 1 for Z in (18) we obtain

$$(\tau[Z := 1])[x := \theta] \leq \delta,$$

i.e., $w \in L_K(G)$.

Conversely, let $w = \sigma_1 \cdots \sigma_n \in L_K(G)$, $\tau_i \in I(\sigma_i)$, $i = 1, \ldots, n$, be such that $\tau_1 \cdots \tau_n$ has k occurrences of x, $k = 0, \ldots, K$, and

$$(\tau_1 \cdots \tau_n)[x := \theta] \leq \delta,$$

and let

$$0 = i_0 < i_1 < \cdots < i_j < \cdots < i_k < i_{k+1} = n+1$$

be such that that $\tau_{i_j} \in I(\sigma_{i_j})$, $j = 1, \ldots, k$, is of the form (iii). Then, for $\tau_i' \in I'(\sigma_i)$ defined by

$$\tau_i' = \begin{cases} (Z^r)^j \tau_i Z^{j+1}, \text{ if } i = i_j,\ j = 1, \ldots, k \\ (Z^r)^j \tau_i Z^j \quad \text{if } i_{j-1} < i < i_j,\ j = 1, \ldots, k+1 \end{cases},$$

we have

$$(\tau_1' \cdots \tau_n')[x := \theta] \leq \delta Z^k.$$

That is, $w \in L(G')$.

For the proof of the strict inclusion of the hierarchy levels consider the languages

$$L_{e,K} = \{(ab^n)^K : n = 1, 2, \ldots\},$$

where K is a positive integer. It can be readily seen that, for all positive integers K, $L_{e,K}$ is a K-restricted BAPGG language. For example,

$$L_{e,3} = L_3(G_{e,3}) = L(G_{e,3})$$

for the BAPGG $G_{e,3} = \langle\{a, b\}, \{B, S, T\}, =, \{B\}, I, \{T^3\}\rangle$, where I is defined by

- $I(a) = \{S\}$ and
- $I(b) = \{S^r B^\ell S, S^r xT\}$.

In particular, $abbbabbbabbb \in L_{e,3}$ can be derived as follows. The lexical category assignment is

$$\overbrace{S}^{a}\ \overbrace{S^r B^\ell S}^{b}\ \overbrace{S^r B^\ell S}^{b}\ \overbrace{S^r xT}^{b}\ \overbrace{S}^{a}\ \overbrace{S^r B^\ell S}^{b}\ \overbrace{S^r B^\ell S}^{b}\ \overbrace{S^r xT}^{b}\ \overbrace{S}^{a}\ \overbrace{S^r B^\ell S}^{b}\ \overbrace{S^r B^\ell S}^{b}\ \overbrace{S^r xT}^{b}$$

and, substituting $\theta = BB\,(\in \mathcal{B}'^{+})$ for x, we obtain (by **(con)**s)

$$SS^r B^\ell SS^r B^\ell SS^r BBTSS^r B^\ell SS^r B^\ell SS^r BBTSS^r B^\ell SS^r B^\ell SS^r BBT \leq TTT.$$

The definition of the BAPGG $G_{e,3}$ and the lexical category assignment above naturally extend to all positive integers K and all elements of $L_{e,K}$, implying $L_{e,K} \subseteq L(G_{e,K})$. The proof of the converse inclusion is equally easy and is omitted.

It easily follows from the pumping lemma for restricted BAPGG languages that $L_{e,K+1}$ is not a K-restricted BAPGG languages. Thus, the "K–hierarchy" of restricted BAPGG languages is strict.

8 An Extension of BAPGGs

It can be readily seen that all results of this paper also hold for "multi-buffer augmented" pregroup grammars defined below and their corresponding K-restricted languages.

Definition 5. (Cf. Definition 3.) *A q-buffer augmented pregroup grammar (q-BAPGG) is a tuple $G = \langle\Sigma, \mathcal{B}, \leq, \mathcal{B}', \mathcal{V}, I, \Delta\rangle$, where the components of G are as follows.*

- *Σ is a finite set of terminals (the* alphabet*).*
- *$\langle\mathcal{B}, \leq\rangle$ is a partially ordered finite set.*
- *$\mathcal{B}' \subseteq \mathcal{B}$ is the set of the* buffer *elements.*
- *$\mathcal{V} = \{x_1, \ldots, x_q\}$ is a set of* v*ariables (buffers) disjoint from $\kappa(\mathcal{B})$.*
- *I is a mapping that assigns to each element of Σ a finite set of categories from $\kappa(\mathcal{B} \cup \mathcal{V})$ such that for all $\sigma \in \Sigma$, each $\tau \in I(\sigma)$ is of one of the following forms:*

(i) $\tau \in \kappa(\mathcal{B} \setminus \mathcal{B}')$,
(ii) $\tau = \alpha A^{(\pm 1)} \beta$, *where* $A \in \mathcal{B}'$, $\alpha, \beta \in \kappa(\mathcal{B} \setminus \mathcal{B}')$, *or*
(iii) $\tau = \alpha x_i \beta$, *where* $\alpha, \beta \in \kappa(\mathcal{B} \setminus \mathcal{B}')$ *and* $i = 1, \ldots, q$.
In addition,
- *for each* $\tau = \alpha A^{(\pm 1)} \beta \in I(\sigma)$ *there is* $\tau' = \alpha A^{(\pm 1)} \beta' \in I(\sigma)$ *such that* $\beta' \alpha \leq 1$ *or there is* $\tau' = \alpha' A^{(\pm 1)} \beta \in I(\sigma)$ *such that* $\beta \alpha' \leq 1$, *and*
- *if* $I(\sigma)$ *contains a category of the form* (i), *then it contains no category of the form* (ii).

– $\Delta \subset \kappa(\mathcal{B} \setminus \mathcal{B}')$ *is a finite set of* distinguished *categories.*

The language generated by G is defined by

$$L(G) = \{w : \text{there exist } \tau \in I(w),\ \theta_i \in {\mathcal{B}'}^+,\ i = 1, \ldots, q, \text{ and } \delta \in \Delta \text{ such that } \tau[x_i := \theta_i, i = 1, \ldots, m] \leq \delta\},$$

where $\tau[x_i := \theta_i, i = 1, \ldots, q]$ is the result of simultaneous substitution of θ_i for x_i, $i = 1, \ldots, q$, in τ.

We end this section with the question whether the class of q-BAPGG languages is a proper subclass of the $(q+1)$-BAPGG ones.

9 Concluding Remarks

In our paper we argued that pregroup based grammars are a very convenient tool for describing mildly context-sensitive languages, introduced a new model of such grammars called (restricted) buffer augmented pregroup grammars, and established some of their basic properties. These grammars have a natural automaton counterpart, called (restricted) *buffer augmented pushdown automata* which are pushdown automata augmented with only once written additional memory – the buffer. The class of languages accepted by (restricted) *buffer augmented pushdown automata* coincide with the class of (mildly context-sensitive) languages generated by (restricted) buffer augmented pregroup grammars. This automaton model of computation will be published elsewhere.

References

1. Aizikowitz, T., Francez, N., Genkin, D., Kaminski, M.: Extending free pregroups with lower bounds. Studia Logica (to appear)
2. Buszkowski, W.: Lambek grammars based on pregroups. In: Groote, P.D., Morrill, G., Retorè, C. (eds.) LACL 2001. LNCS (LNAI), vol. 2099, pp. 95–109. Springer, Heidelberg (2001)
3. Buszkowski, W., Moroz, K.: Pregroup grammars and context-free grammars. In: Pre-proceedings of the international workshop on non-classical formal languages in linguistics, Tarragona, Spain, Research Group on Mathematical Linguistics, Universitat Rovira i Virgili (2007)
4. Francez, N., Kaminski, M.: Commutation-extended pregroup grammars and mildly context-sensitive languages. Studia Logica 87, 295–321 (2007)

5. Francez, N., Kaminski, M.: Commutation-augmented pregroup grammars and pushdown automaton with cancellation. Information and Computation 206, 1018–1032 (2008)
6. Hopcroft, J., Ullman, J.: Introduction to automata theory, languages, and computation. Addison-Wesley, Reading (1979)
7. Lambek, J.: Type grammars revisited. In: Lecomte, A., Perrier, G., Lamarche, F. (eds.) LACL 1997. LNCS (LNAI), vol. 1582, pp. 1–27. Springer, Heidelberg (1999)
8. Michaelis, J., Kracht, M.: Semilinearity as a syntactic invariant. In: Retoré, C. (ed.) LACL 1996. LNCS (LNAI), vol. 1328, pp. 329–345. Springer, Heidelberg (1997)
9. Vijay-Shanker, K., Weir, D.: The equivalence of four extensions of context-free grammars. Mathematical Systems Theory 27(6), 511–546 (1994)

Inference Rules for Proving the Equivalence of Recursive Procedures

Benny Godlin[1] and Ofer Strichman[2]

[1] Computer Science, Technion, Haifa, Israel
bgodlin@cs.technion.ac.il
[2] Information Systems engineering, IE, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Abstract. We present two proof rules for the equivalence of recursive procedures, in the style of Hoare's rule for recursive invocation of procedures. The first rule can be used for proving partial equivalence of programs; the second can be used for proving their mutual termination. There are various applications to these rules, such as proving backward compatibility.

1 Introduction

The ability to prove equivalence of programs can be useful for maintaining backward compatibility, or as a means for proving functional correctness where an earlier version of the program is trusted as a reference. Like functional verification, equivalence checking of general programs is undecidable, but it has the potential of being easier both because it does not require specifying what the system should do (the previous version of the code serves as the reference), and because it offers various opportunities for abstraction if the two programs share code or have similar call graphs, as we showed in [8].

The idea of proving equivalence between programs is not new, and in fact preceded the idea of functional verification. Indeed, in his pivotal 1969 paper about axiomatic basis for computer programming [9], Hoare points to previous works from the late 1950's on axiomatic treatment of the problem of proving equivalence between programs. It is a rather old challenge in the theorem-proving community. For example, a lot of attention has been given to this problem in the ACL2 community (some recent examples are [2,13,14]), although there the problem of proving equivalence between programs was a case study for using generic proof techniques, i.e., not specific for proving equivalence of similar programs. There is also a large body of work on proving equivalence between a source and a target of a compiler or a code generator in a process called *translation validation* [16]. This line of work focuses on mechanically generated programs, and on translation between languages (in some cases even translation between synchronous and sequential languages [15]). In contrast we are interested here in comparing two versions of a program, where the changes are manual and are thus arbitrary. In particular we are interested in *regression verification*, which is a process of proving the equivalence of two closely-related programs. Our position paper in [6] argues for the importance of this problem.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 167–184, 2010.

In this work we propose two proof rules for establishing equivalence between recursive procedures, which are inspired by Hoare's rule for recursive invocation of procedures [10]. We begin with defining various notions of equivalence.

1.1 Notions of Equivalence

We define six notions of Input/Output equivalence between P_1 and P_2.

1. **Partial equivalence:** Given the same inputs, any two terminating executions of P_1 and P_2 return the same value.
2. **Mutual termination:** Given the same inputs, P_1 terminates if and only if P_2 terminates.
3. **Reactive equivalence:** Given the same inputs, P_1 and P_2 emit the same output sequence.
4. **k-equivalence:** Given the same inputs, every two executions of P_1 and P_2 where
   - each loop iterates up to k times, and
   - each recursive call is not deeper than k,

   generate the same output.
5. **Full equivalence:** The two programs are partially equivalent and mutually terminate.
6. **Total equivalence:** The two programs are partially equivalent and both terminate.

Comments on this list:

- k-equivalence is the only notion of equivalence in this list that poses a decidable problem, assuming the program variables range over finite and discrete domains. For example, in the case of C programs k-equivalence can be proven with a tool such as CBMC [11].
- Reactive equivalence is targeted at reactive programs. In such programs it is assumed that inputs are read and outputs are written during the (presumably infinite) execution of the program.
- Full equivalence is very similar to 'strong equivalence' and 'functional equivalence' that were defined already in the early 1970's by Luckham et al. in [12] and by Pratt et al. in [17], respectively. The only difference is that in these early works there is an additional requirement that the two programs have the same set of variables.
- Total equivalence resembles the definition of program equivalence suggested by Bouge and Cachera [1].

We developed a proof rule for each of the first three notions of equivalence, which, note, are also useful for proving full and total equivalence (the fifth and sixth notions in the list above). Due to lack of space in this article we only focus on the first two rules, however. For the same reason we also do not include proofs of soundness, and a description of how these rules can be used with a language such as C. This missing information can be found in the long version of this article [7] and in the thesis of the first author [5].[1] Our main purpose in this article is to describe the rules and demonstrate how they can be used with a simple programming language.

[1] [7] and [5] can be downloaded from the 2nd author's home page at ie.technion.ac.il/∼ofers.

2 Preliminaries

2.1 The Programming Language

Let $Proc = \{p_0, \ldots, p_m\}$ denote a set of procedure names, where p_0 has a special role as the *root procedure* (the equivalent of 'main' in C). Let $\mathbb{D}$ be a domain that contains the constants TRUE and FALSE, and no subtypes. Let $O_{\mathbb{D}}$ be a set of operations (functions and predicates) over $\mathbb{D}$. We define a set of variables over this domain: $V = \bigcup_{p \in Proc} V_p$, where V_p is the set of variables of a procedure p. The sets V_p, $p \in Proc$ are pairwise disjoint.

The LPL language is modeled after PLW [4], but is different in various aspects. For example, it does not contain loops and allows only procedure calls by value-return.

Definition 1 (Linear Procedure Language (LPL)). *The* linear procedure language *(LPL) is defined by the following grammar, where lexical elements are bold and S denotes a statement:*

$$\begin{array}{l} \mathit{Program} :: \ \langle \textbf{procedure}\ p(\textbf{val}\ \overline{\mathit{arg\text{-}r_p}};\ \textbf{ret}\ \overline{\mathit{arg\text{-}w_p}}) : S_p \rangle_{p \in Proc} \\ S :: \ x := e \mid S;S \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi} \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{fi} \mid \\ \qquad \textbf{call}\ p(\overline{e};\overline{x}) \mid \textbf{return} \end{array}$$

where e is an expression over $O_{\mathbb{D}} \cup V_p$, and B is a predicate over $O_{\mathbb{D}} \cup V_p$. $\overline{\mathit{arg\text{-}r_p}}, \overline{\mathit{arg\text{-}w_p}}$ are vectors of V_p variables called, respectively, read formal arguments *and* write formal arguments*, and are used in the* body *S_p of the procedure named p. In a procedure call "***call** *$p(\overline{e};\overline{x})$", the expressions $\overline{e}$ are called the* actual input arguments *and $\overline{x}$ are called the* actual output variables*. The following constraints are assumed:*

1. *The only variables that can appear in the* procedure body *S_p are from V_p.*
2. *For each procedure call "***call** *$p(\overline{e},\overline{x})$" the lengths of $\overline{e}$ and $\overline{x}$ are equal to the lengths of $\overline{\mathit{arg\text{-}r_p}}$ and $\overline{\mathit{arg\text{-}w_p}}$, respectively.*
3. **return** *must appear at the end of any procedure body S_p ($p \in Proc$).*

For simplicity LPL is defined so it does not permit global variables and iterative expressions like **while** loops. Both of these syntactic restrictions do not constrain the expressive power of the language: global variables can be passed as part of the list of arguments of each procedure, and loops can be rewritten as recursive expressions.

The semantics of LPL follows what one would expect from such keywords in imperative languages. The formal operational semantics is given in Appendix A.

The equivalent prototype assumption. Two procedures

$$\begin{array}{l} \textbf{procedure}\ F(\textbf{val}\ \overline{\mathit{arg\text{-}r_F}};\textbf{ret}\ \overline{\mathit{arg\text{-}w_F}}), \\ \textbf{procedure}\ G(\textbf{val}\ \overline{\mathit{arg\text{-}r_G}};\textbf{ret}\ \overline{\mathit{arg\text{-}w_G}}) \end{array}$$

are said to have an equivalent prototype if $|\overline{\mathit{arg\text{-}r_F}}| = |\overline{\mathit{arg\text{-}r_G}}|$ and $|\overline{\mathit{arg\text{-}w_F}}| = |\overline{\mathit{arg\text{-}w_G}}|$.

We will assume that the two LPL programs P_1 and P_2 that we compare have the following property: $|Proc[P_1]| = |Proc[P_2]|$, and there is a 1-1 and onto mapping $map : Proc[P_1] \mapsto Proc[P_2]$ such that if $\langle F, G \rangle \in map$ then F and G have an equivalent prototype.

Programs that we wish to prove equivalent and do not fulfill this requirement can sometimes be brought to this state by applying inlining of procedures that cannot be mapped.

2.2 Computations and Subcomputations of LPL Programs

A computation of a program P in LPL is a sequence of configurations $\overline{C} = \langle C_0, C_1, \ldots \rangle$, where each configuration consists of the stack and the current state (valuation of the variables). For a variable v we denote its value in configuration C_i by $C_i.v$.

Definition 2 (Subcomputation). *A continuous subsequence of a computation is a* subcomputation *from level d if its first configuration C_0 is just before the call to a procedure p and has stack depth d, and all its configurations have a stack depth of at least d.*

Definition 3 (Maximal Subcomputation). *A* maximal subcomputation *from level d is a subcomputation from level d which is either*

- *infinite, or*
- *finite, and,*
  - *if $d > 0$ the successor of its last configuration has stack-depth smaller than d, and*
  - *if $d = 0$, then its last configuration is terminal.*

A finite maximal subcomputation is also called *closed*.

Example 1. In Fig. 1, the subsequence 8 – 11 is a finite (but not maximal) subcomputation from level $d+1$, and the subsequence 2 – 4 is a maximal subcomputation from level $d+1$.

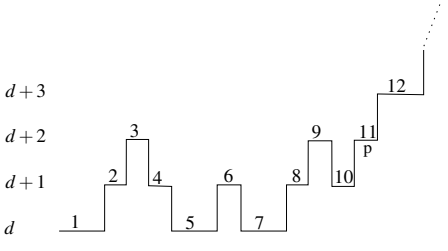


Fig. 1. A computation through various stack levels. Each rise corresponds to a procedure call, and each fall to a **return** statement.

Let π be a computation and π' a subcomputation of π. We denote by $\mathbf{first}(\pi')$ the first configuration in π', and by $\mathbf{last}(\pi')$ the last configuration in π', if π' is finite.

3 A Proof Rule for Partial Procedure Equivalence

We now proceed to define a proof rule for the partial equivalence of two LPL procedures. The rule refers to finite computations only.

Our running example for this section will be the two programs in Fig. 2, which compute recursively yet in different ways the GCD (Greatest Common Divisor) of two positive integers. We would like to prove their partial equivalence, namely that when they are called with the same inputs and terminate, they return the same result.

```
procedure gcd1(val a,b; ret g):
    if b = 0 then
        g := a
    else
        a := a mod b;
        call gcd1(b, a; g)
    fi;
    return
```

```
procedure gcd2(val x,y; ret z):
    z := x;
    if y > 0 then
        call gcd2(y, z mod y; z)
    fi;
    return
```

Fig. 2. Two procedures to calculate GCD of two positive integers

$$\frac{\forall \langle F, G\rangle \in map.\ p\text{-}equiv(\textbf{call}\ F, \textbf{call}\ G) \vdash p\text{-}equiv(F\ \textbf{body}, G\ \textbf{body})}{\forall \langle F, G\rangle \in map.\ p\text{-}equiv(\textbf{call}\ F, \textbf{call}\ G)}\ (\text{PROC-P-EQ})$$

Fig. 3. An inference rule for proving partial equivalence

3.1 Definitions

We now define more formally the notion of partial equivalence. The definitions that follow refer to subcomputations that begin right before the first statement in the procedure and end just before the **return** statement (of the same procedure at the same level in the stack), and use the formal arguments of the procedure.

Definition 4 (Argument-equivalence of Subcomputations). *Given two procedures $F \in Proc[P_1]$ and $G \in Proc[P_2]$ such that $\langle F, G\rangle \in map$, for any two computations π_1 in P_1 and π_2 in P_2, π_1' and π_2' are* argument-equivalent with respect to F and G *if the following holds:*

1. *π_1' and π_2' are maximal subcomputations of π_1 and π_2,*
2. *π_1' begins just before the call to F, and π_2' begins just before the call to G,*
3. *Equal read arguments:*

$$first(\pi_1').\overline{arg\text{-}r_F} = first(\pi_2').\overline{arg\text{-}r_G}$$

(here and later on we overload the equality sign to denote pairwise equivalence)

Definition 5 (Partial Equivalence). *If for every argument-equivalent finite subcomputations π_1' and π_2' with respect to two procedures F and G,*

$$last(\pi_1').\overline{arg\text{-}w_F} = last(\pi_2').\overline{arg\text{-}w_G}$$

then F and G are partially equivalent.

Denote by $p\text{-}equiv(F, G)$ the fact that F and G are partially equivalent. The equivalence is only partial because it does not consider infinite computations. Note that the definition of partial equivalence refers to closed subcomputations.

3.2 Rule (PROC-P-EQ)

Rule (PROC-P-EQ) appears in Fig. 3. The reader may notice the similarity to Hoare's rule for recursive invocation of procedures from almost four decades ago:

$$\frac{\{p\}\ \textbf{call}\ proc\ \{q\} \vdash_H \{p\} proc\ \textbf{body}\ \{q\}}{\{p\}\ \textbf{call}\ proc\ \{q\}}\ \text{(REC)}$$

This unintuitive rule was described by Hoare in [10] as follows: *The solution... is simple and dramatic: to permit the use of the desired conclusion as a hypothesis in the proof of the body itself.* Rule (PROC-P-EQ) uses exactly the same principle.

Whereas (REC) was defined by Hoare in the context of *deductive* proofs[2], we use (PROC-P-EQ) in a context of a *model-theoretic* proof. A possible way to prove the premise in this context is to replace the recursive calls with calls to two functions that are

- partially equivalent by construction, and,
- they abstract (over-approximates) the original functions.

Obvious candidates for such replacements are *uninterpreted procedures*. An uninterpreted procedure U is the same as an empty procedure in LPL (a procedure with a single statement – **return**), other than the fact that it preserves partial equivalence: For every two subcomputations π_1 and π_2 through U,

$$\begin{array}{l} \text{first}(\pi_1).\overline{arg\text{-}r_U} = \text{first}(\pi_2).\overline{arg\text{-}r_U} \\ \rightarrow \\ \text{last}(\pi_1).\overline{arg\text{-}w_U} = \text{last}(\pi_2).\overline{arg\text{-}w_U}\ . \end{array} \tag{1}$$

(this is similar to the classical congruence axiom that is defined for uninterpreted functions).

Let UP be a mapping of the procedures in $Proc[P_1] \cup Proc[P_2]$ to respective uninterpreted procedures, such that:

$$\langle F, G\rangle \in map \iff \text{UP}(F) = \text{UP}(G)\ , \tag{2}$$

and such that each procedure is mapped to an uninterpreted procedure with an equivalent prototype.

Definition 6 (Isolated Procedure). *The* isolated *version of a procedure F, denoted F^{UP}, is derived from F by replacing all of its procedure calls by calls to the corresponding uninterpreted procedures, i.e., $F^{UP} \doteq F[f \leftarrow UP(f) | f \in Proc[P]]$.*

For example, Fig. 5 presents an isolated version of the programs in Fig. 2.

Fig. 4 presents a reformulation of rule (PROC-P-EQ). This version of the rule is theoretically weaker than the original one, because it enforces a specific method for proving the premise (namely, by proving partial equivalence of the isolated procedures). On the other hand it reflects the way we actually use it in our system.

[2] To use it one should: 1) assume p as a premise in the beginning of *proc*, 2) prove that p holds right before calling *proc*, 3) assume q as a premise right after the call to *proc*, and 4) prove that q holds in the end of *proc*. Then by rule (REC) $\{p\}$ **call** $\{q\}$ holds.

$$\frac{(1)\ \forall\langle F,G\rangle \in map.\ p\text{-}equiv(\textbf{call}\ F^{UP}, \textbf{call}\ G^{UP})}{(2)\qquad \forall\langle F,G\rangle \in map.\ p\text{-}equiv(\textbf{call}\ F, \textbf{call}\ G)}$$

Fig. 4. A variation on rule (PROC-P-EQ) that suggests a specific method to prove the premise

Proving the premise of this rule can be done with any sound proof system for a restricted version of the programming language, in which there are no calls to interpreted procedures, and hence, in particular, no recursion[3]. In addition, such a proof system has to be able to reason about uninterpreted procedures.

Example 2. Following are two instantiations of rule (PROC-P-EQ). We omit the **call** keyword.

- The two programs contain one recursive procedure each, called f and g such that $map = \{\langle f,g\rangle\}$.
$$\frac{p\text{-}equiv(f[f \leftarrow \mathrm{UP}(f)], g[g \leftarrow \mathrm{UP}(g)])}{p\text{-}equiv(f,g)}$$
Recall that the isolation $f[f \leftarrow \mathrm{UP}(f)]$ means that the call to f inside f is replaced with a call to UP(f).
- The two compared programs contain two mutually recursive procedures each, f_1, f_2 and g_1, g_2 respectively, such that $map = \{\langle f_1,g_1\rangle, \langle f_2,g_2\rangle\}$, and f_1 calls f_2, f_2 calls f_1, g_1 calls g_2 and g_2 calls g_1.
$$\frac{\begin{array}{l} p\text{-}equiv(f_1[f_2 \leftarrow \mathrm{UP}(f_2)], g_1[g_2 \leftarrow \mathrm{UP}(g_2)]) \wedge \\ p\text{-}equiv(f_2[f_1 \leftarrow \mathrm{UP}(f_1)], g_2[g_1 \leftarrow \mathrm{UP}(g_1)]) \end{array}}{p\text{-}equiv(f_1,g_1) \wedge p\text{-}equiv(f_2,g_2)}$$

□

Example 3. Consider once again the two programs in Fig. 2. There is only one procedure in each program, which we naturally map to one another. Let H be the uninterpreted procedure to which we map gcd_1 and gcd_2, i.e., $H = \mathrm{UP}(gcd_1) = \mathrm{UP}(gcd_2)$. Figure 5 presents the isolated programs.

To prove the partial equivalence of the two procedures, we need to first translate them to formulas expressing their respective transition relations. A convenient way to do so is to use Static Single Assignment (SSA) [3]. Briefly, this means that in each assignment of the form `x = exp;` the left-hand side variable `x` is replaced with a new variable, say $\mathtt{x}_1$. Any reference to `x` after this line and before `x` is potentially assigned again, is replaced with the new variable $\mathtt{x}_1$ (recall that this is done in a context of a program without loops). In addition, assignments are guarded according to the control flow. After this transformation, the statements are conjoined: the resulting equation represents the states

[3] In LPL there are no loops, but in case (PROC-P-EQ) is applied to other languages, the proof engine is required to handle a restricted version of the language with no procedure calls, recursion or loops. Under this restriction there are sound and complete decision procedures for deciding the validity of assertions over popular programming languages such as C.

```
procedure gcd1(val a,b; ret g):
    if b = 0 then
        g := a
    else
        a := a mod b;
        call H(b, a; g)
    fi;
    return
```

```
procedure gcd2(val x,y; ret z):
    z := x;
    if y > 0 then
        call H(y, z mod y; z)
    fi;
    return
```

Fig. 5. After isolation of the procedures, i.e., replacing their procedure calls with calls to the uninterpreted procedure H

of the original program. If a subcomputation through a procedure is valid then it can be associated with an assignment that satisfies the SSA form of this procedure.

The SSA form of gcd_1 and gcd_2 is:

$$T_{gcd_1} = \begin{pmatrix} a_0 = a & \wedge \\ b_0 = b & \wedge \\ b_0 = 0 \rightarrow \ g_0 = a_0 & \wedge \\ (b_0 \neq 0 \rightarrow a_1 = (a_0 \mod b_0)) & \wedge \\ (b_0 = 0 \rightarrow a_1 = a_0) & \wedge \\ (b_0 \neq 0 \rightarrow H(b_0, a_1; g_1)) & \wedge \\ (b_0 = 0 \rightarrow g_1 = g_0) & \wedge \\ g = g_1 & \end{pmatrix} \tag{3}$$

$$T_{gcd_2} = \begin{pmatrix} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ (y_0 > 0 \rightarrow H(y_0, (z_0 \mod y_0); z_1)) & \wedge \\ (y_0 \leq 0 \rightarrow z_1 = z_0) & \wedge \\ z = z_1 & \end{pmatrix} \tag{4}$$

The premise of rule (PROC-P-EQ) requires proving partial equivalence of the isolated procedures (see Definition 5), which in this case amounts to proving the validity of the following formula over positive integers:

$$(a = x \wedge b = y \wedge T_{gcd_1} \wedge T_{gcd_2}) \quad \rightarrow \quad g = z\,. \tag{5}$$

Many theorem provers can prove such formulas fully automatically, and hence establish the partial equivalence of gcd_1 and gcd_2. □

It is important to note that while the premise refers to procedures that are isolated from other procedures, the consequent refers to the original procedures. Hence, while our decision procedure is required to reason about executions of bounded length (the length of one procedure body) the consequent refers to unbounded executions.

In [7] we prove:

Theorem 1 (Soundness). *Rule* (PROC-P-EQ) *is sound.*

4 A Proof Rule for Mutual Termination of Procedures

Rule (PROC-P-EQ) only proves partial equivalence, because it only refers to finite computations. It is desirable, in the context of equivalence checking, to prove that the two procedures mutually terminate.

4.1 Definitions

Definition 7 (Mutual Termination of Procedures). *If for every pair of argument-equivalent subcomputations π'_1 and π'_2 with respect to two procedures F and G, it holds that π'_1 is finite if and only if π'_2 is finite, then F and G are* mutually terminating.

Denote by $mutual\text{-}terminate(F,G)$ the fact that F and G are mutually terminating.

Definition 8 (Reach Equivalence of Procedures). *Procedures F and G are* reach-equivalent *if for every pair of argument-equivalent subcomputations π'_1 and π'_2 through F and G respectively, for every call statement $c_F =$ "**call** p_1" in F there exists a call statement $c_G =$ "**call** p_2" in G (and vice versa) such that $\langle p_1, p_2\rangle \in map$, and π'_1 and π'_2 reach c_F and c_G respectively with the same read arguments, or do not reach them at all.*

Denote by $reach\text{-}equiv(F,G)$ the fact that F and G are reach-equivalent. Note that checking for reach-equivalence amounts to proving the equivalence of the 'guards' leading to each of the mapped procedure calls (i.e., the conjunction of conditions that need to be satisfied in order to reach these program locations), and the equivalence of the arguments before the calls. This will be demonstrated later on.

4.2 Rule (M-TERM)

The mutual termination rule (M-TERM) is stated in Fig. 6. It is interesting to note that unlike proofs of procedure termination, here we do not rely on well-founded sets (see, for example, Sect.3.4 of [4]).

Example 4. Continuing Example 3, we now prove the mutual termination of the two programs in Fig. 2. Since we already proved $p\text{-}equiv(gcd_1, gcd_2)$ in Example 3, it is left to check Premise (3), i.e.,

$$reach\text{-}equiv({gcd_1}^{UP}, {gcd_2}^{UP}) \ .$$

$$\frac{\begin{array}{ll}(1) & \forall\langle F,G\rangle \in map.\ \{\\ (2) & \quad p\text{-}equiv(F,G) \wedge\\ (3) & \quad reach\text{-}equiv(F^{UP},G^{UP})\ \}\end{array}}{(4)\quad \forall\langle F,G\rangle \in map.\ mutual\text{-}terminate(F,G)}\ \text{(M-TERM)}$$

Fig. 6. Rule (M-TERM): An inference rule for proving the mutual termination of procedures. Note that Premise (2) can be proven by the (PROC-P-EQ) rule.

Since in this case we only have a single procedure call in each side, the only thing we need to check in order to establish reach-equivalence, is that the guards controlling their calls are equivalent, and that they are called with the same input arguments. The verification condition is thus:

$$\begin{array}{l}(T_{gcd_1} \wedge T_{gcd_2} \wedge (a = x) \wedge (b = y)) \rightarrow \\ (\,((b_0 \neq 0) \leftrightarrow (y_0 > 0)) \wedge \qquad \textit{//Equal guards} \\ \;\;((b_0 \neq 0) \rightarrow ((b_0 = y_0) \wedge (a_1 = z_0 \bmod y_0)))\,) \;\; \textit{//Equal inp.}\end{array} \tag{6}$$

where T_{gcd_1} and T_{gcd_2} are as defined in Eq. (3) and (4). □

In [7] we prove:

Theorem 2 (Soundness). *Rule* (M-TERM) *is sound.*

4.3 Using Rule (M-TERM): A Long Example

In this example we set the domain $\mathbb{D}$ to be the set of binary trees with natural values in the leafs and the + and * operators at internal nodes.[4]

Let $t_1, t_2 \in \mathbb{D}$. We define the following operators:

- isleaf(t_1) returns TRUE if t_1 is a leaf and FALSE otherwise.
- isplus(t_1) returns TRUE if t_1 has '+' in its root node and FALSE otherwise.
- leftson(t_1) returns FALSE if t_1 is a leaf, and the tree which is its left son otherwise.
- doplus(l_1, l_2) returns a leaf with a value equal to the sum of the values in l_1 and l_2, if l_1 and l_2 are leafs, and FALSE otherwise.

The operators ismult(t_1), rightson(t_1) and domult(t_1, t_2) are defined similarly to isplus, leftson and doplus, respectively.

The two procedures in Fig. 7 calculate the value of an expression tree.

We introduce three uninterpreted procedures E, P and M and set the mapping UP to satisfy

$$\begin{array}{l}\mathrm{UP}(Eval_1) = \mathrm{UP}(Eval_2) = E, \\ \mathrm{UP}(Plus_1) = \mathrm{UP}(Plus_2) = P, \\ \mathrm{UP}(Mult_1) = \mathrm{UP}(Mult_2) = M\,.\end{array}$$

The SSA form of the formulas which represent the possible computations of the isolated procedure bodies are presented in Fig. 8.

Proving partial equivalence for each of the procedure pairs amounts to proving the following formulas to be valid:

$$\begin{array}{l}(a = x \wedge T_{Eval_1} \wedge T_{Eval_2}) \rightarrow r = y \\ (a = x \wedge T_{Plus_1} \wedge T_{Plus_2}) \rightarrow r = y \\ (a = x \wedge T_{Mult_1} \wedge T_{Mult_2}) \rightarrow r = y\,.\end{array}$$

To prove these formulas it is enough for L_{UF} to know the following facts about the operators of the domain:

[4] To be consistent with the definition of LPL (Definition 1), the domain must also include TRUE and FALSE. Hence we also set the constants TRUE and FALSE to be the leafs with 1 and 0 values respectively.

```
procedure Eval1(val a; ret r):
    if isleaf(a) then
        r := a
    else
        if isplus(a) then
            call Plus1(a; r)
        else
            if ismult(a) then
                call Mult1(a; r)
            fi
        fi
    fi
    return

procedure Eval2(val x; ret y):
    if isleaf(x) then
        y := x
    else
        if ismult(x) then
            call Mult2(x; y)
        else
            if isplus(x) then
                call Plus2(x; y)
            fi
        fi
    fi
    return

procedure Plus1(val a; ret r):
    call Eval1(leftson(a); v);
    call Eval1(rightson(a); u);
    r := doplus(v, u);
    return

procedure Plus2(val x; ret y):
    call Eval2(rightson(x); w);
    call Eval2(leftson(x); z);
    y := doplus(w, z);
    return

procedure Mult1(val a; ret r):
    call Eval1(leftson(a); v);
    call Eval1(rightson(a); u);
    r := domult(v, u);
    return

procedure Mult2(val x; ret y):
    call Eval2(rightson(x); w);
    call Eval2(leftson(x); z);
    y := domult(w, z);
    return
```

Fig. 7. Two procedures to calculate the value of an expression tree

$$
\begin{array}{ll}
\forall l_1, l_2 & (doplus(l_1, l_2) = doplus(l_2, l_1) \wedge \\
& \quad domult(l_1, l_2) = domult(l_2, l_1)) \\
\forall t_1 & (isleaf(t_1) \rightarrow \neg isplus(t_1) \wedge \neg ismult(t_1)) \\
\forall t_1 & (isplus(t_1) \rightarrow \neg ismult(t_1) \wedge \neg isleaf(t_1)) \\
\forall t_1 & (ismult(t_1) \rightarrow \neg isleaf(t_1) \wedge \neg isplus(t_1))
\end{array}
$$

This concludes the proof of partial equivalence using rule (PROC-P-EQ). To prove mutual termination using the (M-TERM) rule we need in addition to verify reach equivalence of each pair of procedures.

To check reach-equivalence we should check that the guards and the read arguments of related calls are equivalent. This can be expressed by the following formulas:

$$
\begin{array}{ll}
\varphi_1 = (& \\
guard_{Plus_1} = (\neg isleaf(a_0) \wedge isplus(a_0)) & \wedge \\
guard_{Plus_2} = (\neg isleaf(x_0) \wedge \neg ismult(x_0) \wedge isplus(x_0)) & \wedge \\
guard_{Mult_1} = (\neg isleaf(a_0) \wedge \neg isplus(a_0) \wedge ismult(a_0)) & \wedge \\
guard_{Mult_2} = (\neg isleaf(x_0) \wedge ismult(x_0)) & \wedge \\
guard_{Plus_1} \leftrightarrow guard_{Plus_2} & \wedge \\
guard_{Mult_1} \leftrightarrow guard_{Mult_2} & \wedge \\
guard_{Plus_1} \rightarrow a_0 = x_0 & \wedge \\
guard_{Mult_1} \rightarrow a_0 = x_0) &
\end{array}
$$

$$T_{Eval_1} = \begin{pmatrix} a_0 = a & \wedge \\ (isleaf(a_0) \rightarrow r_1 = a_0) & \wedge \\ (\neg isleaf(a_0) \wedge isplus(a_0) \rightarrow P(a_0, r_1)) & \wedge \\ (\neg isleaf(a_0) \wedge \neg isplus(a_0) \wedge ismult(a_0) \rightarrow & \\ \quad M(a_0, r_1)) & \wedge \\ r = r_1 & \end{pmatrix}$$

$$T_{Eval_2} = \begin{pmatrix} x_0 = x & \wedge \\ (isleaf(x_0) \rightarrow y_1 = x_0) & \wedge \\ (\neg isleaf(x_0) \wedge ismult(x_0) \rightarrow M(x_0, y_1)) & \wedge \\ (\neg isleaf(x_0) \wedge \neg ismult(x_0) \wedge isplus(x_0) \rightarrow & \\ \quad P(x_0, y_1)) & \wedge \\ y = y_1 & \end{pmatrix}$$

$$T_{Plus_1} = \begin{pmatrix} a_0 = a & \wedge \\ E(leftson(a_0), v_1) & \wedge \\ E(rightson(a_0), u_1) & \wedge \\ r_1 = doplus(v_1, u_1) & \wedge \\ r = r_1 & \end{pmatrix} \quad T_{Plus_2} = \begin{pmatrix} x_0 = x & \wedge \\ E(rightson(x_0), w_1) & \wedge \\ E(leftson(x_0), z_1) & \wedge \\ y_1 = doplus(w_1, z_1) & \wedge \\ y = y_1 & \end{pmatrix}$$

$$T_{Mult_1} = \begin{pmatrix} a_0 = a & \wedge \\ E(leftson(a_0), v_1) & \wedge \\ E(rightson(a_0), u_1) & \wedge \\ r_1 = domult(v_1, u_1) & \wedge \\ r = r_1 & \end{pmatrix} \quad T_{Mult_2} = \begin{pmatrix} x_0 = x & \wedge \\ E(rightson(x_0), w_1) & \wedge \\ E(leftson(x_0), z_1) & \wedge \\ y_1 = domult(w_1, z_1) & \wedge \\ y = y_1 & \end{pmatrix}$$

Fig. 8. SSA form of the two calculators

The guards at all labels in $Plus_1, Plus_2, Mult_1$ and $Mult_2$ are all true, therefore the reach-equivalence formulas for these procedures collapse to checking consistency between the read arguments of the procedures:

$$\begin{aligned} &\varphi_2 = \varphi_3 = \\ &(leftson(a_0) = rightson(x_0) \wedge rightson(a_0) = leftson(x_0)) \quad \vee \\ &(leftson(a_0) = leftson(x_0) \wedge (rightson(a_0) = rightson(x_0))) \end{aligned}$$

Finally, the formulas that need to be validated are:

$$\begin{aligned} (a = x \wedge T_{Eval_1} \wedge T_{Eval_2}) &\rightarrow \varphi_1 \\ (a = x \wedge T_{Plus_1} \wedge T_{Plus_2}) &\rightarrow \varphi_2 \\ (a = x \wedge T_{Mult_1} \wedge T_{Mult_2}) &\rightarrow \varphi_3 \ . \end{aligned}$$

5 What the Rules Cannot Prove

The two inference rules rely on a 1-1 and onto mapping of the procedures (possibly after inlining of some of them, as mentioned in the introduction), such that every pair of mapped procedures are partially equivalent after isolation. Various semantic-preserving code transformations do not satisfy this requirement. Here are a few examples:

1. Consider the following equivalent procedures, which, for a natural number n, compute $\sum_{i=1}^{n} i$.

```
procedure F(val n; ret r):
    if n ≤ 1 then r := n
    else
        call F(n−1, r);
        r := n + r
    fi
    return
```

```
procedure G(val n; ret r):
    if n ≤ 1 then r := n
    else
        call G(n−2, r);
        r := n + (n−1) + r
    fi
    return
```

Since the two procedures are called with different arguments, their partial equivalence cannot be proven with rule (PROC-P-EQ).

2. Consider a similar pair of equivalent procedures, that this time make recursive calls with equivalent arguments:

```
procedure F(val n; ret r):
    if n ≤ 0 then r := n
    else
        call F(n−1, r);
        r := n + r
    fi
    return
```

```
procedure G(val n; ret r):
    if n ≤ 1 then r := n
    else
        call G(n−1, r);
        r := n + r
    fi
    return
```

The premise of (PROC-P-EQ) fails due to the case of $n == 1$.

3. We now consider an example in which the two programs are both computational equivalent and reach-equivalent, but still our rules fail to prove it.

```
procedure F(val n; ret r):
    if n ≤ 0 then r := 0
    else
        call F(n−1, r);
        r := n + r
    fi;
    return
```

```
procedure G(val n; ret r):
    if n ≤ 0 then r := 0
    else
        call G(n−1, r);
        if r ≥ 0 then
            r := n + r;
        fi
    fi
    return
```

In this case the 'if' condition in the second program always holds. Yet since the uninterpreted functions return arbitrary, although equal, values, they can return a negative value, which will make this 'if' condition not hold and as a result make the two isolated functions return different values.

6 Summary

We presented two proof rules in the style of Hoare's rule for recursive procedures: rule (PROC-P-EQ) proves partial equivalence between programs and rule (M-TERM) proves mutual termination of such programs.

These rules can be used in any one of the scenarios described in the introduction. We are using them as part of an automated regression-verification tool for C programs that we currently develop, and so far used them for proving such equivalence of several non-trivial programs. In [5] we describe a recursive algorithm that traverses the call graphs of the compared programs, and each time attempts to prove the equivalence of mapped procedures after abstracting with uninterpreted functions callees that were already proven to be equal.

Our experience so far is that deriving the proper verification conditions automatically is easy in the isolated procedures, and the verification conditions themselves are typically not hard to solve with CBMC, the underlying proof engine for C that we use.[5] Once this system will be capable of handling a larger set of real programs it will be interesting to see if real changes made between versions of real programs can be proven to be equal with the rules described here.

References

1. Bouge, L., Cachera, D.: A logical framework to prove properties of alpha programs (revised version). Technical Report RR-3177 (1997)
2. Craciunescu, S.: Proving the equivalence of CLP programs. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 287–301. Springer, Heidelberg (2002)
3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
4. Francez, N.: Program Verification. Addison-Wesley, Reading (1993)
5. Godlin, B.: Regression verification: Theoretical and implementation aspects. Master's thesis, Technion, Israel Institute of Technology (2008)
6. Godlin, B., Strichman, O.: Regression verification - a practical way to verify programs. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 496–501. Springer, Heidelberg (2008); Conf in 2005, published in 2007
7. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Informatica 45(6), 403–439 (2008)
8. Godlin, B., Strichman, O.: Regression verification. In: 46th Design Automation Conference, DAC (2009)
9. Hoare, C.: An axiomatic basis for computer programming. ACM Comm. 12(10), 576–580 (1969)
10. Hoare, C.: Procedures and parameters: an axiomatic approach. In: Proc. Sym. on semantics of algorithmic languages, vol. (188) (1971)
11. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of DAC 2003, pp. 368–371. ACM Press, New York (2003)
12. Luckham, D., Park, D., Paterson, M.: On formalized computer programs. J. Comp. Systems Sci. 4(3), 220–249 (1970)
13. Manolios, P., Kaufmann, M.: Adding a total order to acl2. In: The Third International Workshop on the ACL2 Theorem Prover (2002)
14. Manolios, P., Vroon, D.: Ordinal arithmetic: Algorithms and mechanization. Journal of Automated Reasoning (to appear, 2006)
15. Pnueli, A., Siegel, M., Shtrichman, O.: Translation validation for synchronous languages. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 235–246. Springer, Heidelberg (1998)
16. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. Technical report, Sacres and Dept. of Comp. Sci., Weizmann Institute (April 1997)
17. Pratt, T.W.: Kernel equivalence of programs and proving kernel equivalence and correctness by test cases. In: International Joint Conference on Artificial Intelligence (1971)

[5] This depends more on the type of operators there are in the procedures than the sheer size of the program. For example, a short program that includes a multiplication between two integers is hard to reason about regardless of the length of the procedure.

A Operational Semantics of LPL

A.1 Notation of Sequences

In the following an n-long sequence is denoted by $\langle l_0,\ldots,l_{n-1}\rangle$ or by $\langle l_i\rangle_{i\in\{0,\ldots,n-1\}}$. If the sequence is infinite we write $\langle l_i\rangle_{i\in\{0,\ldots\}}$. Given two sequences $\overline{a}=\langle a_i\rangle_{i\in\{0,\ldots,n-1\}}$ and $\overline{b}=\langle b_i\rangle_{i\in\{0,\ldots,m-1\}}$,

$$\overline{a}\cdot\overline{b}$$

is their concatenation of length $n+m$.

We overload the equality sign ($=$) to denote sequence equivalence. Given two finite sequences $\overline{a}$ and $\overline{b}$

$$(\overline{a}=\overline{b})\Leftrightarrow(|\overline{a}|=|\overline{b}|\wedge\forall i\in\{0,\ldots,|\overline{a}|-1\}.\ a_i=b_i)\ ,$$

where $|\overline{a}|$ and $|\overline{b}|$ denote the number of elements in $\overline{a}$ and $\overline{b}$, respectively.

If both $\overline{a}$ and $\overline{b}$ are infinite then

$$(\overline{a}=\overline{b})\Leftrightarrow(\forall i\geq 0.\ a_i=b_i)\ ,$$

and if exactly one of $\{\overline{a},\overline{b}\}$ is infinite then $\overline{a}\neq\overline{b}$.

A.2 Operational Semantics

We denote the set of labels in the body of procedure $p\in Proc$ by PC_p. Together the set of all labels is $PC\doteq\bigcup_{p\in Proc}PC_p$.

A computation of a program P in LPL is a sequence of configurations. Each configuration $C=\langle d,O,\overline{pc},\sigma\rangle$ contains the following elements:

1. The natural number d is the depth of the stack at this configuration.
2. The function $O:\{0,\ldots,d\}\mapsto Proc$ is the *order of procedures* in the stack at this configuration.
3. $\overline{pc}=\langle pc_0,pc_1\ldots,pc_d\rangle$ is a vector of program location labels[6] such that $pc_0\in PC_0$ and for each call level $i\in\{1,\ldots,d\}$ $pc_i\in PC_{O[i]}$ (i.e., pc_i "points" into the procedure body that is at the i^{th} place in the stack).
4. The function $\sigma:\{0,\ldots,d\}\times V\mapsto\mathbb{D}\cup\{nil\}$ is a *valuation* of the variables V of program P at this configuration. The value of variables which are not active at the i-th call level is invalid i.e., for $i\in\{0,\ldots,d\}$, if $O[i]=p$ and $v\in V\setminus V_p$ then $\sigma[\langle i,v\rangle]=nil$ where $nil\notin\mathbb{D}$ denotes an invalid value.

A valuation is implicitly defined over a configuration. For an expression e over $\mathbb{D}$ and V, we define the value of e in σ in the natural way, i.e., each variable evaluates according to the procedure and the stack depth defined by the configuration. More formally, for a configuration $C=\langle d,O,\overline{pc},\sigma\rangle$ and a variable x:

$$\sigma[x]\doteq\begin{cases}\sigma[\langle d,x\rangle] & if\ \ x\in V_p\ and\ p=O[d]\\ nil & otherwise\end{cases}$$

[6] $\overline{pc}$ can be thought of as a stack of program counters, hence the notation.

This definition extends naturally to a vector of expressions.

When referring to a specific configuration C, we denote its elements $d, O, \overline{pc}, \sigma$ with $C.d, C.O, C.\overline{pc}, C.\sigma[x]$ respectively.

For a valuation σ, expression e over $\mathbb{D}$ and V, levels $i, j \in \{0,\ldots,d\}$, and a variable x, we denote by $\sigma[\langle i,e\rangle|\langle j,x\rangle]$ a valuation identical to σ other than the valuation of x at level j, which is replaced with the valuation of e at level i. When the respective levels are clear from the context, we may omit them from the notation.

Finally, we denote by $\sigma|_i$ a valuation σ *restricted to level* i, i.e., $\sigma|_i[v] \doteq \sigma[\langle i,v\rangle]$ $(v \in V)$.

For a configuration $C = \langle d, O, \overline{pc}, \sigma\rangle$ we denote by **current-label**$[C]$ the program location label at the procedure that is topmost on the stack, i.e., current-label$[C] \doteq pc_d$.

Definition 9 (Initial and Terminal Configurations in LPL). *A configuration* $C = \langle d, O, \overline{pc}, \sigma\rangle$ *with current-label*$[C] = before[S_{p_0}]$ *is called the* initial *configuration and must satisfy* $d = 0$ *and* $O[0] = p_0$. *A configuration with current-label*$[C] = after[S_{p_0}]$ *is called the* terminal *configuration.*

Definition 10 (Transition Relation in LPL). *Let '*$\rightarrow$*' be the least relation among configurations which satisfies: if* $C \rightarrow C'$, $C = \langle d, O, \overline{pc}, \sigma\rangle$, $C' = \langle d', O', \overline{pc}', \sigma'\rangle$ *then:*

1. *If current-label*$[C] = before[S]$ *for some assign construct* $S =$ "$x := e$" *then* $d' = d$, $O' = O$, $\overline{pc}' = \langle pc_i\rangle_{i\in\{0,\ldots,d-1\}} \cdot \langle after[S]\rangle$, $\sigma' = \sigma[e|x]$.
2. *If current-label*$[C] = before[S]$ *for some construct*

$$S \;=\; \text{“}\textbf{if}\ B\ \ \textbf{then}\ \ S_1\ \textbf{else}\ S_2\ \textbf{fi}\text{”}$$

then

$$d' = d,\ O' = O,\ \overline{pc}' = \langle pc_i\rangle_{i\in\{0,\ldots,d-1\}} \cdot \langle lab_B\rangle,\ \sigma' = \sigma$$

where

$$lab_B = \begin{cases} before[S_1] & \text{if } \sigma[B] = \text{TRUE} \\ before[S_2] & \text{if } \sigma[B] = \text{FALSE} \end{cases}$$

3. *If current-label*$[C] = after[S_1]$ *or current-label*$[C] = after[S_2]$ *for some construct*

$$S \;=\; \text{“}\textbf{if}\ B\ \ \textbf{then}\ \ S_1\ \textbf{else}\ S_2\ \textbf{fi}\text{”}$$

then

$$d' = d,\ O' = O,\ \overline{pc}' = \langle pc_i\rangle_{i\in\{0,\ldots,d-1\}} \cdot \langle after[S]\rangle,\ \sigma' = \sigma$$

4. *If current-label*$[C] = before[S]$ *for some call construct* $S =$ "**call** $p(\overline{e};\overline{x})$" *then* $d' = d+1$, $O' = O \cdot \langle p\rangle$, $\overline{pc}' = \langle pc_i\rangle_{i\in\{0,\ldots,d-1\}} \cdot \langle after[S]\rangle \cdot \langle before[S_p]\rangle$, $\sigma' = \sigma[\langle d, e_1\rangle |\langle d+1, (arg\text{-}r_p)_1\rangle] \ldots [\langle d, e_l\rangle|\langle d+1, (arg\text{-}r_p)_l\rangle]$ *where* $\overline{arg\text{-}r_p}$ *is the vector of formal read variables of procedure p and l is its length.*
5. *If current-label*$[C] = before[S]$ *for some return construct* $S =$ "**return**" *and* $d > 0$ *then* $d' = d-1$, $O' = \langle O_i\rangle_{i\in\{1,\ldots,d-1\}}$, $\overline{pc}' = \langle pc_i\rangle_{i\in\{0,\ldots,d-1\}}$, $\sigma' = \sigma[\langle d, (arg\text{-}w_p)_1\rangle |\langle d-1, x_1\rangle] \ldots [\langle d, (arg\text{-}w_p)_l\rangle|\langle d-1, x_l\rangle]$ *where* $\overline{arg\text{-}w_p}$ *is the vector of formal write variables of procedure p, l is its length, and* $\overline{x}$ *are the actual output variables of the* **call** *statement immediately before* pc_{d-1}.

6. *If current-label*$[C] =$ *before*$[S]$ *for some return construct* $S =$ "**return**" *and* $d = 0$ *then* $d' = 0$, $O' = \langle p_0 \rangle$, $\overline{pc}' = \langle after[S_{p_0}] \rangle$ *and* $\sigma' = \sigma$.

Note that the case of current-label$[C] =$ before$[S]$ for a construct $S = S_1;S_2$ is always covered by one of the cases in the above definition.

Another thing to note is that all write arguments are copied to the actual variables following a **return** statement. This solves possible problems that may occur if the same variable appears twice in the list of write arguments.

Amir Pnueli

Years ago, when I was doing my masters degree, I worked with an advisor who came from the industry and kept me rather isolated from the academic world. As part of my thesis I used a variant of temporal logic, but without ever reading Pnueli's articles and books. When I graduated and looked for a PhD advisor someone asked me: why don't you go directly to the top? Try Pnueli. Pnueli? I asked. For me this name was so mythological that it did not occur to me that there is a living, active person behind this name, and not only that: he happens to live not more than a 100 Km from my home.

It took me time to realize what a scientific giant he really was. One doesn't see clearly such things from up close. Several months after I started working with him he received the Turing award, and further down the line came the other prizes: The Israel prize, the invitation to join the Israel Academy of Science, the invitation to join the American academy of Engineering, the honorary doctorates from various places, and so on. All of these prizes made it, let's say, quite clear. But it became clearer than ever to me only when I graduated and joined, as a postdoc, Clarke's group in CMU. In one of the first group meetings there that I attended, they discussed the new four servers that were just bought for the group. And the names they chose for the servers? Tarski, Bool, Pnueli and Hilbert. There we go: a living myth. You are not really big until someone names a server after you. I was amused by this naming and thought of future conversations: Pnueli is the fastest, Pnueli is busy, Pnueli is overloaded... all too familiar adjectives.

Alan Lightman, an MIT physicist and a novelist, once wrote a popular book called *Einstein's dreams*, which is dedicated to implications of time being different than the way we know it. What would happen if we could slide backward and forward as we wish along the time line? What would it feel like if time were to move faster when we moved faster? what would happen if time moved faster the closer we are to the center of the city? One of the chapters is dedicated to how the world would look like if people could stop time. The first and only example he gives is that of a PhD student that decides to stop time so he can stay such a student forever. I completely related to this. Who wants to go out there? who wants to work with anyone else after working with Amir? When I graduated I sent this book to Amir and wrote him a note that this student is me: I would stop the time if I only could.

It was clear to me that after being Amir's student, any alternative is a downfall. I believe this feeling is common to all his former students. Amir was not only the smartest person I ever met, but also the most generous, kind, and above all modest. To me and to many others Amir is the ultimate role model, not only as a scientist, but also as a

person. In the same note I wrote to him something to that effect (I used the Hebrew word 'Mofet', which seems to be more majestic), but in retrospect I should have said more. So few are the opportunities we have to thank people for what they are and so many are the excuses we have to miss them. I am grateful for the chance I had to say what I said, but at the same time I am sorry for not expressing all of these thoughts when they still mattered. A eulogy is no replacement.

Ofer Strichman

The connection of this article to Amir. Amir was on the thesis committee of Benny Godlin, and hence the first to read about this work. I asked Amir to be on this committee because his focus was always more on software verification rather than on hardware. In his introductory talk in VSTTE'05, the conference dedicated to the grand challenge, he said that the community was distracted by the success of hardware verification, and perhaps it should refocus on software, the original, important goal of this community. In my last visit of him in New-York, indeed we discussed this work and various possibilities to strengthen the inference rules. As always he was incredibly positive and insightful.

Some Thoughts on the Semantics of Biocharts

David Harel[1] and Hillel Kugler[2]

[1] The Weizmann Institute of Science, Rehovot, Israel
dharel@weizmann.ac.il
[2] Microsoft Research, Cambridge, UK
hkugler@microsoft.com

Dedicated to the dear memory of Amir Pnueli: Friend, mentor, colleague, and a truly towering figure in computer science

Abstract. This paper combines three topics to which Amir Pnueli contributed significantly: the semantics of languages for concurrency, the semantics of statecharts, and reactive and hybrid systems. It is also no accident that the main motivation of our paper comes from biological systems: in recent years Amir became interested in these too. In [KLH10] we introduced Biocharts, a fully executable, two-tier compound visual language for modeling complex biological systems. The high-level part of the language is a version of statecharts, which have been shown to be successful in software and systems engineering. These statecharts can then be combined with any appropriately well-defined language (preferably a diagrammatic one) for specifying the low-level dynamics of the biological pathways and networks. The purpose of [KLH10] was to present the main concepts through a biological example and to illustrate the feasibility and usefulness of the approach. Here we discuss some of the questions that arise when one attempts to provide a careful definition of the semantics of Biocharts. We also compare the main requirements needed in a language for modeling biology with the way statecharts are used in software and system engineering.

1 Introduction

In recent years it is becoming clearer that understanding and predicting the behavior of complex biological systems requires developing and using new computational modeling languages and tools. In addition to the reductionist approach, which has been very successful in uncovering biological mechanisms, there is a need to integrate and synthesize the knowledge gained through reductionism to build system-level models that can explain and predict the behavior of a system as a whole and not only focus on very specific parts or aspects of the system behavior.

The language of *Statecharts*, which has proven to be very useful for specifying complex reactive software and systems, has been applied in the past ten years to modeling biological systems [KCH01, FPH+05, EHC07, SCDH08]. There are many aspects for which the statecharts approach is well suited to biological

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 185–194, 2010.

modeling, but also several major challenges specific to biology that led us to introduce *Biocharts*, a variant of statecharts geared towards biological modeling. Here we explain to a non-biological reader some of the challenges Biocharts aims to address and outline a definition of the semantics of the language. We have not yet built a dedicated tool to support Biocharts (in [KLH10] we used Rhapsody for demonstrating the feasibility of the approach), and some of the semantic decisions are still to be made. For various issues we point out here different semantic options and their implications.

2 Biological Modeling with Differential Equations

The traditional mathematical approach for modeling biological systems has been to use tools from classical continuous mathematics, mainly differential equations, and despite its successful application in many cases, this approach suffers from several limitations. First, differential equations require numeric values for the coefficients in the equations, values which are often unknown and can be very difficult to determine experimentally. Second, the continuous assumption is sometimes not valid when we consider a small number of molecules or cells. Third, often the level of abstraction that biologists use when thinking about their systems is discrete; for example, considering a gene to be either on or off, a signal to be low, medium or high and the differentiated fate of a cell to be primary, secondary or tertiary. Biologists know that these discrete values are only an abstraction but since this way of thinking is useful to them, we deem it an important facet of any approach to modeling that the languages and tools support such abstractions in a natural way. Finally, our computational ability to handle large systems of differential equations is often extremely limited.

3 Challenges in Biological Modeling

There are many reasons why modeling biological systems is extremely challenging. First and foremost is the inherent complexity of biological artifacts. Even when one focusses "merely" on modeling the behavior of a single cell, the process is likely to result in a model that is more complex than most engineered software systems. There are certain aspects of biology that are best treated using a continuous approximation, or differential equations, and there are other aspects where discrete methods are more suitable. A combination of these approaches would lead to hybrid models, which may play an important role in biological modeling, but only if suitable languages and tools are developed to integrate the approaches in an intuitive and rigorous manner.

A biological system can be examined and modeled on many scales: the molecular scale, the cellular scale, the scale of a tissue or an organ, or the scale of an organism or an entire population. For certain purposes, focusing on one scale is sufficient, while for other purposes building system-level models that incorporate several or even all of these scales is a must.

Now, multi-scale models can be constructed using the same language for describing each of the scales, or using different languages to describe the behavior on each scale. Whenever several languages are used, the interaction between the different scales described in different languages and sub-models must be clearly defined and has to be well integrated in the supporting tools. These types of multi-scale models are also challenging due to the computational resources needed to run them effectively.

We now discuss in more detail some of the common aspects and differences between biology and reactive software and the influence they have on the appropriate task of language design.

4 Biology vs. Software

A fundamental observation regarding biology and software, which leads to the idea of using languages that have proven themselves effective in software design also for biological modeling, is the fact that both types of systems (in the case of software this applies to many of the most complex kinds of systems) are *reactive* in nature. Reactive systems [HP85] are those whose role is to maintain an ongoing interaction with their environment, rather than produce a final result upon termination. Both types of systems are composed of many different parts, that act together in order to maintain the desired interaction with the environment and to achieve some required high-level system goals.

At the heart of the statechart language, which was designed specifically to deal with the dynamic behaviors of reactive systems, are states and transitions and ways to describe them in an intuitive and concise manner. This turns out to be very important also in biological modeling. In fact, many biologists are already used to thinking, and informally describing, their systems in terms of states and changes between states in response to the occurrence of certain events. Thus, adopting statecharts for biological modeling is quite natural.

A main difference between the software and biology domains is that for the former the main goal is constructing a system that will satisfy a set of requirements whereas for the latter the main goal is to understand how an existing biological system works. This is really the difference between engineering and reverse-engineering. It is interesting to observe that the emerging field of synthetic biology [End05] aims to go a step forward and engineer new biological systems to achieve given goals, typically by modifying certain aspects of existing systems, yet even for this direction the ability to understand and predict the behavior of existing biological systems is crucial.

While modeling a biological system, the model can be considered as a theory aiming to explain the behavior of the system, so that one's confidence in the theory increases if the model can predict behaviors that have not been observed yet. Such behaviors are sometimes considered to be *emergent properties* of the model, since despite not being explicitly programmed they emerge as a result of the combined interactive behavior of many components. Such behaviors may be hard to predict and understand without a fully executable model.

All relevant and interesting predictions arising from the model must be verified experimentally, since even models that seem very plausible could well be wrong and thus the model and its result can only serve as a guide towards performing interesting experiments in the lab. Biological models are more valuable if in addition to their predictive capabilities they have the ability to explain the phenomena by uncovering hidden mechanisms and underlying principles. In general, the full correctness of a model can never be established; hence the main goal is to refute potential models and thus to try and rule out hypotheses about the principles of the system behavior. This is a very Popperian approach to modeling. See [Har05].

Building models for biological systems is also different from software construction due to the fact that there are many aspects of biology that we still do not understand, and many units and components whose role in certain processes in still largely unknown. Thus, almost any attempt at biological modeling must involve dealing with such "black boxes". In software modeling and design there is a much clearer understanding of the system being developed, including the requirements, architecture and implementation, yet it is still very helpful to use abstraction and "black boxes", which provide freedom from the bias of implementation and thus help develop more robust software. In software models one aims to simplify and to avoid redundancy, whereas biological systems have many inherent redundancies. Hence, when modeling biology any simplifications to the model should be done very carefully, considering the assumptions made in the model and their implications, since a model that can reproduce biological behavior is not a goal in itself. Rather, we are mainly interested in what can be learned from the model and from its predictive capabilities.

In both software and biology, the ability to involve the domain experts, the various stake-holders in a software project and experimental biologists in biological modeling, is key to the success of the effort. In this way, developing languages and tools that are useful and intuitive for non-programmers is essential, which appears to be one of the major strengths of both Statecharts and Biocharts.

5 Semantics Outline

Biocharts is a fully executable, two-tier compound visual language for modeling complex biological systems. The high-level part of our language is a version of statecharts, which can be combined with any appropriately well-defined language (preferably a diagrammatic one) for specifying the low-level dynamics of the biological pathways and networks. We now outline the semantics of the variant of statechart we propose to use in Biocharts, and discuss how it integrates with the lower level languages.

5.1 The Basics

The statechart itself is similar to the original description in [Har87], and to that of Statemate [HN96] and Rhapsody [HG97], in that there are three types of states:

OR-states, *AND-states* and *basic states*. The OR-states have *substates* related to each other by "exclusive or", AND-states have *orthogonal components* that are related by "and", while basic states have no substates, and are the lowest in the state hierarchy. When building a statechart there is an implicit additional state, the root state, which is the highest in the hierarchy. The *active configuration* is a maximal set of states that the statechart can be in simultaneously, including the root state, exactly one substate for each OR-state in the set, all substates for each AND-state in it and no additional states. The general syntax of an expression labeling a transition in a statechart is "$m[c]/a$" where m is the message that triggers the transition, c is a condition that guards the transition from being taken unless it is true when m occurs, and a is an action that is carried out if and when the transition is taken. All of these parts are optional.

5.2 Classes and Objects

For Biocharts we adapt some of the basic principles of the Rhapsody semantics of statecharts, as described in [HG97, HK04], especially the way statecharts are incorporated into an object-oriented framework. The motivation for this decision is that typical biological models require specifying many entities (e.g., cells) with the same specification but each one in a different active configuration. These entities (e.g., cells) can be born and may die during model execution, so the object oriented framework is a natural one for representing such models.

A system is composed of classes. A statechart describes the modal behavior of the class; that is, how it reacts to messages it receives by defining the actions taken and the new mode entered. A class can have an associated statechart describing its behavior. These classes are called *reactive classes*. During runtime there can exist many objects of the same class, called *instances*, and each can be in a different active configuration – a set of states in which the instance resides. Thus, a new statechart is "born" for each new instance of the class, and it runs independently of the others. When a new instance is created, the statechart enters its initial states by taking default transitions recursively until it is in an active configuration.

A new feature that we propose for Biocharts, building on our experience from previous biological projects, is to enable an object to dynamically create a new object of the same class in exactly the same active state as it is in at the moment of creation. This is useful in several biological contexts; for example, during cell division, where daughter cells typically inherit the state of the mother cell. In this case, the statechart of the daughter cell is "born" in an active configuration identical to its mother cell, and no default transitions are taken as part of this initialization.

5.3 Messages and Actions

As mentioned above, the general syntax of an expression labeling a transition in a statechart is "$m[c]/a$", for message m, condition c and action a. We now describe each of these parts in more detail. The message m is either an event

or a triggered operation. Consider a simple transition between states $S1$ and $S2$ labeled "$m[c]/a$". Regardless of whether m is an event or a triggered operation, if the statechart is in state $S1$, message m occurs and the condition c holds, state $S1$ is exited, the action a is performed and then state $S2$ is entered.

A practical question concerns the language in which the conditions and actions should be written. Statemate introduces a special action language for this purpose, whereas Rhapsody, which is geared towards software development, allows using the implementation language produced by the code generator, e.g., C++, as the action language. We leave this as an open decision for Biocharts. On the one hand, the main advantage of using an existing programming language for the action language is easy integration with other software modules and tools, which is an important aspect of Biocharts since one of the main ideas is that the lower level modules can be described in appropriate existing languages and tools. However, on the other hand, defining a special-purpose action language may be very beneficial, as it makes the models more language and platform independent. Also, by carefully restricting the expressive power of the special action language compared to a language like C++ , verification and analysis tools can be more effective, which is also an important consideration for biological modeling.

Events represent asynchronous communication, while triggered operations represent synchronous communication. Both are supported in Rhapsody and we suggest that they also be supported by Biocharts. Both an event and a triggered operation are invoked by a sender object that invokes the destination object. For an event a special event instance is created and is placed in an event queue. The sender object can then continue its work without waiting for the receiver to consume the event, which will take place later when the event reaches the top of the queue and is dispatched to the destination object, potentially triggering a transition. Even though an event queue does not seem to have any direct counterpart in biological systems, our current experience shows that events are a practical solution for modelers, since the event does not have to worry about the receiver being able to communicate. This fits well with the modeling view that considers entities in the model as autonomous agents.

A triggered operation is a synchronous operation, thus the sender object must wait until the receiver object responds to the invocation, possibly causing a transition. Triggered operations may return a value to the calling object, the value being specified on the transition. If no transition is triggered by invoking the triggered operation, a special value is returned and the sender object can proceed. Our current experience from several statechart-based models that used Rhapsody shows that events were used more often than triggered operations, although we believe both are useful for biological modeling.

Events can also have attributes (variables), which the sender object sets to concrete values when sending an event. This feature is useful for biological modeling where one needs to deal with more quantitative information about the strength of certain signalling events. Events can be sub-classed, a mechanism that can be used in order to add attributes. In particular, if event e' is derived from event e in this way, e' will trigger any transition that has e as a trigger.

5.4 Inheritance and Statechart Modification

A basic feature of object-oriented programs is inheritance, which allows class B to be a subclass of class A, thus inheriting its variables and methods, which can later be modified by the programmer of class B. Rhapsody deals with inheritance of reactive classes (ones that have a statechart), by copying the statechart of the superclass to the derived class, and allowing the modeler to perform certain restricted changes in the derived class. In particular, B inherits all As states and transitions, and these cannot be removed, though certain refinements are allowed.

Inherited states can be modified in three ways: 1) decomposing a basic state by Or (into substates) or by And (into orthogonal components); 2) adding substates to an Or state; and 3) adding orthogonal components to any state. Transitions can be added to the derived statechart, and certain modifications are allowed in the original inherited ones: the target state of an inherited transition can be changed, for example, and certain changes are also allowed in the guard and action.

Our current experience in biological modeling points towards allowing in Biocharts the modeler to perform any changes he would like to the derived statechart, including building it from scratch. In case the modeler decides to indeed perform only very well controlled changes in the derived call, it will be beneficial if the tool can provide traceability and show visually where exactly changes were made.

5.5 Dynamic Changes to a Statechart

We propose a new feature for Biocharts, which we believe will become useful for biological modeling but which was not supported in previous statechart semantics. This is the ability to change the statechart itself during runtime. The motivation for this feature is that in biology there is a stronger connection than in software between the structure of the system and its behavior, and a natural way to represent changes in the structure of the biological system is by adding or removing a transition, adding a new basic-state, or changing the target of a default transition.

We propose to support such a dynamic change on the object level or on the class level. If the change is on the object level, it can be invoked by calling, for example, the method $O.RemoveTransition(t)$ for object O that contains transition t, and the result at runtime would be that for object O transition t from this point onwards will not be considered. Invoking such a call on the class level, for example by calling $C.RemoveTransition(t)$, will remove transition t from all existing objects of class C, and future instances of the class will be created without this transition.

Although supporting this feature in an implementing tool, such as Rhapsody, will require overcoming various technical challenges, we believe it is a very natural representation of the dynamic changes in a biological system and will allow one to perform *in-silico* mutations and various perturbations in a natural and elegant way.

5.6 Steps

At the heart of statechart semantics is the precise definition of the effect of a step, which takes the system from one stable configuration to the next one. In general, we propose to adapt the definitions of the Rhapsody semantics; for a detailed description see [HK04]. However, we leave open two key issues, for which we suggest to carefully consider whether to adopt the Rhapsody semantics or the Statemate semantics [HN96].

These issues are these: (i) Should changes made in a given step take effect in the current step or only in the next step; (ii) does a step take zero time or can it take more than zero time.

In Rhapsody, changes made in a given step take effect in the current step, and a step can take more than zero time, whereas in Statemate changes made in a given step take effect only in the next step and a step takes zero time. In Rhapsody, as mentioned earlier, the action language is the programming language that serves as the target for the code generator, thus it would have been difficult to postpone to the next step the effects of changes caused by running part of an action in a given step. Also the zero time step assumption does not hold for such general action languages. However, if a special action language is defined for Biocharts, this opens the way to consider adapting the Statemate semantics for these two issues.

Another issue involves how to resolve conflicting transitions. Roughly speaking, Rhapsody gives priority to lower level source states, while Statemate gives priority to higher level ones. We propose to adopt the Rhapsody priority scheme, since it is intuitive in an object-oriented setting, in that it allows to "override" behavior in lower level states. Rhapsody tries to detect and disallow transitions that have the same trigger and guard and have the same priority, with the motivation that the generated code is intended to serve as a final implementation and for most embedded software systems such nondeterminism is not acceptable. For biological modeling nondeterminism is very common and useful, so we suggest to support it in Biocharts too.

We also think that Biocharts should include support for the *clear history* operator, which erases the history of a state so that the next time the history connector for this state is entered the default transition will be taken. The motivation for supporting this feature is that it corresponds to biological phenomena that may be modeled in an easier way with clear history. We also suggest to revisit the way null transitions are handled in Rhapsody to ensure that one can specify a null transition with a guard and it will be taken immediately when the guard becomes true. In Rhapsody, such guards were evaluated on entering the state, so later changes did not always take effect unless they were associated with a visible event.

5.7 Low Level Modules

We now explain how the low level part of the two-tier language of Biocharts, which will typically describe the dynamics of the biological pathways and networks, is integrated with the high-level statechart part.

A state in a Biochart's statechart can include a 'low-level' module, which is a program P. This P is activated on entering the state, by calling $P.Start$, and is stopped when the state is exited, by calling $P.Stop$. The program P has input variables $x_1, x_2, \cdots x_l$, output variables $y_1, y_2, \cdots y_m$ and local variables $z_1, z_2, \cdots z_n$. The input and output variables are part of the object's variables, so that, for example, another program P' activated in an orthogonal state can use an output variable of P as one of its input variables.

Input variables are accessed by the program P by calling $x_i.get()$ initially and at any stage of its computation. Similarly, P can set the value of the output variables by calling $y_j.set(val)$. A Biocharts framework for supporting complicated scenarios with shared variables will need to support locking and notification of value changes for shared variables. To support hybrid modeling some of the local variables z_i can be continuous, and their dynamics would then be determined by a set of differential equations.

6 A Pledge

We are fully aware of the fact that in this paper we have only touched upon some of the issues around the semantics of Biocharts. In fact, there are several issues about what to include in, or exclude from, the language itself prior to defining the semantics rigorously. Obviously, all this has to be done before the language can be viewed as a complete modeling medium for biological systems.

The truth is that not only do we deeply miss Amir Pnueli personally, but we are confident that he would have been the ideal colleague with whom to continue this line of work. His pioneering research on hybrid systems, his unparalleled understanding of semantic issues for reactivity, and his rare scientific wisdom, would all have made the future work on this topic far easier, and the results far better than we can ever expect them to become in his absence.

Nevertheless, we pledge to forge forward with this work, with whatever talents and abilities we can muster, and bring it to a state where it can be seriously evaluated and hopefully then adopted and used beneficially.

References

[EHC07] Efroni, S., Harel, D., Cohen, I.R.: Emergent Dynamics of Thymocyte Development and Lineage Determination. PLoS Computational Biology 3(1), 3–13 (2007)

[End05] Endy, D.: Foundations for engineering biology. Nature 438, 449–453 (2005)

[FPH+05] Fisher, J., Piterman, N., Hubbard, E.J.A., Stern, M.J., Harel, D.: Computational Insights into C. *elegans* Vulval Development. Proceedings of the National Academy of Sciences 102(6), 1951–1956 (2005)

[Har87] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8, 231–274 (1987); Preliminary version: Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel (February 1984)

[Har05] Harel, D.: A Turing-Like Test for Biological Modeling. Nature Biotechnology 23, 495–496 (2005)

[HG97] Harel, D., Gery, E.: Executable object modeling with statecharts. Computer 30(7), 31–42 (1997); Also in: Proc.18th Int. Conf. Soft. Eng., Berlin, pp. 246–257. IEEE Press, Los Alamitos (March 1996)

[HK04] Harel, D., Kugler, H.: The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML). In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 325–354. Springer, Heidelberg (2004)

[HN96] Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Trans. Software Engin. Methods 5(4), 293–333 (1996)

[HP85] Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems, New York. NATO ASI Series, vol. F-13, pp. 477–498. Springer, Heidelberg (1985)

[KCH01] Kam, N., Cohen, I.R., Harel, D.: The immune system as a reactive system: Modeling t cell activation with statecharts. In: IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2001), pp. 15–22. IEEE Computer Society Press, Los Alamitos (2001)

[KLH10] Kugler, H., Larjo, A., Harel, D.: Biocharts: A Visual Formalism for Complex Biological Systems. J. R. Soc. Interface (2010)

[SCDH08] Setty, Y., Cohen, I.R., Dor, Y., Harel, D.: Four-Dimensional Realistic Modeling of Pancreatic Organogenesis. Proceedings of the National Academy of Sciences (2008)

Unraveling a Card Trick*

Tony Hoare[1] and Natarajan Shankar[2]

[1] Microsoft Research
Cambridge CB3 0FB UK
http://research.microsoft.com/en-us/people/thoare/
[2] Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
http://www.csl.sri.com/users/shankar

Dedicated to the memory of Amir Pnueli.

In one version of Gilbreath's card trick, a deck of cards is arranged as a series of quartets, where each quartet contains a card from each suit and all the quartets feature the same ordering of the suits . For example, the deck could be a repeating sequence of spades, hearts, clubs, and diamonds, in that order, as in the deck below.

$$\langle 5\spadesuit\rangle, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle, \langle 8\diamondsuit\rangle,$$
$$\langle K\spadesuit\rangle, \langle 2\heartsuit\rangle, \langle 7\clubsuit\rangle, \langle 4\diamondsuit\rangle,$$
$$\langle 8\spadesuit\rangle, \langle J\heartsuit\rangle, \langle 9\clubsuit\rangle, \langle A\diamondsuit\rangle$$

The deck is then cut into two (not necessarily equal) half-decks, possibly as $\langle 5\spadesuit\rangle, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle, \langle 8\diamondsuit\rangle, \langle K\spadesuit\rangle$ and $\langle 2\heartsuit\rangle, \langle 7\clubsuit\rangle, \langle 4\diamondsuit\rangle, \langle 8\spadesuit\rangle, \langle J\heartsuit\rangle, \langle 9\clubsuit\rangle, \langle A\diamondsuit\rangle$.

The order of one of the half-decks is then reversed. Either half-deck could be reversed. We can pick the smaller one, i.e., the first one, and reverse it to obtain $\langle K\spadesuit\rangle, \langle 8\diamondsuit\rangle, \langle Q\clubsuit\rangle, \langle 3\heartsuit\rangle, \langle 5\spadesuit\rangle$. The two half-decks are then shuffled in a (not necessarily perfect) riffle-shuffle. One such shuffle is shown below, where the underlined cards are drawn from the second half-deck.

$$\langle 2\heartsuit\rangle, \langle 7\clubsuit\rangle, \underline{\langle K\spadesuit\rangle}, \underline{\langle 8\diamondsuit\rangle},$$
$$\langle 4\diamondsuit\rangle, \langle 8\spadesuit\rangle, \underline{\langle Q\clubsuit\rangle}, \langle J\heartsuit\rangle,$$
$$\underline{\langle 3\heartsuit\rangle}, \langle 9\clubsuit\rangle, \underline{\langle 5\spadesuit\rangle}, \langle A\diamondsuit\rangle$$

The quartets in the shuffled deck are displayed to demonstrate that each quartet contains a card from each suit. This turns out to be inevitable no matter how the original deck is cut and the order in which the two decks are shuffled. The principle underlying the card trick can be proved in a number of ways. We present the argument as a series of transformations that demystify the trick and describe its formalization.

1 Informal Proof

We outline the informal argument before formalizing it. The trick can be explained in a number of ways. The proof below works by transforming (i.e., demystifying) the trick

* The second author was supported by NSF Grants CSR-EHCS(CPS)-0834810 and CNS-0917375. Sam Owre commented on earlier drafts of the paper.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 195–201, 2010.

in stages. A deck of cards is *suit-ordered* if the deck is a sequence of quartets where there is some ordering of the suits so that the (suits of the) cards in each quartet are arranged in exactly this order. A deck is *suit-sorted* if each quartet is *pairwise suit-distinct*, i.e., it contains exactly one card from each suit, or equivalently, it does not contain two cards of the same suit. The card trick above takes a suit-ordered deck and produces a suit-sorted deck by cutting the deck into two half-decks, reversing one of the half-decks, and riffle-shuffling the two half-decks.

First, we observe that since we are only interested in the set of cards that occur in each quartet in the final shuffled deck, and not in their relative order, we can therefore replace the shuffle by a selection. Let us assume that the given suit-ordered deck A is cut into C and D so that $A = C; D$, and deck D is reversed to get D^-. The final deck B is the result of shuffling C and D^-, where we write the set of possible shuffles of C and D^- as $C \bowtie D^-$. Let the deck B be partitioned as $B[\rightarrow 4], B[4 \rightarrow]$, where $B[\rightarrow 4]$ represents the prefix containing the first four cards, and $B[4 \rightarrow]$ is the corresponding suffix. Then there is some $k \leq 4$ such that $B[\rightarrow 4]$ is in $C[\rightarrow k] \bowtie D^-[\rightarrow 4-k]$ and $B[4 \rightarrow]$ is in $C[k \rightarrow] \bowtie D^-[4-k \rightarrow]$. We can see that the cards in $B[\rightarrow 4]$ are pairwise suit-distinct if the cards in $C[\rightarrow k]; D^-[\rightarrow 4-k]$ are pairwise suit-distinct.

Let $D[\leftarrow j]$ be the deck D with the last j cards removed, and $D[j \leftarrow]$ be the deck consisting of just the last j cards of D. Next, we observe that $D^-[\rightarrow 4-k] = (D[4-k \leftarrow])^-$ and $D^-[4-k \rightarrow] = (D[\leftarrow 4-k])^-$ since picking $4-k$ cards from the top of the half-deck D^- is the same as picking these cards from the bottom of D.

At this point, the proof might already be quite apparent, but we persist. We have, at this point, dispensed with the shuffle and the reversal of the half-deck D and essentially reduced the problem to checking if $C[\rightarrow k]; D[4-k \leftarrow]$ is pairwise suit-distinct. Next, we also get rid of the cut. Given that $A = C; D$, the construction of B_0 is equivalent to shuffling $A[\rightarrow k] \bowtie A[4-k \leftarrow]$. Then B_0 is pairwise suit-distinct if $A[\rightarrow k]; A[4-k \leftarrow]$ is pairwise suit-distinct. This is indeed the case since $A[\rightarrow k]; A[4-k \leftarrow]$ has the same sequence of suits as $A[\rightarrow 4]$. The remainder of the deck A is $A[k \rightarrow][\leftarrow 4-k]$ which is suit-ordered, and hence by induction $B[4 \rightarrow]$ is suit-sorted, and hence B itself is suit-sorted.

The original card trick is thus reducible to one where the suit-sorted deck B is obtained by contructing quartets B_i by successively picking cards from the top and bottom of the input deck A. A perhaps more intuitive explanation is that if the deck A is viewed as a suit-ordered circular deck of cards, and B_0 contains a set of 4 contiguous cards extracted from this circular deck, then B_0 is pairwise suit-distinct. The cards remaining in A form a suit-ordered circular deck. Obviously, the latter trick is not as surprising as the one involving the cut, reversal, and shuffle. It is also worth noting that it does not matter which of the half-decks is reversed since in either case the cards that form the block B_0 are taken from some contiguous block of the circular deck A.

The trick can be generalized to any sequence over an alphabet of size m, where $0 < m$. In the above case, m equals 4. The case for $m = 1$ is trivial, but when $m = 2$, we have an interesting variant that has already been subject to machine verification. In that trick, the initial deck is arranged in alternating red and black cards. The deck is cut into two half-decks and shuffled. If the bottom cards of the half-decks were of the same color, then the bottom card of the shuffled deck is moved to the top. Otherwise,

we retain the same shuffled deck. The resulting deck is a sequence of pairs, where each pair consists of a red and a black card. For example, if the deck is as before, i.e.,

$$\langle 5\spadesuit\rangle, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle, \langle 8\diamondsuit\rangle,$$
$$\langle K\spadesuit\rangle, \langle 2\heartsuit\rangle, \langle 7\clubsuit\rangle, \langle 4\diamondsuit\rangle,$$
$$\langle 8\spadesuit\rangle, \langle J\heartsuit\rangle, \langle 9\clubsuit\rangle, \langle A\diamondsuit\rangle$$

The cut deck consists of the two half-decks $\langle 5\spadesuit\rangle, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle, \langle 8\diamondsuit\rangle, \langle K\spadesuit\rangle, \langle 2\heartsuit\rangle$ and $\langle 7\clubsuit\rangle, \langle 4\diamondsuit\rangle, \langle 8\spadesuit\rangle, \langle J\heartsuit\rangle, \langle 9\clubsuit\rangle, \langle A\diamondsuit\rangle$. Note that the bottom cards of the two half-decks have the same color, namely, red. These half-decks can, for example, be shuffled as shown below with the underlined cards being drawn from the second half-deck.

$$\langle 5\spadesuit\rangle, \underline{\langle 7\clubsuit\rangle}, \underline{\langle 4\diamondsuit\rangle}, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle, \underline{\langle 8\spadesuit\rangle},$$
$$\underline{\langle J\heartsuit\rangle}, \langle 8\diamondsuit\rangle, \langle K\spadesuit\rangle, \underline{\langle 9\clubsuit\rangle}, \langle 2\heartsuit\rangle, \underline{\langle A\diamondsuit\rangle}$$

Since the bottom cards of the half-decks from the cut were the same color, we move the bottom card of the shuffled deck to the top to get

$$\underline{\langle A\diamondsuit\rangle}, \langle 5\spadesuit\rangle, \underline{\langle 7\clubsuit\rangle}, \underline{\langle 4\diamondsuit\rangle}, \langle 3\heartsuit\rangle, \langle Q\clubsuit\rangle,$$
$$\underline{\langle 8\spadesuit\rangle}, \underline{\langle J\heartsuit\rangle}, \langle 8\diamondsuit\rangle, \langle K\spadesuit\rangle, \underline{\langle 9\clubsuit\rangle}, \langle 2\heartsuit\rangle$$

The resulting deck is a sequence of matched red/black pairs.

The shuffled deck could be produced by first moving the card $\underline{\langle A\diamondsuit\rangle}$ to the top of the second deck, and then shuffling. For the case of an alternating red/black deck, moving the bottom card of the half-deck to the top corresponds to a reversal of the deck. This is because the two half-decks from the cut will have bottom cards of the same color only when the half-decks contain an even number of cards. When the bottom two cards of the half-deck are of different colors, each half-deck must contain an odd number of cards, and the rotation of one of the half-decks leaves the color pattern unchanged. This variant of the trick is thus an instance of the earlier one.

One variant of this trick is when the deck is ordered to first contain the ordered sequence of cards in spades, then hearts, clubs, and diamonds. The result of cutting the deck, reversing one of the half-decks, and then shuffling yields a deck in which each block of 13 cards contains a full sequence from A to K albeit with the suits mixed. In this version of the trick, $m = 13$ and $n = 4$. Another interesting variant of the trick is when we have two full decks where one deck has the cards in the reverse order from the other one. The two decks are then shuffled and the resulting double deck is cut exactly in half to reveal two complete decks. This is just an instance of the trick with $m = 52$ and $n = 2$.

We learnt of this version of Gilbreath's card trick from Professor Madhavan Mukund of the Chennai Mathematical Institute during the TECSWeek school of January 2010 in Pune, India. The $m = 2$ version of the card trick (the first Gilbreath principle) was discovered by Norman Gilbreath in 1958 [Gil58], and popularized by Martin Gardner in his August 1960 *Mathematical Games* column in the Scientific American [Gar66]. The trick is described on pages 126–127 of Martin Gardner's book *aha! Gotcha* [Gar82]. Colm Mulcahy's *Card Colm* 2005 article (`http://www.maa.org/columns/colm/cardcolm200508.html`)

explains some of the background behind the trick. The generalization to $m > 0$ (the second Gilbreath principle) was discovered by Gilbreath in 1966. It is described in a 2006 column by Colm Mulcahy (http://www.maa.org/columns/colm/cardcolm200608.html) where he outlines an argument similar to the one above. In Mulcahy's argument, the deck C is reversed in which case, the top k cards of C and the bottom $4 - k$ cards of D are from a sequence of 4 contiguous cards in the middle of the pack. Any sequence of 4 cards from a suit-ordered deck must contain a card of each suit. When this middle segment of 4 cards is removed, the resulting deck remains suit-ordered. Some variations of the trick are described by Wilf Hey in his Magic & Mathematics column (`http://www.bitwisemag.com/2/Garden-of-Gilbreath`) in **bitwise** magazine. Most researchers in formal methods learned of the first Gilbreath principle from Huet [Hue91] where he carries out a machine-checked proof of the first Gilbreath principle. Other proofs have been done by Boyer using ACL2 [Boy91] and Bouhoula and Rusinowitch using SPIKE [BR95]. Huet learned of the Gilbreath principle from de Bruijn [dB87]. The paper by de Bruijn provides an automata-theoretic proof of the principle and its extensions. Given two decks that each contain alternating red and black cards so that the bottom cards of the decks are of distinct colors, the shuffle can be represented as a nondeterministic automaton that has four states where the bottom two cards of the respective decks are red-black, black-red, red-red, or black-black. The first two states are the start states as well as the final states, and the language generated by this automaton clearly is $(RB \mid BR)^*$, where R represents a red card and B a black card.

2 The Formalization

Formalizing a concrete problem such as one about cards naturally requires the explication of a number of implicit assumptions. Our formalization deals with the general case of a repeating pattern n sub-decks of m distinct cards, with $m > 0$. The card deck is captured as a finite sequence. Other representations such as lists will also work for this purpose. A *order-repeating* sequence is a finite sequence A such that for any indices i and j, $A(i) = A(j)$ iff $i = j (\mathrm{mod}\, m)$. A finite sequence B of length nm is said to be *block-sorted* if it is made up of n segments of length m where each segment contains no duplicates. The objective is to demonstrate that if B is obtained by shuffling C and D, where $A = C; reverse(D)$, then if A is a order-repeating sequence, then B is block-sorted.

There are a couple of ways to capture the shuffling operation. One method is to define a predicate that asserts that B is the result of shuffling C and D. The other is to use an oracle variable that specifies the selection pattern for the shuffle. We use the latter approach although we still specify the shuffle by means of a predicate. The shuffle pattern is given by a finite sequence *map* of Booleans so that the i'th element of B is selected from C when *map*(i) is true, and from D when it is false. The operation shuffle is defined to use two counters *countC* and *countD* that are used to track the number of cards that have already been shuffled out of the half-decks C and D, respectively. The operation shuffle is then defined so that *shuffle*$(B, C, D, countC, countD, map)$ holds when either

1. $countC + countD = nm$, or
2. $map(countC + countD) = true$,
   $countC < length(C)$, $B(countC + countD) = C(countC)$, and
   shuffle$(B, C, D, countC + 1, countD, map)$ holds, or
3. $map(countC + countD) = false$,
   $countD < length(D)$, $B(countC + countD) = D(countD)$, and
   shuffle$(B, C, D, countC, countD + 1, map)$ holds.

Then, B is the result of shuffling C and D with respect to the oracle map when ***shuffle***$(B, C, D, 0, 0, map)$ holds.

We then define an *up-down* shuffle operation on A where the deck B is constructed so that $B(i)$ is selected from the top of A when ***map***(i) is true, and from the bottom of A, when ***map***(i) is false. The up-down shuffle also maintains two counters up and $down$ to keep track of how many cards have been added to the shuffle from the top and the bottom of deck A, respectively. Thus ***updownShuffle***$(B, A, up, down, map)$ holds when either

1. $up + down = nm$, or
2. $map(up + down) = true$,
   $B(up + down) = A(up)$, and
   updownShuffle$(B, A, up + 1, down, map)$ holds, or
3. $map(up + down) = false$,
   $B(up + down) = A(nm - down - 1)$, and
   updownShuffle$(B, A, up, down + 1, map)$ holds.

When ***updownShuffle***$(B, A, 0, 0, map)$ holds with respect to the oracle map, then B is the result of up-down shuffling A

We can see by induction that if $A = C; reverse(D)$ and B is the result of shuffling C and D with respect to ***map*** with $countC = up$ and $countD = down$, then B is the result of up-down shuffling A.

The next part of the proof is to demonstrate that when B is the up-down shuffle of A and A is order-repeating, then B is block-sorted. For this, we first need a variant of up-down shuffle called block-shuffle that performs the operation in a blockwise manner. For this purpose, the counters up and $down$ in the up-down shuffle are replaced by counters U and u and V and v with the invariant that $up = U + u$ and $down = V + v$, where $U + V$ is a multiple of m and $u + v < m$. If the deck B is the result of up-down shuffling A, then B is also the result of block-shuffling A.

Finally, we can demonstrate that if B is obtained by block-shuffling A, then B is block-sorted. Let A' be the sequence A with the first U and the last V elements removed. If A is order-repeating, then so is A', since any prefix or suffix of an order-repeating sequence is also order-repeating. The main claim that B is block-sorted requires an auxiliary invariant asserting that if $U + V = im$ and , then the elements of B from the $im + u + v$ to $(i + 1)m - 1$ appear in A' in the positions u to $m - v - 1$. With this invariant, it can be shown that element B at position $im + u + v - 1$, for $u + v > 0$ which appears either position $u - 1$ or $m - v$ of A' cannot appear in the remainder of the block within B. This is because the remainder of the block within B, i.e., those elements at position $im + u + v$ to $(i + 1)m - 1$ appear in index positions u

to $m - v - a$ of A' and these indices are distinct (mod m) from $u - 1$ and $m - v$. Thus when $u = v = 0$, the invariant ensures that the block of B from im to $(i + 1)m - 1$ contains no duplicates.

The main claim follows from this auxiliary invariant by induction on $n - i$ since if $i < n$ and the blocks of B from $(n - i + 1)m$ to $nm - 1$ are block-sorted, and the block of B from $(n - i)m$ to $(n - i + 1)m - 1$ satisfies the auxiliary invariant, then the blocks from $(n - i)m$ to $nm - 1$ are block-sorted. When $i = 0$, all of B is block-sorted.

The above formalization has been verified using SRI's Prototype Verification System (PVS) [ORSvH95].

3 Conclusions

Amir Pnueli was a scientist of uncommon brilliance and versatility. His humility, humanity, and dedication to perfection was, and will remain, an inspiration for all whose lives he touched. His contributions span every aspect of program correctness covering hardware, software, real-time and hybrid systems, static analysis, temporal logic, model checking, deductive verification, concurrency, abstraction, composition, synthesis, synchronous languages, translation validation, and automated deduction. In all of his work, Amir Pnueli was acutely attentive to the need for *usable* methods that did not sacrifice rigor. Indeed, he saw both usability and rigor as by-products of simplicity. He saw no contradiction between good theory and effective practice. In the years ahead, as we continue to explore the rich legacy of his ideas, his focus on rigor, simplicity, and usability will continue to guide and shape the future of computing.

For Amir, automation was an important driving concern. He famously remarked that *interactive proof is the greatest waste of human talent*. Interaction is not the enemy of automation, nor is automation necessarily the friend of interaction. The different levels and ranges of automation in formal proof construction are similar to the suite of tools available to a machinist or a carpenter. A good craftsman will marshall the different tools and techniques as appropriate. With proofs, the automation that is available today in the form of model checkers and decision procedures is impressive. However, such automation does hit some limits through infeasibility or undecidability when reasoning with quantification, nonlinear arithmetic, and advanced operations on data types such as arrays, lists, and sets. The most effective tools are generally those that directly extend human capabilities. This is best accomplished through a happy fusion of interaction and automation that has not yet been achieved.

With Gilbreath's card trick, we have outlined a simple approach that demystifies the trick. After we have applied our transformations, we are left with a version of the trick that is somewhat unsurprising. When we pick k cards from the top of an order-repeating deck and $4-k$ cards from the bottom of the deck, we expect to get 4 cards that represent a permutation of the suits. However, even with these simplifications, the formalization does pose some challenges. Proper representations for concepts such as card decks, shuffles, and blocks can only be found after a bit of trial and error. Interaction is quite helpful in experimenting with different ways of formalizing these concepts and their associated proofs. However, the ultimate goal of constructing such proofs with a usable, automated method remains a challenge of the kind that Amir Pnueli would have relished.

References

[Boy91] Boyer, R.S.: A mechanical checking of a theorem about a card trick. ACL2 formalization (May 22, 1991)

[BR95] Bouhoula, A., Rusinowitch, M.: Implicit induction in conditional theories. Journal of Automated Reasoning 14(2), 189–235 (1995)

[dB87] de Bruijn, N.G.: A riffle shuffle card trick and its relation to quasi-crystal theory. Nieuw Archief voor Wiskunde 5(3), 285–301 (1987)

[Gar66] Gardner, M.: New Mathematical Diversions from Scientific American. Simon & Schuster, New York (1966); Republished by the Mathematical Association of America (June 1995)

[Gar82] Gardner, M.: aha! Gotcha. W. H. Freeman & Co., New York (1982); Republished as part of a two-volume collection with *aha!* Insight by the Mathematical Association of America (2006)

[Gil58] Gilbreath, N.: Magnetic colors. The Linking Ring 38(5), 60 (1958)

[Hue91] Huet, G.: The Gilbreath trick: A case study in axiomatisation and proof development in the Coq proof assistant. Technical Report RR-1511, INRIA, Institut National de Recherche en Informatique et en Automatique (September 1991)

[ORSvH95] Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering 21(2), 107–125 (1995); PVS home page: `http://pvs.csl.sri.com`

An Automata-Theoretic Approach to Infinite-State Systems*

Orna Kupferman[1], Nir Piterman[2], and Moshe Y. Vardi[3]

[1] Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
orna@cs.huji.ac.il
http://www.cs.huji.ac.il/∼orna
[2] Imperial College London, Department of Computing, London SW7 2AZ, UK
nir.piterman@doc.ic.ac.uk
http://www.doc.ic.ac.uk/∼npiterma
[3] Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
vardi@cs.rice.edu
http://www.cs.rice.edu/∼vardi

Amir has had a profound influence on the three of us, as a teacher, an advisor, a mentor, and a collaborator. His fundamental ideas on the temporal logics of programs have, to a large extent, set the course for our professional careers. His sudden passing away has deprived us of many more years of wonderful interaction, intellectual engagement, and friendship. We miss him profoundly. His wisdom and pleasantness will stay with us forever.

Abstract. In this paper we develop an automata-theoretic framework for reasoning about infinite-state sequential systems. Our framework is based on the observation that states of such systems, which carry a finite but unbounded amount of information, can be viewed as nodes in an infinite tree, and transitions between states can be simulated by finite-state automata. Checking that a system satisfies a temporal property can then be done by an alternating two-way tree automaton that navigates through the tree. We show how this framework can be used to solve the model-checking problem for μ-calculus and LTL specifications with respect to pushdown and prefix-recognizable systems. In order to handle model checking of linear-time specifications, we introduce and study *path automata on trees*. The input to a path automaton is a tree, but the automaton cannot split to copies and it can read only a single path of the tree.

As has been the case with finite-state systems, the automata-theoretic framework is quite versatile. We demonstrate it by solving the realizability and synthesis problems for μ-calculus specifications with respect to prefix-recognizable environments, and extending our framework to handle systems with *regular labeling regular fairness constraints* and μ-calculus with *backward modalities*.

1 Introduction

One of the most significant developments in the area of formal design verification is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [CES86, LP85, QS82, VW86a]. In temporal-logic *model checking*, we verify

* The paper is based on the papers [KV00a, KPV02].

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 202–259, 2010.

the correctness of a finite-state system with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior (for a survey, see [CGP99]). Symbolic methods that enable model checking of very large state spaces, and the great ease of use of fully algorithmic methods, led to industrial acceptance of temporal model checking [BLM01, CFF+01].

An important research topic over the past decade has been the application of model checking to infinite-state systems. Notable success in this area has been the application of model checking to real-time and hybrid systems (cf. [HHWT95, LPY97]). Another active thrust of research is the application of model checking to *infinite-state sequential systems*. These are systems in which a state carries a finite, but unbounded, amount of information, e.g., a pushdown store. The origin of this thrust is the important result by Müller and Schupp that the monadic second-order theory (MSO) of *context-free graphs* is decidable [MS85]. As the complexity involved in that decidability result is nonelementary, researchers sought decidability results of elementary complexity. This started with Burkart and Steffen, who developed an exponential-time algorithm for model-checking formulas in the *alternation-free* μ-calculus with respect to context-free graphs [BS92]. Researchers then went on to extend this result to the μ-calculus, on one hand, and to more general graphs on the other hand, such as *pushdown graphs* [BS95, Wal96], *regular graphs* [BQ96], and *prefix-recognizable graphs* [Cau96].

On the theoretical side, the limits of MSO decidability have been pushed forward. Walukiewicz and Caucal show that MSO decidability is maintained under certain operations on graphs [Wal02, Cau03]. Further studies of these graphs show that they are the configuration graphs of *high-order pushdown automata* [CW03], and provide an elementary time solution for model checking μ-calculus over these graphs [Cac03]. Recently, the decidability of MSO and μ-calculus with respect to graphs produced by higher-order recursion was established [KNUW05, Ong06].

From a practical point of view, model checking of pushdown graphs (or pushdown systems) provides a framework for software model checking where the store of the pushdown system corresponds to the function call stack. This led to the implementation of pushdown model-checkers such as Mops [CW02], Moped [ES01, Sch02], and Bebop [BR00] (to name a few). Of the mentioned three, the industrial application, Bebop, enables only model checking of safety properties. Successful applications of these model-checkers to the verification of software are reported, for example, in [BR01, CW02, EKS06]. Researchers then considered more expressive logics that are tailored for pushdown graphs [AEM04][1] and showed how to handle restricted cases of communicating pushdown systems [KIG05, BTP06, KG06, KGS06, KG07]. Recently, model checking and analysis of pushdown systems has been shown to have uses also in security and authentication [SSE06, JSWR06]. Extensions like module checking, probabilistic model checking, and exact computational complexity of model checking with respect to branching time logics were studied as well [BMP05, EE05, Boz06].

[1] See also extensive research on visibly pushdown automata and visibly pushdown languages and games that resulted from the research of this logic [AM04, LMS04, BLS06, AM06, ACM06].

In this paper, we develop an automata-theoretic framework for reasoning about infinite-state sequential systems. The automata-theoretic approach uses the theory of automata as a unifying paradigm for system specification, verification, and synthesis [WVS83, EJ91, Kur94, VW94, KVW00]. Automata enable the separation of the logical and the algorithmic aspects of reasoning about systems, yielding clean and asymptotically optimal algorithms. Automata are the key to techniques such as on-the-fly verification [GPVW95], and they are useful also for modular verification [KV98], partial-order verification [GW94, WW96], verification of real-time systems and hybrid systems [HKV96, DW99], and verification of open systems [AHK97, KV99]. Many decision and synthesis problems have automata-based solutions and no other solution for them is known [EJ88, PR89, KV00b]. Automata-based methods have been implemented in industrial automated-verification tools (c.f., COSPAN [HHK96] and SPIN [Hol97, VB00]).

The automata-theoretic approach, however, has long been thought to be inapplicable for effective reasoning about infinite-state systems. The reason, essentially, lies in the fact that the automata-theoretic techniques involve constructions in which the state space of the system directly influences the state space of the automaton (e.g., when we take the product of a specification automaton with the graph that models the system). On the other hand, the automata we know to handle have finitely many states. The key insight, which enables us to overcome this difficulty, and which is implicit in all previous decidability results in the area of infinite-state sequential systems, is that in spite of the somewhat misleading terminology (e.g., "context-free graphs" and "pushdown graphs"), the classes of infinite-state graphs for which decidability is known can be described by finite-state automata. This is explained by the fact the the states of the graphs that model these systems can be viewed as nodes in an infinite tree and transitions between states can be expressed by finite-state automata. As a result, automata-theoretic techniques can be used to reason about such systems. In particular, we show that various problems related to the analysis of such systems can be reduced to the membership and emptiness problems for *alternating two-way tree automata*, which was shown to be decidable in exponential time [Var98].

We first show how the automata-theoretic framework can be used to solve the μ-calculus model-checking problem with respect to pushdown and prefix-recognizable systems. As explained, the solution is based on the observation that states of such systems correspond to a location in an infinite tree. Transitions of the system, can be simulated by a finite state automaton that reads the infinite tree. Thus, the model-checking problem of μ-calculus over pushdown and prefix-recognizable graphs is reduced to the membership problem of 2-way alternating parity tree automata, namely, the question whether an automaton accepts the tree obtained by unwinding a given finite labeled graph. The complexity of our algorithm matches the complexity of previous algorithms.

The μ-calculus is sufficiently strong to express all properties expressible in the linear temporal logic LTL (and in fact, all properties expressible by an ω-regular language) [Dam94]. Thus, by translating LTL formulas into μ-calculus formulas we can use our solution for μ-calculus model checking in order to solve LTL model checking. This solution, however, is not optimal. This has to do both with the fact that the translation of LTL to μ-calculus is exponential, as well as the fact that our solution

for μ-calculus model checking is based on tree automata. A tree automaton splits into several copies when it runs on a tree. While splitting is essential for reasoning about branching properties, it has a computational price. For linear properties, it is sufficient to follow a single computation of the system, and tree automata seem too strong for this task. For example, while the application of the framework developed above to pushdown systems and LTL properties results in an algorithm that is doubly-exponential in the formula and exponential in the system, the problem is known to be EXPTIME-complete in the formula and polynomial in the system [BEM97].

In order to handle model checking of linear-time properties, we introduce *path automata on trees*. The input to a path automaton is a tree, but the automaton cannot split to copies and it can read only a single path of the tree. In particular, *two-way* nondeterministic path automata enable exactly the type of navigation that is required in order to check linear properties of infinite-state sequential systems. We study the expressive power and the complexity of the decision problems for (two way) path automata. The fact that path automata follow a single path in the tree makes them very similar to two-way nondeterministic automata on infinite words. This enables us to reduce the membership problem (whether an automaton accepts the tree obtained by unwinding a given finite labeled graph) of two-way nondeterministic path automata to the emptiness problem of one-way alternating Büchi automata on infinite words, which was studied in [VW86b]. This leads to a quadratic upper bound for the membership problem for two-way nondeterministic path automata.

Using path automata we are able to solve the problem of LTL model checking with respect to pushdown and prefix-recognizable systems by a reduction to the membership problem of two-way nondeterministic path automata. Usually, automata-theoretic solutions to model checking use the emptiness problem, namely whether an automaton accepts some tree. We note that for (linear-time) model checking of sequential infinite-state system both simplifications, to the membership problem vs. the emptiness problem, and to path automata vs. tree automata are crucial: as we prove the emptiness problem for two-way nondeterministic Büchi path automata is EXPTIME-complete, and the membership problem for two-way alternating Büchi tree automata is also EXPTIME-complete[2]. Our automata-theoretic technique matches the known upper bound for model-checking LTL properties on pushdown systems [BEM97,EHRS00]. In addition, the automata-theoretic approach provides the first solution for the case where the system is prefix-recognizable. Specifically, we show that we can solve the model-checking problem of an LTL formula φ with respect to a prefix-recognizable system R of size n in time and space $2^{O(n+|\varphi|)}$. We also prove a matching EXPTIME lower bound.

Usually, the labeling of the state depends on the internal state of the system and the top of the store. Our framework also handles *regular labeling*, where the label depends

[2] In contract, the membership problem for one-way alternating Büchi tree automata can be reduced to the emptiness problem of the 1-letter alternating word automaton obtained by taking the product of the labeled graph that models the tree with the one-way alternating tree automaton [KVW00]. This technique cannot be applied to two-way automata, since they can distinguish between a graph and its unwinding. For a related discussion regarding past-time connectives in branching temporal logics, see [KP95].

on whether the word on the store is a member in some regular language. The complexity is exponential in the nondeterministic automata that describe the labeling, matching the known bound for pushdown systems and linear-time specifications [EKS01]. The automata-theoretic techniques for handling regular labeling and for handling the regular transitions of a prefix-recognizable system are very similar. This leads us to the understanding that regular labeling and prefix-recognizability have exactly the same power. Formally, we prove that model checking (for either μ-calculus or LTL) with respect to a prefix-recognizable system can be reduced to model checking with respect to a pushdown system with regular labeling, and vice versa. For linear-time properties, it is known that model checking of a pushdown system with regular labeling is EXPTIME-complete [EKS01]. Hence, our reductions suggest an alternative proof of the exponential upper and lower bounds for the problem of LTL model checking of prefix-recognizable systems.

While most of the complexity results established for model checking of infinite-state sequential systems using our framework are not new, it appears to be, like the automata-theoretic framework for finite-state systems, very versatile, and it has further potential applications. We proceed by showing how to solve the *realizability* and *synthesis* problem of μ-calculus formulas with respect to infinite-state sequential environments. Similar methods are used to solve realizability of LTL [ATM03]. We discuss how to extend the algorithms to handle graphs with *regular fairness constraints*, and to μ-calculus with *backward modalities*. In both these problems all we demonstrate is a (fairly simple) extension of the basic algorithm; the (exponentially) hard work is then done by the membership-checking algorithm. The automata-theoretic framework for reasoning about infinite-state sequential systems was also extended to global model checking [PV04] and to classes of systems that are more expressive than prefix-recognizable [Cac03, PV03]. It can be easily extended to handle also CARET specifications [AEM04].

Since the publication of the preliminary versions of this work [KV00a, KPV02], this method has been used extensively. Cachat uses the connection between pushdown-systems and 2-way tree automata to show that μ-calculus model checking over high-order pushdown automata is decidable [Cac03]. Gimbert uses these techniques to consider games over pushdown arenas where the winning conditions are combination of parity and unboundedness [Gim03][3]. Serre shows how these techniques can achieve better upper bounds in the restricted case of counter machines [Ser06].

2 Preliminaries

Given a finite set Σ, a *word* over Σ is a finite or infinite sequence of symbols from Σ. We denote by Σ^* the set of finite sequences over Σ and by Σ^ω the set of infinite sequences over Σ. Given a word $w = \sigma_0\sigma_1\sigma_2 \cdots \in \Sigma^* \cup \Sigma^\omega$, we denote by $w_{\geq i}$ the suffix of w starting at σ_i, i.e., $w_{\geq i} = \sigma_i\sigma_{i+1} \cdots$. The *length* of w is denoted by $|w|$ and is defined to be ω for infinite words.

[3] See also [BSW03] for a different solution when the parity conditions are restricted to index three.

2.1 Labeled Transition Graphs and Rewrite Systems

A *labeled transition graph* is $G = \langle \Sigma, S, L, \rho, s_0 \rangle$, where Σ is a finite set of labels, S is a (possibly infinite) set of states, $L : S \to \Sigma$ is a labeling function, $\rho \subseteq S \times S$ is a transition relation, and $s_0 \in S_0$ is an initial state. When $\rho(s, s')$, we say that s' is a *successor* of s, and s is a *predecessor* of s'. For a state $s \in S$, we denote by $G^s = \langle \Sigma, S, L, \rho, s \rangle$, the graph G with s as its initial state. An *s-computation* is an infinite sequence of states $s_0, s_1, \ldots \in S^\omega$ such that $s_0 = s$ and for all $i \geq 0$, we have $\rho(s_i, s_{i+1})$. An s-computation $s_0, s_1, \ldots$ induces the *s-trace* $L(s_0) \cdot L(s_1) \cdots$. Let $\mathcal{T}_s$ be the set of all s-traces.

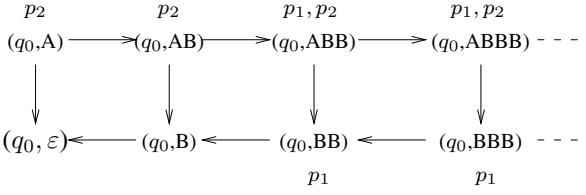
A *rewrite system* is $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$, where Σ is a finite set of labels, V is a finite alphabet, Q is a finite set of states, $L : Q \times V^* \to \Sigma$ is a labeling function, T is a finite set of rewrite rules, to be defined below, q_0 is an initial state, and $x_0 \in V^*$ is an initial word. The set of *configurations* of the system is $Q \times V^*$. Intuitively, the system has finitely many control states and an unbounded store. Thus, in a configuration $(q, x) \in Q \times V^*$ we refer to q as the *control state* and to x as the *store*. A configuration $(q, x) \in Q \times V^*$ indicates that the system is in control state q with store x. We consider here two types of rewrite systems. In a *pushdown* system, each rewrite rule is $\langle q, A, x, q' \rangle \in Q \times V \times V^* \times Q$. Thus, $T \subseteq Q \times V \times V^* \times Q$. In a *prefix-recognizable* system, each rewrite rule is $\langle q, \alpha, \beta, \gamma, q' \rangle \in Q \times \mathit{reg}(V) \times \mathit{reg}(V) \times \mathit{reg}(V) \times Q$, where $\mathit{reg}(V)$ is the set of regular expressions over V. Thus, $T \subseteq Q \times \mathit{reg}(V) \times \mathit{reg}(V) \times \mathit{reg}(V) \times Q$. For a word $w \in V^*$ and a regular expression $r \in \mathit{reg}(V)$ we write $w \in r$ to denote that w is in the language of the regular expression r. We note that the standard definition of prefix-recognizable systems does not include control states. Indeed, a prefix-recognizable system without states can simulate a prefix-recognizable system with states by having the state as the first letter of the unbounded store. We use prefix-recognizable systems with control states for the sake of uniform notation.

We consider two types of labeling functions, *simple* and *regular*. The labeling function associates with a configuration $(q, x) \in Q \times V^*$ a symbol from Σ. A simple labeling function depends only on the first letter of x. Thus, we may write $L : Q \times (V \cup \{\epsilon\}) \to \Sigma$. Note that the label is defined also for the case that x is the empty word ϵ. A regular labeling function considers the entire word x but can only refer to its membership in some regular set. Formally, for every state q there is a partition of V^* to $|\Sigma|$ regular languages $R_1, \ldots R_{|\Sigma|}$, and $L(q, x)$ depends on the regular set that x belongs to. For a letter $\sigma \in \Sigma$ and a state $q \in Q$ we set $R_{\sigma,q} = \{x \mid L(q, x) = \sigma\}$ to be the regular language of store contents that produce the label σ (with state q). We are especially interested in the cases where the alphabet Σ is the powerset 2^{AP} of the set of atomic propositions. In this case, we associate with every state q and proposition p a regular language $R_{p,q}$ that contains all the words w for which the proposition p is true in configuration (q, x). Thus $p \in L(q, x)$ iff $x \in R_{p,q}$. Unless mentioned explicitly, the system has a simple labeling.

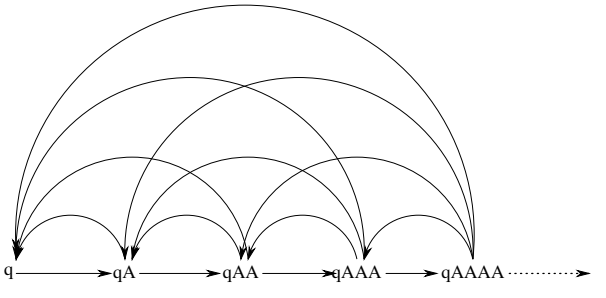
The rewrite system R induces the labeled transition graph whose states are the configurations of R and whose transitions correspond to rewrite rules. Formally, $G_R = \langle \Sigma, Q \times V^*, L, \rho_R, (q_0, x_0) \rangle$, where $Q \times V^*$ is the set of configurations of R and $\langle (q, z), (q', z') \rangle \in \rho_R$ if there is a rewrite rule $t \in T$ leading from configuration (q, z) to

configuration (q', z'). Formally, if R is a pushdown system, then $\rho_R((q, A{\cdot}y), (q', x{\cdot}y))$ if $\langle q, A, x, q' \rangle \in T$; and if R is a prefix-recognizable system, then $\rho_R((q, x \cdot y), (q', x' \cdot y))$ if there are regular expressions α, β, and γ such that $x \in \alpha$, $y \in \beta$, $x' \in \gamma$, and $\langle q, \alpha, \beta, \gamma, q' \rangle \in T$. Note that in order to apply a rewrite rule in state $(q, z) \in Q \times V^*$ of a pushdown graph, we only need to match the state q and the first letter of z with the second element of a rule. On the other hand, in an application of a rewrite rule in a prefix-recognizable graph, we have to match the state q and we should find a partition of z to a prefix that belongs to the second element of the rule and a suffix that belongs to the third element. A labeled transition graph that is induced by a pushdown system is called a *pushdown graph*. A labeled transition system that is induced by a prefix-recognizable system is called a *prefix-recognizable graph*.

Example 1. The pushdown system $P = \langle 2^{\{p_1,p_2\}}, \{A, B\}, \{q_0\}, L, T, q_0, A \rangle$, where L is defined by $R_{q_0,p_1} = \{A, B\}^* \cdot B \cdot B \cdot \{A, B\}^*$ and $R_{q_0,p_2} = A \cdot \{A, B\}^*$ and $T = \{\langle q_0, A, AB, q_0 \rangle, \langle q_0, A, \varepsilon, q_0 \rangle, \langle q_0, B, \varepsilon, q_0 \rangle\}$, induces the labeled transition graph below.



Example 2. The prefix-recognizable system $\langle 2^{\emptyset}, \{A\}, \{q\}, L, T, q_0, A \rangle$, where $T = \{\langle q, A^*, A^*, \varepsilon, q \rangle, \langle q, \varepsilon, A^*, A, q \rangle\}$ induces the labeled transition graph below.



Consider a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$. For a rewrite rule $t_i = \langle s, \alpha_i, \beta_i, \gamma_i, s' \rangle \in T$, let $\mathcal{U}_\lambda = \langle V, Q_\lambda, \eta_\lambda, q^0_\lambda, F_\lambda \rangle$, for $\lambda \in \{\alpha_i, \beta_i, \gamma_i\}$, be the nondeterministic automaton for the language of the regular expression λ. We assume that all initial states have no incoming edges and that all accepting states have no outgoing edges. We collect all the states of all the automata for α, β, and γ regular expressions. Formally, $Q_\alpha = \bigcup_{t_i \in T} Q_{\alpha_i}$, $Q_\beta = \bigcup_{t_i \in T} Q_{\beta_i}$, and $Q_\gamma = \bigcup_{t_i \in T} Q_{\gamma_i}$. We assume that we have an automaton whose language is $\{x_0\}$. We denote the final state of this automaton by x_0 and add all its states to Q_γ. Finally, for a regular labeling function L, a state $q \in Q$, and a letter $\sigma \in \Sigma$, let $\mathcal{U}_{\sigma,q} = \langle V, Q_{\sigma,q}, q^0_{\sigma,q}, \rho_{\sigma,q}, F_{\sigma,q} \rangle$ be the nondeterministic automaton for the language $R_{\sigma,q}$. In a similar way given a state $q \in Q$

and a proposition $p \in AP$, let $\mathcal{U}_{p,q} = \langle V, Q_{p,q}, q^0_{p,q}, \rho_{p,q}, F_{p,q} \rangle$ be the nondeterministic automaton for the language $R_{p,q}$.

We define the *size* $\|T\|$ of T as the space required in order to encode the rewrite rules in T. Thus, in a pushdown system, $\|T\| = \sum_{\langle q,A,x,q' \rangle \in T} |x|$, and in a prefix-recognizable system, $\|T\| = \sum_{\langle q,\alpha,\beta,\gamma,q' \rangle \in T} |\mathcal{U}_\alpha| + |\mathcal{U}_\beta| + |\mathcal{U}_\gamma|$. In the case of a regular labeling function, we also measure the labeling function $\|L\| = \sum_{q \in Q} \sum_{\sigma \in \Sigma} |\mathcal{U}_{\sigma,q}|$ or $\|L\| = \sum_{q \in Q} \sum_{p \in AP} |\mathcal{U}_{p,q}|$.

2.2 Temporal Logics

We give a short introduction to the tempora logics LTL [Pnu77] and μ-calculus [Koz83].

The logic LTL augments propositional logic with temporal quantifiers. Given a finite set AP of propositions, an LTL formula is one of the following.

- $\mathbf{true}$, $\mathbf{false}$, p for all $p \in AP$;
- $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, $\bigcirc \varphi_1$ and $\varphi_1 U \varphi_2$, for LTL formulas φ_1 and φ_2;

The semantics of LTL formulas is defined with respect to an infinite sequence $\pi \in (2^{AP})^\omega$ and a location $i \in \mathbb{N}$. We use $(\pi, i) \models \psi$ to indicate that the word π in the designated location i satisfies the formula ψ.

- For a proposition $p \in AP$, we have $(\pi, i) \models p$ iff $p \in \pi_i$;
- $(\pi, i) \models \neg f_1$ iff not $(\pi, i) \models f_1$;
- $(\pi, i) \models f_1 \vee f_2$ iff $(\pi, i) \models f_1$ or $(\pi, i) \models f_2$;
- $(\pi, i) \models f_1 \wedge f_2$ iff $(\pi, i) \models f_1$ and $(\pi, i) \models f_2$;
- $(\pi, i) \models \bigcirc f_1$ iff $(\pi, i + 1) \models f_1$;
- $(\pi, i) \models f_1 \mathcal{U} f_2$ iff there exists $k \geq i$ such that $(\pi, k) \models f_2$ and for all $i \leq j < k$, we have $(\pi, j) \models f_1$;

If $(\pi, 0) \models \psi$, then we say that π *satisfies* ψ. We denote by $L(\psi)$ the set of sequences π that satisfy ψ.

The *μ-calculus* is a modal logic augmented with least and greatest fixpoint operators. Given a finite set AP of atomic propositions and a finite set Var of variables, a μ-calculus formula (in a positive normal form) over AP and Var is one of the following:

- $\mathbf{true}$, $\mathbf{false}$, p and $\neg p$ for all $p \in AP$, or y for all $y \in Var$;
- $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, for μ-calculus formulas φ_1 and φ_2;
- $\Box\varphi$ or $\Diamond\varphi$ for a μ-calculus formula φ.
- $\mu y.\varphi$ or $\nu y.\varphi$, for $y \in Var$ and a μ-calculus formula φ.

A *sentence* is a formula that contains no free variables from Var (that is, every variable is in the scope of some fixed-point operator that binds it). We define the semantics of μ-calculus with respect to a labeled transition graph $G = \langle 2^{AP}, S, L, \rho, s_0 \rangle$ and a valuation $\mathcal{V} : Var \to 2^S$. Each formula ψ and valuation $\mathcal{V}$ then define a set $[[\psi]]^G_{\mathcal{V}}$ of states of G that satisfy the formula. For a valuation $\mathcal{V}$, a variable $y \in Var$, and a set $S' \subseteq S$, we denote by $\mathcal{V}[y \leftarrow S']$ the valuation obtained from $\mathcal{V}$ by assigning S' to y. The mapping $[[\psi]]^G_{\mathcal{V}}$ is defined inductively as follows:

- $[[\mathbf{true}]]^G_{\mathcal{V}} = S$ and $[[\mathbf{false}]]^G_{\mathcal{V}} = \emptyset$;
- For $y \in Var$, we have $[[y]]^G_{\mathcal{V}} = \mathcal{V}(y)$;
- For $p \in AP$, we have $[[p]]^G_{\mathcal{V}} = \{s \mid p \in L(s)\}$ and $[[\neg p]]^G_{\mathcal{V}} = \{s \mid p \notin L(s)\}$;
- $[[\psi_1 \wedge \psi_2]]^G_{\mathcal{V}} = [[\psi_1]]^G_{\mathcal{V}} \cap [[\psi_2]]^G_{\mathcal{V}}$;
- $[[\psi_1 \vee \psi_2]]^G_{\mathcal{V}} = [[\psi_1]]^G_{\mathcal{V}} \cup [[\psi_2]]^G_{\mathcal{V}}$;
- $[[\Box\psi]]^G_{\mathcal{V}} = \{s \in S :$ for all s' such that $\rho(s, s'),$ we have $s' \in [[\psi]]^G_{\mathcal{V}}\}$;
- $[[\Diamond\psi]]^G_{\mathcal{V}} = \{s \in S :$ there is s' such that $\rho(s, s')$ and $s' \in [[\psi]]^G_{\mathcal{V}}\}$;
- $[[\mu y.\psi]]^G_{\mathcal{V}} = \bigcap\{S' \subseteq S : [[\psi]]^G_{\mathcal{V}[y \leftarrow S']} \subseteq S'\}$;
- $[[\nu y.\psi]]^G_{\mathcal{V}} = \bigcup\{S' \subseteq S : S' \subseteq [[\psi]]^G_{\mathcal{V}[y \leftarrow S']}\}$.

The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. For a full exposition of μ-calculus we refer the reader to [Eme97].

Note that $[[\psi]]^G_{\mathcal{V}}$ depends only on the valuations of the free variables in ψ. In particular, no valuation is required for a sentence and we write $[[\psi]]^G$. For a state $s \in S$ and a sentence ψ, we say that ψ holds at s in G, denoted $G, s \models \psi$ iff $s \in [[\psi]]^G$. Also, $G \models \psi$ iff $G, s_0 \models \psi$. We say that a rewrite system R satisfies a μ-calculus formula ψ if $G_R \models \psi$.

While LTL is a linear temporal logic and we have defined its semantics with respect to infinite sequences, we often refer also to satisfaction of LTL formulas in labeled transition graphs. Intuitively, all the sequences induced by computations of the graph should satisfy the formula. Formally, given a graph G and a state s of G, we say that s satisfies an LTL formula φ, denoted $(G, s) \models \varphi$, iff $\mathcal{T}_s \subseteq \mathcal{L}(\varphi)$. A graph G satisfies an LTL formula φ, denoted $G \models \varphi$, iff its initial state satisfies it; that is $(G, s_0) \models \varphi$.

The *model-checking problem* for a labeled transition graph G and an LTL or μ-calculus formula φ is to determine whether G satisfies φ. Note that the transition relation of R need not be total. There may be finite paths but satisfaction is determined only with respect to infinite paths. In particular, if the graph has only finite paths, its set of traces is empty and the graph satisfies every LTL formula [4]. We say that a rewrite system R satisfies an LTL formula φ if $G_R \models \varphi$.[5]

Theorem 1. *The model-checking problem for a pushdown system R and a formula φ is solvable*

- *in time $\|T\|^3 \cdot 2^{O(|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|)}$ in the case φ is an LTL formula and L is a simple labeling function* [EHRS00].
- *in time $\|T\|^3 \cdot 2^{O(\|L\|+|\varphi|)}$ and space $\|T\|^2 \cdot 2^{O(\|L\|+|\varphi|)}$ in the case φ is an LTL formulas and L is a regular labeling function. The problem is EXPTIME-hard in $\|L\|$ even for a fixed formula* [EKS01].
- *in time $2^{O(\|T\| \cdot |\psi| \cdot k)}$ in the case φ is a μ-calculus formula with alternation depth k* [Wal96, Bur97].

[4] It is also possible to consider finite paths. In this case, the nondeterministic Büchi automaton in Theorem 5 has to be modified so that it can recognize also finite words (cf. [GO03]). Our results are easily extended to consider also finite paths.

[5] Some work on verification of infinite-state system (e.g., [EHRS00]), consider properties given by nondeterministic Büchi word automata, rather than LTL formulas. Since we anyway translate LTL formulas to automata, we can easily handle also properties given by automata.

2.3 Alternating Two-Way Automata

Given a finite set Υ of directions, an *Υ-tree* is a set $T \subseteq \Upsilon^*$ such that if $\upsilon \cdot x \in T$, where $\upsilon \in \Upsilon$ and $x \in \Upsilon^*$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $\upsilon \in \Upsilon$ and $x \in T$, the node x is the *parent* of $\upsilon \cdot x$. Each node $x \neq \varepsilon$ of T has a *direction* in Υ. The direction of the root is the symbol $\perp$ (we assume that $\perp \notin \Upsilon$). The direction of a node $\upsilon \cdot x$ is υ. We denote by $dir(x)$ the direction of node x. An Υ-tree T is a *full infinite tree* if $T = \Upsilon^*$. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$ there exists a unique $\upsilon \in \Upsilon$ such that $\upsilon \cdot x \in \pi$. Note that our definitions here reverse the standard definitions (e.g., when $\Upsilon = \{0, 1\}$, the successors of the node 0 are 00 and 10 (rather than 00 and 01)[6].

Given two finite sets Υ and Σ, a *Σ-labeled Υ-tree* is a pair $\langle T, \tau \rangle$ where T is an Υ-tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, \tau \rangle$ a labeled tree. We say that an $((\Upsilon \cup \{\perp\}) \times \Sigma)$-labeled Υ-tree $\langle T, \tau \rangle$ is *Υ-exhaustive* if for every node $x \in T$, we have $\tau(x) \in \{dir(x)\} \times \Sigma$.

A labeled tree is *regular* if it is the unwinding of some finite labeled graph. More formally, a *transducer* $\mathcal{D}$ is a tuple $\langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, where Υ is a finite set of directions, Σ is a finite alphabet, Q is a finite set of states, $\eta : Q \times \Upsilon \to Q$ is a deterministic transition function, $q_0 \in Q$ is a start state, and $L : Q \to \Sigma$ is a labeling function. We define $\eta : \Upsilon^* \to Q$ in the standard way: $\eta(\varepsilon) = q_0$ and $\eta(ax) = \eta(\eta(x), a)$. Intuitively, a transducer is a labeled finite graph with a designated start node, where the edges are labeled by Υ and the nodes are labeled by Σ. A Σ-labeled Υ-tree $\langle \Upsilon^*, \tau \rangle$ is regular if there exists a transducer $\mathcal{D} = \langle \Upsilon, \Sigma, Q, \eta, q_0, L \rangle$, such that for every $x \in \Upsilon^*$, we have $\tau(x) = L(\eta(x))$. The size of $\langle \Upsilon^*, \tau \rangle$, denoted $\|\tau\|$, is $|Q|$, the number of states of $\mathcal{D}$.

Alternating automata on infinite trees generalize nondeterministic tree automata and were first introduced in [MS87]. Here we describe alternating *two-way* tree automata. For a finite set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas $\mathbf{true}$ and $\mathbf{false}$, and, as usual, $\wedge$ has precedence over $\vee$. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ iff assigning $\mathbf{true}$ to elements in Y and assigning $\mathbf{false}$ to elements in $X \setminus Y$ makes θ true. For a set Υ of directions, the *extension* of Υ is the set $ext(\Upsilon) = \Upsilon \cup \{\varepsilon, \uparrow\}$ (we assume that $\Upsilon \cap \{\varepsilon, \uparrow\} = \emptyset$). An *alternating two-way automaton* over Σ-labeled Υ-trees is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(ext(\Upsilon) \times Q)$ is the transition function, $q_0 \in Q$ is an initial state, and F specifies the acceptance condition.

A run of an alternating automaton $\mathcal{A}$ over a labeled tree $\langle \Upsilon^*, \tau \rangle$ is a labeled tree $\langle T_r, r \rangle$ in which every node is labeled by an element of $\Upsilon^* \times Q$. A node in T_r, labeled by (x, q), describes a copy of the automaton that is in the state q and reads the node x of Υ^*. Note that many nodes of T_r can correspond to the same node of Υ^*; there is no one-to-one correspondence between the nodes of the run and the nodes of the tree. The labels of a node and its successors have to satisfy the transition function. Formally,

[6] As will get clearer in the sequel, the reason for that is that rewrite rules refer to the prefix of words.

a run $\langle T_r, r\rangle$ is a Σ_r-labeled Γ-tree, for some set Γ of directions, where $\Sigma_r = \Upsilon^* \times Q$ and $\langle T_r, r\rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (\varepsilon, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, \tau(x)) = \theta$. Then there is a (possibly empty) set $S \subseteq ext(\Upsilon) \times Q$, such that S satisfies θ, and for all $\langle c, q'\rangle \in S$, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and the following hold:
   - If $c \in \Upsilon$, then $r(\gamma \cdot y) = (c \cdot x, q')$.
   - If $c = \varepsilon$, then $r(\gamma \cdot y) = (x, q')$.
   - If $c = \uparrow$, then $x = \upsilon \cdot z$, for some $\upsilon \in \Upsilon$ and $z \in \Upsilon^*$, and $r(\gamma \cdot y) = (z, q')$.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $c = \uparrow$, we require that $x \neq \varepsilon$.

A run $\langle T_r, r\rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. We consider here *parity* acceptance conditions [EJ91]. A parity condition over a state set Q is a finite sequence $F = \{F_1, F_2, \ldots, F_m\}$ of subsets of Q, where $F_1 \subseteq F_2 \subseteq \ldots \subseteq F_m = Q$. The number m of sets is called the *index* of $\mathcal{A}$. Given a run $\langle T_r, r\rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in \Upsilon^* \times \{q\}$. That is, $inf(\pi)$ contains exactly all the states that appear infinitely often in π. A path π satisfies the condition F if there is an even i for which $inf(\pi) \cap F_i \neq \emptyset$ and $inf(\pi) \cap F_{i-1} = \emptyset$. An automaton accepts a labeled tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. The automaton $\mathcal{A}$ is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$. *Büchi* acceptance condition [Büc62] is a private case of parity of index 3. Büchi condition $F \subseteq Q$ is equivalent to parity condition $\langle \emptyset, F, Q\rangle$. A path π satisfies Büchi condition F iff $inf(\pi) \cap F \neq \emptyset$. *Co-Büchi* acceptance condition is the dual of Büchi. Co-Büchi condition $F \subseteq Q$ is equivalent to parity condition $\langle F, Q\rangle$. A path π satisfies co-Büchi condition F iff $inf(\pi) \cap F = \emptyset$.

The size of an automaton is determined by the number of its states and the size of its transition function. The size of the transition function is $\eta = \Sigma_{q\in Q}\Sigma_{\sigma\in\Sigma}|\eta(q, a)|$ where, for a formula in $B^+(ext(\Upsilon) \times Q)$ we define $|(\Delta, q)| = |\mathbf{true}| = |\mathbf{false}| = 1$ and $|\theta_1 \vee \theta_2| = |\theta_1 \wedge \theta_2| = |\theta_1| + |\theta_2| + 1$.

We say that $\mathcal{A}$ is advancing if δ is restricted to formulas in $B^+((\Upsilon \cup \{\varepsilon\}) \times Q)$, it is one-way if δ is restricted to formulas in $B^+(\Upsilon \times Q)$. We say that $\mathcal{A}$ is nondeterministic if its transitions are of the form $\bigvee_{i\in I} \bigwedge_{\upsilon\in\Upsilon}(\upsilon, q^i_\upsilon))$, in such cases we write $\delta : Q \times \Sigma \to 2^{Q^{|\Upsilon|}}$. In particular, a nondeterministic automaton is 1-way. It is easy to see that a run tree of a nondeterministic tree automaton visits every node in the input tree exactly once. Hence, a run of a nondeterministic tree automaton on tree $\langle T, \tau\rangle$ is $\langle T, r\rangle$ where $r : T \to Q$. Note, that τ and r use the same domain T. In the case that $|\Upsilon| = 1$, $\mathcal{A}$ is a word automaton. In the run of a word automaton, the location of the automaton on the word is identified by the length of its location. Hence, instead of marking the location by υ^i, we mark it by i. Formally, a run of a word automaton is $\langle T, r\rangle$ where $r : T \to \mathbb{N} \times Q$ and a node $x \in T$ such that $r(x) = (i, q)$ signifies that the automaton in state q is reading the ith letter of the word. In the case of word automata, there is only one direction $\upsilon \in \Upsilon$. Hence, we replace the atoms $(d, q) \in ext(\Upsilon) \times Q$ in the

transition of $\mathcal{A}$ by atoms from $\{-1, 0, 1\} \times Q$ where -1 means read the previous letter, 0 means read again the same letter, and 1 means read the next letter. Accordingly, the pair $(i, q), (j, q')$ satisfies the transition of $\mathcal{A}$ if there exists $(d, q') \in \delta(q, w_i)$ such that $j = i + d$. In the case that the automaton is 1-way the length of x uniquely identifies the location in the word. That is, we use $r : T \to Q$ and $r(x) = q$ signifies that state q is reading letter $|x|$. In the case that a word automaton is nondeterministic, its run is an infinite sequence of locations and states. Namely, $r = (0, q_0), (i_1, q_1), \ldots$. In addition, if the automaton is 1-way the location in the sequence identifies the letter read by the automaton and we write $r = q_0, q_1, \ldots$.

Theorem 2. *Given an alternating two-way parity tree automaton $\mathcal{A}$ with n states and index k, we can construct an equivalent nondeterministic one-way parity tree automaton whose number of states is exponential in nk and whose index is linear in nk* [Var98]*, and we can check the nonemptiness of $\mathcal{A}$ in time exponential in nk* [EJS93].

We use acronyms in $\{2, \varepsilon, 1\} \times \{A, N, D\} \times \{P, B, C, F\} \times \{T, W\}$ to denote the different types of automata. The first symbol stands for the type of movement: 2 for 2-way automata, ε for advancing, and 1 for 1-way (we often omit the 1). The second symbol stands for the branching mode: A for alternating, N for nondeterministic, and D for deterministic. The third symbol stands for the type of acceptance: P for parity, B for Büchi, C for co-Büchi, and F for finite (i.e., automata that read finite words or trees), and the last symbol stands for the object the automaton is reading: T for trees and W for words. For example, a 2APT is a 2-way alternating parity tree automaton and an NBW is a 1-way nondeterministic Büchi word automaton.

The *membership problem* of an automaton $\mathcal{A}$ and a regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine whether $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$; or equivalently whether $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(A)$. It is not hard to see that the membership problem for a 2APT can be solved by a reduction to the emptiness problem. Formally we have the following.

Theorem 3. *Given an alternating two-way parity tree automaton $\mathcal{A}$ with n states and index k, and a regular tree $\langle \Upsilon^*, \tau \rangle$ we can check whether $\mathcal{A}$ accepts $\langle \Upsilon^*, \tau \rangle$ in time $(\|\tau\| nk)^{O((nk)^2)}$.*

Proof: Let $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ be a 2APT and $\langle \Upsilon^*, \tau \rangle$ be a regular tree. Let the transducer inducing the labeling of τ be $\mathcal{D}_\tau = \langle \Upsilon, \Sigma, D, \eta, d_0, L \rangle$. According to Theorem 2, we construct a 1NPT $N = \langle \Sigma, S, \rho, s_0, \alpha \rangle$ that accepts the language of A.

Consider the 1NPT $N' = \langle \{a\}, D \times S, \rho', (d_0, s_0), \alpha' \rangle$ where $\rho'(d, s)$ is obtained from $\rho(s, L(d))$ by replacing every atom (υ, s') by $(\upsilon, (\eta(d, \upsilon), s'))$ and α' is obtained from α by replacing every set F by the set $D \times F$. It follows that $\langle \Upsilon^*, \tau \rangle$ is accepted by $\mathcal{A}$ iff N' is not empty. The number of states of N' is $\|\tau\|(nk)^{O(nk)}$ and its index is $O(nk)$. □

2.4 Alternating Automata on Labeled Transition Graphs

Consider a labeled transition graph $G = \langle \Sigma, S, L, \rho, s_0 \rangle$. Let $\Delta = \{\varepsilon, \Box, \Diamond\}$. An alternating automaton on labeled transition graphs (*graph automaton*, for short) [JW95][7]

[7] The graph automata in [JW95] are different than these defined here, but this is only a technical difference.

is a tuple $\mathcal{S} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ, Q, q_0, and F are as in alternating two-way automata, and $\delta : Q \times \Sigma \to \mathcal{B}^+(\Delta \times Q)$ is the transition function. Intuitively, when $\mathcal{S}$ is in state q and it reads a state s of G, fulfilling an atom $\langle \Diamond, t \rangle$ (or $\Diamond t$, for short) requires $\mathcal{S}$ to send a copy in state t to some successor of s. Similarly, fulfilling an atom $\Box t$ requires $\mathcal{S}$ to send copies in state t to all the successors of s. Thus, like symmetric automata [DW99, Wil99], graph automata cannot distinguish between the various successors of a state and treat them in an existential or universal way.

Like runs of alternating two-way automata, a run of a graph automaton $\mathcal{S}$ over a labeled transition graph $G = \langle S, L, \rho, s_0 \rangle$ is a labeled tree in which every node is labeled by an element of $S \times Q$. A node labeled by (s, q), describes a copy of the automaton that is in the state q of $\mathcal{S}$ and reads the state s of G. Formally, a run is a Σ_r-labeled Γ-tree $\langle T_r, r \rangle$, where Γ is an arbitrary set of directions, $\Sigma_r = S \times Q$, and $\langle T_r, r \rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (s_0, q_0)$.
2. Consider $y \in T_r$ with $r(y) = (s, q)$ and $\delta(q, L(s)) = \theta$. Then there is a (possibly empty) set $S \subseteq \Delta \times Q$, such that S satisfies θ, and for all $\langle c, q' \rangle \in S$, the following hold:
   - If $c = \varepsilon$, then there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s, q')$.
   - If $c = \Box$, then for every successor s' of s, there is $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.
   - If $c = \Diamond$, then there is a successor s' of s and $\gamma \in \Gamma$ such that $\gamma \cdot y \in T_r$ and $r(\gamma \cdot y) = (s', q')$.

A run $\langle T_r, r \rangle$ is *accepting* if all its infinite paths satisfy the acceptance condition. The graph G is accepted by $\mathcal{S}$ if there is an accepting run on it. We denote by $\mathcal{L}(\mathcal{S})$ the set of all graphs that $\mathcal{S}$ accepts. We denote by $\mathcal{S}^q = \langle \Sigma, Q, \delta, q, F \rangle$ the automaton $\mathcal{S}$ with q as its initial state.

We say that a labeled transition graph G satisfies a graph automaton $\mathcal{S}$, denoted $G \models \mathcal{S}$, if $\mathcal{S}$ accepts G. It is shown in [JW95] that graph automata are as expressive as μ-calculus. In particular, we have the following.

Theorem 4. [JW95] *Given a μ-calculus formula ψ, of length n and alternation depth k, we can construct a graph parity automaton $\mathcal{S}_\psi$ such that $L(\mathcal{S}_\psi)$ is exactly the set of graphs satisfying ψ. The automaton $\mathcal{S}_\psi$ has n states and index k.*

A graph automaton whose transitions are restricted to disjunctions over $\{\Diamond\} \times Q$ is in fact a nondeterministic automaton. We freely confuse between such graph automata with the Büchi acceptance condition and NBW. It is well known that every LTL formula can be translated into an NBW that accepts all traces that satisfy the formula. Formally, we have the following.

Theorem 5. [VW94] *For every LTL formula φ, we can construct an NBW N_φ with $2^{O(|\varphi|)}$ states such that $L(N_\varphi) = L(\varphi)$.*

3 Model-Checking Branching-Time Properties

In this section we present an automata-theoretic approach solution to model-checking branching-time properties of pushdown and prefix-recognizable graphs. We start by

demonstrating our technique on model checking of pushdown systems. Then we show how to extend it to prefix-recognizable systems. Consider a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and let $G_R = \langle \Sigma, Q \times V^*, L, \rho_R, (q_0, x_0) \rangle$ be its induced graph. recall that a configuration of G_R is a pair $(q, x) \in Q \times V^*$. Thus, the store x corresponds to a node in the full infinite V-tree. An automaton that reads the tree V^* can memorize in its state space the state component of the configuration and refer to the location of its reading head in V^* as the store. We would like the automaton to "know" the location of its reading head in V^*. A straightforward way to do so is to label a node $x \in V^*$ by x. This, however, involves an infinite alphabet, and results in trees that are not regular.

We show that labeling every node in V^* by its direction is sufficiently informative to provide the 2-way automaton with the information it needs in order to simulate transitions of the rewrite system. Thus, if R is a pushdown system and we are at node $A \cdot y$ of the V-tree (with state q memorized), an application of the transition $\langle q, A, x, q' \rangle$ takes us to node $x \cdot y$ (with state q' memorized). If R is a prefix-recognizable system and we are at node y of the V-tree (with state q memorized), an application of the transition $\langle q, \alpha, \beta, \gamma, q' \rangle$ takes us to node xz (with state q' memorized) where $x \in \gamma$, $z \in \beta$, and $y = z'z$ for some $z' \in \alpha$. Technically, this means that we first move up the tree, and then move down. Such a navigation through the V-tree can be easily performed by two-way automata.

3.1 Pushdown Graphs

We present our solution for pushdown graphs in details. Let $\langle V^*, \tau_V \rangle$ be the V-labeled V-tree such that for every $x \in V^*$ we have $\tau_V(x) = dir(x)$ ($\langle V^*, \tau_V \rangle$ is the exhaustive V-labeled V-tree). Note that $\langle V^*, \tau_V \rangle$ is a regular tree of size $|V| + 1$. We construct a 2APT $\mathcal{A}$ that reads $\langle V^*, \tau_V \rangle$. The state space of $\mathcal{A}$ contains a component that memorizes the current state of the rewrite system. The location of the reading head in $\langle V^*, \tau_V \rangle$ represents the store of the current configuration. Thus, in order to know which rewrite rules can be applied, $\mathcal{A}$ consults its current state and the label of the node it reads (note that $dir(x)$ is the first letter of x). Formally, we have the following.

Theorem 6. *Given a pushdown system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\bot\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $G_{\mathcal{R}}$ satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\|)$ states, and has the same index as $\mathcal{S}$.*

Proof: We define $\mathcal{A} = \langle V \cup \{\bot\}, P, \eta, p_0, \alpha \rangle$ as follows.

- $P = (W \times Q \times heads(T))$, where $heads(T) \subseteq V^*$ is the set of all prefixes of words $x \in V^*$ for which there are states $q, q' \in Q$ and $A \in V$ such that $\langle q, A, x, q' \rangle \in T$. Intuitively, when $\mathcal{A}$ visits a node $x \in V^*$ in state $\langle w, q, y \rangle$, it checks that G_R with initial state $(q, y \cdot x)$ is accepted by $\mathcal{S}^s$. In particular, when $y = \varepsilon$, then G_R with initial state (q, x) (the node currently being visited) needs to be accepted by $\mathcal{S}^w$. States of the form $\langle w, q, \varepsilon \rangle$ are called *action states*. From these states $\mathcal{A}$ consults δ and T in order to impose new requirements on the exhaustive V-tree. States of the form $\langle w, q, y \rangle$, for $y \in V^+$, are called *navigation states*. From these states $\mathcal{A}$ only navigates downwards y to reach new action states.

– In order to define $\eta : P \times \Sigma \to \mathcal{B}^+(ext(V) \times P)$, we first define the function $apply_T : \Delta \times W \times Q \times V \to \mathcal{B}^+(ext(V) \times P)$. Intuitively, $apply_T$ transforms atoms participating in δ to a formula that describes the requirements on G_R when the rewrite rules in T are applied to words of the form $A \cdot V^*$. For $c \in \Delta$, $w \in W$, $q \in Q$, and $A \in V$ we define

$$apply_T(c, w, q, A) = \begin{cases} \langle \varepsilon, (w, q, \varepsilon) \rangle & \text{if } c = \varepsilon \\ \bigwedge_{\langle q, A, y, q' \rangle \in T} \langle \uparrow, (w, q', y) \rangle & \text{if } c = \Box \\ \bigvee_{\langle q, A, y, q' \rangle \in T} \langle \uparrow, (w, q', y) \rangle & \text{if } c = \Diamond \end{cases}$$

Note that T may contain no tuples in $\{q\} \times \{A\} \times V^* \times Q$ (that is, the transition relation of G_R may not be total). In particular, this happens when $A = \bot$ (that is, for every state $q \in Q$ the configuration (q, ε) of G_R has no successors). Then, we take empty conjunctions as $\mathbf{true}$, and take empty disjunctions as $\mathbf{false}$.

In order to understand the function $apply_T$, consider the case $c = \Box$. When $\mathcal{S}$ reads the configuration $(q, A \cdot x)$ of the input graph, fulfilling the atom $\Box w$ requires $\mathcal{S}$ to send copies in state w to all the successors of $(q, A \cdot x)$. The automaton $\mathcal{A}$ then sends to the node x copies that check whether all the configuration $(q', y \cdot x)$, with $\rho_R((q, A \cdot x), (q', y \cdot x))$, are accepted by $\mathcal{S}$ with initial state w.

Now, for a formula $\theta \in \mathcal{B}^+(\Delta \times S)$, the formula $apply_T(\theta, q, A) \in \mathcal{B}^+(ext(V) \times P)$ is obtained from θ by replacing an atom $\langle c, w \rangle$ by the atom $apply_R(c, w, q, A)$. We can now define η for all $A \in V \cup \{\bot\}$ as follows.

- $\eta(\langle w, q, \varepsilon \rangle, A) = apply_T(\delta(w, L(q, A)), q, A)$.
- $\eta(\langle w, q, y \cdot B \rangle, A) = (B, \langle w, q, y \rangle)$.

Thus, in action states, $\mathcal{A}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to δ. In navigation states, $\mathcal{A}$ needs to go downwards $y \cdot B$, so it continues in direction B.

– $p_0 = \langle w_0, q_0, x_0 \rangle$. Thus, in its initial state $\mathcal{A}$ checks that G_R with initial configuration (q_0, x_0) is accepted by $\mathcal{S}$ with initial state w_0.
– α is obtained from F by replacing each set F_i by the set $S \times F_i \times heads(T)$.

We show that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \models \mathcal{S}$. Assume that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $\langle T, r \rangle$ of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subtree of nodes labeled by action states. That is, consider the following tree $\langle T', r' \rangle$ defined by induction. We know that $r(\varepsilon) = (\varepsilon, (w_0, q_0, x_0))$. It follows that there exists a unique minimal (according to the inverse lexicographic order on the nodes of T) node $y \in T$ labeled by an action state. In our case, $r(y) = (x_0, (w_0, q_0, \varepsilon))$. We add ε to T' and label it $r'(\varepsilon) = ((q_0, x_0), w_0)$. Consider a node z' in T' labeled $r'(z') = ((q, x), w)$. By the construction of $\langle T', r' \rangle$ there exists $z \in T$ such that $r(z) = (x, (w, q, \epsilon))$. Let $\{z_1 \cdot z, \ldots, z_k \cdot z\}$ be the set of minimal nodes in T such that $z_i \cdot z$ is labeled by an action state, $r(z_i \cdot z) = (x_i, (w_i, q_i, \varepsilon))$. We add k successors $a_1 z', \ldots a_k z'$ to z' in T' and set $r'(a_i z') = ((q_i, x_i), w_i)$. By the definition of η, the tree $\langle T', r' \rangle$ is a valid run tree of $\mathcal{S}$ on G_R. Consider an infinite path π' in $\langle T', r' \rangle$. The labels of nodes in π' identify a unique path π in $\langle T, r \rangle$. It follows that the minimal rank appearing infinitely often along π' is the minimal rank appearing infinitely often along π. Hence, $\langle T', r' \rangle$ is accepting and $\mathcal{S}$ accepts G_R.

Assume now that $G_R \models \mathcal{S}$. Then, there exists an accepting run tree $\langle T', r' \rangle$ of $\mathcal{S}$ on G_R. The tree $\langle T', r' \rangle$ serves as the action state skeleton to an accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. A node $z \in T'$ labeled by $((q, x), w)$ corresponds to a copy of $\mathcal{A}$ in state (w, q, ε) reading node x of $\langle V^*, \tau_V \rangle$. It is not hard to extend this skeleton into a valid and accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. □

Pushdown systems can be viewed as a special case of prefix-recognizable systems. In the next subsection we describe how to extend the construction described above to prefix-recognizable graphs, and we also analyze the complexity of the model-checking algorithm that follows for the two types of systems.

3.2 Prefix-Recognizable Graphs

We now extend the construction described in Subsection 3.1 to prefix-recognizable systems. The idea is similar: two-way automata can navigate through the full V-tree and simulate transitions in a rewrite graph by a chain of transitions in the tree. While in pushdown systems the application of rewrite rules involved one move up the tree and then a chain of moves down, here things are a bit more involved. In order to apply a rewrite rule $\langle q, \alpha, \beta, \gamma, q' \rangle$, the automaton has to move upwards along a word in α, check that the remaining word leading to the root is in β, and move downwards along a word in γ. As we explain below, $\mathcal{A}$ does so by simulating automata for the regular expressions participating in T.

Theorem 7. *Given a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, T, L, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $(V \cup \{\bot\})$-labeled V-trees such that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff G_R satisfies $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\|)$ states, and its index is the index of $\mathcal{S}$ plus 1.*

Proof: As in the case of pushdown systems, $\mathcal{A}$ uses the labels on $\langle V^*, \tau_V \rangle$ in order to learn the location in V^* that each node corresponds to. As there, $\mathcal{A}$ applies to the transition function δ of $\mathcal{S}$ the rewrite rules of R. Here, however, the application of the rewrite rules on atoms of the form $\Diamond w$ and $\Box w$ is more involved, and we describe it below. Assume that $\mathcal{A}$ wants to check whether $\mathcal{S}^w$ accepts $G_R^{(q,x)}$, and it wants to proceed with an atom $\Diamond w'$ in $\delta(w, L(q, x))$. The automaton $\mathcal{A}$ needs to check whether $\mathcal{S}^{w'}$ accepts $G_R^{(q',y)}$ for some configuration (q', y) reachable from (q, x) by applying a rewrite rule. That is, a configuration (q', y) for which there is $\langle q, \alpha, \beta, \gamma, q' \rangle \in T$ and partitions $x' \cdot z$ and $y' \cdot z$, of x and y, respectively, such that x' is accepted by $\mathcal{U}_\alpha$, z is accepted by $\mathcal{U}_\beta$, and y' accepted by $\mathcal{U}_\gamma$. The way $\mathcal{A}$ detects such a state y is the following. From the node x, the automaton $\mathcal{A}$ simulates the automaton $\mathcal{U}_\alpha$ upwards (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_\alpha$ on the word it reads as it proceeds on direction $\uparrow$ from x towards the root of the V-tree). Suppose that on its way up to the root, $\mathcal{A}$ encounters a state in F_α as it reads the node $z \in V^*$. This means that the word read so far is in α, and can serve as the prefix x' above. If this is indeed the case, then it is left to check that the word z is accepted by $\mathcal{U}_\beta$, and that there is a state that is obtained from z by prefixing it with a word $y' \in \gamma$ that is accepted by $\mathcal{S}^{w'}$. To check the first condition, $\mathcal{A}$ sends a copy in direction $\uparrow$ that simulates a run of $\mathcal{U}_\beta$, hoping to reach a state in F_β as it reaches

the root (that is, $\mathcal{A}$ guesses a run of $\mathcal{U}_\beta$ on the word it reads as it proceeds from z up to the root of $\langle V^*, \tau_V \rangle$). To check the second condition, $\mathcal{A}$ simulates the automaton $\mathcal{U}_\gamma$ downwards starting from F_γ. A node $y' \cdot z \in V^*$ that $\mathcal{A}$ reads as it encounters q^0_γ can serve as the state y we are after. The case for an atom $\Box w'$ is similar, only that here $\mathcal{A}$ needs to check whether $\mathcal{S}^{w'}$ accepts $G_R^{(q',y)}$ for all configurations (q', y) reachable from (q, x) by applying a rewrite rule, and thus the choices made by $\mathcal{A}$ for guessing the partition $x' \cdot z$ of x and the prefix y' of y are now treated dually. More formally, we have the following.

We define $\mathcal{A} = \langle V \cup \{\bot\}, P, \eta, p_0, \alpha \rangle$ as follows.

- $P = P_1 \cup P_2$ where $P_1 = \{\exists, \forall\} \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma)$ and $P_2 = \{\exists, \forall\} \times T \times Q_\beta$. States in P_1 serve to simulate automata for α and γ regular expressions and states in P_2 serve to simulate automata for β regular expressions. A state marked by $\exists$ participates in the simulation of a $\Diamond s$ atom of $\mathcal{S}$, and a state marked by $\forall$ participates in the simulation of a $\Box s$ atom of $\mathcal{S}$. A state in P_1 marked by the transition $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ and a state $s \in Q_{\alpha_i}$ participates in the simulation of a run of $\mathcal{U}_{\alpha_i}$. When $s \in F_{\alpha_i}$ (recall that states in F_{α_i} have no outgoing transitions) $\mathcal{A}$ spawns a copy (in a state in P_2) that checks that the suffix is in β_i and continues to simulate $\mathcal{U}_{\gamma_i}$. A state in P_1 marked by the transition $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$ and a state $s \in Q_{\gamma_i}$ participates in the simulation of a run of U_{γ_i}. When $s = q_o^{\gamma_i}$ (recall that the initial state $q^0_{\gamma_i}$ has no incoming transitions) the state is an action state, and $\mathcal{A}$ consults δ and T in order to impose new restrictions on $\langle V^*, \tau_V \rangle$.[8]
- In order to define $\eta : P \times \Sigma \to \mathcal{B}^+(ext(V) \times P)$, we first define the function $apply_T : \Delta \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma) \to \mathcal{B}^+(ext(V) \times P)$. Intuitively, $apply_T$ transforms atoms participating in δ to a formula that describes the requirements on G_R when the rewrite rules in T are applied to words from V^*. For $c \in \Delta$, $w \in W$, $q \in Q$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T$, and $s = q^0_{\gamma_i}$ we define

$$apply_T(c, w, q, t_i, s) = \begin{cases} \langle \varepsilon, (\exists, w, q, t_i, s) \rangle & \text{if } c = \varepsilon \\ \bigwedge_{t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle \in T} (\varepsilon, (\forall, w, q', t_{i'}, q^0_{\alpha_{i'}})) & \text{if } c = \Box \\ \bigvee_{t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle \in T} (\varepsilon, (\exists, w, q', t_{i'}, q^0_{\gamma_{i'}})) & \text{if } c = \Diamond \end{cases}$$

  In order to understand the function $apply_T$, consider the case $c = \Box$. When $\mathcal{S}$ reads the configuration (q, x) of the input graph, fulfilling the atom $\Box w$ requires $\mathcal{S}$ to send copies in state w to all the successors of (q, x). The automaton $\mathcal{A}$ then sends copies that check whether all the configurations (q', y') with $\rho_R((q, x), (q', y'))$ are accepted by $\mathcal{S}$ with initial state w.

  For a formula $\theta \in \mathcal{B}^+(\Delta \times W)$, the formula $apply_T(\theta, q, t_i, s) \in \mathcal{B}^+(ext(V) \times P)$ is obtained from θ by replacing an atom $\langle c, w \rangle$ by the atom $apply_T(c, w, q, t_i, s)$. We can now define η for all $w \in W$, $q \in Q$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T$, $s \in Q_{\alpha_i} \cup Q_{\gamma_i}$, and $A \in V \cup \{\bot\}$ as follows.

[8] Note that a straightforward representation of P results in $O(|W| \cdot |Q| \cdot |T| \cdot \|T\|)$ states. Since, however, the states of the automata for the regular expressions are disjoint, we can assume that the rewrite rule in T that each automaton corresponds to is uniquely defined from it.

- $\eta((\exists, w, q, t_i, s), A) =$

$$\begin{cases} apply_T(\delta(w, L(q, A)), q, t_i, s) & \text{if } s = q^0_{\gamma_i} \\ \bigvee_{B \in V} \bigvee_{s \in \eta_{\gamma_i}(s', B)} (B, (\exists, w, q, t_i, s')) & \text{if } s \in Q_{\gamma_i} \setminus \{q^0_{\gamma_i}\} \\ \bigvee_{s' \in \eta_{\alpha_i}(s, A)} (\uparrow, (\exists, w, q, t_i, s')) & \text{if } s \in Q_{\alpha_i} \setminus F_{\alpha_i} \\ (\varepsilon, (\exists, t_i, q^0_{\beta_i})) \wedge \left(\bigvee_{s' \in F_{\gamma_i}} (\varepsilon, (\exists, w, q, t_i, s')) \right) & \text{if } s \in F_{\alpha_i} \end{cases}$$

- $\eta((\forall, w, q, t_i, s), A) =$

$$\begin{cases} apply_T(\delta(w, L(q, A)), q, t_i, s) & \text{if } s = q^0_{\gamma_i} \\ \text{if } \bigwedge_{B \in V} \bigwedge_{s \in \eta_{\gamma_i}(s', B)} (B, (\forall, w, q, t_i, s')) & \text{if } s \in Q_{\gamma_i} \setminus \{q^0_{\gamma_i}\} \\ \text{if } \bigwedge_{s' \in \eta_{\alpha_i}(s, A)} (\uparrow, (\forall, w, q, t_i, s')) & \text{if } s \in Q_{\alpha_i} \setminus F_{\alpha_i} \\ (\varepsilon, (\forall, t_i, q^0_{\beta_i})) \vee \left(\bigwedge_{s' \in F_{\gamma_i}} (\varepsilon, (\forall, w, q, t_i, s')) \right) & \text{if } t \in F_{\alpha_i} \end{cases}$$

Thus, when $s \in Q_\alpha$ the 2APT $\mathcal{A}$ either chooses a successor s' of s and goes up the tree or in case s is an accepting state of $\mathcal{U}_{\alpha_i}$, it spawns a copy that checks that the suffix is in β_i and moves to a final state of $\mathcal{U}_{\gamma_i}$.

When $s \in Q_\gamma$ the 2APT $\mathcal{A}$ either chooses a direction B and chooses a predecessor s' of s or in case that $s = q^0_{\gamma_i}$ is the initial state of $\mathcal{U}_{\gamma_i}$, the automaton $\mathcal{A}$ uses the transition δ to impose new restrictions on $\langle V^*, \tau_V \rangle$.
We define η for all $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$, $s \in Q_{\beta_i}$, and $A \in V \cup \{\bot\}$ as follows.

$$\eta((\exists, t_i, s), A) = \begin{cases} \bigvee_{s' \in \eta_{\beta_i}(s, A)} (\uparrow, (\exists, t_i, s')) & \text{if } A \neq \bot \\ \mathbf{true} & \text{if } s \in F_{\beta_i} \text{ and } A = \bot \\ \mathbf{false} & \text{if } s \notin F_{\beta_i} \text{ and } A = \bot \end{cases}$$

$$\eta((\forall, t_i, s), A) = \begin{cases} \bigwedge_{s' \in \eta_{\beta_i}(s, A)} (\uparrow, (\forall, t_i, s')) & \text{if } A \neq \bot \\ \mathbf{false} & \text{if } s \in F_{\beta_i} \text{ and } A = \bot \\ \mathbf{true} & \text{if } s \notin F_{\beta_i} \text{ and } A = \bot \end{cases}$$

If $s \in Q_\beta$, then in existential mode, the automaton $\mathcal{A}$ makes sure that the suffix is in β and in universal mode it makes sure that the suffix is not in β.

– $p_0 = \langle \exists, w_0, q_0, t, x_0 \rangle$. Thus, in its initial state $\mathcal{A}$ starts a simulation (backward) of the automaton that accepts the unique word x_0. It follows that $\mathcal{A}$ checks that G_R with initial configuration (q_0, x_0) is accepted by $\mathcal{S}$ with initial state w_0.
– Let $F_\gamma = \bigcup_{t_i \in T} F_{\gamma_i}$. The acceptance condition α is obtained from F by replacing each set F_i by the set $\{\exists, \forall\} \times F_i \times Q \times T \times F_\gamma$. We add to α a maximal odd set and include all the states in $\{\exists\} \times W \times Q \times T \times (Q_\gamma \setminus F_\gamma)$ in this set. We add to α a maximal even set and include all the states in $\{\forall\} \times W \times Q \times T \times (Q_\gamma \setminus F_\gamma)$ in this set[9]. The states in $\{\exists, \forall\} \times W \times Q \times T \times Q_\alpha$ and P_2 are added to the maximal set (notice that states marked by a state in Q_α appear in finite sequences and states in P_2 appear only in suffixes of finite paths in the run tree).

[9] Note that if the maximal set in F is even then we only add to α a maximal odd set. Dually, if the maximal set in F is odd then we add to α a maximal even set.

Thus, in a path that visits infinitely many action states, the action states define it as accepting or not accepting. A path that visits finitely many action states is either finite or ends in an infinite sequence of Q_γ labeled states. If these states are existential, then the path is rejecting. If these states are universal, then the path is accepting.

We show that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \models \mathcal{S}$. Assume that $\mathcal{A}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $\langle T, r \rangle$ of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subtree of nodes labeled by action states. Denote this tree by $\langle T', r' \rangle$. By the definition of δ, the tree $\langle T', r' \rangle$ is a valid run tree of $\mathcal{S}$ on G_R. Consider an infinite path π' in $\langle T', r' \rangle$. The labels of nodes in π' identify a unique path π in $\langle T, r \rangle$. As π' is infinite, it follows that π visits infinitely many action states. As all navigation states are added to the maximal ranks the minimal rank visited along π must be equal to the minimal rank visited along π'. Hence, $\langle T', r' \rangle$ is accepting and $\mathcal{S}$ accepts G_R.

Assume now that $G_R \models \mathcal{S}$. Then, there exists an accepting run tree $\langle T', r' \rangle$ of $\mathcal{S}$ on G_R. The tree $\langle T', r' \rangle$ serves as the action state skeleton to an accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$. A node $z \in T'$ labeled by $((q, x), w)$ corresponds to a copy of $\mathcal{A}$ in state (d, w, q, t, s) reading node x of $\langle V^*, \tau_V \rangle$ for some $d \in \{\exists, \forall\}$, $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T$ and $s = q^0_{\gamma_i}$. In order to extend this skeleton into a valid and accepting run tree of $\mathcal{A}$ on $\langle V^*, \tau_V \rangle$ we have to complete the runs of the automata for the different regular expressions appearing in T. □

The constructions described in Theorems 6 and 7 reduce the model-checking problem to the membership problem of $\langle V^*, \tau_V \rangle$ in the language of a 2APT. By Theorem 3, we then have the following.

Theorem 8. *The model-checking problem for a pushdown or a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$ and k is the index of $\mathcal{S}$.*

Together with Theorem 4, we can conclude with an EXPTIME bound also for the model-checking problem of μ-calculus formulas matching the lower bound in [Wal96]. Note that the fact the same complexity bound holds for both pushdown and prefix-recognizable rewrite systems stems from the different definition of $\|T\|$ in the two cases.

4 Path Automata on Trees

We would like to enhance the approach developed in Section 3 to linear time properties. The solution to μ-calculus model checking is exponential in both the system and the specification and it is EXPTIME-complete [Wal96]. On the other hand, model-checking linear-time specifications is polynomial in the system [BEM97]. As we discuss below, both the emptiness and membership problems for 2APT are EXPTIME-complete. While 2APT can reason about many computation paths simultaneously, in linear-time model-checking we need to reason about a single path that does not satisfy a specification. It follows, that the extra power of 2APT comes at a price we cannot pay. In this section we introduce *path automata* and study them. In Section 5 we show that path automata give us the necessary tool in order to reason about linear specifications.

Path automata resemble *tree walking automata*. These are automata that read finite trees and expect the nodes of the tree to be labeled by the direction and by the set of successors of the node. Tree walking automata are used in XML queries. We refer the reader to [EHvB99, Nev02].

4.1 Definition

Path automata on trees are a hybrid of nondeterministic word automata and nondeterministic tree automata: they run on trees but have linear runs. Here we describe *two-way* nondeterministic Büchi path automata.

A *two-way nondeterministic Büchi path automaton* (2NBP, for short) on Σ-labeled Υ-trees is in fact a 2ABT whose transitions are restricted to disjunctions. Formally, $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$, where Σ, P, p_0, and F are as in an NBW, and $\delta : P \times \Sigma \to 2^{(ext(\Upsilon) \times P)}$ is the transition function. A path automaton that is in state p and reads the node $x \in T$ chooses a pair $(d, p') \in \delta(p, \tau(x))$, and then follows direction d and moves to state p'. It follows that a *run* of a 2NBP $\mathcal{P}$ on a labeled tree $\langle \Upsilon^*, \tau \rangle$ is a sequence of pairs $r = (x_0, p_0), (x_1, p_1), \ldots$ where for all $i \geq 0$, $x_i \in \Upsilon^*$ is a node of the tree and $p_i \in P$ is a state. The pair (x, p) describes a copy of the automaton that reads the node x of Υ^* and is in the state p. Note that many pairs in r may correspond to the same node of Υ^*; Thus, $\mathcal{S}$ may visit a node several times. The run has to satisfy the transition function. Formally, $(x_0, p_0) = (\varepsilon, q_0)$ and for all $i \geq 0$ there is $d \in ext(\Upsilon)$ such that $(d, p_{i+1}) \in \delta(p_i, \tau(x_i))$ and

- If $\Delta \in \Upsilon$, then $x_{i+1} = \Delta \cdot x_i$.
- If $\Delta = \varepsilon$, then $x_{i+1} = x_i$.
- If $\Delta = \uparrow$, then $x_i = \upsilon \cdot z$, for some $\upsilon \in \Upsilon$ and $z \in \Upsilon^*$, and $x_{i+1} = z$.

Thus, ε-transitions leave the automaton on the same node of the input tree, and $\uparrow$-transitions take it up to the parent node. Note that the automaton cannot go up the root of the input tree, as whenever $d = \uparrow$, we require that $x_i \neq \varepsilon$. A run r is *accepting* if it visits $\Upsilon^* \times F$ infinitely often. An automaton accepts a labeled tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(\mathcal{P})$ the set of all Σ-labeled trees that $\mathcal{P}$ accepts. The automaton $\mathcal{P}$ is *nonempty* iff $\mathcal{L}(\mathcal{P}) \neq \emptyset$. We measure the size of a 2NBP by two parameters, the number of states and the size, $|\delta| = \Sigma_{p \in P} \Sigma_{a \in \Sigma} |\delta(s, a)|$, of the transition function.

Readers familiar with tree automata know that the run of a tree automaton starts in a single copy of the automaton reading the root of the tree, and then the copy splits to the successors of the root and so on, thus the run simultaneously follows many paths in the input tree. In contrast, a path automaton has a single copy at all times. It starts from the root and it always chooses a single direction to go to. In two-way path automata, the direction may be "up", so the automaton can read many paths of the tree, but it cannot read them simultaneously.

The fact that a 2NBP has a single copy influences its expressive power and the complexity of its nonemptiness and membership problems. We now turn to study these issues.

4.2 Expressiveness

One-way nondeterministic path automata can read a single path of the tree, so it is easy to see that they accept exactly all languages $\mathcal{T}$ of trees such that there is an ω-regular language L of words and $\mathcal{T}$ contains exactly all trees that have a path labeled by a word in L. For two-way path automata, the expressive power is less clear, as by going up and down the tree, the automaton can traverse several paths. Still, a path automaton cannot traverse all the nodes of the tree. To see that, we prove that a 2NBP cannot recognize even very simple properties that refer to all the branches of the tree (*universal* properties for short).

Theorem 9. *There are no 2NBP $\mathcal{P}_1$ and $\mathcal{P}_2$ over the alphabet $\{0,1\}$ such that $L(\mathcal{P}_1) = L_1$ and $L(\mathcal{P}_2) = L_2$ where $|\Upsilon| > 1$ and*

- $L_1 = \{\langle \Upsilon^*, \tau\rangle : \tau(x) = 0$ *for all* $x \in T\}$.
- $L_2 = \{\langle \Upsilon^*, \tau\rangle :$ *for every path* $\pi \subseteq T$, *there is* $x \in \pi$ *with* $\tau(x) = 0\}$.

Proof: Suppose that there exists a 2NBP $\mathcal{P}_1$ that accepts L_1. Let $T = \langle \Upsilon^*, \tau\rangle \in L_1$ be some tree accepted by $\mathcal{P}_1$. There exists an accepting run $r = (x_0, p_0), (x_1, p_1), \ldots$ of $\mathcal{P}_1$ on T. It is either the case that r visits some node in Υ^* infinitely often or not.

- Suppose that there exists a node $x \in \Upsilon^*$ visited infinitely often by r. There must exist $i < j$ such that $x_i = x_j = x$, $p_i = p_j$, and there exists $i \leq k < j$ such that $p_k \in F$. Consider the run $r' = (x_0, p_0), \ldots, (x_{i-1}, p_{i-1})\left((x_i, p_i), \ldots, (x_{j-1}, p_{j-1})\right)^\omega$. Clearly, it is a valid and accepting run of $\mathcal{P}_1$ on T. However, r' visits only a finite number of nodes in T. Let $W = \{x_i | x_i$ visited by $r'\}$. It is quite clear that the same run r' is an accepting run of $\mathcal{P}_1$ on the tree $\langle \Upsilon, \tau'\rangle$ such that $\tau'(x) = \tau(x)$ for $x \in W$ and $\tau'(x) = 1$ for $x \notin W$. Clearly, $\langle \Upsilon^*, \tau'\rangle \notin L_1$.
- Suppose that every node $x \in \Upsilon^*$ is visited only a finite number of times. Let (x_i, p_i) be the last visit of r to the root. It must be the case that $x_{i+1} = \upsilon$ for some $\upsilon \in \Upsilon$. Let $\upsilon' \neq \upsilon$ be a different element in Υ. Let $W = \{x_{i'} \in \Upsilon^* \cdot \upsilon' \mid x_{i'}$ visited by $r\}$ be the set of nodes in the subtree of υ' visited by r. Clearly, W is finite and we proceed as above.

The proof for the case of $\mathcal{P}_2$ and L_2 is similar. □

There are, however, universal properties that a 2NBP can recognize. Consider a language $L \subseteq \Sigma^\omega$ of infinite words over the alphabet Σ. A finite word $x \in \Sigma^*$ is a *bad prefix* for L iff for all $y \in \Sigma^\omega$, we have $x \cdot y \notin L$. Thus, a bad prefix is a finite word that cannot be extended to an infinite word in L. A language L is a *safety* language iff every $w \notin L$ has a finite bad prefix. A language $L \subseteq \Sigma^\omega$ is *clopen* if both L and its complement are safety languages, or, equivalently, L corresponds to a set that is both closed and open in Cantor space. It is known that a clopen language is bounded: there is an integer k such that after reading a prefix of length k of a word $w \in \Sigma^\omega$, one can determine whether w is in L [KV01]. A 2NBP can then traverse all the paths of the input tree up to level k (given L, its bound k can be calculated), hence the following theorem.

Theorem 10. *Let $L \subseteq \Sigma^\omega$ be a clopen language. There is a 2NBP $\mathcal{P}$ such that $L(\mathcal{P}) = \{\langle \Upsilon^*, \tau\rangle : \text{for all paths } \pi \subseteq \Upsilon^*, \text{ we have } \tau(\pi) \in L\}$.*

Proof: Let k be the bound of L and $\Upsilon = \{\upsilon_1, \ldots, \upsilon_m\}$. Consider, $w = w_0, \ldots, w_r \in \Upsilon^*$. Let i be the maximal index such that $w_i \neq \upsilon_m$. Let $succ(w)$ be as follows

$$succ(w) = w_0, \ldots w_{i-1}, w'_i, w_{i+1}, \ldots, w_r,$$

where if $w_i = \upsilon_j$ then $w'_i = \upsilon_{j+1}$. That is, if we take $w = (\upsilon_1)^k$ then by using the $succ$ function we pass on all elements in Υ^k according to the lexicographic order (induced by $\upsilon_1 < \upsilon_2 < \ldots < \upsilon_m$). Let $\mathcal{N} = \langle \Sigma, N, \delta, n_0, F\rangle$ be an NBW accepting L. According to [KV01], $\mathcal{N}$ is cycle-free and has a unique accepting sink state. Formally, $\mathcal{N}$ has an accepting state n_{acc} such that for every $\sigma \in \Sigma$ we have $\delta(n_{acc}, \sigma) = \{n_{acc}\}$ and for every run $r = n_0, n_1, \ldots$ and every $i < j$ either $n_i \neq n_j$ or $n_i = n_{acc}$.

We construct a 2NBP that scans all the paths in Υ^k according to the order induced by using $succ$. The 2NBP scans a path and simulates $\mathcal{N}$ on this path. Once our 2NBP ensures that this path is accepted by $\mathcal{N}$ it proceeds to the next path. Consider the following 2NBP $\mathcal{P} = \langle \Sigma, Q, \eta, q_0, \{q_{acc}\}\rangle$ where

- $Q = (\{u, d\} \times \Upsilon^k \times [k] \times N) \cup \{q_{acc}\}$. A state consists of 4 components. The symbols u and d are acronyms for up and $down$. A state marked by d means that the 2NBP is going down the tree while scanning a path. A state marked by u means that the 2NBP is going up towards the root where it starts scanning the next path. The word $w \in \Upsilon^k$ is the current explored path. The number $i \in [k]$ denotes the location in the path w. The state $n \in N$ denotes the current state of the automaton $\mathcal{N}$.
- For every state $q \in Q$ and letter $\sigma \in \Sigma$, the transition function $\eta : Q \times \Sigma \to 2^{ext(\Upsilon) \times Q}$ is defined as follows:

$$\eta((d, w, i, n), \sigma) = \begin{cases} \{(w_{i+1}, (d, w, i{+}1, n')) \mid n' \in \delta(n, \sigma)\} & \text{if } i \neq k \\ \emptyset & \text{if } i = k \text{ and } n \neq n_{acc} \\ \{(\varepsilon, (u, succ(w), i, n))\} & \text{if } i = k, n = n_{acc}, \text{ and } w \neq (\upsilon_m)^k \\ \{(\varepsilon, q_{acc})\} & \text{if } i = k, n = n_{acc}, \text{ and } w = (\upsilon_m)^k \end{cases}$$

$$\eta((u, w, i, n), \sigma) = \begin{cases} \{(\uparrow, (u, w, i-1, n))\} & \text{if } i \neq 0 \\ \{(\varepsilon, (d, w, 0, n_0))\} & \text{if } i = 0 \end{cases}$$

$$\eta(q_{acc}, \sigma) = \{(\varepsilon, q_{acc})\}$$

  Intuitively, in d-states the automaton goes in the direction dictated by w and simulates $\mathcal{N}$ on the labeling of the path w. Once the path w is explored, if the $\mathcal{N}$ component is not in n_{acc} this means the run of $\mathcal{N}$ on w failed and the run is terminated. If the $\mathcal{N}$ component reaches n_{acc} this means that the run of $\mathcal{N}$ on w succeeded and the 2NBP proceeds to a u-state with $succ(w)$. If $succ(w)$ does not exist (i.e., $w = (\upsilon_m)^k$) the 2NBP accepts. In u-states the 2NBP goes up towards the root; when it reaches the root it initiates a run of $\mathcal{N}$ on the word w.
- $q_0 = (d, (\upsilon_1)^k, 0, n_0)$. Thus, in the initial state, $\mathcal{P}$ starts to simulate $\mathcal{N}$ on the first path $(\upsilon_1)^k$.

Let $\mathcal{L} = \{\langle \Upsilon^*, \tau\rangle : \text{ for all paths } \pi \subseteq \Upsilon^*, \text{we have } \tau(\pi) \in L\}$. Consider a tree t in $\mathcal{L}$. We show that t is accepted by $\mathcal{P}$. Consider a path w in t. Let $r_w = n_0 \cdots n_k$ be

the accepting run of $\mathcal{N}$ on the word labeling the path w in t. We use the sequence $(d, w, n_0, 0) \cdots (d, w, n_k, k)$ as part of the run of $\mathcal{P}$ on t. We add the parts $(u, w, k-1, n) \cdots (u, w, 0, n)$ that connect these different sequences and finally add an infinite sequence of q_{acc}.

In the other direction consider a tree t accepted by $\mathcal{P}$. For a path w in t we extract from the accepting run of $\mathcal{P}$ the part that relates to w. By the definition of the transition it follows that if we project this segment on the states of $\mathcal{N}$ we get an accepting run of $\mathcal{N}$ on the word labeling w in t. As w is arbitrary it follows that every path in tree is labeled by a word in L and that $t \in \mathcal{L}$. □

Recently, it was shown that deterministic walking tree automata are less expressive than nondeterministic walking tree automata [BC04] and that nondeterministic walking tree automata do not accept all regular tree languages [BC05]. That is, there exist languages recognized by nondeterministic walking tree automata that cannot be recognized by deterministic walking tree automata and there exist languages accept by deterministic tree automata that cannot be recognized by nondeterministic walking tree automata. Using standard techniques to generalize results about automata over finite objects to automata over infinite objects we can show that 2DBP are less expressive than 2NBP. Similarly, the algorithms described in the next subsection can be modified to handle the respective problems for walking tree automata.

4.3 Decision Problems

Given a 2NBP $\mathcal{S}$, the *emptiness problem* is to determine whether $\mathcal{S}$ accepts some tree, or equivalently whether $\mathcal{L}(\mathcal{S}) = \emptyset$. The *membership problem* of $\mathcal{S}$ and a regular tree $\langle \Upsilon^*, \tau \rangle$ is to determine whether $\mathcal{S}$ accepts $\langle \Upsilon^*, \tau \rangle$, or equivalently $\langle \Upsilon^*, \tau \rangle \in \mathcal{L}(\mathcal{S})$. The fact that 2NBP cannot spawn new copies makes them very similar to word automata. Thus, the membership problem for 2NBP can be reduced to the emptiness problem of εABW over a 1-letter alphabet (cf. [KVW00]). The reduction yields a polynomial time algorithm for solving the membership problem. In contrast, the emptiness problem of 2NBP is EXPTIME-complete.

We show a reduction from the membership problem of 2NBP to the emptiness problem of εABW with a 1-letter alphabet. The reduction is a generalization of a construction that translates 2NBW to εABW [PV03]. The emptiness of εABW with a 1-letter alphabet is solvable in quadratic time and linear space [KVW00]. We show that in our case the membership problem of a 2NBP is solved in cubic time and quadratic space in the size of the original 2NBP. Formally, we have the following.

Theorem 11. *Consider a 2NBP $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$. The membership problem of the regular tree $\langle \Upsilon^*, \tau \rangle$ in the language of $\mathcal{S}$ is solvable in time $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$ and space $O(|P|^2 \cdot \|\tau\|)$.*

Proof: We construct an εABW on 1-letter alphabet $\mathcal{A} = \langle \{a\}, Q, \eta, q_0, \alpha \rangle$ such that $L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$. The εABW $\mathcal{A}$ has $O(|P|^2 \cdot \|\tau\|)$ states and the size of its transition function is $O(|P|^2 \cdot |\delta| \cdot \|\tau\|)$. As $\langle \Upsilon^*, \tau \rangle$ is a regular tree, there exists a

transducer that produces it. In order to construct $\mathcal{A}$ we combine this transducer with a construction that converts 2-way automata to 1-way automata. In [PV03] we show that given a 2NBW we can construct an ϵABW that accepts the same language. The conversion of 2-way movement to 1-way movement relies on the following basic paradigm. We check whether the 2-way automaton accepts the word aw from state s by checking that it can get from state s to state t reading aw and that it accepts aw from state t. In order to check that the 2-way automaton can get from state s to state t reading a suffix aw, the 1-way automaton either guesses that the 2-way automaton gets from s to some state p and from p to t, or that there is a transition from s reading a and going forward to state s', a transition from some state t' reading the first letter of w going backward to t, and that the 2-way automaton can get from s' to t' reading w. We use a similar idea here. Consider a regular tree that is the unwinding of a transducer from state d. The 2NBP accepts this tree from state s if there exists a state t such that the 2NBP reaches from s to t reading the tree and accepts the tree starting from t. In order to get from s to t reading the tree the 2NBP either reaches the root again in state p (i.e., reach from s to p and from p to t) or there is a transition from s reading the label of d and going in direction γ to state s', a transition from some state t' reading the label of the γ successor of d going backward to t, and that the 2-way automaton can get from s' to t' reading the regular tree that is the unwinding of the transducer from state d'.

Let $\mathcal{D}_\tau = \langle \Upsilon, \Sigma, D_\tau, \rho_\tau, d_0^\tau, L_\tau \rangle$ be the transducer that generates the labels of τ. For a word $w \in \Upsilon^*$ we denote by $\rho_\tau(w)$ the unique state that $\mathcal{D}_\tau$ gets to after reading w. We construct the εABW $\mathcal{A} = \langle \{a\}, Q, \eta, q_0, \alpha \rangle$ as follows.

- $Q = (P \cup (P \times P)) \times D_\tau \times \{\bot, \top\}$. States in $P \times D_\tau \times \{\bot, \top\}$, which hold a single state from P, are called *singleton states*. Similarly, we call states in $P \times P \times D_\tau \times \{\bot, \top\}$ *pair states*.
- $q_0 = (p_0, d_0^\tau, \bot)$.
- $\alpha = (F \times D_\tau \times \{\bot\}) \cup (P \times D_\tau \times \{\top\})$.

In order to define the transition function we have the following definitions. Two functions $f_\alpha : P \times P \to \{\bot, \top\}$ where $\alpha \in \{\bot, \top\}$, and for every state $p \in P$ and alphabet letter $\sigma \in \Sigma$ the set C_p^σ is the set of states from which p is reachable by a sequence of ϵ-transitions reading letter σ and one final $\uparrow$-transition reading σ. Formally,

$$f_\bot(p, p') = \bot.$$

$$f_\top(p, p') = \begin{cases} \bot & \text{if } p \in F \text{ or } p' \in F \\ \top & \text{otherwise.} \end{cases}$$

$$C_p^\sigma = \left\{ p' \;\middle|\; \begin{array}{l} \exists t_0, t_1, \ldots, t_n \in P^+ \text{ such that } t_0 = p',\ t_n = p, \\ (\epsilon, t_i) \in \delta(t_{i-1}, \sigma) \text{ for all } 0 < i < n, \text{ and } (\uparrow, p_n) \in \delta(p_{n-1}, \sigma) \end{array} \right\}.$$

Now η is defined for every state in Q as follows (recall that $\mathcal{A}$ is a word automaton, hence we use directions 0 and 1 in the definition of η, as $\Sigma = \{a\}$, we omit the letter a from the definition of η).

$$\eta(p,d,\alpha) = \bigvee_{p'\in P}\bigvee_{\beta\in\{\bot,\top\}} (0,(p,p',d,\beta)) \wedge (0,(p',d,\beta)) \quad \vee$$
$$\bigvee_{\upsilon\in\Upsilon}\bigvee_{(\upsilon,p')\in\delta(p,L_\tau(d))} (1,(p',\rho_\tau(d,\upsilon),\bot)) \quad \vee$$
$$\bigvee_{\langle\epsilon,p'\rangle\in\delta(p,L_\tau(d))} (0,(p',d,\bot))$$

$$\eta(p_1,p_2,d,\alpha) = \bigvee_{\langle\epsilon,p'\rangle\in\delta(p_1,L_\tau(d))} (0,(p',p_2,d,f_\alpha(p',p_2))) \quad \vee$$
$$\bigvee_{p'\in P}\bigvee_{\beta_1+\beta_2=\alpha} (0,(p_1,p',d,f_{\beta_1}(p_1,p')) \wedge (0,(p',p_2,d,f_{\beta_2}(p',p_2))) \vee$$
$$\bigvee_{\upsilon\in\Upsilon}\bigvee_{\langle\upsilon,p'\rangle\in\delta(p_1,L_\tau(d))}\bigvee_{p''\in C_{p_2}^{L_\tau(d)}} (1,(p',p'',\rho_\tau(d,\upsilon),f_\alpha(p',p'')))$$

Finally, we replace every state of the form $\{(p,p,d,\alpha) \mid \text{either } p\in P \text{ and } \alpha=\bot \text{ or } p\in F \text{ and } \alpha=\top\}$ by $\mathbf{true}$.

Claim. $L(\mathcal{A})\neq\emptyset$ iff $\langle\Upsilon^*,\tau\rangle\in L(\mathcal{P})$.

The proof is very similar to the proof in [PV03] and is described in detail in Appendix A.

The emptiness of an εABW can be determined in linear space [EL86]. For an εABW $\mathcal{A}$ with 1-letter alphabet, we have the following.

Theorem 12. [VW86b] *Given an εABW over 1-letter alphabet $\mathcal{A}=\langle\{a\},Q,\eta,q_0,\alpha\rangle$ we can check whether $L(\mathcal{A})$ is empty in time $O(|Q|\cdot|\eta|)$ and space $O(|Q|)$.*

Vardi and Wolper give an algorithm that solves the emptiness problem of an ABW over 1-letter alphabet [VW86b]. We note that emptiness of εABW over 1-letter alphabet can be reduced to that of an ABW over 1-letter alphabet by replacing every ϵ-transition by a transition that advances to the next letter. As the input word is infinite, there is no difference between advancing and not advancing.

The automaton $\mathcal{A}$ constructed above has a special structure. The transition of $\mathcal{A}$ from states of the form $P\times P\times D_\tau\times\{\bot,\top\}$ includes only states of the same form. In addition, all these states are not accepting. This suggests that if in the emptiness algorithm we handle these states first, the quadratic part of the algorithm can be applied only to the states of the form $P\times D_\tau\times\{\bot,\top\}$. Using these ideas, we show in [PV03] that the emptiness of $\mathcal{A}$ can be decided in time $O(|\eta|)$ and space $O(|Q|)$. □

Theorem 13. *The emptiness problem for 2NBP is EXPTIME-complete.*

Proof: The upper bound follows immediately from the exponential time algorithm for the emptiness for 2APT [Var98].

For the lower bound we use the EXPTIME-hard problem of whether a linear space alternating Turing machine accepts the empty tape [CKS81]. We reduce this problem to the emptiness problem of a 2NBP with a polynomial number of states. We start with definitions of alternating linear space Turing machines. An alternating Turing machine

is $M = \langle \Gamma, S_u, S_e, \mapsto, s_0, F_{acc}, F_{rej} \rangle$, where the four sets of states S_u, S_e, F_{acc}, and F_{rej} are disjoint, and contain the universal, the existential, the accepting, and the rejecting states, respectively. We denote their union (the set of all states) by S. Our model of alternation prescribes that $\mapsto \subseteq S \times \Gamma \times S \times \Gamma \times \{L, R\}$ has a binary branching degree. When a universal or an existential state of M branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(s, a) \mapsto^l (s_l, b_l, \Delta_l)$ and $(s, a) \mapsto^r (s_r, b_r, \Delta_r)$ to indicate that when M is in state $s \in S_u \cup S_e$ reading input symbol a, it branches to the left with (s_l, b_l, Δ_l) and to the right with (s_r, b_r, Δ_r). (Note that the directions left and right here have nothing to do with the movement direction of the head; these are determined by Δ_l and Δ_r.)

Recall that we consider here alternating linear-space Turing machines. Let $f : \mathbb{N} \to \mathbb{N}$ be the linear function such that M uses $f(n)$ cells in its working tape in order to process an input of length n. We encode a configuration of M by a string in $\{\sharp\} \cdot \Gamma^i \cdots (S \times \Gamma) \cdot \Gamma^{f(n)-i-1}$. That is, a configuration starts with the symbol $\sharp$, all its other letters are in Γ, except for one letter in $S \times \Gamma$. The meaning of such a configuration is that the j^{th} cell in the configuration, for $1 \leq j \leq f(n)$, is labeled γ_j, the reading head points at cell $i{+}1$, and M is in state s. For example, the initial configuration of M is $\sharp \cdot (s_0, b)b \cdots b$ (with $f(n){-}1$ occurrences of b's) where b stands for an empty cell. A configuration c' is a successor of configuration c if c' is a left or right successor of c. We can encode now a computation of M by a tree whose branches describe sequences of configurations of M. The computation is legal if a configuration and its successors satisfy the transition relation.

Note that though M has an existential (thus nondeterministic) mode, there is a single computation tree that describes all the possible choices of M. Each run of M corresponds to a pruning of the computation tree in which all the universal configurations have both successors and all the existential configurations have at least one successor. The run is accepting if all the branches in the pruned tree reach an accepting configuration.

We encode the full run tree of M into the labeling of the full infinite binary tree. We construct a 2NBP that reads an input tree and checks that it is indeed a correct encoding of the run tree of M. In case the input tree is a correct encoding, the 2NBP further checks that there exists a subtree that represents an accepting computation of M.

We now explain how the labeling of the full binary tree is used to encode the run tree of M. Let $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ be a configuration and $\sharp \cdot \sigma_1^l \cdots \sigma_{f_n}^l$ be its left successor. We set σ_0 and σ_0^l to $\sharp$. Formally, let $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma)$ and let $next_l : V^3 \to V$ where $next_l(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$ denotes our expectation for σ_i^l. We define $next_l(\sigma, \sharp, \sigma') = \sharp$ and

$$next_l(\sigma, \sigma', \sigma'') = \begin{cases} \sigma' & \text{if } \{\sigma, \sigma', \sigma''\} \subseteq \{\sharp\} \cup \Gamma \\ \sigma' & \text{if } \sigma'' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', R) \\ (s', \sigma') & \text{if } \sigma'' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', L) \\ \sigma' & \text{if } \sigma = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', L) \\ (s', \sigma') & \text{if } \sigma = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', R) \\ \gamma' & \text{if } \sigma' = (s, \gamma) \text{ and } (s, \gamma) \to^l (s', \gamma', \alpha) \end{cases}$$

The expectation $next_r : V^3 \to V$ for the letters in the right successor is defined analogously.

The run tree of M is encoded in the full binary tree as follows. Every configuration is encoded by a string of length $f(n)+1$ in $\{\sharp\} \times \Gamma^* \times (S \times \Gamma) \times \Gamma^*$. The encoding of a configuration $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ starts in a node x that is labeled by $\sharp$. The 0 successor of x, namely $0 \cdot x$, is labeled by σ_1 and so on until $0^{f(n)} \cdot x$ that is labeled by $\sigma_{f(n)}$. The configuration $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)}$ has its right successor $\sharp \cdot \sigma_1^r \cdots \sigma_{f(n)}^r$ and its left successor $\sharp \cdot \sigma_1^l \cdots \sigma_{f(n)}^l$. The encoding of $\sharp \cdot \sigma_1^r \cdots \sigma_{f(n)}^r$ starts in $1 \cdot 0^{f(n)} \cdot x$ (that is labeled by $\sharp$) and the encoding of $\sharp \cdot \sigma_1^l \cdots \sigma_{f(n)}^l$ starts in $0 \cdot 0^{f(n)} \cdot x$ (that is labeled by $\sharp$). We also demand that every node be labeled by its direction. This way we can infer from the label of the node labeled by $\sharp$ whether its the first letter in the left successor or the first letter in the right successor. For example, the root of the tree is labeled by $\langle \perp, \sharp \rangle$, the node 0 is labeled by $\langle 0, (s_0, b) \rangle$ and for every $1 < i \leq f(n)$ the node 0^i is labeled by $\langle 0, b \rangle$ (here b stands for the blank symbol). We do not care about the labels of other nodes. Thus, the labeling of 'most' nodes in the tree does not interest us.

The 2NBP reads an infinite binary tree. All trees whose labeling does not conform to the above are rejected. A tree whose labeling is a correct encoding of the run tree of M is accepted only if there exists an accepting pruning tree. Thus, the language of the 2NBP is not empty iff the Turing machine M accepts the empty tape.

In order to check that the input tree is a correct encoding of the run tree of M, the 2NBP has to check that every configuration is followed by its successor configurations. When checking location i in configuration a, the NBW memorizes the three letters around location i ($i-1$, i, $i+1$), it goes $f(n)$ steps forward to the next configuration and checks that it finds there the correct $next_l$ or $next_r$ successor. Then the 2NBP returns to location $i+1$ in configuration a and updates its three letters memory to check consistency of this next location.

We now explain the construction in more detail. The 2NBP has two main modes of operation. In *forward* mode, the 2NBP checks that the next (right or left) configuration is indeed the correct successor. Then it moves to check the next configuration. If it reaches an accepting configuration, this means that the currently scanned pruning tree may still be accepting. Then it moves to *backward* mode and remembers that it should check other universal branches. If it reaches a rejecting configuration, this means that the currently scanned pruning tree is rejecting. The 2NBP has to move to the next pruning tree. It moves to *backward* mode and remembers that it has to check other existential branches. In *backward universal* mode, the 2NBP goes backward until it gets to a universal configuration and the only configuration to be visited below it is the left successor. Then the 2NBP goes back to forward mode but remembers that the next configuration to visit is the right successor. If the root is reached in backward universal mode then there are no more branches to check, the pruning tree is accepting and the 2NBP accepts. In *backward existential* mode, the 2NBP goes backward until it gets to an existential configuration and the only configuration to be visited below it is the left successor. Then the 2NBP goes to forward mode but remembers that the next configuration to visit is the right successor. If the root is reached in backward existential mode then there are no more pruning trees to check and the 2NBP rejects.

The full formal construction is given in Appendix B. □

We note that the membership problem for 2-way alternating Büchi automata on trees is EXPTIME-complete. Indeed, CTL model checking of pushdown systems, proven to be EXPTIME-hard in [Wal00], can be reduced to the membership problem of a regular tree in the language of a 2ABT. Given a pushdown system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a CTL formula φ, we can construct a graph automaton $\mathcal{S}$ accepting the set of graphs that satisfy φ [KVW00]. This graph automaton is linear in φ and it uses the Büchi acceptance condition. Using the construction in Section 3, CTL model checking then reduces to the membership problem of $\langle V^*, \tau_V \rangle$ in the language of a 2ABT. EXPTIME-hardness follows. Thus, path automata capture the computational difference between linear and branching specifications.

5 Model-Checking Linear-Time Properties

In this section we solve the LTL model-checking problem by a reduction to the membership problem of 2NBP. We start by demonstrating our technique on LTL model checking of pushdown systems. Then we show how to extend it to prefix-recognizable systems. For an LTL formula φ, we construct a 2NBP that navigates through the full infinite V-tree and simulates a computation of the rewrite system that does not satisfy φ. Thus, our 2NBP accepts the V-tree iff the rewrite system does not satisfy the specification. Then, we use the results in Section 4: we check whether the given V-tree is in the language of the 2NBP and conclude whether the system satisfies the property. For pushdown systems we show that the tree $\langle V^*, \tau_V \rangle$ gives sufficient information in order to let the 2NBP simulate transitions. For prefix-recognizable systems the label is more complex and reflects the membership of a node x in the regular expressions that are used in the transition rules and the regular labeling.

5.1 Pushdown Graphs

Recall that in order to apply a rewrite rule of a pushdown system from configuration (q, x), it is sufficient to know q and the first letter of x. We construct a 2NBP $\mathcal{P}$ that reads $\langle V^*, \tau_V \rangle$. The state space of $\mathcal{P}$ contains a component that memorizes the current state of the rewrite system. The location of the reading head in $\langle V^*, \tau_V \rangle$ represents the store of the current configuration. Thus, in order to know which rewrite rules can be applied, $\mathcal{P}$ consults its current state and the label of the node it reads (note that $dir(x)$ is the first letter of x). Formally, we have the following.

Theorem 14. *Given a pushdown system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$ and an LTL formula φ, there is a 2NBP $\mathcal{P}$ on V-trees such that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff $G_R \not\models \varphi$. The automaton $\mathcal{P}$ has $|Q| \cdot \|T\| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Proof: According to Theorem 5, there is an NBW $\mathcal{S}_{\neg\varphi} = \langle 2^{AP}, W, \eta_{\neg\varphi}, w_0, F \rangle$ such that $L(\mathcal{S}_{\neg\varphi}) = (2^{AP})^\omega \setminus L(\varphi)$. The 2NBP $\mathcal{P}$ tries to find a trace in G_R that satisfies $\neg\varphi$. The 2NBP $\mathcal{P}$ runs $\mathcal{S}_{\neg\varphi}$ on a guessed (q_0, x_0)-computation in R. Thus, $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff there exists an (q_0, x_0)-trace in G_R accepted by $\mathcal{S}_{\neg\varphi}$. Such a (q_0, x_0)-trace does not satisfy φ, and it exists iff $R \not\models \varphi$. We define $\mathcal{P} = \langle \{V \cup \{\bot\}, P, \delta, p_0, \alpha \rangle$, where

- $P = W \times Q \times heads(T)$, where $heads(T) \subseteq V^*$ is the set of all prefixes of words $x \in V^*$ for which there are states $q, q' \in Q$ and $A \in V$ such that $\langle q, A, x, q' \rangle \in T$. Intuitively, when $\mathcal{P}$ visits a node $x \in V^*$ in state $\langle w, q, y \rangle$, it checks that R with initial configuration $(q, y \cdot x)$ is accepted by $\mathcal{S}^w_{\neg\varphi}$. In particular, when $y = \varepsilon$, then R with initial configuration (q, x) needs to be accepted by $\mathcal{S}^w_{\neg\varphi}$. States of the form $\langle w, q, \varepsilon \rangle$ are called *action states*. From these states $\mathcal{S}$ consults $\eta_{\neg\varphi}$ and T in order to impose new requirements on $\langle V^*, \tau_V \rangle$. States of the form $\langle w, q, y \rangle$, for $y \in V^+$, are called *navigation states*. From these states $\mathcal{P}$ only navigates downwards y to reach new action states.
- The transition function δ is defined for every state in $\langle w, q, x \rangle \in S \times Q \times heads(T)$ and letter in $A \in V$ as follows.
  - $\delta(\langle w, q, \epsilon \rangle, A) =$

    $$\{(\uparrow, \langle w', q', y \rangle) \ : \ w' \in \eta_{\neg\varphi}(w, L(q, A)) \text{ and } \langle q, A, y, q' \rangle \in T\}.$$

  - $\delta(\langle w, q, y \cdot B \rangle, A) = \{(B, \langle w, q, y \rangle)\}$.

  Thus, in action states, $\mathcal{P}$ reads the direction of the current node and applies the rewrite rules of R in order to impose new requirements according to $\eta_{\neg\varphi}$. In navigation states, $\mathcal{P}$ needs to go downwards $y \cdot B$, so it continues in direction B.
- $p_0 = \langle w_0, q_0, x_0 \rangle$. Thus, in its initial state $\mathcal{P}$ checks that R with initial configuration (q_0, x_0) contains a trace that is accepted by $\mathcal{S}$ with initial state w_0.
- $\alpha = \{\langle w, q, \epsilon \rangle \ : \ w \in F \text{ and } \ q \in Q\}$. Note that only action states can be accepting states of $\mathcal{P}$.

We show that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$ iff $R \not\models \varphi$. Assume first that $\mathcal{P}$ accepts $\langle V^*, \tau_V \rangle$. Then, there exists an accepting run $(p_0, x_0), (p_1, x_1), \ldots$ of $\mathcal{P}$ on $\langle V^*, \tau_V \rangle$. Extract from this run the subsequence of action states $(p_{i_1}, x_{i_1}), (p_{i_2}, x_{i_2}), \ldots$. As the run is accepting and only action states are accepting states, we know that this subsequence is infinite. Let $p_{i_j} = \langle w_{i_j}, q_{i_j}, \varepsilon \rangle$. By the definition of δ, the sequence $(q_{i_1}, x_{i_1}), (q_{i_2}, x_{i_2}), \ldots$ corresponds to an infinite path in the graph G_R. Also, by the definition of α, the run $w_{i_1}, w_{i_2}, \ldots$ is an accepting run of $\mathcal{S}_{\neg\varphi}$ on the trace of this path. Hence, G_R contains a trace that is accepted by $\mathcal{S}_{\neg\varphi}$, thus $R \not\models \varphi$.

Assume now that $R \not\models \varphi$. Then, there exists a path $(q_0, x_0), (q_1, x_1), \ldots$ in G_R whose trace does not satisfy φ. There exists an accepting run $w_0, w_1, \ldots$ of $\mathcal{S}_{\neg\varphi}$ on this trace. The combination of the two sequences serves as the subsequence of action states in an accepting run of $\mathcal{P}$. It is not hard to extend this subsequence to an accepting run of $\mathcal{P}$ on $\langle V^*, \tau_V \rangle$. □

5.2 Prefix-Recognizable Graphs

We now turn to consider prefix-recognizable systems. Again a configuration of a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ consists of a state in Q and a word in V^*. So, the store content is still a node in the tree V^*. However, in order to apply a rewrite rule it is not enough to know the direction of the node. Recall that in order to represent the configuration $(q, x) \in Q \times V^*$, our 2NBP memorizes the state q as part of its state space and it reads the node $x \in V^*$. In order to apply the rewrite rule $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$, the 2NBP has to go up the tree along a word $y \in \alpha_i$. Then, if

$x = y \cdot z$, it has to check that $z \in \beta_i$, and finally guess a word $y' \in \gamma_i$ and go downwards y' to $y' \cdot z$. Finding a prefix y of x such that $y \in \alpha_i$, and a new word $y' \in \gamma_i$ is not hard: the 2NBP can emulate the run of the automaton $\mathcal{U}_{\alpha_i}$ while going up the tree and the run of the automaton $\mathcal{U}_{\gamma_i}$ backwards while going down the guessed y'. How can the 2NBP know that $z \in \beta_i$? In Subsection 3.2 we allowed the 2APT to branch to two states. The first, checking that $z \in \beta_i$ and the second, guessing y'. With 2NBP this is impossible and we provide a different solution. Instead of labeling each node $x \in V^*$ only by its direction, we can label it also by the regular expressions β for which $x \in \beta$. Thus, when the 2NBP runs $\mathcal{U}_{\alpha_i}$ up the tree, it can tell, in every node it visits, whether z is a member of β_i or not. If $z \in \beta_i$, the 2NBP may guess that time has come to guess a word in γ_i and run $\mathcal{U}_{\gamma_i}$ down the guessed word.

Thus, in the case of prefix-recognizable systems, the nodes of the tree whose membership is checked are labeled by both their directions and information about the regular expressions β. Let $\{\beta_1, \ldots, \beta_n\}$ be the set of regular expressions β_i such that there is a rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle \in T$. Let $\mathcal{D}_{\beta_i} = \langle V, D_{\beta_i}, \eta_{\beta_i}, q^0_{\beta_i}, F_{\beta_i} \rangle$ be the deterministic automaton for the reverse of the language of β_i. For a word $x \in V^*$, we denote by $\eta_{\beta_i}(x)$ the unique state that $\mathcal{D}_{\beta_i}$ reaches after reading the word x^R. Let $\Sigma = V \times \Pi_{1 \leq i \leq n} D_{\beta_i}$. For a letter $\sigma \in \Sigma$, let $\sigma[i]$, for $i \in \{0, \ldots n\}$, denote the i-th element in σ (that is, $\sigma[0] \in V$ and $\sigma[i] \in D_{\beta_i}$ for $i > 0$). Let $\langle V^*, \tau_\beta \rangle$ denote the Σ-labeled V-tree such that $\tau_\beta(\epsilon) = \langle \bot, q^0_{\beta_1}, \ldots, q^0_{\beta_n} \rangle$, and for every node $A \cdot x \in V^+$, we have $\tau_\beta(A \cdot x) = \langle A, \eta_{\beta_1}(A \cdot x), \ldots, \eta_{\beta_n}(A \cdot x) \rangle$. Thus, every node x is labeled by $dir(x)$ and the vector of states that each of the deterministic automata reach after reading x. Note that $\tau_\beta(x)[i] \in F_{\beta_i}$ iff x is in the language of β_i. Note also that $\langle V^*, \tau_\beta \rangle$ is a regular tree whose size is exponential in the sum of the lengths of the regular expressions $\beta_1, \ldots, \beta_n$.

Theorem 15. *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and an LTL formula φ, there is a 2NBP $\mathcal{P}$ such that $\mathcal{P}$ accepts $\langle V^*, \tau_\beta \rangle$ iff $R \not\models \varphi$. The automaton $\mathcal{P}$ has $|Q| \cdot (|Q_\alpha| + |Q_\gamma|) \cdot |T| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.*

Proof: As before we use the NBW $\mathcal{S}_{\neg\varphi} = \langle 2^{AP}, W, \eta_{\neg\varphi}, w_0, F \rangle$.
We define $\mathcal{P} = \langle \Sigma, P, \delta, p_0 \alpha \rangle$ as follows.

- $\Sigma = V \times \Pi_{i=1}^{n} D_{\beta_i}$.
- $P = \{ \langle w, q, t_i, s \rangle \mid w \in W,\ q \in Q,\ t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle \in T,\ \text{and } s \in Q_{\alpha_i} \cup Q_{\gamma_i} \}$
  Thus, $\mathcal{P}$ holds in its state a state of $\mathcal{S}_{\neg\varphi}$, a state in Q, the current rewrite rule being applied, and the current state in Q_α or Q_γ. A state $\langle w, q, \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle, s \rangle$ is an action state if s is the initial state of $\mathcal{U}_{\gamma_i}$, that is $s = q^0_{\gamma_i}$. In action states, $\mathcal{P}$ chooses a new rewrite rule $t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle$. Then $\mathcal{P}$ updates the $\mathcal{S}_{\neg\varphi}$ component according to the current location in the tree and moves to $q^0_{\alpha_{i'}}$, the initial state of $\mathcal{U}_{\alpha_{i'}}$. Other states are navigation states. If $s \in Q_{\gamma_i}$ is a state in $\mathcal{U}_{\gamma_i}$ (that is not initial), then $\mathcal{P}$ chooses a direction in the tree, a predecessor of the state in Q_{γ_i} reading the chosen direction, and moves in the chosen direction. If $s \in Q_{\alpha_i}$ is a state of $\mathcal{U}_{\alpha_i}$ then $\mathcal{P}$ moves up the tree (towards the root) while updating the state of $\mathcal{U}_{\alpha_i}$. If $s \in F_{\alpha_i}$ is an accepting state of $\mathcal{U}_{\alpha_i}$ and $\tau(x)[i] \in F_{\beta_i}$ marks the current node x as a member of the language of β_i then $\mathcal{P}$ moves to some accepting state $s \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$ (recall that initial states and accepting states have no incoming / outgoing edges respectively).

– The transition function δ is defined for every state in P and letter in $\Sigma = V \times \Pi_{i=1}^{n} D_{\beta_i}$ as follows.
  • If $s \in Q_\alpha$ then

$$\delta(\langle w, q, t_i, s\rangle, \sigma) = \left\{ (\uparrow, \langle w, q, t_i, s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \\ s' \in \eta_{\alpha_i}(s, \sigma[0]) \end{array} \right\} \bigcup \left\{ (\epsilon, \langle w, q, t_i, s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle, \\ s \in F_{\alpha_i},\ s' \in F_{\gamma_i}, \\ \text{and } \sigma[i] \in F_{\beta_i} \end{array} \right\}$$

  • If $s \in Q_\gamma$, then

$$\delta(\langle w, q, t_i, s\rangle, \sigma) = \left\{ (B, \langle w, q, t_i, s'\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \\ s \in \eta_{\gamma_i}(s', B) \text{ and } B \in V \end{array} \right\} \bigcup \left\{ (\epsilon, \langle w', q'', t_{i'}, s_0\rangle) \;\middle|\; \begin{array}{l} t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle, \\ t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q''\rangle, \\ w' \in \eta_{\neg\varphi}(w, L(q, \sigma[0])), \\ s = q^0_{\gamma_i} \text{ and } s_0 = q^0_{\alpha_{i'}} \end{array} \right\}$$

  Thus, when $s \in Q_\alpha$ the 2NBP $\mathcal{P}$ either chooses a successor s' of s and goes up the tree or in case s is the final state of $\mathcal{U}_{\alpha_i}$ and $\sigma[i] \in F_{\beta_i}$ then $\mathcal{P}$ chooses an accepting state $s' \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$.

  When $s \in Q_\gamma$ the 2NBP $\mathcal{P}$ either guesses a direction B and chooses a predecessor s' of s reading B or in case $s = q^0_{\gamma_i}$ is the initial state of $\mathcal{U}_{\gamma_i}$, the automaton $\mathcal{P}$ updates the state of $\mathcal{S}_{\neg\varphi}$, chooses a new rewrite rule $t_{i'} = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q''\rangle$ and moves to $q^0_{\alpha_{i'}}$, the initial state of $\mathcal{U}_{\alpha_{i'}}$.

– $p_0 = \langle w_0, q_0, t, x_0\rangle$, where t is an arbitrary rewrite rule.
  Thus, $\mathcal{P}$ navigates down the tree to the location x_0. There, it chooses a new rewrite rule and updates the state of $\mathcal{S}_{\neg\varphi}$ and the Q component accordingly.
– $\alpha = \{\langle w, q, t_i, s\rangle \mid w \in F,\ q \in Q,\ t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q\rangle,\ \text{and } s = q^0_{\gamma_i}\}$
  Only action states may be accepting. As initial states have no incoming edges, in an accepting run, every navigation stage is finite.

As before we can show that a trace that violates φ and the rewrite rules used to create this trace can be used to produce a run of $\mathcal{P}$ on $\langle V^*, \tau_\beta\rangle$

Similarly, an accepting run of $\mathcal{P}$ on $\langle V^*, \tau_\beta\rangle$ is used to find a trace in G_R that violates φ. □

We can modify the conversion of 2NBP to ϵABW described in Section 4 for this particular problem. Instead of keeping in the state of the ϵABW a component of the direction of the node $A \in V \cup \{\bot\}$ we keep the letter from Σ (that is, the tuple $\langle A, q_1, \ldots, q_n\rangle \in V \times \Pi_{i=1}^{n} D_{\beta_i}$). When we take a move forward in the guessed direction $B \in V$ we update $\langle A, q_1, \ldots, q_n\rangle$ to $\langle B, \eta_{\beta_1}(q_1, B), \ldots, \eta_{\beta_n}(q_n, B)\rangle$. This way, the state space of the resulting ϵABW does not contain $(\Pi_{i=1}^{n} D_{\beta_i})^2$ but only $\Pi_{i=1}^{n} D_{\beta_i}$.

Combining Theorems 14, 15, and 11, we get the following.

Theorem 16. *The model-checking problem for a rewrite system R and an LTL formula φ is solvable in*

- *time* $\|T\|^3 \cdot 2^{O(|\varphi|)}$ *and space* $\|T\|^2 \cdot 2^{O(|\varphi|)}$*, if* R *is a pushdown system.*
- *time* $\|T\|^3 \cdot 2^{O(|\varphi|+|Q_\beta|)}$ *and space* $\|T\|^2 \cdot 2^{O(|\varphi|+|Q_\beta|)}$*, if* R *is a prefix-recognizable system. The problem is EXPTIME-hard in* $|Q_\beta|$ *even for a fixed formula.*

For pushdown systems (the first setting), our complexity coincides with the one in [EHRS00]. In Appendix C, we prove the EXPTIME lower bound in the second setting by a reduction from the membership problem of a linear space alternating Turing machine. Thus, our upper bounds are tight.

6 Relating Regular Labeling with Prefix-Recognizability

In this section we consider systems with regular labeling. We show first how to extend our approach to handle regular labeling. Both for branching-time and linear-time, the way we adapt our algorithms to handle regular labeling is very similar to the way we handle prefix-recognizability. In the branching-time framework the 2APT guesses a label and sends a copy of the automaton for the regular label to the root to check its guess. In the linear-time framework we include in the labels of the regular tree also data regarding the membership in the languages of the regular labeling. Based on these observations we proceed to show that the two questions are intereducible. We describe a reduction from μ-calculus (resp., LTL) model checking with respect to a prefix-recognizable system with simple labeling function to μ-calculus (resp., LTL) model checking with respect to a pushdown system with regular labeling. We also give reductions in the other direction. We note that we cannot just replace one system by another, but we also have to adjust the μ-calculus (resp., LTL) formula.

6.1 Model-Checking Graphs with Regular Labeling

We start by showing how to extend the construction in Subsection 3.2 to include also regular labeling. In order to apply a transition of the graph automaton $\mathcal{S}$, from configuration (q, x) our 2APT $\mathcal{A}$ has to guess a label $\sigma \in \Sigma$, apply the transition of $\mathcal{S}$ reading σ, and send an additional copy to the root that checks that the guess is correct and that indeed $x \in R_{\sigma,q}$. The changes to the construction in Subsection 3.1 are similar.

Theorem 17. *Given a prefix-recognizable rewrite system* $R = \langle \Sigma, V, Q, T, L, q_0, x_0 \rangle$ *where* L *is a regular labeling function and a graph automaton* $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$*, we can construct a 2APT* $\mathcal{A}$ *over* $(V \cup \{\bot\})$*-labeled* V*-trees such that* $\mathcal{A}$ *accepts* $\langle V^*, \tau_V \rangle$ *iff* G_R *satisfies* $\mathcal{S}$*. The automaton* $\mathcal{A}$ *has* $O(|Q| \cdot (\|T\| + \|L\|) \cdot |V|)$ *states, and its index is the index of* $\mathcal{S}$ *plus 1.*

Proof: We take the automaton constructed for the case of prefix-recognizable systems with simple labeling $\mathcal{A} = \langle V \cup \{\bot\}, P, \eta, p_0, \alpha \rangle$ and modify slightly its state set P and its transition η.

- $P = P_1 \cup P_2 \cup P_3$ where $P_1 = \{\exists, \forall\} \times W \times Q \times T \times (Q_\alpha \cup Q_\gamma)$ and $P_2 = \{\exists, \forall\} \times T \times Q_\beta$ are just like in the previous proof and $P_3 = \bigcup_{\sigma \in \Sigma} \bigcup_{q \in Q} Q_{\sigma,q}$ includes all the states of the automata for the regular expressions appearing in L.

– The definition of $apply_T$ does not change and so does the transition of all navigation states. In the transition of action states, we include a disjunction that guesses the correct labeling. For a state $(d, w, q, t_i, s) \in P_1$ such that $t_i = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$ and $s = q^0_{\gamma_i}$ we have

$$\eta((d, w, q, t_i, s), A) = \bigvee_{\sigma \in \Sigma} \Big(q^0_{\sigma,q} \wedge apply_T(\delta(w, \sigma), t_i, s) \Big).$$

For a state $s \in Q_{\sigma,q}$ and letter $A \in V \cup \{\bot\}$ we have

$$\eta(s, A) = \begin{cases} \bigvee_{s' \in \rho_{\sigma,q}(s,A)} (\uparrow, s') & \text{if } A \neq \bot \\ \mathbf{true} & \text{if } A = \bot \text{ and } s \in F_{\sigma,q} \\ \mathbf{false} & \text{if } A = \bot \text{ and } s \notin F_{\sigma,q} \end{cases}$$

□

Theorem 18. *The model-checking problem for a pushdown or a prefix-recognizable rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ with a regular labeling L and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V| + \|L\| \cdot |V|$ and k is the index of $\mathcal{S}$.*

We show how to extend the construction in Subsection 5.2 to include also regular labeling. We add to the label of every node in the tree V^* also the states of the deterministic automata that recognize the reverse of the languages of the regular expressions of the labels. The navigation through the V-tree proceeds as before, and whenever the 2NBP needs to know the label of the current configuration (that is, in action states, when it has to update the state of $\mathcal{S}_{\neg\varphi}$), it consults the labels of the tree.

Formally, let $\{R_1, \ldots, R_n\}$ denote the set of regular expressions R_i such that there exist some state $q \in Q$ and proposition $p \in AP$ with $R_i = R_{p,q}$. Let $\mathcal{D}_{R_i} = \langle V, D_{R_i}, \eta_{R_i}, q^0_{R_i}, F_{R_i} \rangle$ be the deterministic automaton for the reverse of the language of R_i. For a word $x \in V^*$, we denote by $\eta_{R_i}(x)$ the unique state that $\mathcal{D}_{R_i}$ reaches after reading the word x^R. Let $\Sigma = V \times \Pi_{1 \leq i \leq n} D_{R_i}$. For a letter $\sigma \in \Sigma$ let $\sigma[i]$, for $i \in \{0, \ldots, n\}$, denote the i-th element of σ. Let $\langle V^*, \tau_{_L} \rangle$ be the Σ-labeled V-tree such that $\tau_{_L}(\epsilon) = \langle \bot, q^0_{R_1}, \ldots, q^0_{R_n} \rangle$ and for every node $A \cdot x \in V^+$ we have $\tau_{_L}(A \cdot x) = \langle A, \eta_{R_1}(A \cdot x), \ldots, \eta_{R_n}(A \cdot x) \rangle$. The 2NBP $\mathcal{P}$ reads $\langle V^*, \tau_{_L} \rangle$. Note that if the state space of $\mathcal{P}$ indicates that the current state of the rewrite system is q and $\mathcal{P}$ reads the node x, then for every atomic proposition p, we have that $p \in L(q, x)$ iff $\tau_{_L}(x)[i] \in F_{R_i}$, where i is such that $R_i = R_{p,q}$. In action states, $\mathcal{P}$ needs to update the state of $\mathcal{S}_{\neg\varphi}$, which reads the label of the current configuration. Based on its current state and $\tau_{_L}$, the 2NBP $\mathcal{P}$ knows the letter with which $\mathcal{S}_{\neg\varphi}$ proceeds.

If we want to handle a prefix-recognizable system with regular labeling we have to label the nodes of the tree V^* by both the deterministic automata for regular expressions β_i and the deterministic automata for regular expressions $R_{p,q}$. Let $\langle V^*, \tau_{\beta,L} \rangle$ be the composition of $\langle V^*, \tau_\beta \rangle$ with $\langle V^*, \tau_{_L} \rangle$. Note that $\langle V^*, \tau_{_L} \rangle$ and $\langle V^*, \tau_{\beta,L} \rangle$ are regular, with $\|\tau_{_L}\| = 2^{O(\|L\|)}$ and $\|\tau_{\beta,L}\| = 2^{O(|Q_\beta| + \|L\|)}$.

Theorem 19. *Given a prefix-recognizable system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ where L is a regular labeling and an LTL formula φ, there is a 2NBP $\mathcal{S}$ such that $\mathcal{S}$ accepts*

$\langle V^*, \tau_{\beta,L}\rangle$ iff $R \not\models \varphi$. The automaton $\mathcal{S}$ has $|Q| \cdot (|Q_\alpha| + |Q_\gamma|) \cdot \|T\| \cdot 2^{O(|\varphi|)}$ states and the size of its transition function is $\|T\| \cdot 2^{O(|\varphi|)}$.

Note that Theorem 19 differs from Theorem 15 only in the labeled tree whose membership is checked. Combining Theorems 19 and 11, we get the following.

Theorem 20. *The model-checking problem for a prefix-recognizable system R with regular labeling L and an LTL formula φ is solvable in time $\|T\|^3 \cdot 2^{O(|\varphi|+|Q_\beta|+\|L\|)}$ and space $\|T\|^2 \cdot 2^{O(|\varphi|+|Q_\beta|+\|L\|)}$.*

For pushdown systems with regular labeling an alternative algorithm is given in Theorem 1. This, together with the lower bound in [EKS01], implies EXPTIME-hardness in terms of $\|L\|$. Thus, our upper bound is tight.

6.2 Prefix-Recognizable to Regular Labeling

We reduce μ-calculus (resp., LTL) model checking of prefix-recognizable systems to μ-calculus (resp., LTL) model checking of pushdown systems with regular labeling. Given a prefix-recognizable system we describe a pushdown system with regular labeling that is used in both reductions. We then explain how to adjust the μ-calculus or LTL formula.

Theorem 21. *Given a prefix-recognizable system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0\rangle$, a graph automaton $\mathcal{S}$, and an LTL formula φ, there is a pushdown system $R' = \langle 2^{AP'}, V, Q', L', T', q'_0, x_0\rangle$ with a regular labeling function, a graph automaton $\mathcal{S}'$, and an LTL formula φ', such that $R \models \mathcal{S}$ iff $R' \models \mathcal{S}'$ and $R \models \varphi$ iff $R' \models \varphi'$. Furthermore, $|Q'| = |Q| \times |T| \times (|Q_\alpha| + |Q_\gamma|)$, $\|T'\| = O(\|T\|)$, $\|L\| = |Q_\beta|$, $|\mathcal{S}'| = O(|\mathcal{S}|)$, the index of $\mathcal{S}'$ equals the index of $\mathcal{S}$ plus one, and $|\varphi'| = O(|\varphi|)$. The reduction is computable in logarithmic space.*

The idea is to add to the configurations of R labels that would enable the pushdown system to simulate transitions of the prefix-recognizable system. Recall that in order to apply the rewrite rule $\langle q, \alpha, \beta, \gamma, q'\rangle$ from configuration (q, x), the prefix-recognizable system has to find a partition $y \cdot z$ of x such that the prefix y is a word in α and the suffix z is a word in β. It then replaces y by a word $y' \in \gamma$. The pushdown system can remove the prefix y letter by letter, guess whether the remaining suffix z is a word in β, and add y' letter by letter. In order to check the validity of guesses, the system marks every configuration where it guesses that the remaining suffix is a word in β. It then consults the regular labeling function in order to single out traces in which a wrong guess is made. For that, we add a new proposition, not_wrong, which holds in a configuration iff it is not the case that pushdown system guesses that the suffix z is in the language of some regular expression r and the guess turns out to be incorrect. The pushdown system also marks the configurations where it finishes handling some rewrite rule. For that, we add a new proposition, $ch\text{-}rule$, which is true only when the system finishes handling some rewrite rule and starts handling another.

The pushdown system R' has four modes of operation when it simulates a transition that follows a rewrite rule $\langle q, \alpha, \beta, \gamma, q'\rangle$. In *delete* mode, R' deletes letters from the store x while emulating a run of $\mathcal{U}_{\alpha_i}$. Delete mode starts from the initial state of

$\mathcal{U}_{\alpha_i}$, from which R' proceeds until it reaches a final state of $\mathcal{U}_{\alpha_i}$. Once the final state of $\mathcal{U}_{\alpha_i}$ is reached, R' transitions to *change-direction* mode, where it does not change the store and just moves to a final state of $\mathcal{U}_{\gamma_i}$, and transitions to *write* mode. In write mode, R' guesses letters in V and emulates the run of $\mathcal{U}_{\gamma_i}$ on them backward, while adding them to the store. From the initial state of $\mathcal{U}_{\gamma_i}$ the pushdown system R' transitions to *change-rule* mode, where it chooses a new rewrite rule $\langle q', \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q'' \rangle$ and transitions to delete mode. Note that if delete mode starts in configuration (q, x) it cannot last indefinitely. Indeed, the pushdown system can remove only finitely many letters from the store. On the other hand, since the store is unbounded, write mode can last forever. Hence, traces along which *ch-rule* occurs only finitely often should be singled out.

Singling out of traces is done by the automaton $\mathcal{S}'$ and the formula φ' which restrict attention to traces in which *not_wrong* is always asserted and *ch-rule* is asserted infinitely often.

Formally, R' has the following components.

- $AP' = AP \cup \{not\_wrong, ch\text{-}rule\}$.
- $Q' = Q \times T \times (\{ch\text{-}dir, ch\text{-}rule\} \cup Q_\alpha \cup Q_\gamma)$. A state $\langle q, t, s \rangle \in Q'$ maintains the state $q \in Q$ and the rewrite rule t currently being applied. the third element s indicates the mode of R'. Change-direction and change-rule modes are indicated by a marker. In delete and write modes, R' also maintains the current state of $\mathcal{U}_\alpha$ and $\mathcal{U}_\gamma$.
- For every proposition $p \in AP$, we have $p \in L'(q, x)$ iff $p \in L(q, x)$. We now describe the regular expression for the propositions *ch-rule* and *not_wrong*. The proposition *ch-rule* holds in all the configuration in which the system is in change-rule mode. Thus, for every $q \in Q$ and $t \in T$, we have $R_{\langle q,t,ch\text{-}rule\rangle, ch\text{-}rule} = V^*$ and $R_{\langle q,t,\zeta\rangle, ch\text{-}rule} = \emptyset$ for $\zeta \neq ch\text{-}rule$. The proposition *not_wrong* holds in configurations in which we are not in change-direction mode, or configuration in which we are in change-direction mode and the store is in β, thus changing direction is possible in the configuration. Formally, for every $q \in Q$ and $t = \langle q', \alpha, \beta, \gamma, q \rangle \in T$, we have $R_{\langle q,t,ch\text{-}dir\rangle, not\_wrong} = \beta$ and $R_{\langle q,t,\zeta\rangle, not\_wrong} = V^*$ for $\zeta \neq ch\text{-}dir$.
- $q'_0 = \langle q_0, t, ch\text{-}rule \rangle$ for some arbitrary rewrite rule t.

The transition function of R' includes four types of transitions according to the four operation modes. In change-direction mode, in configuration $(\langle q, t, ch\text{-}dir\rangle, x)$ that applies the rewrite rule $t = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$, the system R' does not change x, and moves to a final state $s \in F_{\gamma_i}$ of $\mathcal{U}_{\gamma_i}$. In change rule mode, in configuration $(\langle q, t, ch\text{-}rule\rangle, x)$, the system R' does not change x, it chooses a new rewrite rule $t' = \langle q, \alpha_{i'}, \beta_{i'}, \gamma_{i'}, q' \rangle$, changes the Q component to q', and moves to the initial state $q^0_{\alpha_{i'}}$ of $\mathcal{U}_{\alpha_{i'}}$. In delete mode, in configuration $(\langle q, t, s\rangle, x)$, for $t = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$ and $s \in Q_{\alpha_i}$, the system R' proceeds by either removing one letter from x and continuing the run of $\mathcal{U}_{\alpha_i}$, or if $s \in F_{\alpha_i}$ is an accepting state of $\mathcal{U}_{\alpha_i}$ then R' leaves x unchanged, and changes s to *ch-dir*. In write mode, in configuration $(\langle q, t, s\rangle, x)$, for $t = \langle q', \alpha_i, \beta_i, \gamma_i, q \rangle$ and $s \in Q_{\gamma_i}$, the system R' proceeds by either extending x with a guessed symbol from V and continuing the run of $\mathcal{U}_{\gamma_i}$ backward using the guessed symbol, or if $s = q^0_{\gamma_i}$, then R' leaves x unchanged and just replaces s by *ch-rule*. Formally, $T' = T'_{ch\text{-}rule} \cup T'_{ch\text{-}dir} \cup T'_\alpha \cup T'_\gamma$, where

– $T'_{ch\text{-}rule} =$

$$\left\{(\langle q,t,\mathit{ch\text{-}rule}\rangle, A, A, \langle q',t',s\rangle) \;\middle|\; t' = \langle q,\alpha_i,\beta_i,\gamma_i,q'\rangle,\ s = q^0_{\alpha_i} \text{ and } A \in V\right\}.$$

– $T'_{ch\text{-}dir} =$

$$\{(\langle q,t,\mathit{ch\text{-}dir}\rangle, A, A, \langle q,t,s\rangle) \mid t = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s \in F_{\gamma_i}, \text{ and } A \in V\}.$$

Note that the same letter A is removed from the store and added again. Thus, the store content of the configuration does not change.

– $T'_\alpha =$

$$\left\{(\langle q,t,s\rangle, A, \epsilon, \langle q,t,s'\rangle) \;\middle|\; \begin{array}{l} t = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s \in Q_\alpha, \\ s' \in \rho_{\alpha_i}(s,A), \text{ and } A \in V \end{array}\right\} \cup$$
$$\left\{(\langle q,t,s\rangle, A, A, \langle q,t,\mathit{ch\text{-}dir}\rangle) \;\middle|\; \begin{array}{l} t = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s \in Q_\alpha, \\ s \in F_{\alpha_i}, \text{ and } A \in V \end{array}\right\}.$$

– $T'_\gamma =$

$$\left\{(\langle q,t,s\rangle, A, AB, \langle q,t,s'\rangle) \;\middle|\; \begin{array}{l} t = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s \in Q_\gamma, \\ s \in \rho_{\gamma_i}(s',B), \text{ and } A,B \in V \end{array}\right\} \cup$$
$$\left\{(\langle q,t,s\rangle, A, A, \langle q,t,\mathit{ch\text{-}rule}\rangle) \;\middle|\; \begin{array}{l} t = \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle,\ s \in Q_\gamma, \\ s = q^0_{\gamma_i} \text{ and } A \in V \end{array}\right\}.$$

As final states have no outgoing edges, after a state $\langle q, \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, s\rangle$ for $s \in F_{\alpha_i}$ we always visit the state $\langle q,t,\mathit{ch\text{-}dir}\rangle$. Recall that initial states have no incoming edges. It follows that we always visit the state $\langle q,t,\mathit{ch\text{-}rule}\rangle$ after visiting a state $\langle q, \langle q',\alpha_i,\beta_i,\gamma_i,q\rangle, q^0_{\gamma_i}\rangle$.

The automaton $\mathcal{S}'$ adjusts $\mathcal{S}$ to the fact that every transition in R corresponds to multiple transitions in R'. Accordingly, when $\mathcal{S}$ branches universally, infinite navigation stages and states not marked by $\mathit{not\_wrong}$ are allowed. Dually, when $\mathcal{S}$ branches existentially, infinite navigation stages and states not marked by $\mathit{not\_wrong}$ are not allowed.

Formally, let $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F\rangle$. We define, $\mathcal{S}' = \langle \Sigma, W', \delta', w_0, \alpha\rangle$ where

– $W' = W \cup (\{\forall,\exists\} \times W)$ Intuitively, when $\mathcal{S}'$ reads configuration (q,x) and transitions to $\exists w$ it is searching for a successor of (q,x) that is accepted by S^w. The state $\exists w$ navigates to some configuration reachable from (q,x) of R' marked by $\mathit{ch\text{-}rule}$. Dually, when $\mathcal{S}'$ reads configuration (q,x) and transitions to $\forall w$ it is searching for all successors of (q,x) and tries to ensure that they are accepted by S^w. The state $\forall w$ navigates to all configurations reachable from (q,x) of R' marked by $\mathit{ch\text{-}rule}$.
– For every state $w \in W$ and letter $\sigma \in \Sigma$, the transition function δ' is obtained from δ by replacing every atom of the form $\Box w$ by $\Box(\forall w)$ and every atom of the form $\Diamond w$ by $\Diamond(\exists w)$.

For every state $w \in W$ and letter $\sigma \in \Sigma$, we have

$$\delta'(\forall w,\sigma) = \begin{bmatrix} \mathbf{true} & \text{if } \sigma \not\models \mathit{not\_wrong} \\ (\varepsilon, w) & \text{if } \sigma \models \mathit{not\_wrong} \wedge \mathit{ch\text{-}rule} \\ (\Box, \forall w) & \text{if } \sigma \models \mathit{not\_wrong} \wedge \neg \mathit{ch\text{-}rule} \end{bmatrix}$$

$$\delta'(\exists w, \sigma) = \begin{bmatrix} \mathbf{false} & \text{if } \sigma \not\models \mathit{not\_wrong} \\ (\varepsilon, w) & \text{if } \sigma \models \mathit{not\_wrong} \wedge \mathit{ch\text{-}rule} \\ (\Diamond, \exists w) & \text{if } \sigma \models \mathit{not\_wrong} \wedge \neg \mathit{ch\text{-}rule} \end{bmatrix}$$

– The set α is obtained from F by including all states in $\{\forall\} \times W$ as the maximal even set and all states in $\{\exists\} \times W$ as the maximal odd set.

Claim. $G_R \models \mathcal{S}$ iff $G_{R'} \models \mathcal{S}'$

Proof: Assume that $G_{R'} \models \mathcal{S}'$. Let $\langle T', r' \rangle$ be an accepting run of $\mathcal{S}'$ on $G_{R'}$. We construct an accepting run $\langle T, r \rangle$ of $\mathcal{S}$ on G_R based on the subtree of nodes in T' labeled by states in W (it follows that these nodes are labeled by configurations with state $\mathit{ch\text{-}rule}$). Formally, we have the following. We have $r'(\varepsilon) = ((q_0, x_0), w_0)$. We add to T the node ε and label it $r(\varepsilon) = ((q_0, x_0), w_0)$. Given a node $z \in T$ labeled by $r(z) = ((q, x), w)$, it follows that there exists a node $z' \in T'$ labeled by $r'(z') = ((q, x), w)$. Let $\{((q_i, x_i), w_i)\}_{i \in I}$ be the labels of the minimal nodes in T' labeled by states in W. We add $|I|$ successors $\{a_i z\}_{i \in I}$ to z in T and label them $r(a_i z) = ((q_i, x_i), w_i)$. From the definition of R' it follows that $\langle T, r \rangle$ is a valid run of $\mathcal{S}$ on G_R. As every infinite path in T corresponds to an infinite path in T' all whose nodes are marked by configurations marked by $\mathit{not\_wrong}$ and infinitely many configurations are marked by $\mathit{ch\text{-}rule}$ it follows that $\langle T, r \rangle$ is an accepting run.

In the other direction, we extend an accepting run tree $\langle T, r \rangle$ of $\mathcal{S}$ on G_R into an accepting run tree of $\mathcal{S}'$ on $G_{R'}$ by adding transitions to $\{\forall, \exists\} \times W$ type states. □

Corollary 1. *Given a prefix-recognizable system R and a graph automaton $\mathcal{S}$ with n states and index k, we can model check $\mathcal{S}$ with respect to R in time exponential in $n \cdot k \cdot \|T\|$.*

Finally, we proceed to the case of an LTL formula φ. The formula φ' is the implication $\varphi'_1 \rightarrow \varphi'_2$ of two formulas. The formula φ'_1 holds in computations of R' that correspond to real computations of R. Thus, $\varphi'_1 = \Box \mathit{not\_wrong} \wedge \Box\Diamond \mathit{ch\text{-}rule}$. Then, φ'_2 adjusts φ to the fact that a single transition in R corresponds to multiple transitions in R'. Formally, $\varphi'_2 = f(\varphi)$, for the function f defined below.

– $f(p) = p$ for a proposition $p \in AP$.
– $f(\neg a) = \neg f(a)$, $f(a \vee b) = f(a) \vee f(b)$, and $f(a \wedge b) = f(a) \wedge f(b)$.
– $f(a \mathcal{U} b) = (\mathit{ch\text{-}rule} \rightarrow f(a)) \mathcal{U} (\mathit{ch\text{-}rule} \wedge f(b))$.
– $f(\bigcirc a) = \bigcirc((\neg \mathit{ch\text{-}rule}) \mathcal{U} (\mathit{ch\text{-}rule} \wedge f(a)))$.

Claim. $G_R \models \varphi$ iff $G_{R'} \models \varphi'$

We first need some definitions and notations. We define a partial function g from traces in $G_{R'}$ to traces in G_R. Given a trace π' in $G_{R'}$, if $\pi' \not\models \varphi'_1$ then $g(\pi')$ is undefined. Otherwise, denote $\pi' = (p'_0, w_0), (p'_1, w_1), \ldots$ and

$$g(\pi') = \begin{bmatrix} (p, w_0), g(\pi'_{\geq 1}) & \text{if } p'_0 = \langle p, t, \mathit{ch\text{-}rule} \rangle \\ g(\pi'_{\geq 1}) & \text{if } p'_0 = \langle p, t, \alpha \rangle \text{ and } \alpha \neq \mathit{ch\text{-}rule} \end{bmatrix}$$

Thus, g picks from π' only the configurations marked by $\mathit{ch\text{-}rule}$, it then takes the state from Q that marks those configurations and the store. Furthermore given two traces π'

and $g(\pi')$ we define a matching between locations in π' in which the configuration is marked by $ch\text{-}rule$ and the locations in $g(\pi')$. Given a location i in $g(\pi')$ we denote by $ch(i)$ the location in π' of the i-th occurrence of $ch\text{-}rule$ along π'.

Lemma 1. *1. For every trace π' of $G_{R'}$, $g(\pi')$ is either not defined or a valid trace of G_R.*
2. The function g is a bijection between $domain(g)$ and the traces of G_R.
3. For every trace π' of $G_{R'}$ such that $g(\pi')$ is defined, we have $(\pi', ch(i)) \models f(\varphi)$ iff $(g(\pi'), i) \models \varphi$

Proof: 1. Suppose $g(\pi')$ is defined, we have to show that it is a trace of G_R. The first pair in π' is $(\langle q_0, t, ch\text{-}rule\rangle, x_0)$. Hence $g(\pi')$ starts from (q_0, x_0). Assume by induction that the prefix of $g(\pi')$ up to location i is the prefix of some computation in G_R. We show that also the prefix up to location $i+1$ is a prefix of a computation. Let $(\langle q, t, ch\text{-}rule\rangle, x)$ be the i-th $ch\text{-}rule$ appearing in π', then the i-th location in $g(\pi')$ is (q, x). The computation of R' chooses some rewrite rule $t_i = \langle q, \alpha_i, \beta_i, \gamma_i, q'\rangle \in T$ and moves to state $\langle q', t_i, s\rangle$ where $s = q^0_{\alpha_i}$. It must be the case that a state $\langle q', t_i, ch\text{-}dir\rangle$ appears in the computation of R' after location $ch(i)$. Otherwise, the computation is finite and does not interest us. The system R' can move to a state marked by $ch\text{-}dir$ only from $s \in F_{\alpha_i}$, an accepting state of $\mathcal{U}_{\alpha_i}$. Hence, we conclude that $x = y \cdot z$ where $y \in \alpha_i$. As not_wrong is asserted everywhere along π' we know that $z \in \beta_i$. Now R' adds a word y' in γ_i to z and reaches state $(\langle q', t', ch\text{-}rule\rangle, y' \cdot z)$. Thus, the transition t is possible also in R and can lead from $(q, y \cdot z)$ to $(q', y' \cdot z)$.
2. It is quite clear that g is an injection. As above, given a trace π in G_R we can construct the trace π' in $G_{R'}$ such that $g(\pi') = \pi$.
3. We prove that $(\pi, i) \models \varphi$ iff $(\pi', ch(i)) \models \varphi$ by induction on the structure of φ.
   - For a boolean combination of formulas the proof is immediate.
   - For a proposition $p \in AP$, it follows from the proof above that if state (q, x) appears in location i in $g(\pi')$ then state $(\langle q, t, ch\text{-}rule\rangle, x)$ appears in location $ch(i)$ in π'. By definition $p \in L(q, x)$ iff $p \in L'(\langle q, t, ch\text{-}rule\rangle, x)$.
   - For a formula $\varphi = \psi_1 \mathcal{U} \psi_2$. Suppose $(g(\pi'), i) \models \varphi$. Then there exists some $j \geq i$ such that $(g(\pi'), j) \models \psi_2$ and for all $i \leq k < j$ we have $(g(\pi'), k) \models \psi_1$. By the induction assumption we have that $(\pi', ch(j)) \models f(\psi_2)$ (and clearly, $(\pi', ch(j)) \models ch\text{-}rule$), and for all $i \leq j < k$ we have $(\pi', ch(k)) \models \psi_1$. Furthermore, as every location marked by $ch\text{-}rule$ is associated by the function ch to some location in $g(\pi')$ all other locations are marked by $\neg ch\text{-}rule$. Hence, $(\pi', ch(i)) \models (ch\text{-}rule \rightarrow f(\psi_1))\mathcal{U}(f(\psi_2) \wedge ch\text{-}rule)$.
     The other direction is similar.
   - For a formula $\varphi = \bigcirc\psi$ the argument resembles the one above for $\mathcal{U}$. □

We note that for every trace π' and $g(\pi')$ we have that $ch(0) = 0$. Claim 6.2 follows immediately.

If we use this construction in conjunction with Theorem 1, we get an algorithm whose complexity coincides with the one in Theorem 16.

Corollary 2. *Given a prefix-recognizable system R and an LTL formula φ we can model check φ with respect to R in time $O(\|T\|^3)\cdot 2^{O(|Q_\beta|)}\cdot 2^{O(|\varphi|)}$ and space $O(\|T\|^2)\cdot 2^{O(|Q_\beta|)}\cdot 2^{O(|\varphi|)}$.*

Note that for LTL, we change the formula itself while for μ-calculus we change the graph automaton resulting from the formula. Consider the following function from μ-calculus formulas to μ-calculus formulas.

- For $p \in AP$ we have $f(p) = ch\text{-}rule \wedge p$.
- $f(\neg a) = \neg f(a)$, $f(a \vee b) = f(a) \vee f(b)$, and $f(a \wedge b) = f(a) \wedge f(b)$.
- $f(\Box a) = \Box \nu X(f(a) \wedge ch\text{-}rule \vee \neg not\_wrong \vee \neg ch\text{-}rule \wedge \Box X)$.
- $f(\Diamond a) = \Diamond \mu X(f(a) \wedge ch\text{-}rule \wedge not\_wrong \vee \neg ch\text{-}rule \wedge not\_wrong \wedge \Diamond X)$.
- $f(\mu X a(X)) = \mu X(ch\text{-}rule \wedge f(a(X)))$.
- $f(\nu X a(X)) = \nu X(ch\text{-}rule \wedge f(a(X)))$.

We claim that $R \models \psi$ iff $R' \models f(\psi)$. However, the alternation depth of $f(\psi)$ my be much larger than that of ψ. For example, $\varphi = \mu X(p \wedge \Box(\neg p \wedge \Box(X \wedge \mu Y(q \vee \Box Y))))$ is alternation free, while $f(\varphi)$ is of alternation depth 3. This kind of transformation is more appropriate with the equational form of μ-calculus where we can declare all the newly added fixpoints as minimal and incur only an increase of 1 in the alternation depth.

We note that since we end up with a pushdown system with regular labeling, it is easy to extend the reduction to start with a prefix-recognizable system with regular labeling. It is left to show the reduction in the other direction.

We can also reduce the problem of μ-calculus (resp., LTL) model checking of pushdown graphs with regular labeling, to the problem of μ-calculus (resp., LTL) model checking of prefix-recognizable graphs. This is summarized in the following two theorems.

Theorem 22. *Given a pushdown system $R = \langle \Sigma, V, Q, T, L, q_0, x_0\rangle$ with a regular labeling function and a graph automaton $\mathcal{S}$, there is a prefix-recognizable system $R' = \langle \Sigma, V, Q', T', L', q'_0, x_0\rangle$ with simple labeling and a graph automaton $\mathcal{S}'$ such that $R \models \mathcal{S}$ iff $R' \models \mathcal{S}'$. Furthermore, $|Q'| = |Q|+|\Sigma|$, $|Q'_\alpha|+|Q'_\gamma| = O(\|T\|)$, and $|Q'_\beta| = \|L\|$. The reduction is computable in logarithmic space.*

Theorem 23. *Given a pushdown system $R = \langle 2^{AP}, V, Q, T, L, q_0, x_0\rangle$ with a regular labeling function and an LTL formula φ, there is a prefix-recognizable system $R' = \langle 2^{AP'}, V, Q', T', L', q'_0, x_0\rangle$ with simple labeling and an LTL formula φ' such that $R \models \varphi$ iff $R' \models \varphi'$. Furthermore, $|Q'| = O(|Q| \cdot |AP|)$, $|Q'_\alpha| + |Q'_\gamma| = O(\|T\|)$, and $|Q'_\beta| = 2^{\|L\|}$ yet the automata for Q'_β are deterministic. The reduction is computable in polynomial space.*

For the full constructions and proofs we refer the reader to [Pit04].

7 Realizability and Synthesis

In this section we show that the automata-theoretic approach can be used also to solve the realizability and synthesis problems for branching time and linear time specifications of pushdown and prefix-recognizable systems. We start with a definition of the

realizability and synthesis problems and then proceed to give algorithms that solve these problems for μ-calculus and LTL.

Given a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a partition $\{T_1, \ldots, T_m\}$ of T, a *strategy* of R is a function $f : Q \times V^* \to [m]$. The function f restricts the graph G_R so that from a configuration $(q, x) \in Q \times V^*$, only $f(q, x)$ transitions are taken. Formally, R and f together define the graph $G_{R,f} = \langle \Sigma, Q \times V^*, \rho, (q_0, x_0), L \rangle$, where $\rho((q, x), (q', y))$ iff $f(q, x){=}i$ and there exists $t \in T_i$ such that $\rho_t((q, x), (q', y))$. Given R and a specification ψ (either a graph automaton or an LTL formula), we say that a strategy f of R is *winning* for ψ iff $G_{R,f}$ satisfies ψ. Given R and ψ the problem of *realizability* is to determine whether there is a winning strategy of R for ψ. The problem of *synthesis* is then to construct such a strategy.[10] The setting described here corresponds to the case where the system needs to satisfy a specification with respect to environments modeled by a rewrite system. Then, at each state, the system chooses the subset of transitions to proceed with and the environment provides the rules that determine the successors of the state.

Similar to Theorems 7 and 15, we construct automata that solve the realizability problem and provide winning strategies. The idea is simple: a strategy $f : Q \times V^* \to [m]$ can be viewed as a $V \times [m]$-labeled V-tree. Thus, the realizability problem can be viewed as the problem of determining whether we can augment the labels of the tree $\langle V^*, \tau_V \rangle$ by elements in $[m]$, and accept the augmented tree in a run of $\mathcal{A}$ in which whenever $\mathcal{A}$ reads an entry $i \in [m]$, it applies to the transition function of the specification graph automaton only rewrite rules in T_i.

We give the solution to the realizability and synthesis problems for branching-time specifications. Given a rewrite system R and a graph automaton $\mathcal{S}$, we show how to construct a 2APT $\mathcal{A}$ such that the language of $\mathcal{A}$ is not empty iff $\mathcal{S}$ is realizable over R.

Theorem 24. *Given a rewrite system $R = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$, a partition $\{T_1, \ldots, T_m\}$ of T, and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, we can construct a 2APT $\mathcal{A}$ over $((V \cup \{\bot\}) \times [m])$-labeled V-trees such that $L(\mathcal{A})$ contains exactly all the V-exhaustive trees whose projection on $[m]$ is a winning strategy of R for $\mathcal{S}$. The automaton $\mathcal{A}$ has $O(|W| \cdot |Q| \cdot \|T\| \cdot |V|)$ states, and its index is the index of $\mathcal{S}$ (plus 1 for a prefix-recognizable system).*

Proof: Unlike Theorem 7 here we use the emptiness problem of 2APT instead of the membership problem. It follows that we have to construct a 2APT that ensures that

[10] Note that we define here only memoryless strategies. The strategy depends solely on the current configuration and not on the history of the computations. In general, in order to realize some specifications, strategies that depend on the history of the computation may be required. In order to solve realizability and synthesis for specifications that require memory we have to use a more complex algorithm. In the case of branching time specifications, we have to combine the rewrite system with the graph automaton for the specification and analyze the resulting game. In the case of linear time specifications, we have to combine the rewrite system with a deterministic parity automaton for the specification and analyze the resulting game. In both cases the analysis of the game can be done using 2-way tree automata. In the linear-time framework, the deterministic automaton may be doubly exponential larger than the LTL formula; and the total complexity of this algorithm is triple exponential. For further details and a matching lower bound we refer the reader to [LMS04].

its input tree is V-exhaustive and that the strategy encoded in the tree is winning. The modification to the construction in the proof of Theorem 7 are simple. Let $\mathcal{A}'$ denote the result of the construction in Theorem 6 or Theorem 7 with the following modification to the function $apply_T$. From action states we allow to proceed only with transitions from T_i, where i is the $[m]$ element of the letter we read. For example, in the case of a pushdown system, we would have for $c \in \Delta$, $w \in W$, $q \in Q$, $A \in V$ and $i \in [m]$ (the new parameter to $apply_T$, which is read from the input tree),

$$apply_T(c, w, q, A, i) = \begin{cases} \langle \varepsilon, (w, q, \varepsilon) \rangle & \text{if } c = \varepsilon \\ \bigwedge_{\langle q, A, y, q' \rangle \in T_i} \langle \uparrow, (w, q', y) \rangle & \text{if } c = \Box \\ \bigvee_{\langle q, A, y, q' \rangle \in T_i} \langle \uparrow, (w, q', y) \rangle & \text{if } c = \Diamond \end{cases}$$

We now construct the automaton $\mathcal{A}'' = \langle (V \cup \{\bot\} \times [m]), (V \cup \{\bot\}), \rho, bot, \{V\} \rangle$ of index 1 (i.e., every valid run is an accepting run) such that for every $A, B \in V \cup \{\bot\}$ and $i \in [m]$ we have

$$\rho(A, (B, i) = \begin{cases} \bigwedge_{C \in V} (C, C) & \text{if } A = B \\ \mathbf{false} & \text{if } A \neq B \end{cases}$$

It follows that $\mathcal{A}'$ accepts only V-exhaustive trees. Finally, we take $\mathcal{A} = \mathcal{A}' \wedge \mathcal{A}''$ the conjunction of the two automata. □

Let $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$, let k be the index of $\mathcal{S}$, and let $\Gamma = (V \cup \{\bot\}) \times [m]$. By Theorem 2, we can transform $\mathcal{A}$ to a nondeterministic one-way parity tree automaton $\mathcal{N}$ with $2^{O(nk)}$ states and index $O(nk)$.[11] By [Rab69, Eme85], if $\mathcal{N}$ is nonempty, there exists a Γ-labeled V-tree $\langle V^*, f \rangle$ such that for all $\gamma \in \Gamma$, the set X_γ of nodes $x \in V^*$ for which $f(x) = \gamma$ is a regular set. Moreover, the nonemptiness algorithm of $\mathcal{N}$, which runs in time exponential in nk, can be easily extended to construct, within the same complexity, a deterministic word automaton $\mathcal{U}_\mathcal{A}$ over V such that each state of $\mathcal{U}_\mathcal{A}$ is labeled by a letter $\gamma \in \Gamma$, and for all $x \in V^*$, we have $f(x) = \gamma$ iff the state of $\mathcal{U}_\mathcal{A}$ that is reached by following the word x is labeled by γ. The automaton $\mathcal{U}_\mathcal{A}$ is then the answer to the synthesis problem. Note that since the transitions in $G_{\mathcal{R},f}$ take us from a state $x \in V^*$ to a state $y \in V^*$ such that x is not necessarily the parent of y in the V-tree, an application of the strategy f has to repeatedly run the automaton $\mathcal{U}_\mathcal{A}$ from its initial state resulting in a strategy whose every move is computed in time proportional to the length of the configuration. We can construct a strategy that computes the next step in time proportional to the difference between x and y. This strategy uses a pushdown store. It stores the run of $\mathcal{U}_\mathcal{A}$ on x on the pushdown store. In order compute the strategy in node y, we retain on the pushdown store only the part of the run of $\mathcal{U}_\mathcal{A}$ that relates to the common suffix of x and y. We then continue the run of $\mathcal{U}_\mathcal{A}$ on the prefix of y while storing it on the pushdown store.

The construction described in Theorems 6 and 7 implies that the realizability and synthesis problem is in EXPTIME. Thus, it is not harder than in the satisfiability problem for the μ-calculus, and it matches the known lower bound [FL79]. Formally, we have the following.

[11] Note that the automaton $\mathcal{A}''$ is in fact a 1NPT of index 1. We can improve the efficiency of the algorithm by first converting $\mathcal{A}'$ into a 1NPT and only then combining the result with $\mathcal{A}''$. This would result in $|V|$ being removed from the figure describing the index of $\mathcal{N}$.

Theorem 25. *The realizability and synthesis problems for a pushdown or a prefix-recognizable rewrite system $\mathcal{R} = \langle \Sigma, V, Q, L, T, q_0, x_0 \rangle$ and a graph automaton $\mathcal{S} = \langle \Sigma, W, \delta, w_0, F \rangle$, can be solved in time exponential in nk, where $n = |W| \cdot |Q| \cdot \|T\| \cdot |V|$, and k is the index of $\mathcal{S}$.*

By Theorem 4, if the specification is given by a μ-calculus formula ψ, the bound is the same, with $n = |\psi| \cdot |Q| \cdot \|T\| \cdot |V|$, and k being the alternation depth of ψ.

In order to use the above algorithm for realizability of linear-time specifications we cannot use the 'usual' translations of LTL to μ-calculus [Dam94, dAHM01]. The problem is with the fact that these translations are intended to be used in μ-calculus model checking. The translation from LTL to μ-calculus used for model checking [Dam94] cannot be used in the context of realizability [dAHM01]. We have to use a doubly exponential translation intended for realizability [dAHM01], this, however, results in a triple exponential algorithm which is, again, less than optimal.

Alur et al. show that LTL realizability and synthesis can be exponentially reduced to μ-calculus realizability [ATM03]. Given an LTL formula φ, they construct a graph automaton $\mathcal{S}_\varphi$ such that $\mathcal{S}_\varphi$ is realizable over R iff φ is realizable over R. The construction of the graph automaton proceeds as follows. According to Theorem 5, for every LTL formula ψ we can construct an NBW N_ψ such that $L(N_\psi) = L(\psi)$. We construct an NBW $N_{\neg\varphi} = \langle \Sigma, W, \eta, w_0, F \rangle$ from $\neg\varphi$. We then construct the graph automaton $\mathcal{S}_\varphi = \langle \Sigma, W, \rho, w_0, \{F, W\} \rangle$ where $\rho(w, \sigma) = \bigwedge_{w' \in \eta(w,\sigma)} \Box w'$ and the parity condition $\{F, W\}$ is equivalent to the co-Büchi condition F. It follows that $\mathcal{S}_\varphi$ is a universal automaton and has a unique run over every trace. Alur et al. show that the fact that $\mathcal{S}_\varphi$ has a unique run over every trace makes it adequate for solving the realizability of φ [ATM03]. The resulting algorithm is exponential in the rewrite system and doubly exponential in the LTL formula. As synthesis of LTL formulas with respect to finite-state environments is already 2EXPTIME-hard [PR89], this algorithm is optimal. Note that realizability with respect to LTL specifications is exponential in the system already for pushdown systems and exponential in all components of the system for prefix-recognizable systems.

8 Discussion

The automata-theoretic approach has long been thought to be inapplicable for effective reasoning about infinite-state systems. We showed that infinite-state systems for which decidability is known can be described by finite-state automata, and therefore, the states and transitions of such systems can be viewed as nodes in an infinite tree and transitions between states can be expressed by finite-state automata. As a result, automata-theoretic techniques can be used to reason about such systems. In particular, we showed that various problems related to the analysis of such systems can be reduced to the membership or emptiness problems for alternating two-way tree automata. Our framework achieves the same complexity bounds of known model-checking algorithms and gives the first solution to model-checking LTL with respect to prefix-recognizable systems. In [PV04] we show how to extend it also to global model checking. In [Cac03, PV04] the scope of automata-theoretic reasoning is extended beyond prefix-recognizable systems.

We have shown that the problems of model checking with respect to pushdown systems with regular labeling and model checking with respect to prefix-recognizable systems are intimately related. We give reductions between model checking of pushdown systems with regular labeling and model checking of prefix-recognizable systems with simple labeling.

The automata-theoretic approach offers several extensions to the model checking setting. The systems we want to reason about are often augmented with *fairness constraints*. Like state properties, we can define a *regular fairness constraint* by a regular expression α, where a computation of the labeled transition graph is fair iff it contains infinitely many states in α (this corresponds to weak fairness; other types of fairness can be defined similarly). It is easy to extend our model-checking algorithm to handle fairness (that is, let the path quantification in the specification range only on fair paths[12]). In the branching-time framework, the automaton $\mathcal{A}$ can guess whether the state currently visited is in α, and then simulate the word automaton $\mathcal{U}_\alpha$ upwards, hoping to visit an accepting state when the root is reached. When $\mathcal{A}$ checks an existential property, it has to make sure that the property is satisfied along a fair path, and it is therefore required to visit infinitely many states in α. When $\mathcal{A}$ checks a universal property, it may guess that a path it follows is not fair, in which case $\mathcal{A}$ eventually always send copies that simulate the automaton for $\neg\alpha$. In the linear-time framework, we add the automata for the fairness constraints to the tree whose membership is checked. The guessed path violating the property must visit infinitely many fair states. The complexity of the model-checking algorithm stays the same.

Another extension is the treatment of μ-calculus specifications with *backwards modalities*. While forward modalities express weakest precondition, backward modalities express strongest postcondition, and they are very useful for reasoning about the past [LPZ85]. In order to adjust graph automata to backward reasoning, we add to Δ the "directions" $\Diamond^-$ and $\Box^-$. This enables the graph automata to move to predecessors of the current state. More formally, if a graph automaton reads a state x of the input graph, then fulfilling an atom $\Diamond^- t$ requires $\mathcal{S}$ to send a copy in state t to some predecessor of x, and dually for $\Box^- t$. Theorem 4 can then be extended to μ-calculus formulas and graph automata with both forward and backward modalities [Var98]. Extending our solution to graph automata with backward modalities is simple. Consider a configuration $(q, x) \in Q \times V^*$ in a prefix-recognizable graph. The predecessors of (q, x) are configurations $(q'y)$ for which there is a rule $\langle q', \alpha_i, \beta_i, \gamma_i, q\rangle \in T$ and partitions $x' \cdot z$ and $y' \cdot z$, of x and y, respectively, such that x' is accepted by $\mathcal{U}_{\gamma_i}$, z is accepted by $\mathcal{U}_{\beta_i}$, and y' is accepted by $\mathcal{U}_{\alpha_i}$. Hence, we can define a mapping T^- such that $\langle q, \gamma, \beta, \alpha, q'\rangle \in T^-$ iff $\langle q, \alpha, \beta, \gamma, q\rangle \in T$, and handle atoms $\Diamond^- t$ and $\Box^- t$ exactly as we handle $\Diamond t$ and $\Box t$, only that for them we apply the rewrite rules in T^- rather than these in T. The complexity of the model-checking algorithm stays the same. Note that the simple solution relies on the fact that the structure of the rewrite rules in a prefix-recognizable rewrite

[12] The exact semantics of *fair graph automata* as well as *fair μ-calculus* is not straightforward, as they enable cycles in which we switch between existential and universal modalities. To make our point here, it is simpler to assume in the branching-time framework, say, graph automata that correspond to CTL* formulas.

system is symmetric (that is, switching α and γ results in a well-structured rule), which is not the case for pushdown systems[13].

Recently, Alur et al. suggested the logic CARET, that can specify non-regular properties [AEM04]. Our algorithm generalizes to CARET specifications as well. Alur et al. show how to combine the specification with a pushdown system in a way that enables the application of our techniques. The logic CARET is tailored for use in conjunction with pushdown systems. It is not clear how to modify CARET in order to apply to prefix-recognizable systems. Other researchers have used the versatility of the automata-theoretic framework for reasoning about infinite-state systems. Cachat shows how to model check μ-calculus specifications with respect to high order pushdown graphs [Cac03]. Gimbert shows how to solve games over pushdown graphs where the winning conditions are combinations of parity and unboundedness [Gim03].

References

[ACM06] Alur, R., Chaudhuri, S., Madhusudan, P.: Languages of nested trees. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 329–342. Springer, Heidelberg (2006)

[AEM04] Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 67–79. Springer, Heidelberg (2004)

[AHK97] Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. In: Proc. 38th IEEE Symp. on Foundations of Computer Science, pp. 100–109 (1997)

[AM04] Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proc. 36th ACM Symp. on Theory of Computing. ACM press, New York (2004)

[AM06] Alur, R., Madhusudan, P.: Adding nesting structure to words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 1–13. Springer, Heidelberg (2006)

[ATM03] Alur, R., La Torre, S., Madhusudan, P.: Modular strategies for infinite games on recursive game graphs. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 67–79. Springer, Heidelberg (2003)

[BC04] Bojanczyk, M., Colcombet, T.: Tree-walking automata cannot be determinized. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 246–256. Springer, Heidelberg (2004)

[BC05] Bojanczyk, M., Colcombet, T.: Tree-walking automata do not recognize all regular languages. In: Proc. 37th ACM Symp. on Theory of Computing. ACM press, New York (2005)

[BEM97] Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)

[BLM01] Biesse, P., Leonard, T., Mokkedem, A.: Finding bugs in an alpha microprocessor using satisfiability solvers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 454–464. Springer, Heidelberg (2001)

[13] Note that this does not mean we cannot model check specifications with backwards modalities in pushdown systems. It just means that doing so involves rewrite rules that are no longer pushdown. Indeed, a rule $\langle q, A, x, q' \rangle \in T$ in a pushdown system corresponds to the rule $\langle q, A, V^*, x, q' \rangle \in T$ in a prefix-recognizable system, inducing the rule $\langle q', x, V^*, A, q \rangle \in T^{-1}$.

[BLS06] Bárány, V., Löding, C., Serre, O.: Regularity problem for visibly pushdown languages. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 420–431. Springer, Heidelberg (2006)

[BMP05] Bozzelli, L., Murano, A., Peron, A.: Pushdown module checking. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 504–518. Springer, Heidelberg (2005)

[Boz06] Bozzelli, L.: Complexity results on branching-time pushdown model checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 65–79. Springer, Heidelberg (2005)

[BQ96] Burkart, O., Quemener, Y.-M.: Model checking of infinite graphs defined by graph grammars. In: Proc. 1st Int. workshop on verification of infinite states systems. ENTCS, p. 15. Elsevier, Amsterdam (1996)

[BR00] Ball, T., Rajamani, S.: Bebop: A symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)

[BR01] Ball, T., Rajamani, S.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)

[BS92] Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)

[BS95] Burkart, O., Steffen, B.: Composition, decomposition and model checking of pushdown processes. Nordic J. Comut. 2, 89–125 (1995)

[BSW03] Bouquet, A.-J., Serre, O., Walukiewicz, I.: Pushdown games with unboundedness and regular conditions. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 88–99. Springer, Heidelberg (2003)

[BTP06] Bozzelli, L., La Torre, S., Peron, A.: Verification of well-formed communicating recursive state machines. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 412–426. Springer, Heidelberg (2005)

[Büc62] Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. Int. Congress on Logic, Method, and Philosophy of Science, pp. 1–12. Stanford University Press, Stanford (1960)

[Bur97] Burkart, O.: Model checking rationally restricted right closures of recognizable graphs. In: Moller, F. (ed.) Proc. 2nd Int. workshop on verification of infinite states systems (1997)

[Cac03] Cachat, T.: Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 556–569. Springer, Heidelberg (2003)

[Cau96] Caucal, D.: On infinite transition graphs having a decidable monadic theory. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 194–205. Springer, Heidelberg (1996)

[Cau03] Caucal, D.: On infinite transition graphs having a decidable monadic theory. Theoretical Computer Science 290(1), 79–115 (2003)

[CES86] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languagues and Systems 8(2), 244–263 (1986)

[CFF+01] Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 436–453. Springer, Heidelberg (2001)

[CGP99] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

[CKS81] Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the Association for Computing Machinery 28(1), 114–133 (1981)

[CW02] Chen, H., Wagner, D.: Mops: an infrastructure for examining security properties of software. In: Proc. 9th ACM conference on Computer and Communications Security, pp. 235–244. ACM, New York (2002)

[CW03] Carayol, A., Wöhrle, S.: The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 112–123. Springer, Heidelberg (2003)

[dAHM01] de Alfaro, L., Henzinger, T.A., Majumdar, R.: From verification to control: dynamic programs for omega-regular objectives. In: Proc. 16th IEEE Symp. on Logic in Computer Science, pp. 279–290. IEEE Computer Society Press, Los Alamitos (2001)

[Dam94] Dam, M.: CTL* and ECTL* as fragments of the modal μ-calculus. Theoretical Computer Science 126, 77–96 (1994)

[DW99] Dickhfer, M., Wilke, T.: Timed alternating tree automata: the automata-theoretic solution to the TCTL model checking problem. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 281–290. Springer, Heidelberg (1999)

[EE05] Esparza, J., Etessami, K.: Verifying probabilistic procedural programs. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 16–31. Springer, Heidelberg (2004)

[EHRS00] Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)

[EHvB99] Engelfriet, J., Hoggeboom, H.J., van Best, J.-P.: Trips on trees. Acta Cybernetica 14, 51–64 (1999)

[EJ88] Emerson, E.A., Jutla, C.: The complexity of tree automata and logics of programs. In: Proc. 29th IEEE Symp. on Foundations of Computer Science, pp. 328–337 (1988)

[EJ91] Emerson, E.A., Jutla, C.: Tree automata, μ-calculus and determinacy. In: Proc. 32nd IEEE Symp. on Foundations of Computer Science, pp. 368–377 (1991)

[EJS93] Emerson, E.A., Jutla, C., Sistla, A.P.: On model-checking for fragments of μ-calculus. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 385–396. Springer, Heidelberg (1993)

[EKS01] Esparza, J., Kucera, A., Schwoon, S.: Model-checking LTL with regular valuations for pushdown systems. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 316–339. Springer, Heidelberg (2001)

[EKS06] Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with craig interpolation and symbolic pushdown systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 489–503. Springer, Heidelberg (2006)

[EL86] Emerson, E.A., Lei, C.-L.: Efficient model checking in fragments of the propositional μ-calculus. In: Proc. 1st IEEE Symp. on Logic in Computer Science, pp. 267–278 (1986)

[Eme85] Emerson, E.A.: Automata, tableaux, and temporal logics. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 79–87. Springer, Heidelberg (1985)

[Eme97] Emerson, E.A.: Model checking and the μ-calculus. In: Immerman, N., Kolaitis, P.G. (eds.) Descriptive Complexity and Finite Models, pp. 185–214. American Mathematical Society, Providence (1997)

[ES01] Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)

[FL79] Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. Journal of Computer and Systems Science 18, 194–211 (1979)

[Gim03] Gimbert, H.: Explosion and parity games over context-free graphs. Technical Report 2003-015, Liafa, CNRS, Paris University 7 (2003)

[GO03] Gastin, P., Oddoux, D.: Ltl with past and two-way very-weak alternating automata. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 439–448. Springer, Heidelberg (2003)

[GPVW95] Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembiski, P., Sredniawa, M. (eds.) Protocol Specification, Testing, and Verification, pp. 3–18. Chapman & Hall, Boca Raton (1995)

[GW94] Godefroid, P., Wolper, P.: A partial approach to model checking. Information and Computation 110(2), 305–326 (1994)

[HHK96] Hardin, R.H., Har'el, Z., Kurshan, R.P.: COSPAN. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 423–427. Springer, Heidelberg (1996)

[HHWT95] Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: A user guide to HYTECH. In: TACAS 1995. LNCS, vol. 1019, pp. 41–71. Springer, Heidelberg (1995)

[HKV96] Henzinger, T.A., Kupferman, O., Vardi, M.Y.: A space-efficient on-the-fly algorithm for real-time model checking. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 514–529. Springer, Heidelberg (1996)

[Hol97] Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)

[JSWR06] Jha, S., Schwoon, S., Wang, H., Reps, T.: Weighted pushdown systems and trust-management systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, p. 126. Springer, Heidelberg (2006)

[JW95] Janin, D., Walukiewicz, I.: Automata for the modal μ-calculus and related results. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)

[KG06] Kahlon, V., Gupta, A.: An automata theoretic approach for model checking threads for ltl properties. In: Proc. 21st IEEE Symp. on Logic in Computer Science, pp. 101–110. IEEE press, Los Alamitos (2006)

[KG07] Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: Proc. 34th ACM Symp. on Principles of Programming Languages (2007)

[KGS06] Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 286–299. Springer, Heidelberg (2006)

[KIG05] Kahlon, V., Ivanĉić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)

[KNUW05] Knapik, T., Niwinski, D., Urzczyn, P., Walukiewicz, I.: Unsafe grammars and panic automata. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1450–1461. Springer, Heidelberg (2005)

[Koz83] Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)

[KP95] Kupferman, O., Pnueli, A.: Once and for all. In: Proc. 10th IEEE Symp. on Logic in Computer Science, pp. 25–35 (1995)

[KPV02] Kupferman, O., Piterman, N., Vardi, M.Y.: Model checking linear properties of prefix-recognizable systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 371–385. Springer, Heidelberg (2002)

[Kur94] Kurshan, R.P.: Computer Aided Verification of Coordinating Processes. Princeton Univ. Press, Princeton (1994)

[KV98] Kupferman, O., Vardi, M.Y.: Modular model checking. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 381–401. Springer, Heidelberg (1998)

[KV99] Kupferman, O., Vardi, M.Y.: Robust satisfaction. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 383–398. Springer, Heidelberg (1999)

[KV00a] Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to reasoning about infinite-state systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 36–52. Springer, Heidelberg (2000)

[KV00b] Kupferman, O., Vardi, M.Y.: Synthesis with incomplete information. In: Advances in Temporal Logic, pp. 109–127. Kluwer Academic Publishers, Dordrecht (2000)

[KV01] Kupferman, O., Vardi, M.Y.: On bounded specifications. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 24–38. Springer, Heidelberg (2001)

[KVW00] Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM 47(2), 312–360 (2000)

[LMS04] Löding, C., Madhusudan, P., Serre, O.: Visibly pushdown games. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 408–420. Springer, Heidelberg (2004)

[LP85] Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proc. 12th ACM Symp. on Principles of Programming Languages, pp. 97–107 (1985)

[LPY97] Larsen, K.G., Petterson, P., Yi, W.: UPPAAL: Status & developments. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 456–459. Springer, Heidelberg (1997)

[LPZ85] Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)

[MS85] Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and second-order logic. Theoretical Computer Science 37, 51–75 (1985)

[MS87] Muller, D.E., Schupp, P.E.: Alternating automata on infinite trees. Theoretical Computer Science 54, 267–276 (1987)

[Nev02] Neven, F.: Automata, logic, and XML. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471, pp. 2–26. Springer, Heidelberg (2002)

[Ong06] Ong, L.: On model-checking trees generated by higher-order recursion schemes. In: Proc. 21st IEEE Symp. on Logic in Computer Science, pp. 81–90. IEEE Computer Society Press, Los Alamitos (2006)

[Pit04] Piterman, N.: Verification of Infinite-State Systems. PhD thesis, Weizmann Institute of Science (2004)

[Pnu77] Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. on Foundations of Computer Science, pp. 46–57 (1977)

[PR89] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages, pp. 179–190 (1989)

[PV03] Piterman, N., Vardi, M.Y.: From bidirectionality to alternation. Theoretical Computer Science 295(1-3), 295–321 (2003)

[PV04] Piterman, N., Vardi, M.: Global model-checking of infinite-state systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 387–400. Springer, Heidelberg (2004)

[QS82] Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: Proc. 8th ACM Symp. on Principles of Programming Languages. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)

[Rab69] Rabin, M.O.: Decidability of second order theories and automata on infinite trees. Transaction of the AMS 141, 1–35 (1969)

[Sch02] Schwoon, S.: Model-checking pushdown systems. PhD thesis, Technische Universität München (2002)

[Ser06] Serre, O.: Parity games played on transition graphs of one-counter processes. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 337–351. Springer, Heidelberg (2006)

[SSE06] Suwimonteerabuth, D., Schwoon, S., Esparza, J.: Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 141–153. Springer, Heidelberg (2006)

[Var98] Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)

[VB00] Visser, W., Barringer, H.: Practical CTL* model checking: Should SPIN be extended? Software Tools for Technology Transfer 2(4), 350–365 (2000)

[VW86a] Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. 1st IEEE Symp. on Logic in Computer Science, pp. 332–344 (1986)

[VW86b] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and Systems Science 32(2), 182–221 (1986)

[VW94] Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

[Wal96] Walukiewicz, I.: Pushdown processes: games and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996)

[Wal00] Walukiewicz, I.: Model checking ctl properties of pushdown systems. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 127–138. Springer, Heidelberg (2000)

[Wal02] Walukiewicz, I.: Monadic second-order logic on tree-like structures. Theoretical Computer Science 275(1-2), 311–346 (2002)

[Wil99] Wilke, T.: CTL$^+$ is exponentially more succinct than CTL. In: Pandu Rangan, C., Raman, V., Sarukkai, S. (eds.) FST TCS 1999. LNCS, vol. 1738, pp. 110–121. Springer, Heidelberg (1999)

[WVS83] Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proc. 24th IEEE Symp. on Foundations of Computer Science, pp. 185–194 (1983)

[WW96] Willems, B., Wolper, P.: Partial-order methods for model checking: From linear time to branching time. In: Proc. 11th IEEE Symp. on Logic in Computer Science, pp. 294–303 (1996)

A Proof of Claim 4.3

The proof of the claim is essentially equivalent to the same proof in [PV03].

Claim. $L(\mathcal{A}) \neq \emptyset$ iff $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$.

Proof: We prove that $\langle \Upsilon^*, \tau \rangle \in L(\mathcal{P})$ implies $L(\mathcal{A}) \neq \emptyset$. Let $r = (p_0, w_0) \cdot (p_1, w_1) \cdot (p_2, w_2) \cdots$ be an accepting run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$. We add the annotation of the locations in the run $(p_0, w_0, 0) \cdot (p_1, w_1, 1) \cdot (p_2, w_2, 2) \cdots$. We construct the run $\langle T', r' \rangle$ of $\mathcal{A}$.

For every node $x \in T'$, if x is labeled by a singleton state we add a tag to x some triplet from the run r. If x is labeled by a pair state we add two tags to x, two triplets from the run r. The labeling and the tagging conform to the following.

- Given a node x labeled by state (p, d, α) and tagged by the triplet (p', w, i) from r, we build r' so that $p = p'$ and $d = \rho_\tau(w)$. Furthermore all triplets in r whose third element is greater than i have their second element greater or equal to w (Υ^* is ordered according to the lexical order on the reverse of the words).
- Given a node x labeled by state (q, p, d, α) and tagged by the triplets (q', w, i) and (p', w', j) from r, we build r' so that $q = q'$, $p = p'$, $w = w'$, $d = \rho_\tau(w)$, and $i < j$. Furthermore all triplets in r whose third element k is between i and j, have their second element greater or equal to w. Also, if $j > i + 1$ then $w_{j-1} = \upsilon \cdot w_j$ for some $\upsilon \in \Upsilon$.

Construct the run tree $\langle T', r' \rangle$ of $\mathcal{A}$ as follows. Label the root of T' by $(p_0, d^0_\tau, \bot)$ and tag it by $(p_0, \varepsilon, 0)$. Given a node $x \in T'$ labeled by (p, d, α) tagged by (p, w, i). Let (p_j, w_j, j) be the minimal $j > i$ such that $w_j = w$. If $j = i + 1$ then add one son to x, label it $(p_j, d, \bot)$ and tag it (p_j, w, j). If $j > i + 1$, then $w_{j-1} = \upsilon \cdot w_i$ for some $\upsilon \in \Upsilon$ and we add two sons to x, label them (p, p_j, d, β) and (p_j, d, β). We tag (p_i, p_j, d, β) by (p, w, i) and (p_j, w, j), and tag (p_j, d, β) by (p_j, w, j), β is $\top$ if there is a visit to F between locations i and j in r. If there is no other visit to w then $w_{i+1} = \upsilon \cdot w$ for some $\upsilon \in \Upsilon$. We add one son to x and label it $(p_{i+1}, \rho_\tau(d, \upsilon), \bot)$ and tag it $(p_{i+1}, \upsilon \cdot w, i+1)$. Obviously the labeling and the tagging conform to the assumption.

Given a node x labeled by a state (p, q, d, α) and tagged by (p, w, i) and (q, w, j). Let (p_k, w, k) be the first visit to w between i and j. If $k = i + 1$ then add one son to x, label it $(p_k, q, d, f_\alpha(p_k, q))$, and tag it by (p_k, w, k) and (q, w, j). If $k > i + 1$ then add two sons to x and label them $(p, p_k, d, f_{\beta_1}(p, p_k))$ and $(p_k, q, d, f_{\beta_2}(p_k, q))$ where β_1, β_2 are determined according to the visits to F between i and j. We tag the state $(p, p_k, d, f_{\beta_1}(p, p_k))$ by (p, w, i) and (p_k, w, k) and tag $(p_k, t, d, f_{\beta_2}(p_k, q))$ by (p_k, w, k) and (q, w', j).

If there is no visit to w between i and j it must be the case that all triplets in r between i and j have the same suffix $\upsilon \cdot w$ for some $\upsilon \in \Upsilon$ (otherwise w is visited). We add a son to x labeled $(p_{i+1}, q_{j-1}, \rho_\tau(d, \upsilon), f_\alpha(p', q'))$ and tagged by $(p_{i+1}, \upsilon \cdot w, i + 1)$ and $(p_{j-1}, \upsilon \cdot w, j - 1)$. We are ensured that $p_{j-1} \in C_q^{L_\tau(\rho_\tau(d,\upsilon))}$ as $(\uparrow, p_j) \in \delta(p_{j-1}, \tau(\upsilon \cdot w))$.

In the other direction, given an accepting run $\langle T', r' \rangle$ of $\mathcal{A}$ we use the recursive algorithm in Figure 1 to construct a run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$.

A node $x \cdot a$ in T' is *advancing* if the transition from x to $x \cdot a$ results from an atom $(1, r'(x \cdot a))$ that appears in $\eta(r'(x))$. An advancing node that is the immediate successor of a singleton state satisfies the disjunct $\bigvee_{\upsilon \in \Upsilon} \bigvee_{(\upsilon, p') \in \delta(p, L(d))} (1, (p', \rho_\tau(d, \upsilon), \bot))$ in η. We tag this node by the letter υ that was used to satisfy the transition. Similarly, an advancing node that is the immediate successor of a pair state satisfies the disjunct $\bigvee_{\upsilon \in \Upsilon} \bigvee_{\langle \upsilon, p' \rangle \in \delta(p_1, L_\tau(d))} \bigvee_{p'' \in C_{p_2}^{L_\tau(d)}} (1, (p', p'', \rho_\tau(d, \upsilon), f_\alpha(p', p''))$ in η. We tag this node by the letter υ that was used to satisfy the transition. We use these tags in order to build the run of $\mathcal{P}$. When handling advancing nodes we update the location on the tree Υ^* according to the tag. For an advancing node x we denote by $tag(x)$ the letter in

build_run $(x, r'(x) = (p, d, \alpha), w, add_l, add_r)$

if (advancing(x))
  $w := tag(x) \cdot w$;
if (add_l)
  $r := r \cdot (w, p)$;
if (x has one son $x \cdot a$)
  build_run $(x \cdot a, r'(x \cdot a), w, 1, 0)$
if (x has two sons $x \cdot a$ and $x \cdot b$)
  build_run $(x \cdot a, r'(x \cdot a), w, 0, 1)$
  build_run $(x \cdot b, r'(x \cdot b), w, 0, 0)$

handle_C_q $(r'(x) = (p', p, d, \alpha), q, w)$

Let $t_0, \ldots, t_n \in P^+$ be the sequence of ε-transitions connecting p to q
  $r := r \cdot (w, t_1), \cdots, (w, t_{n-1})$

build_run $(x, r'(x) = (p, q, d, \alpha), w, add_l, add_r)$

if (advancing(x))
  $w := tag(x) \cdot w$;
if (add_l)
  $r := r \cdot (w, p)$;
if (x has one non advancing son $x \cdot a$)
  build_run $(x \cdot a, r'(x \cdot a), w, 1, 0)$
if (x has two sons $x \cdot a$ and $x \cdot b$)
  build_run $(x \cdot a, r'(x \cdot a), w, 0, 1)$
  build_run $(x \cdot b, r'(x \cdot b), w, 0, 0)$
if (x has one advancing son $x \cdot a$)
  build_run $(x \cdot a, r'(x \cdot a), w, 1, 1)$
  handle_C_q $(r'(x \cdot a), q, tag(x \cdot a) \cdot w)$
if (add_r)
  $r := r \cdot (w, q)$;

Fig. 1. Converting a run of A into a run of P

Υ that tags it. A node is *non advancing* if the transition from x to $x \cdot a$ results from an atom $(0, r'(x \cdot a))$ that appears in $\eta(r'(x))$.

The function **build_run** uses the variable w to hold the location in the tree $\langle \Upsilon^*, \tau \rangle$. Working on a singleton (p, d, α) the variable add_l is used to determine whether p was already added to the run. Working on a pair (p, q, d, α) the variable add_l is used to determine whether p was already added to the run and the variable add_r is used to determine whether q was already added to the run.

The intuition behind the algorithm is quite simple. We start with a node x labeled by a singleton (p, d, α). If the node is advancing we update w by $tag(x)$. Now we add p to r (if needed). The case where x has one son matches a transition of the form $(\Delta, p') \in \delta(p, L_\tau(d))$. In this case we move to handle the son of x and clearly p' has to be added to the run r. In case $\Delta = \varepsilon$ the son of x is non advancing and p' reads the same location w. Otherwise, w is updated by Δ and p' reads $\Delta \cdot w$. The case where x has two sons matches a guess that there is another visit to w. Thus, the computation splits into two sons (p, q, d, β) and (q, d, β). Both sons are non advancing. The state p was already added to r and q is added to r only in the first son.

With a node x labeled by a pair (p, q, d, α), the situation is similar. The case where x has one non advancing son matches a transition of the form $(\epsilon, s') \in \delta(p, A)$. Then we move to the son. The state p' is added to r but q is not. The case where x has two non advancing sons matches a split to (p, p', d, α_1) and (p', q, d, α_2). Only p' is added to r as p and q are added by the current call to build_run or by an earlier call to build_run. The case where x has one advancing son matches the move to the state $(p', q', \rho_\tau(d, \upsilon), \alpha)$ and checking that $q' \in C_q^{L_\tau(\rho_\tau(d,\upsilon))}$. Both p' and q' are added to r and **handle_**C_q handles the sequence of ε transitions that connects q' to q.

It is quite simple to see that the resulting run is a valid and accepting run of $\mathcal{P}$ on $\langle \Upsilon^*, \tau \rangle$. □

B Lower Bound on Emptiness of 2NBP

We give the full details of the construction in the proof of Theorem 13 Formally, $\mathcal{P} = \langle \Sigma, P, \delta, p_0, F \rangle$ where

- $\Sigma = \{0, 1, \perp\} \times (\{\sharp\} \cup \Gamma \cup (S \times \Gamma))$.
  Thus, the letters are pairs consisting of a direction and either a $\sharp$, a tape symbol of M, or a tape symbol of M marked by a state of M.
- $P = F \cup B \cup I \cup \{acc\}$ where F is the set of forward states, B is the set of backward states, and I is the set of states that check that the tree starts from the initial configuration of M. All three sets are defined formally below. The state acc is an accepting sink.
- $F = \{acc\}$.

The transition function δ and the initial state p_0 are described below.

We start with forward mode. In forward mode, every state is flagged by either l or r, signaling whether the next configuration to be checked is the left successor or the right successor of the current configuration. The 2NBP starts by memorizing the current location it is checking and the environment of this location (that is for checking location i, memorize the letters in locations $i-1$, i, and $i+1$). For checking the left (resp. right) successor it continues $f(n) - i$ steps in direction 0 then it progresses one step in direction 0 (resp. 1) and then takes i steps in direction 0. Finally, it checks that the letter it is reading is indeed the $next_l$ (resp. $next_r$) successor of the memorized environment. It then goes $f(n) - 1$ steps back, increases the location that it is currently checking and memorizes the environment of the new location. It continues zigzagging between the two configurations until completing the entire configuration and then it starts checking the next.

Thus, the forward states are $F = \{f\} \times \{l, r\} \times [f(n)] \times V^3 \times [f(n)] \times \{x, v\} \times \{0, 1, \perp\}$. Every state is flagged by f and either r or l (next configuration to be checked is either right or left successor). Then we have the current location $i \in [f(n)]$ we are trying to check, the environment $(\sigma, \sigma', \sigma'') \in V^3$ of this location. Then a counter for advancing $f(n)$ steps. Finally, we have x for *still-checking* and v for *checked* (and going backward to the next letter). We also memorize the direction we went to in order to check that every node is labeled by its direction (thus, we have 0 or 1 for forward moves and $\perp$ for backward moves).

The transition of these states is as follows.

- For $0 \leq i \leq f(n)$ and $0 \leq j < f(n)$ we have
  $\delta(\langle f, d, i, \sigma, \sigma', \sigma'', j, x, \Delta \rangle, \langle \Delta, \sigma''' \rangle) =$

$$\begin{bmatrix} \{(1, \langle f, d, i, \sigma, \sigma', \sigma'', j+1, x, 1\rangle\} & \text{if } i + j = f(n) \text{ and } d = r \\ \{(0, \langle f, d, i, \sigma, \sigma', \sigma'', j+1, x, 0\rangle\} & \text{otherwise} \end{bmatrix}$$

  Continue going forward while increasing the counter. If reached the end of configuration and next configuration is the right configuration go in direction 1. Otherwise go in direction 0.

– For $0 \leq i \leq f(n)$ we have
$\delta(\langle f, d, i, \sigma, \sigma', \sigma'', f(n), x, \Delta\rangle, \langle \Delta, \sigma'''\rangle) =$

$$\begin{bmatrix} \emptyset & \text{if } \sigma''' \neq next_d(\sigma, \sigma', \sigma'') \\ \{(\uparrow, \langle f, d, (i+1)_{f(n)}, \sigma', \sigma'', \bot, f(n)-1, v, \bot\rangle)\} & \text{if } \sigma''' = next_d(\sigma, \sigma', \sigma'') \end{bmatrix}$$

If σ''' is not the $next_d$ letter, then abort. Otherwise, change the mode to v and start going back. Push σ' and σ'' to the first two memory locations and empty the third memory location.

– For $0 \leq i \leq f(n)$ and $1 < j \leq f(n)$ we have
$\delta(\langle f, d, i, \sigma, \sigma', \bot, j, v, \bot\rangle, \langle \Delta, \sigma''\rangle) = \{(\uparrow, \langle f, d, i, \sigma, \sigma', \bot, j-1, v, \bot\rangle)\}$.
Continue going backward while updating the counter.

– For $0 \leq i \leq f(n)$ we have
$\delta(\langle f, d, i, \sigma, \sigma', \bot, 1, v, \bot\rangle, \langle \Delta, \sigma''\rangle) =$

$$\begin{bmatrix} \{(\uparrow, \langle b_\forall, \bot, x\rangle)\} & \text{if } \sigma'' \in F_a \times \Gamma \\ \{(\uparrow, \langle b_\exists, \bot, x\rangle)\} & \text{if } \sigma'' \in F_r \times \Gamma \\ \{(\epsilon, \langle f, d, i, \sigma, \sigma', \sigma'', 0, x, \bot\rangle)\} & \text{otherwise} \end{bmatrix}$$

Stop going backward. If the configuration that is checked is either accepting or rejecting go to backward mode (recall that the configuration is already verified as the correct successor of the previous configuration). Otherwise memorize the third letter of the environment and initialize the counter to 0.

– $\delta(\langle f, d, 0, \sharp, \sharp, \bot, 0, x, \Delta\rangle, \langle \Delta, \sigma\rangle) = \{(0, \langle f, d, 0, \sharp, \sharp, \sigma, 1, x, 0\rangle)\}$

This is the first forward state after backward mode and after the initial phase. It starts checking the first letter of the configuration. The 2NBP already knows that the letter it has to check is $\sharp$, it memorizes the current letter (the third letter of the environment) and moves forward while updating the counter.

Note that also the first letter is marked as $\sharp$, this is because when checking location 0 of a configuration we are only checking that the length of the configuration is $f(n)+1$ and that after $f(n)+1$ locations there is another $\sharp$.

Backward mode (either universal or existential) is again flagged by l or r, signaling whether the last configuration the 2NBP saw was the left or right successor. Backward mode starts in a node labeled by a state of M. As the 2NBP goes backward, whenever it passes a $\sharp$ it memorizes its direction. When the 2NBP gets again to a letter that is marked with a state of M, if the memorized direction is l and the type of the state the 2NBP is reading matches the type of backward mode (universal state of M and backward universal or existential state of M and backward existential) then the 2NBP continues going up until the $\sharp$, then it moves to forward mode again (marked by r). Otherwise (i.e. if the memorized direction is r or the type of the state the 2NBP is reading does not match the type of backward mode) then the 2NBP stays in backward mode, when it passes the next $\sharp$ it memorizes the current direction, and goes on moving backward. When returning to the root in backward existential mode, this means that the 2NBP is trying to find a new pruning tree. As no such pruning tree exists the 2NBP rejects. When returning to the root in backward universal mode, this means that all universal choices of the currently explored pruning tree were checked and found accepting. Thus, the pruning tree is accepting and the 2NBP accepts.

The set of backward states is $B = \{b_\forall, b_\exists\} \times \{l, r, \bot\} \times \{x, v\}$. Every state is flagged by $\forall$ (for universal) or $\exists$ (for existential) and by either l or r (the last configuration seen is left successor or right successor, or $\bot$ for unknown). Finally, every state is flagged by either x or v. A state marked by v means that the 2NBP is about to move to forward mode and that it is just going backward until the $\sharp$.

The transition of backward states is as follows.

- $\delta(\langle b_\forall, d, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle b_\forall, l, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \Delta = 0 \\ \{(\uparrow, \langle b_\forall, r, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \Delta = 1 \\ \{(\epsilon, acc)\} & \text{if } \Delta = \bot \\ \{(\uparrow, \langle b_\forall, l, v\rangle)\} & \text{if } \sigma \in S_u \times \Gamma \text{ and } d = l \\ \{(\uparrow, \langle b_\forall, d, x\rangle)\} & \text{otherwise} \end{cases}$

  In backward universal mode reading a $\sharp$ we memorize its direction. If reading the root, we accept. If reading a universal state of M and the last configuration was the left successor then change the x to v. Otherwise, just keep going backward.

- $\delta(\langle b_\exists, d, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle b_\exists, l, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \Delta = 0 \\ \{(\uparrow, \langle b_\exists, r, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \Delta = 1 \\ \emptyset & \text{if } \Delta = \bot \\ \{(\uparrow, \langle b_\exists, l, v\rangle)\} & \text{if } \sigma \in S_e \times \Gamma \text{ and } d = l \\ \{(\uparrow, \langle b_\forall, d, x\rangle)\} & \text{otherwise} \end{cases}$

  In backward existential mode reading a $\sharp$ we memorize its direction. If reading the root, we reject. If reading an existential state of M and the last configuration was the left successor then change x to v. Otherwise, just keep going backward.

- $\delta(\langle b, l, v\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle b, l, v\rangle)\} & \text{if } \sigma \neq \sharp \\ \{(\varepsilon, \langle f, r, 0, \sharp, \sharp, \bot, 0, x, 0\rangle)\} & \text{if } \sigma = \sharp \end{cases}$

  In backward mode marked by v we go backward until we read $\sharp$. When reading $\sharp$ we return to forward mode. The next configuration to be checked is the right successor. The location we are checking is location 0, thus the letter before is not interesting and is filled by $\sharp$. The counter is initialized to 0.

Finally, the set I of 'initial' states makes sure that the first configuration in the tree is indeed $\sharp \cdot (s_0, b) \cdot b^{f(n)-1}$. When finished checking the first configuration $\mathcal{S}$ returns to the node 0 and moves to forward mode.

Formally, $I = \{i\} \times [f(n)] \times \{x, v\}$ with transition as follows.

- $\delta(\langle i, 0, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(0, \langle i, 1, x\rangle)\} & \text{if } \sigma = \sharp \text{ and } \Delta = \bot \\ \emptyset & \text{otherwise} \end{cases}$

  Make sure that the root is labeled by $\langle \bot, \sharp\rangle$.

- $\delta(\langle i, 1, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(0, \langle i, 2, x\rangle)\} & \text{if } \sigma = (s_0, b) \text{ and } \Delta = 0 \\ \emptyset & \text{otherwise} \end{cases}$

  Make sure that the first letter is (s_0, b).

- For $1 < j < f(n)$ we have

$$\delta(\langle i, j, x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(0, \langle i, j+1, x\rangle)\} & \text{if } \sigma = b \text{ and } \Delta = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

  Make sure that all other letters are b.

- $\delta(\langle i, f(n), x\rangle, \langle \Delta, \sigma\rangle) = \begin{cases} \{(\uparrow, \langle i, f(n)-1, v\rangle)\} & \text{if } \sigma = b \text{ and } \Delta = 0 \\ \emptyset & \text{otherwise} \end{cases}$
  Make sure that the last letter is b. The first configuration is correct, start going back to node 0. Change x to v.
- For $2 < j < f(n)$ we have $\delta(\langle i, j, v\rangle, \langle \Delta, \sigma\rangle) = \{(\uparrow, \langle i, j-1, v\rangle)\}$
  Continue going backward while updating the counter.
- $\delta(\langle i, 2, v\rangle, \langle 0, \sigma\rangle) = \{(\uparrow, \langle f, l, 0, \sharp, \sharp, \bot, 0, x, 0\rangle)\}$.
  Finished checking the first configuration. Go up to node 0 in the first state of forward mode.

Last but not least the initial state is $p_0 = \langle i, 0, x\rangle$.

Finally, we analyze the reduction. Given an alternating Turing machine with n states and alphabet of size m we get a 2NBP with $O(n \cdot m)$ states, that reads an alphabet with $O(n \cdot m)$ letters. The 2NBP is actually deterministic. Clearly, the reduction is polynomial.

We note that instead of checking emptiness of $\mathcal{P}$, we can check the membership of some correct encoding of the run tree of M in the language of $\mathcal{P}$. However, the transducer that generates a correct encoding of M is exponential.

C Lower Bound for Linear Time Model-Checking on Prefix-Recognizable Systems

It was shown by [BEM97] that the problem of model-checking an LTL formula with respect to a pushdown graph is EXPTIME-hard in the size of the formula. The problem is polynomial in the size of the pushdown system inducing the graph. Our algorithm for model-checking an LTL formula with respect to a prefix-recognizable graph is exponential both in the size of the formula and in $|Q_\beta|$.

As prefix-recognizable systems are a generalization of pushdown systems the exponential resulting from the formula cannot be improved. We show that also the exponent resulting from Q_β cannot be removed. We use the EXPTIME-hard problem of whether a linear space alternating Turing machine accepts the empty tape [CKS81]. We reduce this question to the problem of model-checking a fixed LTL formula with respect to the graph induced by a prefix-recognizable system with a constant number of states and transitions. Furthermore Q_α and Q_γ depend only on the alphabet of the Turing machine. The component Q_β does 'all the hard work'. Combining this with Theorem 15 we get the following.

Theorem 26. *The problem of linear-time model-checking the graph induced by the prefix-recognizable system $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0\rangle$ is EXPTIME-complete in $|Q_\beta|$.*

Proof: Let $M = \langle \Gamma, S_u, S_e, \mapsto, s_0, F_{acc}, F_{rej}\rangle$ be an alternating linear-space Turing machine. Let $f : \mathbb{N} \to \mathbb{N}$ be the linear function such that M uses $f(n)$ cells in its working tape in order to process an input of length n. In order to make sure that M does not accept the empty tape, we have to check that every legal pruning of the computation tree of M contains one rejecting branch.

Given such an alternating linear-space Turing machine M, we construct a prefix-recognizable system R and an LTL formula φ such that $G_R \models \varphi$ iff M does not accept the empty tape. The system R has a constant number of states and rewrite rules. For every rewrite rule $\langle q, \alpha_i, \beta_i, \gamma_i, q' \rangle$, the languages of the regular expressions α_i and γ_i are subsets of $\Gamma \cup (\{\downarrow\} \times \Gamma) \cup S \cup \{\epsilon\}$. The language of the regular expression β_i, can be encoded by a nondeterministic automaton whose size is linear in n. The LTL formula φ does not depend on the structure of M.

The graph induced by R has one infinite trace. This trace searches for rejecting configurations in all the pruning trees. The trace first explores the left son of every configuration. If it reaches an accepting configuration, the trace backtracks until it reaches a universal configuration for which only the left son was explored. It then goes forward again and explores under the right son of the universal configuration. If the trace returns to the root without finding such a configuration then the currently explored pruning tree is accepting. Once a rejecting configuration is reached, the trace backtracks until it reaches an existential configuration for which only the left son was explored. It then explores under the right son of the existential configuration. In this mode, if the trace backtracks all the way to the root, it means that all pruning trees were checked and that there is no accepting pruning tree for M.

We change slightly the encoding of a configuration by including with the state of M a symbol l or r denoting whether the next explored configuration is the right or left successor. Let $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma \times \{l, r\})$ and let $\sharp \cdot \sigma_1 \cdots \sigma_{f(n)} \cdot \sharp \sigma_1^d \ldots \sigma_{f(n)}^d$ be a configuration of M and its d-successor (where d is either l or r). We also set σ_0 and σ_0^d to $\sharp$. Given σ_{i-1}, σ_i, and σ_{i+1} we know, by the transition relation of M, what σ_i^d should be. In addition the symbol $\sharp$ should repeat exactly every $f(n)+1$ letters. Let $next : V^3 \to V$ denote our expectation for σ_i^d. Note that whenever the triplet σ_{i-1}, σ_i, and σ_{i+1} does not include the reading head of the Turing machine, it does not matter whether d is l or r. In both cases the expectation for σ_i^d is the same. We set $next(\sigma, \sharp, \sigma') = \sharp$, and

$next(\sigma, \sigma', \sigma'') =$

$$\begin{cases} \sigma' & \text{if } \{\sigma, \sigma', \sigma''\} \subseteq \{\sharp\} \cup \Gamma \\ \sigma' & \text{if } \sigma'' = (s, \gamma, d) \text{ and } (s, \gamma) \to^d (s', \gamma', R) \\ (s', \sigma', d') & \text{if } \sigma'' = (s, \gamma, d),\ (s, \gamma) \to^d (s', \gamma', L), \text{ and } d' \in \{l, r\} \\ \sigma' & \text{if } \sigma = (s, \gamma, d) \text{ and } (s, \gamma) \to^d (s', \gamma', L) \\ (s', \sigma', d') & \text{if } \sigma = (s, \gamma, d),\ (s, \gamma) \to^d (s', \gamma', R), \text{ and } d' \in \{l, r\} \\ \gamma' & \text{if } \sigma' = (s, \gamma, d) \text{ and } (s, \gamma) \to^d (s', \gamma', \alpha) \end{cases}$$

Consistency with $next$ now gives us a necessary condition for a sequence in V^* to encode a branch in the computation tree of M. Note that when $next(\sigma, \sigma', \sigma'') \in S \times \Gamma \times \{l, r\}$ then marking it by both l and r is correct.

The prefix-recognizable system starts from the initial configuration of M. It has two main modes, a *forward* mode and a *backward* mode. In forward mode, the system guesses a new configuration. The configuration is guessed one letter at a time, and this letter should match the functions $next_l$ or $next_r$. If the computation reaches an accepting configuration, this means that the currently explored pruning tree might still be

accepting. The system moves to backward mode and remembers that it should explore other universal branches until it finds a rejecting state. In backward universal mode, the system starts backtracking and removes configurations. Once it reaches a universal configuration that is marked by l, it replaces the mark by r, moves to forward mode, and explores the right son. If the root is reached (in backward universal mode), the computation enters a rejecting sink. If in forward mode, the system reaches a rejecting configuration, then the currently explored pruning tree is rejecting. The system moves to backward mode and remembers that it has to explore existential branches that were not explored. Hence, in backward existential mode, the system starts backtracking and removes configurations. Once it reaches an existential configuration that is marked by l, the mark is changed to r and the system returns to forward mode. If the root is reached (in backward existential mode) all pruning trees have been explored and found to be rejecting. Then the system enters an accepting sink. All that the LTL formula has to check is that there exists an infinite computation of the system and that it reaches the accepting sink. Note that the prefix-recognizable system accepts, when the alternating Turing machine rejects and vice versa.

More formally, the LTL formula is $\Diamond reject$ and the rewrite system is $R = \langle 2^{AP}, V, Q, L, T, q_0, x_0 \rangle$, where

- $AP = \{reject\}$
- $V = \{\sharp\} \cup \Gamma \cup (S \times \Gamma \times \{l, r\})$
- $Q = \{forward, backward_{\exists}, backward_{\forall}, sink_a, sink_r\}$
- $L(q, \alpha) = \begin{cases} \emptyset & \text{if } q \neq sink_a \\ \{reject\} & \text{if } q = sink_a \end{cases}$
- $q_0 = forward$
- $x_0 = b \cdots b \cdot (s_0, b, l) \cdot \sharp$

In order to define the transition relation we use the following languages.

- $L^1_{egal} = \{next(\sigma, \sigma', \sigma'') \cdot V^{f(n)-1} \sigma \cdot \sigma' \cdot \sigma''\}$
  $L^2_{egal} = \{w \in V^{f(n)+1} \mid w \notin V^* \cdot \sharp \cdot V^* \cdot \sharp \cdot V^*\}$
  $L^3_{egal} = \{w \in V^{f(n)+1} \mid w \notin V^* \cdot (S \times \Gamma \times \{l, r\}) \cdot V^* \cdot (S \times \Gamma \times \{l, r\}) \cdot V^*\}$
  $L_{egal} = (L^1_{egal} \cap L^2_{egal} \cap L^3_{egal}) \cdot V^*$
  Thus, this language contains all words whose suffix of length $f(n) + 1$ contains at most one $\sharp$ and at most one symbol from $S \times \Gamma \times \{l, r\}$ and the last letter is the $next$ correct successor of the previous configuration.
- $A_{ccept} = V \cdot (\{F_{acc}\} \times \Gamma \times \{l, r\}) \cdot V^*$
  Thus, this language contains all words whose one before last letter is marked by an accepting state[14].
- $R_{eject} = V \cdot (\{F_{rej}\} \times \Gamma \times \{l, r\}) \cdot V^*$
  Thus, this language contains all words whose one before last letter is marked by a rejecting state.

[14] It is important to use the one before last letter so that the state itself is already checked to be the correct next successor of previous configuration.

- $R_{emove}^{S_u \times \{l\}} = V \setminus (S_u \times \Gamma \times \{l\})$
  Thus, this language contains all the letters that are not marked by universal states and the direction l.
- $R_{emove}^{S_e \times \{l\}} = V \setminus (S_e \times \Gamma \times \{l\})$.

  Thus, this language contains all the letters that are not marked by existential states and the direction l.

Clearly the languages L_{egal}, A_{ccept}, and R_{eject} can be accepted by nondeterministic automata whose size is linear in $f(n)$.

The transition relation includes the following rewrite rules:

1. $\langle forward, \{\epsilon\}, L_{egal}, V \setminus (S \times \Gamma \times \{r\}), forward\rangle$ - guess a new letter and put it on the store. States are guessed only with direction l. The fact that L_{egal} is used ensures that the currently guessed configuration (and in particular the previously guessed letter) is the successor of the previous configuration on the store.
2. $\langle forward, \{\epsilon\}, A_{ccept}, \{\epsilon\}, backward_\forall\rangle$ - reached an accepting configuration. Do not change the store and move to backward universal mode.
3. $\langle forward, \{\epsilon\}, R_{eject}, \{\epsilon\}, backward_\exists\rangle$ - reached a rejecting configuration. Do not change the store and move to backward existential mode.
4. $\langle backward_\forall, R_{emove}^{S_u \times \{l\}}, V^*, \{\epsilon\}, backward_\forall\rangle$ - remove one letter that is not in $S_u \times \Gamma \times \{l\}$ from the store.
5. $\langle backward_\forall, S_u \times \Gamma \times \{l\}, V^*, S_u \times \Gamma \times \{r\}, forward\rangle$ - replace the marking l by the marking r and move to forward mode. The state s does not change[15].
6. $\langle backward_\forall, \epsilon, \epsilon, \epsilon, sink_r\rangle$ - when the root is reached in backward universal mode enter the rejecting sink
7. $\langle backward_\exists, R_{emove}^{S_e \times \{l\}}, V^*, \{\epsilon\}, backward_\exists\rangle$ - remove one letter that is not in $S_e \times \Gamma \times \{l\}$ from the store.
8. $\langle backward_\exists, S_e \times \Gamma \times \{l\}, V^*, S_e \times \Gamma \times \{r\}, forward\rangle$ - replace the marking l by the marking r and move to forward mode. The state s does not change.
9. $\langle backward_\exists, \epsilon, \epsilon, \epsilon, sink_a\rangle$ - when the root is reached in backward existential mode enter the accepting sink.
10. $\langle sink_a, \epsilon, \epsilon, \epsilon, sink_a\rangle$ - remain in accepting sink.
11. $\langle sink_r, \epsilon, \epsilon, \epsilon, sink_r\rangle$ - remain in rejecting sink.

□

[15] Actually, we guess all states in S_u. As we change state into $forward$, the next transition verifies that indeed the state is the same state.

On the Krohn-Rhodes Cascaded Decomposition Theorem

Oded Maler

CNRS-VERIMAG, 2 Av. de Vignate
38610 Gières, France

Dedicated to the memory of Amir Pnueli, deeply missed.

Abstract. The Krohn-Rhodes theorem states that any deterministic automaton is a homomorphic image of a cascade of very simple automata which realize either resets or permutations. Moreover, if the automaton is counter-free, only reset automata are needed. In this paper we give a very constructive proof of a variant of this theorem due to Eilenberg.

1 Introduction

More than 20 years ago my PhD advisor Amir Pnueli convinced me to postpone some dead-ends I was pursuing around the middle of my thesis and look at the Krohn-Rhodes decomposition theorem. His correct intuition was that this theorem can help in establishing a lower-complexity translation from automata to temporal logic. The best known complexity at that time was non-elementary [6], based on a series of transformation adapted from the monograph by McNaughton and Papert [9] which dealt with different characterizations (logical, algebraic, language-theoretic, automatic) of the same class of objects, the *star-free regular sets* [13].

The result of Kenneth Krohn and John Rhodes, announced almost 50 years ago [12,5], states that any *deterministic* automaton can be decomposed into a cascade of simple automata, whose structure reflects the algebraic structure of the transformation semigroup associated with the automaton. For some time in the 60s and 70s, their theorem, which got them 2 simultaneous PhD titles from Harvard and MIT, respectively, was considered to be a cornerstone of automata theory. When I started to look at at the topic in the late 80s the results have been practically forgotten in the Computer Science mainstream, excluding some specialized islands.

Although I started to acquaint myself with the algebraic (and French) vocabulary of transformation semigroups, it was not easy for me to understand the purely-algebraic versions of the theorem expressed in terms of *wreath product* of semigroups or groups. Fortunately, the book of Ginzburg [2] gave a clear automata-theoretic presentation from which one could understand that a cascade of automata is a particular type of composition where the automata are ordered and each automaton reads the common input alphabet *and* the states of its preceding automata or, equivalently, the *output* of its predecessor, as illustrated in Fig. 1. The theorem states that any automaton, up to homomorphism, can be realized by a cascade of elementary automata of two types, *permutation*

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 260–278, 2010.

automata where each letter induces a permutation of the state space, and *reset* automata where each letter is either a reset (sends all states to a fixed single state) or an identity (a self-loop from every state). However, as noted in the last paragraph of [2] *"Finally, notice that the above theory does not indicate how many particular basic building blocks are needed to construct a cascade product covering of a given semiautomaton."*

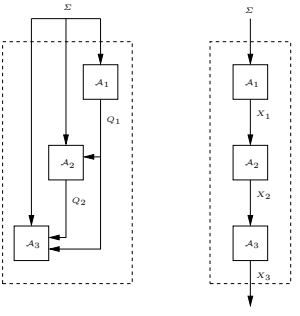


Fig. 1. A cascade product of 3 automata using the state-as-input convention (left) and the input-output convention (right)

I had the privilege to discuss the topic with the late Marcel-Paul Schützenberger who encouraged me to look at the *holonomy* decomposition theorem, a variant of the Krohn-Rhodes theorem, written on eight dense pages of volume B of Eilenberg's book [1] to which he contributed. It took me a *long* time to decipher this motivation-less algebraic prose, translate the construction to my own automata-theoretic language, and verify that it is indeed exponential. From there an exponential translation from counter-free automata to temporal logic, very similar to the construction of Meyer [10] for star-free regular expressions, followed immediately. We also managed to give a lower bound on the size of the decomposition, obtained via a bounded two-way counter, also known as the *elevator* automaton. Apart from a short abstract [7] and a draft on the web [8] we have not published this work, which is what I intend to do presently. Unfortunately due to timing constraints the presentation is not complete, including only the reconstruction of the holonomy decomposition without the lower bound and the translation to temporal logic. The interested reader is referred to [7,8] for those.

The rest of the paper is organized as follows. In Section 2 we give the basic definitions concerning the algebraic theory of automata and semigroups. In section 3 we define the cascade product and state the theorem. Section 4 is devoted to the study of a particular structure, the holonomy tree, tightly related to a cascaded decomposition. It is a combination of a tree whose nodes are labeled by subsets of the states of the automaton, and on which a transition function, satisfying certain constraints is defined. After establishing the close relationship between such a tree and a cascaded decomposition we describe in Section 5 an algorithm for computing the tree and thus completing a constructive version of the proof. The subtle part in these two sections is how to avoid introducing *spurious permutations* and to assure that if the automaton is counter-free the decomposition will consist exclusively of reset automata.

2 Preliminaries

A total function f from X to Y is an *injection* if $f(x) \neq f(x')$ whenever $x \neq x'$. It is a *surjection* if $\forall y \exists x \ f(x) = y$ and a *bijection* if it is both an injection and a surjection. The latter case implies $|X| = |Y|$ as well as the existence of an inverse function $f' : Y \to X$.

2.1 Automata

We assume familiarity with finite automata and regular sets at the level of [4]. We use Σ^* to denote the set of finite sequences over an alphabet Σ and use ϵ for the empty sequence.

Definition 1 (Automaton). *A deterministic automaton is triple $\mathcal{A} = (\Sigma, Q, \delta)$ where Σ is a finite set of symbols called the input alphabet, Q is a finite set of states and $\delta : Q \times \Sigma \to Q$ is the transition function. A partial automaton is such where δ may be undefined for some combinations of states and symbols.*

The transition function can be lifted naturally to sets of states, by letting $\delta(P, \sigma) = \{\delta(q, \sigma) : q \in P\}$, and to input sequences, by letting $\delta(q, w\sigma) = \delta(\delta(q, w), \sigma)$.

An automaton can be made an *acceptor* by choosing an initial state $q_0 \in Q$ and a set of accepting states $F \subseteq Q$. As such it accepts/recognizes a set of sequences, also known as a *language*, defined as $L(\mathcal{A}) = \{w : \delta(q_0, w) \in F\}$. Kleene's Theorem states that the class of languages recognizable by finite automata coincides with the *regular* languages. A subclass of the regular sets is the class of *star-free* sets defined as:

Definition 2 (Star-Free Regular Sets). *The class of star-free regular sets over Σ is the smallest class containing Σ^* and the sets of the form $\{\sigma\}$ where $\sigma \in \Sigma \cup \{\epsilon\}$, which is closed under finitely many applications of concatenation and Boolean operations.*

Star-free sets have additional characterizations to be discussed in the sequel.

Definition 3 (Automaton Homomorphism). *A surjection $\varphi : Q \to Q'$ is an automaton homomorphism from $\mathcal{A} = (\Sigma, Q, \delta)$ to $\mathcal{A}' = (\Sigma, Q', \delta')$ if for every $q \in Q$, $\sigma \in \Sigma$*

$$\varphi(\delta(q, \sigma)) = \delta'(\varphi(q), \sigma)$$

In such a case we say that $\mathcal{A}'$ is homomorphic to $\mathcal{A}$ and denote it by $\mathcal{A}' \leq_\varphi \mathcal{A}$. When φ is a bijection, $\mathcal{A}$ and $\mathcal{A}'$ are said to be isomorphic.

Intuitively $\mathcal{A}' \leq_\varphi \mathcal{A}$ means that $\mathcal{A}'$ is an abstraction of $\mathcal{A}$ and anything that can be expressed using $\mathcal{A}'$ can be expressed, possibly in more detail using $\mathcal{A}$. Homomorphism is transitive and induces a partial-order relation among automata.

2.2 Semigroups

The theory of automata is strongly related to the algebraic theory of *semigroups* dealing with sets closed under an associative (but not necessarily invertible) binary operation.

Two typical examples of semigroups are *sequences* of symbols under the *concatenation* operation and *transformations* (functions from a set to itself) under *function composition*. In fact, the theory of formal languages and automata is, to some extent, a theory about the relation between these two semigroups.

Definition 4 (Semigroups, Monoids and Groups). *A Semigroup is a pair $(S, \cdot)$ where S is a set and $\cdot$ is a binary associative operation ("multiplication") from $S \times S$ to S. A Monoid $(S, \cdot, 1)$ is a semigroup admitting an identity element 1 such that $s \cdot 1 = 1 \cdot s = s$ for every $s \in S$. A group is a monoid such that for every $s \in S$ there exists an element $s^{-1} \in S$ (an inverse) such that $s \cdot s^{-1} = 1$.*

Definition 5 (Subsemigroups, Generators). *A subsemigroup T of S is a subset $T \subseteq S$ which is closed under multiplication, that is, $T^2 \subseteq T$. A subgroup of S is a subsemigroup which is a group. The smallest subsemigroup of S containing a subset $A \subseteq S$ is denoted by A^+ and it consists of all elements of S obtained by finitely many products of elements of A. Any subset $A \subseteq S$ such that $A^+ = S$ is called a generating set of S.*

A finite semigroup can be described by its multiplication table. The trivial semigroup consisting of the singleton set $\{e\}$ is of course a monoid and a group. In the sequel we will not make a distinction between a semigroup and a monoid. As with automata, one can define semigroup homomorphism which is transitive and corresponds to the intuitive notions of refinement/abstraction among structures.

Definition 6 (Semigroup Homomorphisms). *A surjective function $\varphi : S \to S'$ is a semigroup homomorphism from $(S, \cdot)$ to $(S', *)$ if for every $s_1, s_2 \in S$,*

$$\varphi(s_1 \cdot s_2) = \varphi(s_1) * \varphi(s_2)$$

In such a case we say that S' is homomorphic to S and denote it by $S' \leq_\varphi S$. Two mutually homomorphic semigroups are said to be isomorphic.

Let $\mathrm{TR}(Q)$ be the set of all total functions (*transformations*) of the form $s : Q \to Q$ over a finite set Q, $|Q| = n$. One can see that $\mathrm{TR}(Q)$ is a monoid of n^n elements under the operation of *function composition* defined as $s \cdot t(q) = t(s(q))$ for every $q \in Q$. The identity function on Q, 1_Q, is the identity element of $\mathrm{TR}(Q)$. A transformation can be represented as an n-tuple $(q_{i_1}, \ldots, q_{i_n})$ where $q_{i_j} = s(q_j)$.
Remark: There is some conflict between algebraic, functional, and automata-theoretic notational conventions. Algebraically, the *action* of s on q is denoted by qs and the associativity of composition is expressed as $(qs)t = q(s \cdot t)$. On the other hand, the automata-theoretic notation $\delta(q, s)$ is preferable when we have to refer to *several* transition functions. We will try not to confuse the reader.

Definition 7 (Transformation Semigroups). *A transformation semigroup is a pair $X = (Q, S)$ where Q is the underlying set and S is a subsemigroup of $\mathrm{TR}(Q)$, that is, a set of transformations on Q closed under composition. Clearly if Q is finite, so is S.*

The importance of transformation semigroups as more concrete representations of abstract semigroups comes from the following theorem:

Theorem 1 (Cayley). *Every semigroup is isomorphic to a transformation semigroup.*

On the other hand, every automaton gives rise to a transformation semigroup $X_{\mathcal{A}}$ whose generators are the transformations $\{s_\sigma\}_{\sigma\in\Sigma}$ induced by input letters. The following definition gives an intermediate representation of this semigroup.

Definition 8 (Expanded Automaton). *Let $\mathcal{A} = (\Sigma, Q, \delta)$ be an automaton and let $X_{\mathcal{A}} = (Q, S)$ be its transformation semigroup. The expansion of $\mathcal{A}$ is the automaton $\hat{\mathcal{A}} = (S, Q, \delta)$ with $\delta(q, s) = q \cdot s$.*

It can be shown that the existence of a homomorphism between two automata implies the existence of a homomorphism between their corresponding transformation semigroups. On the other hand, a homomorphism from $X = (Q, S)$ to $X' = (Q', S')$ can be obtained without an automaton state-homomorphism, just by taking $Q' \subseteq Q$ and letting S' be the set of transformation on Q' obtained from transformations in S by projection (which consitutes the semigroup homomorphism from S to S'). Mechanically this semigroup can be computed by constructing $\hat{\mathcal{A}} = (S, Q, \delta)$ and then restricting it to Q' and to an alphabet $S' \subseteq S$ consisting of all transformations satisfying $\delta(Q', s) \subseteq Q'$.

Definition 9 (Rank). *The rank of a transformation $s \in \mathrm{TR}(Q)$ is defined as the cardinality of its range $Qs = \{qs : q \in Q\}$.*

Permutations and resets (see Fig. 2) represent two extreme types of transformations in terms of rank. The $n!$ permutations are those in which the domain and the range coincide and the rank is n while the n resets are the constant transformations of rank 1.

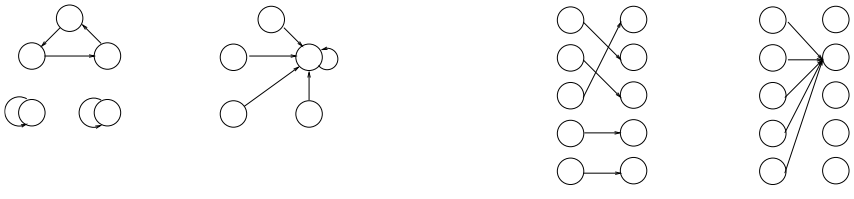


Fig. 2. A permutation and a reset illustrated as transition graphs (left) and as transformations (right)

It is worth looking at the effect of resets and permutations from the following angle, emphasizing what is known about the state of the automaton upon the occurrence of a generic transition $q' = \delta(q, \sigma)$. If σ is a reset we do not need to know q in order to determine q', however knowing q' we *cannot* determine q. On the other hand if σ is a permutation we know nothing about q' if we do not know what q was, but if we know q', q is uniquely determined. In other words, a permutation is reverse-deterministic, while in resets the degree of reverse non-determinism is maximal.

Permutations and resets are closed under composition or more precisely, if we denote a reset by R and a permutation by P we get the following multiplication table:

$*$	P	R
P	P	R
R	R	R

Resets can be obtained by composing non-reset transformations, for example, $(122) \cdot (223) = (222)$, because composition can decrease the rank. On the other hand, because composition cannot increase the rank, a permutation on Q cannot be composed from non-permutations on Q. However a permutation on a subset $R \subseteq Q$ can be composed from non-permutations as can be seen from Fact 1.

Fact 1. *A transformation s permutes a subset $R \subseteq Q$ iff $s = s_1 \cdots s_m$ for some $m > 0$ and there exists a sequence of subsets $\{R_j\}_{j=0..m}$ such that $R_0 = R_m = R$ and the restriction of every s_j to R_j is an injection to R_{j+1}.*

There are various ways to classify finite semigroups and their corresponding automata and regular sets [11]. An important sub-class of semigroups is defined as follows:

Definition 10 (Group-Free Semigroups). *A semigroup S is aperiodic if there exists a number k such that $s^k = s^{k+1}$ for every element $s \in S$. A semigroup is group-free if it has no non-trivial subgroups. An automaton is counter-free if no word induces a permutation other than identity on any subset of Q.*

A semigroup is aperiodic iff it is group-free and an automaton is counter-free iff its transformation semigroup is group-free. The following theorem relates these objects to star-free sets and, consequently, to propositional temporal logic.

Theorem 2 (Schützenberger). *A regular set U is star-free if and only if its syntactic semigroup is aperiodic.*

The syntactic semigroup of a language is the transformation semigroup of the minimal *deterministic* automaton which recognizes it. This automaton is unique and, following the theorem, it is counter-free if the language is star-free.

3 The Krohn-Rhodes Primary Decomposition Theorem

The definition of the cascade product of two or more automata is given below:

Definition 11 (Cascade Product). *Let $\mathcal{B}_1 = (\Sigma, Q_1, \delta_1)$ be an automaton, and let $\mathcal{B}_2 = (Q_1 \times \Sigma, Q_2, \delta_2)$ be a (possibly partial) automaton such that for every $q_1 \in Q_1$ and $q_2 \in Q_2$, either $\delta_2(q_2, \langle q_1, \sigma \rangle)$ is defined for every $\sigma \in \Sigma$ or it is undefined for every σ. The cascade product $\mathcal{B}_1 \circ \mathcal{B}_2$ is the automaton $\mathcal{C} = (\Sigma, P, \bar{\delta})$ where*

$$P = \{(q_1, q_2) : \delta_2(q_2, \langle q_1, \sigma \rangle) \text{ is defined}\}$$

and

$$\bar{\delta}(\langle q_1, q_2 \rangle, \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \langle q_1, \sigma \rangle)).$$

The cascade product of more than two automata is defined as

$$\mathcal{B}_1 \circ \mathcal{B}_2, \ldots \circ \mathcal{B}_k = (\ldots((\mathcal{B}_1 \circ \mathcal{B}_2) \circ \mathcal{B}_3 \ldots) \circ \mathcal{B}_k.$$

Fig. 3 shows a cascade product of two automata. Note that the communication links between $\mathcal{B}_1$ and $\mathcal{B}_2$ are given implicitly via the definition of the input alphabet of $\mathcal{B}_2$ as the product of Σ and the state-space of $\mathcal{B}_1$. A alternative definition using transducers (Mealy machines) is possible as illustrated in Fig. 1.

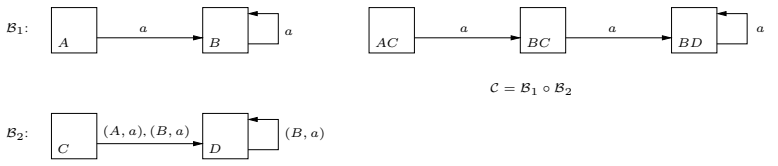


Fig. 3. A cascade $\mathcal{C} = \mathcal{B}_1 \circ \mathcal{B}_2$

Definition 12 (Permutation-Reset Automata). *A (potentially partial) automaton $\mathcal{A} = (\Sigma, Q, \delta)$ is a permutation-reset automaton if for every letter $\sigma \in \Sigma$, σ is either a permutation or reset with respect to the set of states on which it is defined.*[1] *If the only permutations are identities, we call it a reset automaton.*

The Krohn-Rhodes theorem states that one can realize any automaton (up to homomorphism) as a cascade of permutation-reset automata and that non-trivial permutations are required only if the transformation semigroup of the automaton admits non-trivial subgroups. Based on the Jordan-Hölder Theorem, the groups can be decomposed further into a cascade of *simple* groups but we will not be concerned much with the group part of the theorem beyond guaranteeing that it vanishes for counter-free automata. The original formulation of the theorem was stated in terms of semigroups and its automata-theoretic version can be phrased as follows.

Theorem 3 (Krohn-Rhodes: Automata). *For every automaton $\mathcal{A}$ there exists a cascade $\mathcal{C} = \mathcal{B}_1 \circ \mathcal{B}_2 \circ \cdots \circ \mathcal{B}_k$ such that:*

1. *Each $\mathcal{B}_i$ is a permutation-reset automaton;*
2. *There is a homomorphism φ from $\mathcal{C}$ to $\mathcal{A}$;*
3. *Any permutation group in some $\mathcal{B}_i$ is homomorphic to a subgroup of the transformation semigroup of $\mathcal{A}$.*

The pair $(\mathcal{C}, \varphi)$ is called a cascaded decomposition of $\mathcal{A}$.

The third condition implies that if $\mathcal{A}$ is counter-free then each $\mathcal{B}_i$ is a reset automaton. It is this theorem that we are going to prove in constructive detail in the sequel. We sometimes assume an additional trivial one-state automaton $\mathcal{B}_0$ composed in front of the cascade. We will often use a notation of the form $\langle p, q_i \rangle$ for $\langle q_1, q_2, \ldots, q_{i-1}, q_i \rangle$.

4 Structures Associated with a Cascade

Let $\mathcal{C} = (\Sigma, P, \bar{\delta}) = \mathcal{B}_1 \circ \cdots \circ \mathcal{B}_k$ be a cascade, and let φ be a homomorphism from $\mathcal{C}$ to an automaton $\mathcal{A} = (\Sigma, Q, \delta)$. Let $\mathcal{C}_i = (\Sigma, P_i, \bar{\delta}_i) = \mathcal{B}_1 \circ \cdots \circ \mathcal{B}_i$ be the product of the first i components, $i \leq k$. Elements of P_i are called i-configurations and they admit a natural hierarchical structure, the *configuration tree*, where each i-configuration $p_i = \langle p_{i-1}, q_i \rangle$ is an extension of a *parent* configuration p_{i-1}. We associate a family

[1] Partial resets and partial permutations can be completed to full ones by appropriately defining the missing transitions.

of mappings $\varphi_i : P_i \to 2^Q$ indicating for each configuration which states of Q are encoded by its extensions, that is,

$$\varphi_k(p) = \{\varphi(p)\}$$

and

$$\varphi_{i-1}(p) = \bigcup_{\langle p,q\rangle \in P_i} \varphi_i(\langle p, q\rangle).$$

A decomposition is *redundant* if there are two i-configurations $\langle p, q\rangle$ and $\langle p, q'\rangle$ such that $\varphi_i(\langle p, q'\rangle) \subseteq \varphi_i(\langle p, q\rangle)$. In this case we can remove configuration $\langle p, q'\rangle$ by letting $\delta_i(q', \langle p, \sigma\rangle)$ be undefined and redirecting all transitions entering q' to q. The restriction of φ_k to the remaining configurations, those which are not extensions of $\langle p, q'\rangle$, still covers the whole Q and is a homomorphism. Repeating this procedure until all redundancies are removed we can conclude that the existence of a decomposition is equivalent to the existence of a non-redundant decomposition.

The hierarchical relation between cascade configurations and subsets of Q motivates the following definition.

Definition 13 (Subset Transition Tree). *A subset transition tree (STT) for an automaton $\mathcal{A} = (\Sigma, Q, \delta)$ is a tuple $\mathcal{T} = (\Sigma, M, \Delta, \pi, \phi)$ where:*

$M = M_0 \uplus \ldots \uplus M_k$ *is a set of nodes partitioned into levels, with $M_0 = \{m_*\}$;*

- *$\Delta = \Delta_0 \uplus \cdots \uplus \Delta_k$ is a transition function with $\Delta_i : M_i \times \Sigma \to M_i$. Each level can be viewed as an automaton $N_i = (\Sigma, M_i, \Delta_i)$;*
- *$\pi : M - \{m_*\} \to M$ is a parenthood function, mapping every element of M_i to an element of M_{i-1}. We use Π_m to denote the set of sons of a node m, whose elements are called brothers;*
- *The transition function is ancestor-preserving: $\pi(\Delta(m, \sigma) = \Delta(\pi(m), \sigma)$;*
- *The action of every letter on any set Π_m of brothers is either a reset or an injection.*
- *$\phi : M \to 2^Q$ is a function mapping nodes to sets of states whose restriction to M_i is denoted by ϕ_i and which satisfies:*
  - *ϕ_k maps the leaves of the tree to singletons and constitutes a homomorphism from $N_k = (\Sigma, M_k, \Delta_k)$ to $\mathcal{A}$;*
  - *For every $i < k$*

$$\phi_i(m) = \bigcup_{m' \in \Pi_m} \phi_{i+1}(m').$$

  - *No redundancy: $\phi(m) \not\subseteq \phi(m')$ for any pair of brothers.*

Next we prove a weak version of the fundamental fact underlying the decomposition. It is weak because it speaks of a decomposition satisfying only conditions 1-2 of Theorem 3.

Proposition 1. *There exists a cascade decomposition $\mathcal{C} = \mathcal{B}_1 \circ \cdots \circ \mathcal{B}_k \leq_\varphi \mathcal{A}$ with each $\mathcal{B}_i$ being a permutation-reset automaton, iff there exists an STT $\mathcal{T}$ for $\mathcal{A}$ which is isomorphic to the configuration tree.*

Proof. The construction of the STT from the configuration tree of the cascade is straightforward, obtained by letting $M_i = P_i$ and $\Delta_i = \bar{\delta}_i$. The mapping of nodes to states of $\mathcal{A}$ is defined by the encoding, that is, $\phi_i = \varphi_i$, and parenthood is defined naturally as $\pi(\langle p, q\rangle) = p$. The fact that letters induce injections and resets on brothers is obvious and ancestor-preservation follows from:

$$\pi(\bar{\delta}_i(\langle p, q\rangle, \sigma)) = \pi(\langle \bar{\delta}_{i-1}(p, \sigma), \delta_i(q, \langle p, \sigma\rangle)\rangle) = \bar{\delta}_{i-1}(p, \sigma) = \bar{\delta}_{i-1}(\pi(\langle p, \sigma\rangle), \sigma).$$

For the other direction we need to show how to build a cascade from $\mathcal{T}$. Let

$$d_i = \max\{|\Pi_m| : m \in M_{i-1}\}$$

be the size of the largest set of brothers at level i and let $Q_i = \{q_1, \ldots, q_{d_i}\}$. We define for every i a mapping $\theta_i : M_i \to Q_i$ whose restriction to any set of brothers is an injection. This encoding induces a bijection $\psi : M \to P$, which decomposes into $\psi_0 \uplus \cdots \uplus \psi_k$ with $\psi_i : M_i \to P_i$ defined inductively as $\psi_0(m_*) = \theta_0(m_*)$ and

$$\psi_i(m) = \langle \psi_{i-1}(\pi(m)), \theta_i(m)\rangle.$$

The transition function at each level is defined for every $\langle p, q\rangle \in P_i$ as

$$\delta_i(q, \langle p, \sigma\rangle) = \Delta_i(\psi^{-1}(\langle p, q\rangle), \sigma)$$

All that remains to be shown is that $\varphi : P \to Q$, defined as

$$\varphi(p) = \phi_k(\psi_k^{-1}(p))$$

is a homomorphism and this follows from the fact that ψ_k is an isomorphism between $\mathcal{C}$ and N_k and ϕ is an homomorphism from N_k to $\mathcal{A}$. ∎

The idea of the second direction is rather simple. We want to build a cascade whose configurations encode the subsets corresponding to the nodes of the tree. Each level i can be partitioned into several sets of nodes, each of which consisting of all brothers sharing the same ancestor. Let m and m' be nodes at level $i-1$ and let Π_m and $\Pi_{m'}$ be their respective sets of sons. Since m and m' are already encoded by distinct configurations p and p', their sons can be encoded by extensions of p and p' that use *the same* set of states Q_i whose size is the size of the largest set of brothers at level i. Thus we encode elements of Π_m by configurations in $\{p\} \times Q_i$ and elements of $\Pi_{m'}$ by configurations in $\{p'\} \times Q_i$. The transitions that correspond to the former will be labeled by $\langle p, \sigma\rangle$ and those of the latter by $\langle p', \sigma\rangle$. Doing so, every *injection* induced by some σ on some Π_m becomes a *permutation* induced by $\langle p, \sigma\rangle$ on Q_i. The hard part of the proof of the full theorem is to show that this *injection folding* can be done without creating spurious permutations (not implied by permutation subgroups of the automaton) and, in particular, if $\mathcal{A}$ is non-counting there will be no permutations.

Fig. 4 shows how a particular choice of encoding may lead to the introduction of spurious permutations. Automaton $\mathcal{A}$ is a union of two reset automata, hence clearly counter-free. An STT for $\mathcal{A}$ has an upper level with two nodes m and m' mapped naturally to sets $\{q_1, q_3\}$ and $\{q_2, q_4\}$. The first element in the cascade is the reset automaton $\mathcal{B}_1$ whose states A and B encode, respectively, these two subsets. The choice

of the second coordinate makes a difference. Let AC encode q_1 while AD encodes q_3. Then we have two ways to encode q_2 and q_4. Encoding them with BC and BD, respectively, the second element in the cascade is the identity automaton $\mathcal{B}_2$. However, if we choose to encode q_2 by BD and q_4 by BC we obtain $\mathcal{B}'_2$ in which the letter (A, a) induces a non-trivial permutation. As it turns out there is an additional condition on the structure of the STT as well as a general encoding scheme that avoids this phenomenon.

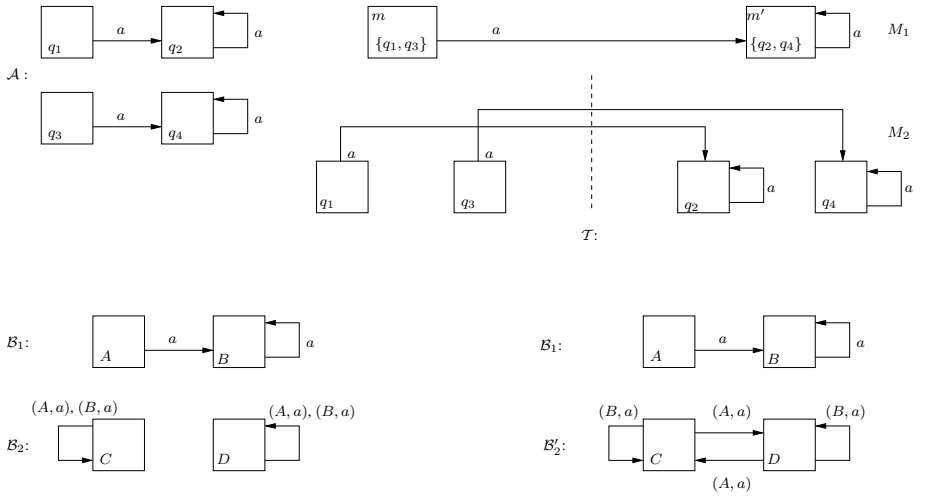


Fig. 4. A counter-free automaton $\mathcal{A}$, an STT with sons of m and m' separated by the dashed line, and two choices of encoding, the second leading to a permutation

Definition 14 (Equivalence). *Let $\mathcal{A} = (\Sigma, Q, \delta)$ be a (complete) automaton and let $\mathcal{T} = (\Sigma, M, \Delta, \pi, \phi)$ be an STT for $\mathcal{A}$.*

1. *Two subsets $R_1, R_2 \subseteq Q$ are equivalent if there exist $w, w' \in \Sigma^*$ such that*
   (a) $\delta(R_1, w) = R_2$ and $\delta(R_2, w') = R_1$;
   (b) ww' and $w'w$ induce identities on R_1 and R_2, respectively;
   This fact is denoted by $R_1 \overset{w,w'}{\sim} R_2$ or simply $R_1 \sim R_2$.
2. *Two nodes $m, m' \in M_i$ are equivalent if there exist $w, w' \in \Sigma^*$ such that:*
   (a) $\Delta(m, w) = m'$ and $\Delta(m', w') = m$;
   (b) $\phi(m) \overset{w,w'}{\sim} \phi(m')$, in the sense of subset equivalence;
   This fact is denoted as well by $m \overset{w,w'}{\sim} m'$ or simply $m \sim m'$.

Note that if w and w' satisfy condition 1-(a) but not 1-(b), then there exist some u, u' satisfying the latter. Since ww' is a permutation on R_1 and $w'w$ is a permutation on R_2, there is some l such that $(ww')^l$ and $(w'w)^l$ are identities. By letting $u = w$ and $u' = w'(ww')^{l-1}$ we have $R_1 \overset{u,u'}{\sim} R_2$. Equivalence between nodes implies an additional constraint on the definition of ϕ over their sons.

Proposition 2. *Let $m \overset{w,w'}{\sim} m'$ be two equivalent nodes in an STT. Then for every $r \in \Pi_m$ there exists $r' \in \Pi_{m'}$ such that $\delta(\phi(r), w) = \phi(r')$. Consequently $|\Pi_m| = |\Pi_{m'}|$.*

Proof: Suppose, on the contrary, that $\delta(\phi(r), w) \subset \phi(r')$ and hence $\delta(\phi(r'), w') \not\subseteq \phi(r)$ which violates the STT definition. ∎

Definition 15 (Holonomy Tree). *A holonomy tree is an STT $\mathcal{T} = (\Sigma, M, \Delta, \pi, \phi)$ such that for every $m, m' \in M_i$ and $\sigma \in \Sigma$, such that $\Delta(m, \sigma) = m'$, σ induces an injection[2] from Π_m to $\Pi_{m'}$ only if $m \sim m'$.*

To motivate this definition let us observe that injection folding is necessary in order to transform an injection from Π_m to $\Pi_{m'}$ to a permutation on Q_i. On the other hand, a reset from Π_m to some $r \in \Pi_{m'}$ remains a reset in $\mathcal{B}_i$ even if Π_m and $\Pi_{m'}$ are encoded using different states in $\mathcal{B}_i$. The essence of the additional condition in the holonomy tree is to restrict injection folding to occur only among sons of equivalent nodes where permutations really exist. If m and m' are not equivalent, σ will induce a *reset* from Π_m to $\Pi_{m'}$.

The proof of the equivalence between the existence of a holonomy tree and a cascaded decomposition satisfying condition 3 of Theorem 3 involves the following steps:

1. Associate with every node m in the holonomy tree a (possibly-trivial) permutation group H_m called a holonomy group;
2. Provide an encoding scheme which guarantees that any permutation subgroup in the cascade is isomorphic to a holonomy group;
3. Show that any holonomy group is homomorphic to a subgroup of $X_{\mathcal{A}}$.

Definition 16 (Holonomy Group). *Let $\mathcal{T} = (\Sigma, M, \Delta, \pi, \phi)$ be a holonomy tree with $N_i = (\Sigma, M_i, \Delta_i)$ being the automaton of level i and let $\hat{N}_i = (S_i, M_i, \Delta_i)$ be its expansion. The holonomy group H_m associated with a node $m \in M_{i-1}$ is the restriction of $\hat{N}_i$ to Π_m and to transformations that induce permutations on it.*

It can be shown that when $m \sim m'$, the groups H_m and $H_{m'}$ are isomorphic. The procedure for state encoding and injection folding for an equivalence class of $\sim$ is given below. The definition is inductive, assuming the encoding of the preceding levels has already been completed.

Definition 17 (Faithful Injection Folding). *Let $\{m_1, \ldots m_l\}$ be an equivalence class of $\sim$ at level $i-1$ whose elements are encoded by configurations $\{p_1, \ldots, p_l\}$, respectively and that for every j, $m_1 \overset{w_j, w'_j}{\sim} m_j$. Let $\mathcal{M} = \bigcup_j \Pi_{m_j}$ be the set of all sons of these nodes. We will use the set $Q_i = \Pi_{m_1}$ to encode the i^{th} coordinate of all elements of $\mathcal{M}$. The function $\theta : \mathcal{M} \to \Pi_{m_1}$ is defined for every m_j and every $r \in \Pi_{m_j}$ as $\theta(r) = \Delta(m, w'_j)$. Then for every $q \in \Pi_{m_1}$ and for every m_j and $m_{j'}$ such that $\Delta_i(m_j, \sigma) = m_{j'}$ we let*

$$\delta_i(q, \langle p_j, \sigma \rangle) = \Delta_i(q, w_j \sigma w'_{j'}).$$

[2] When $|\Pi_m| = 1$ we view σ as a partial reset, not an injection, and $m \sim m'$ is not required.

Proposition 3 (Injection Folding and Holonomy). *For every non-leaf node $m \in M_{i-1}$ the permutation group (induced by letters of the form $\langle p, \sigma \rangle$) in cascade element $\mathcal{B}_i$ constructed according to Definition 17 is isomorphic to H_m.*

Proof: We need to show that each permutation in $\mathcal{B}_i$ is identical to a permutation in H_m and vice versa. One direction follows immediately from the injection folding procedure: the action of $\langle p, \sigma \rangle$ on $Q_i = \Pi_{m_1}$ is defined to be identical to the action of some $w_j \sigma w'_{j'}$ on Π_{m_1}. For the other direction consider a word $u = \sigma_1 \cdot \sigma_2 \cdots \sigma_l$ inducing a cycle from $m = m_1$ to itself passing through nodes $m_2, m_3, \ldots, m_l$. Since each $w'_j w_j$ induces an identity on Π_{m_j}, the word

$$u' = \sigma_1 \cdot w'_2 \cdot w_2 \cdot \sigma_2 \cdot w'_3 \cdot w_3 \cdots \sigma_l$$

induces the same permutation on Π_m as does u. All the remains to be shown is that the word

$$\langle p_1, \sigma_1 \rangle \cdot \langle p_2, \sigma_2 \rangle \cdots$$

induces the same permutation on Π_m as does u' and this follows from defining the action of $\langle p_j, \sigma_j \rangle$ in $\mathcal{A}_i$ to be identical to that of $w_j \sigma_j w'_{j+1}$ in N_i. ▟

Proposition 4 (Holonomy and Subgroups). *The holonomy group H_m is homomorphic to the subgroup of $X_{\mathcal{A}}$ associated with $\phi(m)$.*

Proof: We need to show how to map a permutation $s : \phi(m) \to \phi(m)$ to a permutation $s' : \Pi_m \to \Pi_m$. This is done by letting, for every $r \in \Pi_m$

$$r \cdot s' = \phi^{-1}(\delta(\phi(r), s)),$$

that is, s is applied to the subset $\phi(r)$ associated with r and the resulting set is decoded back into an element of Π_m. The fact that ϕ^{-1} exists and is unique follows from Proposition 2. ▟

To complete the construction of the cascade from the holonomy tree we just need to partition every level in the tree into equivalence classes of $\sim$, build a cascade component with states corresponding to each class and apply to each equivalence class the procedure of Definition 17.

Corollary 1. *There exists a cascaded decomposition $\mathcal{C} = \mathcal{B}_1 \circ \cdots \circ \mathcal{B}_k \leq_\varphi \mathcal{A}$, with each $\mathcal{B}_i$ being a permutation-reset automaton and each permutation group homomorphic to a subgroup of $X_{\mathcal{A}}$, iff there exists a holonomy tree $\mathcal{T}$ for $\mathcal{A}$ isomorphic to the configuration tree.*

5 A Decomposition Algorithm

In this section we show how to build for an automaton $\mathcal{A}$ a holonomy tree whose size is at most exponential in the size of $\mathcal{A}$. The procedure involves the following steps (see Fig. 6):

1. Construct from $\mathcal{A}$ a *tree subset automaton* (TSA). This construction which is similar to the famous subset construction, computes all the subsets reachable from Q, that is, $\{\delta(Q,w) : w \in \Sigma^*\}$, plus the singleton sets which are not reachable from Q. In addition the TSA admits a parenthood function and in order to make its transition function ancestor-preserving, some reachable subsets will be represented by more than one node in the tree;
2. Compute a *height* function over the nodes and rearrange the tree into levels according to the height;
3. Complete the levels by duplicating nodes and redirect transitions to make each level a complete automaton.

It will then remain to show that a holonomy tree is obtained, from which the cascaded decomposition follows. The subtle point is the rearrangement of the tree into levels so as to restrict injections to occur only among sons of equivalent nodes. For a parenthood function π we let $\pi^0(m) = m$ and $\pi^j(m) = \pi(\pi^{j-1}(m))$. We use Π_m, Π^*_m and $\pi^*(m)$, respectively, do denote sons, descendants and ancestors of m.

Definition 18 (Tree Subset Automaton). *A tree subset automaton (TSA) for an automaton $\mathcal{A} = (\Sigma, Q, \delta)$ is a tuple $\mathcal{T} = (\Sigma, M, \Delta, \pi, \phi)$ where:*

- *M is a set of nodes with a distinguished root element m_*;*
- *$\pi : M - \{m_*\} \to M$ is a parenthood function, such that for every $m \neq m_*$, there exists some $j > 0$ such that $\pi^j(m) = m_*$;*
- *$\Delta : M \times \Sigma \to M$ is an ancestor-preserving transition function, that is, for every m, σ there is some j such that $\Delta(\pi(m), \sigma) = \pi^j(\Delta(m, \sigma))$;*
- *$\phi : M \to 2^Q$ is a function mapping nodes to sets of states, satisfying*
  - *The range of ϕ is $\{\delta(Q,w) : w \in \Sigma^*\} \cup \{\{q\} : q \in Q\}$;*
  - *$\phi(m_*) = Q$;*
  - *$\phi(\Delta(m,\sigma)) = \delta(\phi(m), \sigma)$*
  - *$\phi(m) \subseteq \phi(\pi(m))$;*
  - *No redundancy: $\phi(m) \not\subseteq \phi(m')$ for any pair of brothers.*

Algorithm A-TSA for constructing a TSA is depicted in Table 1. It is a typical on-the-fly graph exploration algorithm which uses an auxiliary list L of newly-discovered nodes (those for which the transition function has not yet been computed). The algorithm works in two phases: first it computes nodes that correspond to sets of the form $\delta(Q,w)$, determines their respective parents and computes Δ for them in an ancestor-preserving manner. The determination of the parent for a newly-created node r' is illustrated in Fig. 5. Note that $m' \in \Pi^*_{m'}$ and $F = \delta(\phi(r),\sigma) \subseteq \phi(m')$ so that a node $z \in \Pi^*_{m'}$ satisfying $F = \phi(r') \subseteq \phi(z)$ always exists. Note also that z may be non-unique if m' has two incomparable descendants whose sets contain F and in this case an arbitrary choice of a parent can be made. In the second phase of the algorithm we add to the tree all the remaining *singletons* by adding to each node m sons that correspond to singleton nodes not covered by the union of its existing sons. Then we compute Δ for the newly-added nodes according to the same principle.

Proposition 5. *Algorithm* A-TSA *terminates and produces a TSA for $\mathcal{A}$.*

Table 1. The TSA Construction Algorithm

Algorithm A-TSA

$M := L := \{m_*\}$; $\phi(m_*) := Q$

repeat pick $r \in L$, with $\pi(r) = m$
  for every $\sigma \in \Sigma$
    $F := \delta(\phi(r), \sigma)$
    $m' = \Delta(m, \sigma)$
    if $\neg\exists r' \in M$ s.t. $\phi(r') = F$ and $\pi(r') \in \Pi^*_{m'}$
      create a node r' with $\phi(r') = F$ and insert it to L and M
      let $\pi(r')$ be a minimal $z \in \Pi^*_{m'}$ s.t. $F \subseteq \phi(z)$
      for every node z' s.t. $\pi(z') = z$ and $F \subseteq \phi(z')$
        $\pi(z') := r'$
    endif
    $\Delta(r, \sigma) := r'$
  remove r from L
until $L = \emptyset$

for every $m \in M$
  for every $q \in \phi(m) - \bigcup_{r \in \Pi_m} \phi(r)$
  insert a new node r to M and L
  $\phi(r) := \{q\}$; $\pi(r) := m$
repeat
  take $r \in L$, $m = \pi(r)$
    for every $\sigma \in \Sigma$
      $F := \delta(\phi(r), \sigma)$
      $m' = \Delta(m, \sigma)$
      Let r' be a node with $\phi(r') = F$ and $\pi(r') \in \Pi^*_{m'}$
      $\Delta(r, \sigma) := r'$
  remove r from L
until $L = \emptyset$

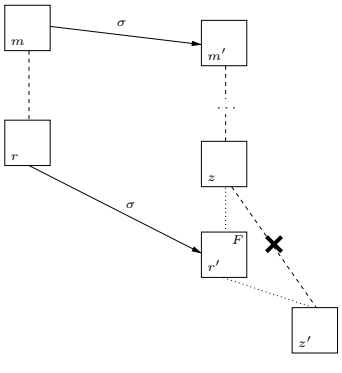


Fig. 5. Determining the parent of r' to be a minimal element of $\Pi^*_{m'}$ which contains $\phi(r')$; Redirecting the parenthood of z' from z to r'

Proof: All sets of the form $\delta(Q,w)$ as well as the other singletons are eventually covered by nodes and ancestor preservation is guaranteed by construction. ∎

The next step involves the rearrangement of the nodes into levels according to a *height function* that we define below. Note that the definition of node equivalence (Definition 14) holds also for TSA, and that apart from transitions among members of an equivalence class of $\sim$, the transition graph of the TSA is acyclic.

Definition 19 (Height). *A height function for a TSA $\mathcal{T} = (\Sigma, M, \Delta, \pi, \phi)$ is a function $h : M \to \mathbb{N}$ defined inductively as*

$$h(m) = 0 \ \textit{if} \ |\phi(m)| = 1$$

$$h(m) = \max \left\{ \begin{array}{l} \max\{h(r)+1 : r \in \Pi_m\}, \\ \max\{h(r)+1 : \exists\sigma \Delta(m,\sigma) = r \not\sim m\}, \\ \max\{h(m') : m \sim m'\} \end{array} \right\}$$

In other words, $h(m)$ is the length of the longest path from m to a singleton node, not counting transitions among equivalent nodes. The height function can be computed polynomially using a shortest-path algorithm variant. After computing the height we partition M into $M_0 \uplus \ldots \uplus M_k$ with $k = h(m_*)$ and $M_i = \{m \in M : h(m) = k - i\}$.

The next step is to transform $\mathcal{T} = (\Sigma, M, \Delta, \pi, \phi)$ into a *balanced* TSA $\mathcal{T}' = (\Sigma, M', \Delta', \pi', \phi')$ in which all the ancestral chains from a singleton to m_* are of the same length. The completion of each level with missing nodes is performed bottom up by letting $M'_k = M_k$ and then computing M'_i based on M'_{i+1} as follows. For every $r \in M'_{i+1}$ such that $\pi(r) \notin M_i$ we create a new node $m \in M'_i$ and let $\phi'(m) = \phi(r)$, $\pi'(m) = \pi(r)$, $\pi'(r) = m$ and $\Delta'(m,\sigma) = \Delta(r,\sigma)$ for every σ. The mapping of existing nodes remains the same, that is, $\phi'(m) = \phi(m)$ when $m \in M$. As a result of this procedure each node has an ancestor (possibly identical to itself) in every level. The final step which transforms $\mathcal{T}'$ into a holonomy tree consists of lifting transitions that go from a node m to a lower-level node m' so that they preserve the level. In other words, for every i, $m \in M_i$ and σ we let $\Delta_i(m,\sigma) = m''$ where m'' is the *ancestor* of $m' = \Delta(m,\sigma)$ at M_i. The whole procedure is demonstrated in Fig. 6-(a,b,c,d).

To prove that we obtain a holonomy tree we need to show that for every two nodes m and m', a letter induces an injection from Π_m to $\Pi_{m'}$ only if $m \sim m'$. Suppose $\Delta(m,\sigma) = m'$ and $m \not\sim m'$, both belonging to level i. This implies that in the TSA there was some node m'' with $h(m'') < h(m)$ such that $\Delta(m,\sigma) = m''$ (Fig. 7-(a)). After the rearrangement and completion procedure, m'' is a node in level $i+1$ and σ induces a *reset* from Π_m to it (Fig. 7-(b)). The injection at level $i+2$ has been *separated* into two resets induced by two *distinct* input letters. Fig. 6-(e,f) shows how the holonomy tree is transformed to a cascade via state encoding. The global automaton associated with the cascade and its homomorphism to the original automaton are shown in Fig. 8.

Corollary 2 (Main Result). *Every automaton $\mathcal{A}$ can be decomposed into a cascade of permutation-reset automata, satisfying the conditions of Theorem 3, whose size is at most exponential in the size of $\mathcal{A}$.*

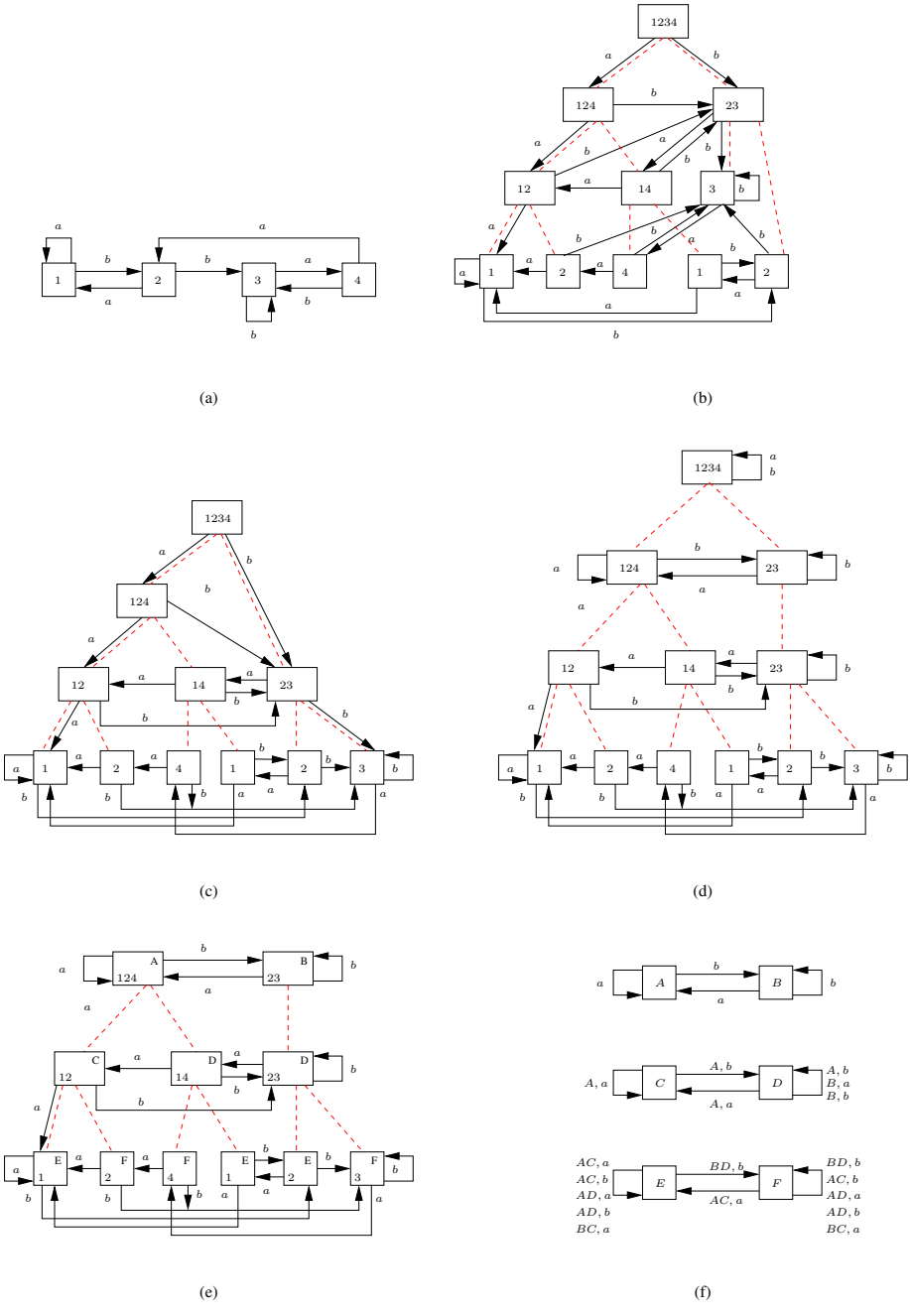


Fig. 6. The decomposition process: (a) An automaton; (b) its TSA (parenthood indicated by dashed lines); (c) the TSA rearranged according to height; (d) the holonomy tree obtained after completion and redirection; (e) state encoding; (f) the decomposition

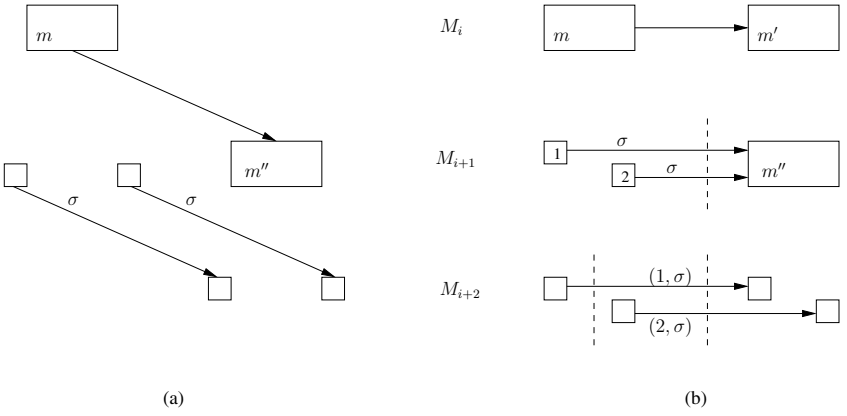


Fig. 7. The crux of the matter: (a) the situation before height rearrangement and completion with an injection between non-equivalent nodes; (b) after the procedure the sons of m make a reset to m'' and the transition functions of their sons are defined over distinct alphabets.

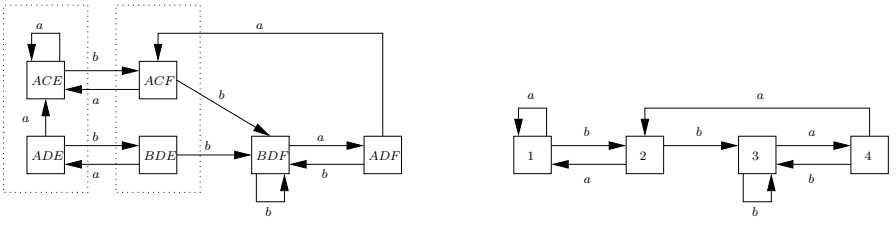


Fig. 8. The global automaton associated with the cascade. The homomorphism to the original automaton is defined as $\varphi(ACE) = \varphi(ADE) = 1$, $\varphi(ACF) = \varphi(BDE) = 2$, $\varphi(BDF) = 3$ and $\varphi(ADF) = 4$

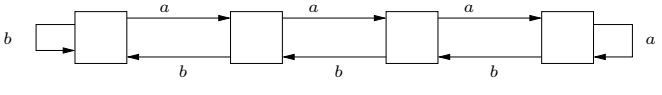


Fig. 9. The elevator automaton, the hardest counter-free automaton to decompose

The reason for the exponential blow-up is that to satisfy ancestor-preservation (which is crucial for the hierarchical coordinate system underlying the cascade) some states may need to split to exponentially-many copies, each representing a different class of input histories that leads to the same state. The reader is invited to construct the holonomy tree for the automaton of Fig. 9.

6 Concluding Remarks

Let us sketch the historical roots of this construction. Among the numerous proofs of the Krohn-Rhodes primary decomposition theorem those of Zeiger [15,16] were more

automata oriented. Zeiger's proof has been corrected and presented more clearly in Ginzburg's book [2] based on some constructs attributed to Yoeli [14]. Ginzburg's proof of the theorem contains some non-deterministic stages concerning the choice of semi-partitions of Q. In addition, it does not discuss complexity issues explicitly. Another incomplete proof in the same spirit appears in [3].

The proof in [2] inspired Eilenberg to give a slight generalization of the primary decomposition, the *holonomy* decomposition ([1], pp. 43-50). The holonomy decomposition is cleaner and determinizes the choice of semi-partitions. Its major drawback is that it is a theorem on coverings of transformation semigroups and as such it pays no attention to the *labels* of the *generators* of the semigroup, that is, the input alphabet. Consequently, the outcome of the decomposition is not given explicitly as a valid automaton over the *original* alphabet. Another cultural problem associated with this construction is the elegant, concise and motivation-less algebraic style in which it is written, which makes it hardly accessible to many. It remains to be seen if the present exposition improves the situation.

As a final note, since this work has not undergone a complete review process, it probably contains inaccuracies for which I apologize and urge the reader to notify me of. I would like to thank O. Gauwin for proofreading and E. Asarin for helping me to catch up with my former self.

References

1. Eilenberg, S.: Automata, Languages, and Machines. Academic Press, London (1976)
2. Ginzburg, A.: Algebraic Theory of Automata. Academic Press, London (1968)
3. Hartmanis, J., Stearns, R.E.: Algebraic Structure Theory of Sequential Machines. Prentice-Hall, Englewood Cliffs (1966)
4. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
5. Krohn, K., Rhodes, J.: Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. Transactions of the American Mathematical Society 116, 450–464 (1965)
6. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
7. Maler, O., Pnueli, A.: Tight bounds on the complexity of cascaded decomposition of automata. In: FOCS, pp. 672–682 (1990)
8. Maler, O., Pnueli, A.: On the cascaded decomposition of automata, its complexity and its application to logic. Unpublished manuscript (1994), `http://www-verimag.imag.fr/~maler/Papers/decomp.pdf`
9. McNaughton, R., Papert, S.: Counter-Free Automata. MIT Press, Cambridge (1971)
10. Meyer, A.R.: A note on star-free events. J. ACM 16(2), 220–225 (1969)
11. Pin, J.-E.: Varieties of Formal Languages. Plenum, New York (1996)
12. Rhodes, J.L., Krohn, K.: Algebraic theory of machines. In: Proc. Symp. on Math. Theory of Automata. Polytechnic Press, Brooklyn (1962)
13. Schützenberger, M.-P.: On finite monoids having only trivial subgroups. Information and Control 8(2), 190–194 (1965)

14. Yoeli, M.: Decomposition of finite automata. Technical Report TR-10, US Office of Naval Research, Information Systems Branch, Hebrew University, Jerusalem (1963)
15. Zeiger, H.P.: Cascade synthesis of finite-state machines. Information and Control 10(4), 419–433 (1967)
16. Zeiger, H.P.: Yet another proof of the cascade decomposition theorem for finite automata. Mathematical Systems Theory 1(3), 225–228 (1967)

Temporal Verification of Reactive Systems: Response

Zohar Manna[1] and Amir Pnueli[2]

[1] Stanford University, Stanford, CA
[2] Weizmann Institute of Science, Israel and New York University, NY, USA

Introduction

Nearly twenty years ago, Amir and I (ZM) decided to write a series of three volumes summarizing the state of the art (at that time) on the verification of Reactive Systems using Temporal Logic techniques. We published the first two volumes:

Volume I: Z. Manna and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems: Specification," Springer-Verlag, New York, Dec. 1991.
Volume II: Z. Manna and A. Pnueli, "Temporal Verification of Reactive Systems: Safety," Springer-Verlag, New York, Sept. 1995.

We had written a complete draft of the first three chapters of **Volume III**. However, the fourth chapter remained incomplete. These chapters are:

Chapter 1: Response
Chapter 2: Response for Parameterized Programs
Chapter 3: Response Under Fairness
Chapter 4: Reactivity (incomplete)

Volume III has never been published, but the first 3 chapters can be found online at `http://theory.stanford.edu/~zm/tvors3.html`. We added exercises to Chapters 1-3, but never wrote the bibliographic remarks and references to this volume. The reader interested in these references should consult the paper: Z. Manna and A. Pnueli, "*Completing the temporal picture*," Theoretical Computer Science Journal, Vol. 83, No. 1, pp. 97–130, 1991.

In this special Springer volume, dedicated to the memory of Amir, I have included the first chapter of Volume III. I have made very few changes to the original version that appeared on August 28, 1996. I would like to thank Sriram Sankaranarayanan, Henny B. Sipma, and Ting Zhang for their help.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 279–361, 2010.

1 Response

In the preceding chapters[1] we discussed methods for proving safety properties. While the class of safety properties is very important, and normally occupies a large portion of a specification, it must be complemented by properties in the other classes.

It is interesting to note that the expression of safety properties and their verification use relatively little temporal logic. The main emphasis there is directed towards finding assertions that are invariant over the computation of a program. Most of the verification efforts are concentrated on showing that the assertions are inductive. This requires proving a set of verification conditions, which are expressed by nontemporal state formulas. Temporal logic is used mainly for stating the final result of invariance of the assertion. It is true that, when considering precedence properties, we extensively use the past part of temporal logic. But as we commented there, an equivalent, though sometimes less elegant, state formulation of these properties can be managed through auxiliary or history variables.

It is only when we enter the realm of more general properties, that temporal logic becomes an essential and irreplaceable tool. Thus if, for some reason, one is willing to restrict himself to the study of safety properties of reactive programs, he does not need the full power of temporal logic.

A related observation is that, only when we go beyond safety properties does fairness become meaningful. Recall the definition of a *run* as a state sequence that satisfies the requirements of initiation and consecution but not necessarily any of the fairness requirements. It can be shown that a safety formula holds over all runs of a program if and only if it holds over all computations, i.e., fair runs. Thus, safety properties cannot distinguish between fair and unfair runs.

This is no longer the case with progress (non safety) properties. Note, for example, that the infinite sequence $s_0, s_0, \ldots$ is a legal run of any program P, provided $s_0 \quad \Theta$. This run is generated by continuously taking the idling transition τ_I. There are very few progress properties that hold over this run.

Consequently, while safety properties do not depend on the fairness requirements for their validity, progress properties do. In Chapters 1–3 we concentrate on the important class of *response* properties, which is one of the progress classes. This class contains properties that can be expressed by a formula of the form

$$p,$$

for a past formula p. In these chapters, we introduce a family of rules for response properties that rely on the different fairness requirements. Chapters 1–2 present rules that rely on *justice*, while Chapter 3 presents rules that rely on (justice and) *compassion*.

Chapter 4 completes the picture by presenting rules for the highest progress class, that of reactivity.

[1] ZM: Cf. Manna & Pnueli, Temporal Verification of Reactive Systems: Safety, Springer-Verlag, New York, Sept. 1995.

Chapter 1 deals with response properties that rely on the just transitions of the system for their validity. Chapter 3 generalizes the treatment to properties that rely on both justice and compassion for their validity.

The remainder of this chapter is organized as follows:

In Section 2 we consider a single-step rule that relies on the activation of a single just transition.

Section 3 shows how to combine several applications of the single-step rule into a rule that relies on a fixed number of activations of just transitions.

Section 4 generalizes the rule to the case that the number of just transition activations necessary to achieve the goal is not fixed and may depend, for example, on an input parameter.

In Section 5, we extend all the above methods to prove properties expressed by response formulas that contain past subformulas.

In Section 6 deals with the class of guarantee formulas, treating them as a special case of response properties.

Similarly, Section 7 considers the class of obligation properties. Again, their verification is based on their consideration as a special case of the response class.

2 Response Rule

Even though there are several different classes of progress properties, their verification is almost always based on the establishment of a single construct — the *response formula*

$$p \Rightarrow q,$$

for past formulas p and q. This formula states that any position in the computation which satisfies p must be followed by a later position which satisfies q. Since the canonical response formula q is equivalent to $\mathrm{T} \Rightarrow q$, it follows that every response property can be expressed by a formula of the form $p \Rightarrow q$.

A general response property allows p and q to be general past formulas. In Sections 2–4 we consider the simpler case that p and q are assertions. In Section 5 we will generalize the treatment to the case that p and q are past formulas.

A Single-Step Rule

A single-step rule, relying on justice, is provided by rule RESP-J presented in Fig. 1.

The rule calls for the identification of an intermediate assertion φ and a just transition $\tau_h \in \mathcal{J}$, to which we refer as the *helpful transition*.

Premise J1 of the rule states that, in any position satisfying p, either the goal formula q already holds, or the intermediate formula φ, bridging the passage from p to q, holds. The q-disjunct of this premise covers the case that the distance between the p-position and the q-position is 0. The φ-disjunct and the other premises cover the case that the distance between these two positions is positive.

Premise J2 requires that every transition leads from a φ-position to a position that satisfies $q \vee \varphi$. That is, either a position satisfying the goal formula q is attained or, if not, then at least the intermediate φ is maintained.

For assertions p, q, φ, and transition $\tau_h \in \mathcal{J}$,

$$\begin{array}{ll} \text{J1.} & p \rightarrow q \vee \varphi \\ \text{J2.} & \{\varphi\}\ \mathcal{T}\ \{q \vee \varphi\} \\ \text{J3.} & \{\varphi\}\ \tau_h\ \{q\} \\ \text{J4.} & \varphi \rightarrow En(\tau_h) \\ \hline & p \Rightarrow \quad q \end{array}$$

Fig. 1. Rule RESP-J (single-step response under justice)

Premise J3 requires that the helpful transition τ_h always leads from a φ-position to a q-position.

Premise J4 requires that the helpful transition τ_h is enabled at every φ-position.

Justification. To justify the rule, consider a computation σ which satisfies the four premises of the rule. Assume that p holds at position k, $k \geq 0$. We wish to show that q holds at some position i, $i \geq k$. Assume, to the contrary, that it does not. That means that for all i, $i \geq k$, q does not hold at i. By J1, φ holds at position k. By J2, every successor of a φ-state is a $(q \vee \varphi)$-state.

Since we assumed that q never occurs beyond k, it follows that φ holds continuously beyond k. By J4, the just transition τ_h must be continuously enabled. However, τ_h is never taken beyond k. This is because if τ_h were taken, it would have been taken from a φ-position, and by J3 the next position would have satisfied q. Thus we have that τ_h is continuously enabled, but never taken beyond k. It follows that the sequence σ is not just with respect to τ_h, and is, therefore, not a computation.

This shows that, for all computations, there must exist an i, $i \geq k$ such that q holds at i. ∎

In applications of the rule, it is sufficient to establish premise J2 for all $\tau \neq \tau_h$, since J2 for $\tau = \tau_h$ is implied by J3. It is also unnecessary to check premise J2 for $\tau = \tau_I$, the idling transition, since $\{\varphi\}\,\tau_I\,\{\varphi\}$ is trivially state valid.

Example (program ANY-Y)

Program ANY-Y of Fig. 2 illustrates a simple program consisting of two processes communicating by the shared variable x, initially set to 0. Process P_1 keeps incrementing variable y as long as $x = 0$. Process P_2 has only one statement, which sets x to 1. Obviously, once x is set to 1, process P_2 terminates, and some time later so does P_1, as soon as it observes that $x \neq 0$.

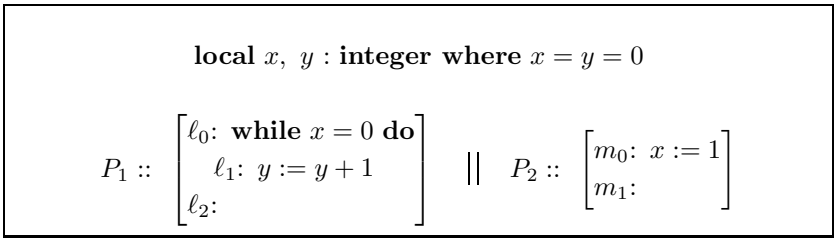


Fig. 2. Program ANY-Y

We illustrate the use of rule RESP-J for proving the response property

$$at\_m_0 \;\Rightarrow\; \Diamond(x = 1)$$

for program ANY-Y.

As the helpful transition τ_h we take m_0. As the intermediate assertion φ we take p: at_m_0. Premise J1 assumes the form

$$\underbrace{at\_m_0}_{p} \;\rightarrow\; \underbrace{\cdots}_{q} \;\vee\; \underbrace{at\_m_0}_{\varphi}\,,$$

which is obviously valid. Premise J2 is established by showing that all transitions, excluding m_0, preserve φ: at_m_0, which is clearly the case.

Premise J3 requires showing that m_0 leads from any φ-state to a q-state, expressed by

$$\underbrace{\cdots \wedge\; x' = 1}_{\rho_{m_0}} \;\wedge\; \underbrace{\cdots}_{\varphi} \;\rightarrow\; \underbrace{x' = 1}_{q'}\,,$$

which is obviously valid. Finally, J4 requires

$$\underbrace{at\_m_0}_{\varphi} \;\rightarrow\; \underbrace{at\_m_0}_{En(m_0)}\,,$$

which is also valid. This establishes that the property specified by the response formula $at\_m_0 \Rightarrow \Diamond(x = 1)$ is valid over program ANY-Y. ∎

Combining Response Properties

Rule RESP-J by itself is not a very strong rule, and is sufficient only for proving *one-step* response properties, i.e., properties that can be achieved by a single activation of a helpful transition. For example, while program ANY-Y always terminates, its termination cannot be proven by a single application of rule RESP-J.

In general, most response properties of the form $p \Rightarrow \Diamond q$ require several helpful steps in order to get from a p-position to a q-position.

To establish such properties we may use several rules that enable us to combine response properties, each established by a single application of rule RESP-J. These rules are based on general properties of response formulas that allow us to form these combinations. We list some of these properties as proof rules. All of these rules can be established as derived rules, using the standard deductive system for temporal logic[2].

Monotonicity

An important property of response formulas is the monotonicity of both the antecedent and the consequent. This can be summarized in the form of the (monotonicity) rule MON-R, presented in Fig. 3.

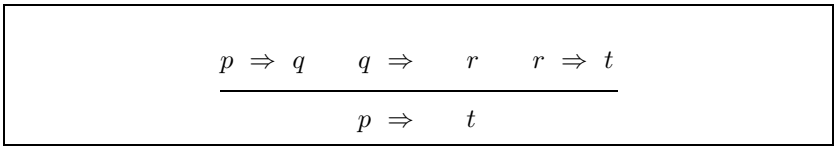


Fig. 3. Rule MON-R (monotonicity of response)

Rule MON-R enables us to strengthen the antecedent and weaken the consequent. Thus, if we managed to prove the response formula

$$at\_\ell_0 \;\Rightarrow\; (x = 1),$$

we can infer from it, using rule MON-R, the formula

$$at\_\ell_0 \;\Rightarrow\; (x > 0).$$

Reflexivity

Property RFLX-R of Fig. 4 states that the operator is reflexive.

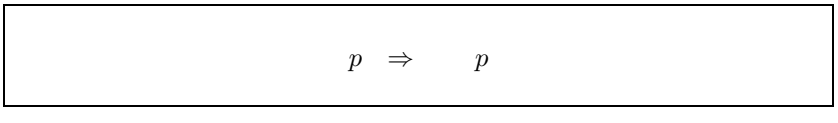


Fig. 4. Property RFLX-R (reflexivity of response)

We may use this property to prove simple response formulas such as

$$x = 0 \;\Rightarrow\; (x = 0).$$

Transitivity

The transitivity property of response formulas is expressed by the (transitivity) rule TRNS-R, presented in Fig. 5.

[2] For example, the one presented in Chapter 3 of Volume I.

$$\frac{p \Rightarrow q \qquad q \Rightarrow r}{p \Rightarrow r}$$

Fig. 5. Rule TRNS-R (transitivity of response)

Thus, if we managed to prove for program ANY-Y the two response formulas

$$at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ (at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1)$$
$$at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ (at\_\ell_2 \ \wedge \ at\_m_1),$$

we may use rule TRNS-R to conclude

$$at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ (at\_\ell_2 \ \wedge \ at\_m_1).$$

The soundness of rule TRNS-R is obvious. Consider a computation σ such that the first two premises are valid over σ. Let i be a position satisfying p. By the first premise, there exists a position j, $j \geq i$, satisfying q. By the second premise, there exists a position k, $k \geq j$, satisfying r. Thus, we are ensured of a position k, $k \geq i$, satisfying r, which establishes $p \Rightarrow \ r$.

Proof by Cases

Another useful property of response formulas is that it is amenable to proof by cases. This possibility is presented by rule CASES-R of Fig. 6.

$$\frac{p \Rightarrow r \qquad q \Rightarrow r}{(p \ \vee \ q) \Rightarrow r}$$

Fig. 6. Rule CASES-R (case analysis for response)

Assume, for example, that we have proved for program ANY-Y the two following response formulas.

$$at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ (at\_\ell_2 \ \wedge \ at\_m_1)$$
$$at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ (at\_\ell_2 \ \wedge \ at\_m_1).$$

Then, we may use rule CASES-R to conclude

$$(at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1) \ \vee \ (at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1) \ \Rightarrow \ (at\_\ell_2 \ \wedge \ at\_m_1),$$

from which, by rule MON-R, we can infer

$$at\_\ell_{0,1} \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ (at\_\ell_2 \ \wedge \ at\_m_1).$$

Example (program ANY-Y)

We will illustrate the use of these rules by proving termination of program ANY-Y. This property can be expressed by the response formula

$$\Theta \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1),$$

where

$$\Theta\colon\quad \pi = \{\ell_0, m_0\} \;\wedge\; x = 0 \;\wedge\; y = 0$$

is the initial condition of program ANY-Y.

The proof consists of the following steps:

1. $at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = 0 \;\Rightarrow\; (at\_\ell_{0,1} \;\wedge\; at\_m_1 \;\wedge\; x = 1)$
   by rule RESP-J, taking τ_h: m_0 and φ: $at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0$
2. $at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule RESP-J, taking τ_h: ℓ_0 and φ: $at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1$
3. $at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; (at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1)$
   by rule RESP-J, taking τ_h: ℓ_1 and φ: $at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1$
4. $at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule TRNS-R, applied to 3 and 2
5. $(at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1) \;\vee\; (at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1) \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule CASES-R, applied to 2 and 4
6. $at\_\ell_{0,1} \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\rightarrow\; (at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1) \;\vee\; (at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1)$
   an assertional validity
7. $at\_\ell_{0,1} \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule MON-R, using 5 and 6
8. $at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = 0 \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule TRNS-R, applied to 1 and 7
9. $\Theta \;\rightarrow\; at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = 0$
   an assertional validity
10. $\Theta \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1)$
    by rule MON-R, applied to 8 and 9. ∎

3 Chain Rule

The proof of the last example follows a very specific pattern that occurs often in proofs of response properties. According to this pattern, to establish $p \Rightarrow q$, we identify a sequence of intermediate situations described by assertions $\varphi_m, \varphi_{m-1}, \ldots, \varphi_0$ such that p implies one of $\varphi_m, \ldots, \varphi_0$, and q is identical with φ_0 (or is implied by φ_0). We then show that for every $i > 0$, being at φ_i implies that eventually we must reach φ_j for some $j < i$.

We can interpret the index i of the intermediate formula φ_i as a measure of the distance of the current state from a state that satisfies the goal q. Thus, the lower the index, the closer we are to achieving the goal q. For a position j, let φ_i be the intermediate formula with the smallest i s.t. φ_i holds at j. We refer to the index i as the *rank* of position j.

This proof pattern is summarized in rule CHAIN-J (Fig. 7).

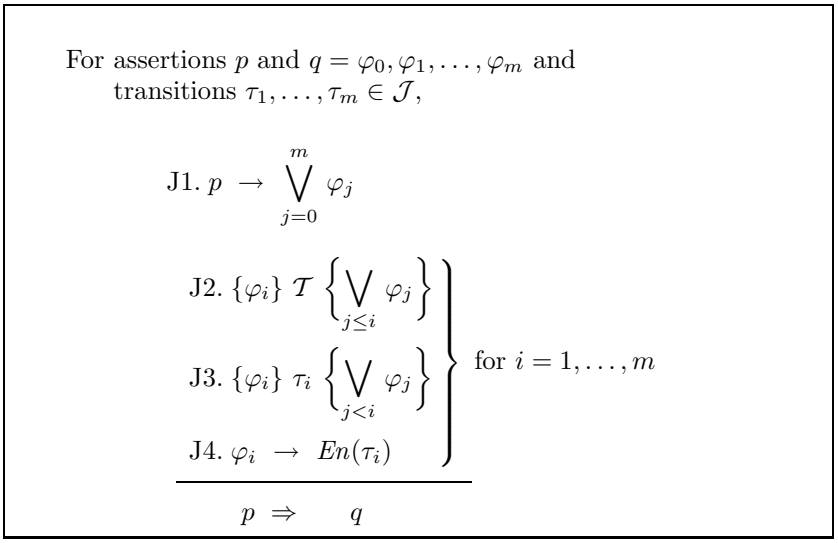


Fig. 7. Rule CHAIN-J (chain rule under justice)

According to premise J1, p implies that one of the intermediate formulas φ_i (possibly φ_0 implying q) holds. Premise J2 requires that taking any transition from a φ_i-position results in a next position which satisfies φ_j, for some $j \le i$. Premise J3 requires that taking the helpful transition τ_i from a φ_i-position results in a next position which satisfies φ_j for $j < i$. We can view premise J2 as stating that the rank never increases, while premise J3 states that the helpful transition guarantees that the rank decreases. Premise J4 claims that the helpful transition τ_i is enabled at every φ_i-position.

Justification. Assume that all four premises are P-state valid. Consider a P-computation σ and a position t that satisfies p. We wish to prove that some later position satisfies q. Assume to the contrary that all positions later than t (including t itself) do not satisfy q. By J1, position t satisfies φ_j for some $j \ge 0$. Index j cannot be 0 because $\varphi_0 = q$ and we assumed that no position beyond t satisfies q. Thus, position t satisfies φ_j, for some $j > 0$. By J2, position $t + 1$ satisfies some φ_k, $k \le j$. Again, $k > 0$ due to $\varphi_0 = q$. Continuing in this manner, it follows that every position beyond t satisfies some φ_j for $j > 0$, to which we refer as the rank of the position.

By J2, the rank of the position can either decrease or remain the same. It follows that there must exist some position $k \ge t$, beyond which the rank never decreases.

Assume that i is the rank of the state at position k. Since q is never satisfied and the rank never decreases beyond position k, it follows (by J2) that φ_i holds continually beyond k. By J3, τ_i cannot be taken beyond k, because that would have led to a rank decrease. By J4, τ_i is continually enabled beyond k yet, by the argument above, it is never taken. This violates the requirement of justice for τ_i.

It follows that if all the premises of the rule are P-state valid then the conclusion $p \Rightarrow \quad q$ is P-valid. ∎

Note that since premise J3 implies premise J2 for $\tau = \tau_i$, it is sufficient to check premise J2 for a given $i = 1, \ldots, m$, only for $\tau \neq \tau_i$. Also, it is unnecessary to check premise J2 for $\tau = \tau_I$, since $\{\varphi_i\}\, \tau_I\, \{\varphi_i\}$ trivially holds.

Example (Reproving termination of program ANY-Y)

Let us show how termination of program ANY-Y can be proved (again) by a single application of rule CHAIN-J.

The property we wish to prove is

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = 0 \ \wedge \ y = 0}_{p = \Theta} \ \Rightarrow \quad \underbrace{at\_\ell_2 \ \wedge \ at\_m_1}_{q} \ .$$

Inspired by our previous proof of this property, we choose four assertions and corresponding helpful transitions as follows:

$$\begin{array}{lll} \varphi_3\colon & at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0 & \tau_3\colon \quad m_0 \\ \varphi_2\colon & at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 & \tau_2\colon \quad \ell_1 \\ \varphi_1\colon & at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 & \tau_1\colon \quad \ell_0 \\ \varphi_0 = q\colon & at\_\ell_2 \ \wedge \ at\_m_1. & \end{array}$$

Let us consider each of the premises of rule CHAIN-J.

- Premise J1
  This premise calls for proving

  $$p \ \rightarrow \ \bigvee_{j=0}^{3} \varphi_j.$$

  We will prove $p \rightarrow \varphi_3$, which amounts to

  $$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = 0 \ \wedge \ y = 0}_{p} \ \rightarrow \ \underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0}_{\varphi_3}$$

  which obviously holds.
- Premises J2–J4 for $i = 3, 2, 1$
  We list below the premises that are proven for each i, $i = 3, 2, 1$.
  - Assertion φ_3: $at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0$

$$\Big\{ \underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0}_{\varphi_3} \Big\} \ \tau \ \Big\{ \underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0}_{\varphi_3} \Big\}$$

for each $\tau \neq m_0$

$$\Big\{ \underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0}_{\varphi_3} \Big\} \ m_0 \ \left\{ \begin{array}{l} \underbrace{at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_2} \ \vee \\ \underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_1} \end{array} \right\}$$

$$\underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0}_{\varphi_3} \ \rightarrow \ \underbrace{at\_m_0}_{En(m_0)} \ .$$

- Assertion φ_2: $at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1$

$$\Big\{ \underbrace{at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_2} \Big\} \ \tau \ \Big\{ \underbrace{at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_2} \Big\}$$

for every $\tau \neq \ell_1$

$$\Big\{ \underbrace{at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_2} \Big\} \ \ell_1 \ \Big\{ \underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_1} \Big\}$$

$$\Big[\underbrace{at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_2} \Big] \ \rightarrow \ \underbrace{at\_\ell_1}_{En(\ell_1)} \ .$$

- Assertion φ_1: $at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1$

$$\Big\{ \underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_1} \Big\} \ \tau \ \Big\{ \underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_1} \Big\}$$

for every $\tau \neq \ell_0$

$$\Big\{ \underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_1} \Big\} \ \ell_0 \ \Big\{ \underbrace{at\_\ell_2 \ \wedge \ at\_m_1}_{\varphi_0} \Big\}$$

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1}_{\varphi_1} \ \rightarrow \ \underbrace{at\_\ell_0}_{En(\ell_0)} \ .$$

All of these implications and verification conditions are obviously state valid, which establishes the conclusion

$$\Theta \;\Rightarrow\; (at\_\ell_2 \;\wedge\; at\_m_1).$$ ∎

Relation to Rule NWAIT

There is a strong resemblance between the premises of rule CHAIN-J and those of rule NWAIT (Fig. 3.6 on page 267 of the SAFETY book). This is not surprising, since in both cases we wish to establish the evolution from p to q (q_0 in the case of NWAIT) by successively passing through $\varphi_m, \varphi_{m-1}, \ldots, \varphi_1, \varphi_0$.

The main difference is that, in rule NWAIT, we are quite satisfied if, from a certain point on, we stay forever within φ_j for some $j > 0$. This is unacceptable in a response rule, where we are anxious to establish eventual arrival at φ_0. This difference is expressed in premise J3 which requires that activation of the helpful transition τ_i takes us out of a φ_i-state, and premise J4 which requires that τ_i is enabled on all φ_i-states. These premises have no counterparts in rule NWAIT. This excludes computations that consist of states whose rank, from a certain point on, never decreases below some $j > 0$.

Another difference is that, in rule NWAIT, we allow a transition to lead from φ_i, $i > 0$, back to φ_i. This is expressed by the disjunction, $\bigvee_{j \le i} \varphi_j$, appearing in the postcondition of premise N3. Rule CHAIN-J allows this in premise J2 for all but the helpful transition, which is required in J3 to lead to a position with a strictly lower rank.

In spite of these differences, many of the approaches used in the study of precedence properties are also applicable to the analysis of response properties. One of these useful approaches is that of *verification diagrams* introduced in Section 3.3 of the SAFETY book.

4 Chain Diagrams

As observed in the previous example (program ANY-Y), in many cases it suffices to specify the intermediate assertions $\varphi_0, \varphi_1, \ldots, \varphi_m$ and to identify the helpful transitions $\tau_1, \ldots, \tau_m$. The proofs of the actual verification conditions is a detail that can be left to skeptical readers, and eventually to an automated system. Only some of the verification conditions raise interesting questions, and those are usually elaborated in a presentation of a proof. However, the main structure of the proof is adequately represented by the list of assertions and their corresponding helpful transitions.

A concise visual summary of this information, and some additional details, is provided by verification diagrams. Verification diagrams were already introduced in Section 3.3 of the SAFETY book to represent proofs using rule NWAIT. However, we give here an independent description of the diagrams that support proofs of response properties by rule CHAIN-J.

Verification Diagrams

A verification diagram is a directed labeled graph constructed as follows:

- *Nodes* in the graph are labeled by assertions. We will often refer to a node by the assertion labeling it.
- *Edges* in the graph represent transitions between assertions. The diagrams presenting proofs by rule CHAIN-J allow edges of two types, represented graphically by *single* (*-lined*) and *dashed* arrows. Each edge of either type departs from one assertion, connects to another, and is labeled by the name of a transition. We refer to an edge labeled by τ as a *τ-edge*.
- One of the nodes may be designated as a *terminal node* ("goal" node). In the graphical representation, this node is distinguished by having a boldface boundary. No edges depart from a terminal node. Terminal nodes correspond to "goal" assertions such as φ_0 in rule CHAIN-J.

Chain Diagrams

A verification diagram is said to be a *chain diagram* if its nodes are labeled by assertions $\varphi_0, \ldots, \varphi_m$, with φ_0 being the terminal node, and if it satisfies the following requirements:

- If a single (-line) edge connects node φ_i to node φ_j, then $i \geq j$.
- If a dashed (-line) edge connects node φ_i to node φ_j, then $i > j$.
- Every node φ_i, $i > 0$, has a dashed edge departing from it. This identifies the transition labeling such an edge as *helpful* for assertion φ_i. All helpful transitions must be just.

The first two requirements ensure that the diagram is weakly acyclic in the sense defined in Section 3.3 of the SAFETY book for WAIT diagrams. That is, the terminal node is labeled by φ_0 and whenever node φ_i is connected by an edge (single or dashed) to node φ_j, then $j \leq i$. The stronger second requirement ensures that the subgraph based on the dashed edges is acyclic, forbidding self-connections by dashed edges. The third requirement demands that every nonterminal assertion (i.e., φ_i for $i > 0$) has at least one helpful transition associated with it.

Verification Conditions for CHAIN Diagrams

The assertions labeling nodes in a diagram are intended to represent the intermediate assertions appearing in a CHAIN-J proof. A τ-labeled edge connecting node φ_i to node φ_j implies that it is possible for a φ_i-state to have a τ-successor satisfying φ_j. A dashed edge departing from node φ and labeled by transition τ identifies τ as helpful for assertion φ. Consequently, we associate verification conditions with nodes and the edges departing from them. These conditions, expressed by implications, represent premises J2–J4 of rule CHAIN-J.

For a node φ_i and transition τ, connecting φ_i to φ_j, we say that φ_j is a τ-successor of φ_i. Let φ be a nonterminal node and $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the τ-successors of φ.

V1. If all the edges connecting φ to its τ-successors are single (-lined), then we associate with φ and τ the verification condition

$$\{\varphi\}\ \tau\ \{\varphi\ \vee\ \varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}.$$

Transition τ, labeling only single edges, is identified as unhelpful for φ. This condition, similar to premise J2, allows τ to lead from a φ-state back to a φ-state, recording no progress.

The case of a transition τ that does not label any edges departing from φ is interpreted as though τ labels $k = 0$ single-lined edges departing from φ. That is, with such a transition we associate the verification condition

$$\{\varphi\}\ \tau\ \{\varphi\}.$$

V2. If some edge departing from φ is dashed (hence $k > 0$), we associate with φ and τ the verification condition

$$\{\varphi\}\ \tau\ \{\varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}.$$

This condition corresponds to premise J3, requiring a transition τ, identified as helpful, to lead away from φ.

V3. If τ labels some dashed edge departing from φ, we require

$$\varphi\ \rightarrow\ En(\tau).$$

This condition corresponds to premise J4, requiring that a transition helpful for φ is enabled on all φ-states. We refer to this requirement as the *enabling requirement*.

Valid CHAIN Diagrams

A CHAIN diagram is said to be *valid over a program P* (*P-valid* for short) if all the verification conditions associated with nodes of the diagram are P-state valid.

The consequences of having a P-valid CHAIN diagram are stated by the following claim:

Claim. (CHAIN diagrams)
A P-valid CHAIN diagram establishes that the response formula

$$\bigvee_{j=0}^{m} \varphi_j\ \Rightarrow\ \ \varphi_0$$

is P-valid.

If, in addition, we can establish the P-state validity of the following implications:

$$\text{(J1)}\quad p\ \rightarrow\ \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \text{(J0)}\quad \varphi_0\ \rightarrow\ q$$

then, we can conclude the P-validity of

$$p\ \Rightarrow\ \ q.$$

Justification. First, we show the first part of the claim, stating the P-validity of

$$\bigvee_{j=0}^{m} \varphi_j\ \Rightarrow\ \ \varphi_0.$$

We use rule CHAIN-J with p: $\bigvee_{j=0}^{m} \varphi_j$, $q = \varphi_0$ and, for each $i = 1, \ldots, m$, we take τ_i (the helpful transition for φ_i) to be the transition labeling the dashed edge departing from φ_i.

For our choice of p and q, premise J1 of rule CHAIN-J assumes the form

$$\text{(J1)} \quad \bigvee_{j=0}^{m} \varphi_j \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j,$$

which is trivially state-valid. We proceed to show that the P-state validity of premises J2–J4 follows from the P-validity of the diagram, for each $i = 1, \ldots, m$.

Premise J2 requires showing

$$\text{(J2)} \quad \rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \varphi'_0 \;\vee\; \varphi'_1 \;\vee\; \cdots \;\vee\; \varphi'_i,$$

for each $\tau \in \mathcal{T}$. Let $\varphi_{i_1}, \ldots, \varphi_{i_k}$ be the τ-successors of φ_i in the P-valid diagram, for $\tau \in \mathcal{T} - \{\tau_i, \tau_I\}$. By the requirement of weak acyclicity $i_1 \leq i, \ldots, i_k \leq i$. Since $\tau \neq \tau_i$, all the τ-edges departing from node φ are single-line edges and the following verification condition holds:

$$\text{V1.} \quad \rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \varphi'_i \;\vee\; \varphi'_{i_1} \;\vee\; \cdots \;\vee\; \varphi'_{i_k}.$$

Since $i_j \leq i$ for each $j = 1, \ldots, k$, the disjunction on the right-hand side of V1 is taken over a subset of the assertions appearing on the right-hand side of premise J2. It follows that J2 is state-valid for assertion φ_i and transition τ. For $\tau = \tau_I$, premise J2 holds trivially since ρ_{τ_I} implies $\varphi'_i = \varphi_i$. For $\tau = \tau_i$, premise J2 is implied by J3.

Premise J3 requires

$$\text{(J3)} \quad \rho_{\tau_i} \;\wedge\; \varphi_i \;\rightarrow\; \varphi'_0 \;\vee\; \varphi'_1 \;\vee\; \cdots \;\vee\; \varphi'_{i-1}.$$

Let $\varphi_{i_1}, \ldots, \varphi_{i_k}$ be the τ_i-successors of φ_i in the P-valid diagram. Since all τ_i-edges departing from φ_i are dashed, $i_j < i$ for $j = 1, \ldots, k$ and the following verification condition holds:

$$\text{V2.} \quad \rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \varphi'_{i_1} \;\vee\; \cdots \;\vee\; \varphi'_{i_k}.$$

Repeating the subset argument, this implies the state validity of premise J3.

Premise J4 is identical to condition V3 for every φ_i and τ_i, $i = 1, \ldots, m$.

Next, we consider the more general case of p and q which are not identical to $\bigvee_{j=0}^{m} \varphi_j$ and φ_0, respectively, but satisfy the implications J1 and J0. Applying rule MON-R to p, $\bigvee_{j=0}^{m} \varphi_j$, φ_0, and q (standing for p, q, r, and t in MON-R), we obtain the conclusion $p \;\Rightarrow\; q$. ∎

Note that chain diagrams and their notion of validity are a conservative extension of the WAIT diagrams, introduced in Section 3.3 of the SAFETY book. The additional requirements that disallow a self-connecting edge all refer to dashed edges

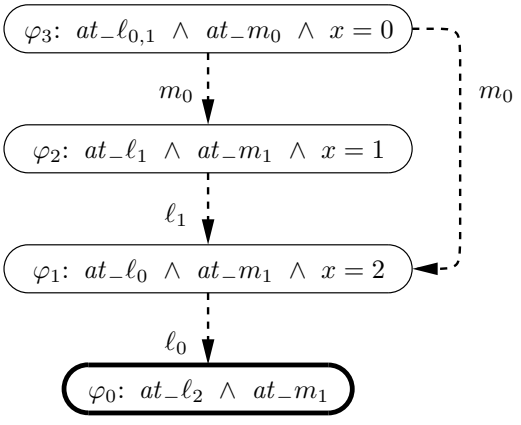


Fig. 8. CHAIN diagram for termination

which are not present in WAIT diagrams. It follows that a P-valid CHAIN diagram is also P-valid for proving the nested waiting-for formula

$$\bigvee_{j=0}^{m} \varphi_j \quad \Rightarrow \quad \varphi_m \ \mathcal{W} \ \varphi_{m-1} \cdots \varphi_1 \ \mathcal{W} \ \varphi_0.$$

Example (program ANY-Y)

Consider again program ANY-Y (Fig. 2). The CHAIN diagram of Fig. 8 provides a graphical representation for the proof of the response property

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = 0}_{p=\Theta} \quad \Rightarrow \quad \Big(\underbrace{at\_\ell_2 \ \wedge \ at\_m_1}_{q=\varphi_0} \Big) \ ,$$

for program ANY-Y.

The diagram identifies $\varphi_3, \ldots, \varphi_0$ as the intermediate assertions and m_0, ℓ_1, ℓ_0 as their corresponding helpful transitions. This CHAIN diagram is valid over program ANY-Y, which establishes the P-validity of

$$\bigvee_{j=0}^{3} \varphi_j \quad \Rightarrow \quad (at\_\ell_2 \ \wedge \ at\_m_1)$$

over this program. Since (as shown above) $\Theta \rightarrow \varphi_3$, the second part of Claim 4 establishes the P-validity of the termination property

$$\Theta \quad \Rightarrow \quad (at\_\ell_2 \ \wedge \ at\_m_1). \qquad \blacksquare$$

The Advantages of Diagrams

One of the advantages of the presentation of a CHAIN-J proof sketch by a CHAIN diagram, over its presentation by a list of assertions and their corresponding helpful transitions is that the diagram provides a stronger (and more detailed)

version of premises J2 and J3 than is standardly provided by rule CHAIN-J and a list of the assertions and helpful transitions.

Consider, for example, the proof presented in Fig. 8. Both the diagram and the textual proof identify φ_2 as $at\_\ell_1 \wedge at\_m_1 \wedge x = 1$ and ℓ_1 as its helpful transition.

However, while rule CHAIN-J suggests that we prove for premise J3 the verification condition

$$\{\varphi_2\}\ \ell_1\ \{\varphi_0 \vee \varphi_1\},$$

the diagram claims that the even stronger condition

$$\{\varphi_2\}\ \ell_1\ \{\varphi_1\}$$

is P-state valid. This results from the fact that there is no ℓ_1-edge connecting φ_2 to φ_0.

Encapsulation Conventions

There are several encapsulation conventions that lead to more structured hierarchical diagrams and improve the readability and manageability of large complex diagrams. These conventions were introduced in Section 3.3 of the SAFETY book and we reproduce them here briefly, to make the presentation self contained. The basic construct of encapsulation is that of a *compound node* that may contain several internal nodes. The encapsulation conventions attribute to a compound node aspects and relations that are common to all of their contained nodes. We refer to the contained nodes as *descendants* of the compound node. Nodes that are not compound are called *basic nodes*. We use three encapsulation conventions.

- *Departing edges*

  An edge departing from a compound node is interpreted as though it departed from each of its descendants. This is represented by the graphical equivalence of Fig. 9.

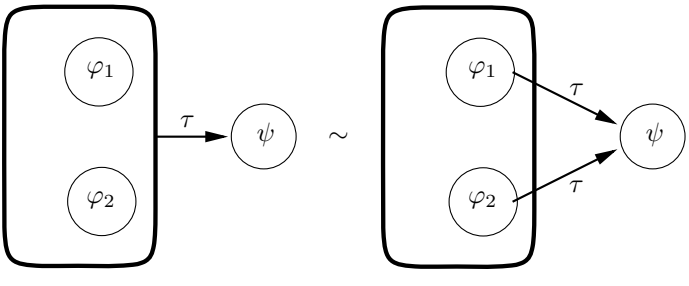


Fig. 9. Departing edges

- *Arriving edges*

  In a similar way, an edge arriving at a compound node is interpreted as though it arrived at each of its descendants. This is represented in the graphical equivalence of Fig. 10.

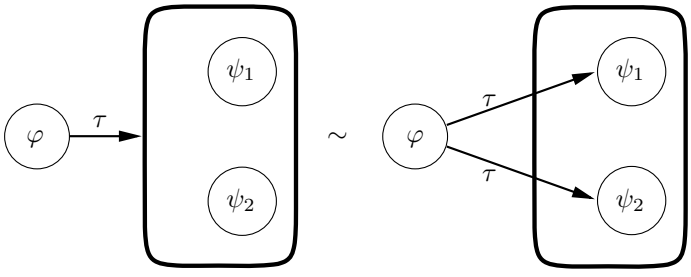


Fig. 10. Arriving edges

- *Common factors*

  An assertion φ labeling a compound node is interpreted as though it were a conjunct added to each of the labels of its descendants. This is represented by the graphical equivalence of Fig. 11. We refer to φ as a *common factor* of the two nodes.

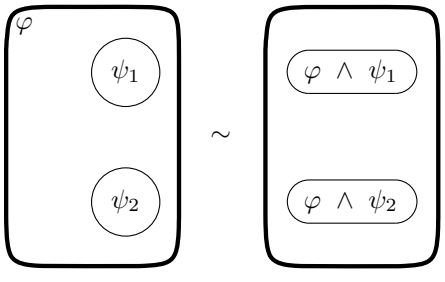


Fig. 11. Common factors

Example. In Fig. 12 we present a verification diagram which is the encapsulated version of the verification diagram of Fig. 8.

This encapsulation uses the arriving edge convention to denote by a single arrow the two edges connecting φ_3 to φ_2 and to φ_1. It uses the common factor convention to simplify the presentation of φ_1 and φ_2. ⌟

Additional Examples

Let us consider a few more examples for the application of rule CHAIN-J and CHAIN diagrams, illustrating the encapsulation conventions.

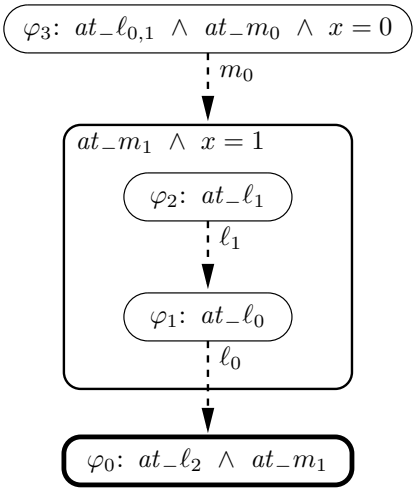


Fig. 12. Encapsulated version of CHAIN diagram for termination

Example (Peterson's Algorithm — version 1)

For the next example, we return to program MUX-PET1, Peterson's algorithm for mutual exclusion (Fig. 13). This program was previously studied in Section 1.4 of the SAFETY book, where we established for it the following invariants:

$$\psi_0: \quad s = 1 \ \vee \ s = 2$$

$$\psi_1: \quad y_1 \ \leftrightarrow \ at\_\ell_{3..5}$$

$$\psi_2: \quad y_2 \ \leftrightarrow \ at\_m_{3..5}.$$

We wish to prove for this program the response property of accessibility, given by

$$\underbrace{at\_\ell_2}_{p} \ \Rightarrow \ \underbrace{at\_\ell_4}_{q} \ .$$

To use rule CHAIN-J or its diagram representation, we have to identify intermediate assertions that characterize the intermediate situations between the starting assertion p: at_ℓ_2 and the goal assertion q: at_ℓ_4. It is obvious that the first helpful step in the progress from ℓ_2 to ℓ_4 is process P_1 moving from ℓ_2 to ℓ_3. Consequently, we can safely take φ_m to be at_ℓ_2 and the helpful transition τ_m to be ℓ_2. We cannot yet determine the value of m because it depends on the number of helpful steps necessary to get from ℓ_3 to ℓ_4. We can now concentrate on showing how to get from ℓ_3 to ℓ_4.

Similar to the heuristics employed in the application of rule NWAIT (Section 3.3 of the SAFETY book), it is often useful to work backwards from the goal assertion at_ℓ_4. Consequently, we take φ_0 to be at_ℓ_4.

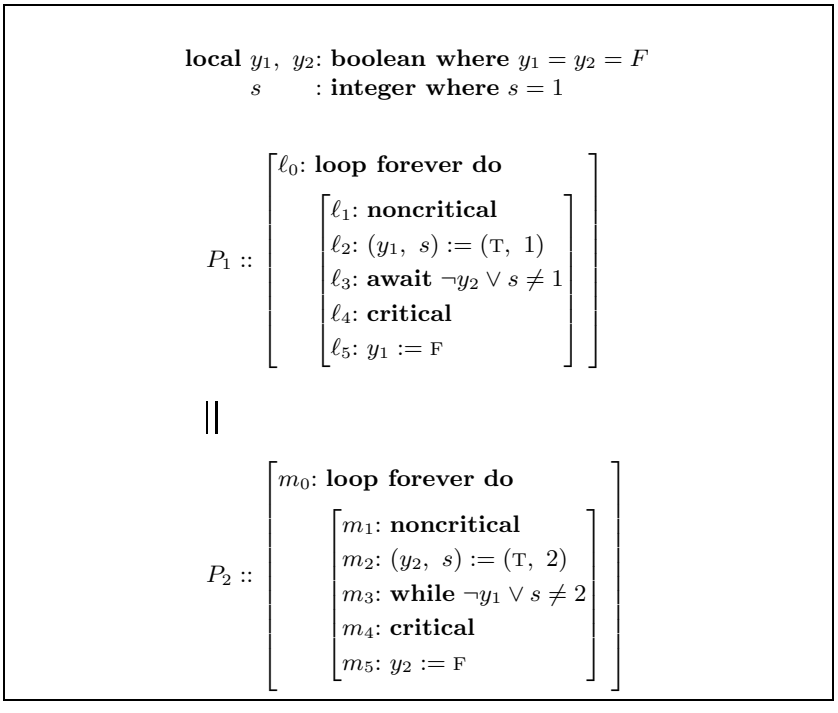


Fig. 13. Program MUX-PET1 (Peterson's algorithm) — version 1

For the previous intermediate assertion φ_1, we look for situations that are only one helpful step away from φ_0. Clearly, the only transition that can accomplish φ_0 in one step is ℓ_3. If we choose the helpful transition τ_1 to be ℓ_3, then the appropriate φ_1 is the assertion characterizing all the states on which ℓ_3 is enabled. Therefore, as φ_1 we take the enabling condition of ℓ_3

$$\varphi_1: \quad at\_\ell_3 \ \wedge\ (\neg y_2 \ \vee\ s \neq 1).$$

Assertion φ_1 does not yet cover all the accessible states satisfying at_ℓ_3. Consequently, we cannot take m to be 2, and must search for additional assertions, characterizing states that satisfy at_ℓ_3 and that are one helpful step away from φ_1. Therefore, we look for transitions of P_2 that may change the disjunction $\neg y_2 \vee s = 2$ from F to T. The only candidate transition is m_5, which sets y_2 to F. Consequently, we take

$$\varphi_2: \quad at\_\ell_3 \ \wedge\ at\_m_5 \ \wedge\ y_2 \ \wedge\ s = 1.$$

The conjunct $y_2 \wedge s = 1$ can be safely added to φ_2 since all the states satisfying $at\_\ell_3 \wedge (\neg y_2 \vee s \neq 1)$ are already covered by φ_1.

Looking for $(at\_\ell_3 \wedge y_2 \wedge s = 1)$-states that are one helpful step away from φ_2, we easily identify

$$\varphi_3: \quad at\_\ell_3 \ \wedge\ at\_m_4 \ \wedge\ y_2 \ \wedge\ s = 1.$$

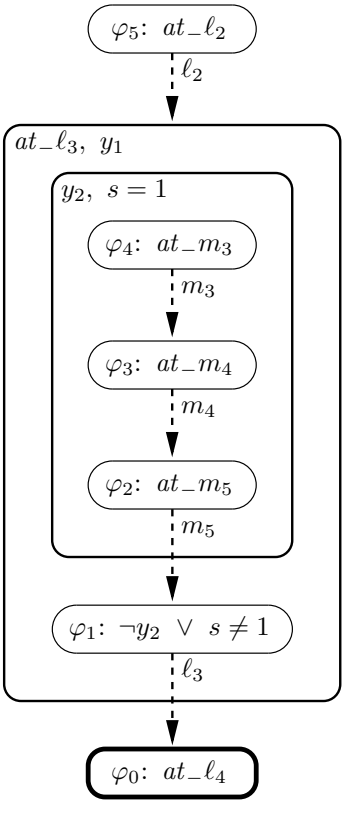


Fig. 14. CHAIN diagram for program MUX-PET1

In a similar way, we can identify the preceding assertion as

$$\varphi_4\colon \quad at\_\ell_3\ \wedge\ at\_m_3\ \wedge\ y_2\ \wedge\ s = 1.$$

Note that m_3, which is the helpful transition for φ_4, is enabled on all states satisfying φ_4.

At this point, we find out that the disjunction $\varphi_1 \vee \cdots \vee \varphi_4$ covers the range of all accessible (at_ℓ_3)-states. This is because P_2 must be in one of the locations $m_0, \ldots, m_4$. Due to ψ_2, the range $m_{0..2}$ is covered by φ_1 under the $\neg y_2$ disjunct. Locations $m_3 \ldots, m_5$ for $s \neq 1$ are covered by φ_1 under the disjunct $s \neq 1$, while the same locations for the case that $s = 1$ are covered by assertions $\varphi_4, \varphi_3, \varphi_2$, respectively.

In Fig. 14 we present a CHAIN diagram using the intermediate assertions constructed through the preceding analysis.

Note that we have grouped under φ_1 many possible states of P_2, and have not represented the movement of P_2 through them. This is justified by the fact that ℓ_3 is enabled on all of these states and is the transition declared as helpful for φ_1. In contrast, we separated m_3, m_4, and m_5, because the helpful transitions there changed from one of these states to the next. ⌟

In **Problem 1**, the reader is requested to establish accessibility for another algorithm for mutual exclusion.

Example (Peterson's Algorithm — Version 2)

Consider the refined program MUX-PET2, version 2 of Peterson's algorithm, presented in Fig. 15.

In Section 1.4 of the SAFETY book, we established the following invariants for this program:

$$\chi_0\colon \quad s = 1 \ \vee \ s = 2$$

$$\chi_1\colon \quad y_1 \ \leftrightarrow \ at\_\ell_{3..6}$$

$$\chi_2\colon \quad y_2 \ \leftrightarrow \ at\_m_{3..6}.$$

We intend to verify the property of accessibility for program MUX-PET2, which can be expressed by the response formula

$$\underbrace{at\_\ell_2}_{p} \ \Rightarrow \ \underbrace{at\_\ell_5}_{q} \ .$$

The construction of the appropriate verification diagram starts in a similar way to the diagram for program MUX-PET1 of the previous example. We take φ_m to be at_ℓ_2. From ℓ_2, process P_1 can proceed at its own pace to ℓ_3, which we take as φ_{m-1}. The next step taken by P_1 leads into ℓ_4 where a more detailed analysis is necessary.

To perform this detailed analysis we take φ_0 to be the goal assertion at_ℓ_5. What should we take as φ_1? In the preceding case, we characterized φ_1 as being one helpful step away from φ_0. This characterization is not sufficient here. Another requirement is that if s' is a successor of a φ_1-state, then s' should satisfy either φ_1 or φ_0: at_ℓ_5. This shows that we cannot take φ_1 to be, as before, the assertion $at\_\ell_4 \wedge (\neg y_2 \vee s \neq 1)$. This is because the accessible state

$$s\colon \quad \langle \pi\colon \{\ell_4, m_2\}, \ y_1\colon \mathrm{T}, \ y_2\colon \mathrm{F}, \ s\colon 1 \rangle$$

satisfies the candidate assertion $at\_\ell_4 \wedge (\neg y_2 \vee s \neq 1)$ but has an m_2-successor given by

$$s'\colon \quad \langle \pi\colon \{\ell_4, m_3\}, \ y_1\colon \mathrm{T}, \ y_2\colon \mathrm{T}, \ s\colon 1 \rangle$$

which satisfies neither the candidate assertion nor φ_0.

We observe that the cause for this problem is the disjunct $\neg y_2$ which can be falsified (changed from T to F) by transition m_2 of P_2. There is no such problem with the disjunct $s \neq 1$ which cannot be falsified by P_2. Consequently, we take

$$\varphi_1\colon \quad at\_\ell_4 \ \wedge \ s \neq 1.$$

The only transition that can lead from a $\neg\varphi_1$-state to a φ_1-state is m_3. Therefore, we take φ_2 to be

$$\varphi_2\colon \quad at\_\ell_4 \ \wedge \ at\_m_3 \ \wedge \ s = 1.$$

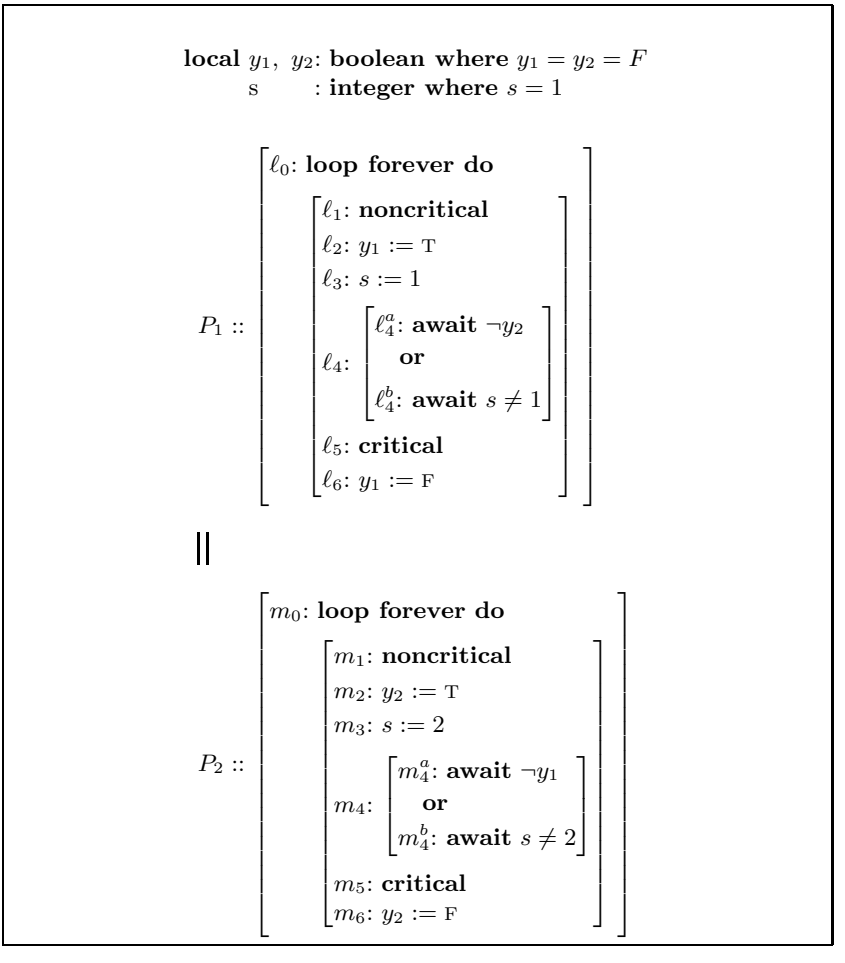


Fig. 15. Program MUX-PET2 (Peterson's algorithm) — version 2

We also realize that the transition leading into φ_2 is m_2 which changes y_2 from F to T and preserves the value of s. Consequently, we take φ_3 to be

$$\varphi_3: \quad at\_m_{0..2} \ \wedge \ \neg y_2 \ \wedge \ s = 1.$$

By now, we have covered all the states satisfying $at\_\ell_4 \wedge (\neg y_2 \vee s \neq 1)$. From now on, the analysis proceeds as it did for program MUX-PET1. The final CHAIN diagram is presented in Fig. 16.

This diagram partitions the range $m_{0..4}$ into three regions. The region $m_{0..2}$, represented by φ_3, guarantees that ℓ_4^a is enabled (but not necessarily that m_2 is enabled, which is therefore drawn as a single edge). However it may evolve into φ_2, where no transition of P_1 is guaranteed to be enabled. Being at φ_2, m_3 is the helpful transition which eventually leads into φ_1. In φ_1, ℓ_4^b is enabled, and since P_2 cannot falsify $s \neq 1$, eventually ℓ_4^b is taken and leads to φ_0.

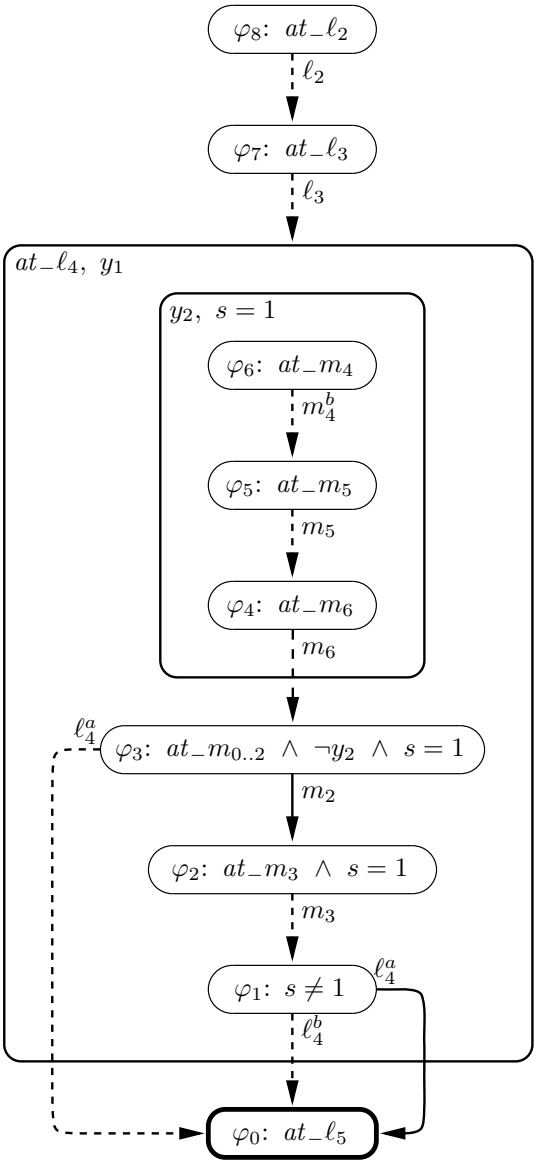


Fig. 16. CHAIN diagram for program MUX-PET2

The edge labeled ℓ_4^a connecting node φ_1 to node φ_0 represents the possibility that ℓ_4^a may be enabled on a state satisfying $s \neq 1$. A more careful analysis shows that φ_1 in this diagram can be strengthened to the assertion

$$\widehat{\varphi}_1\colon \quad at\_m_4 \ \wedge \ s \neq 1 \ \wedge \ y_2,$$

and then this edge is unnecessary. ⌟

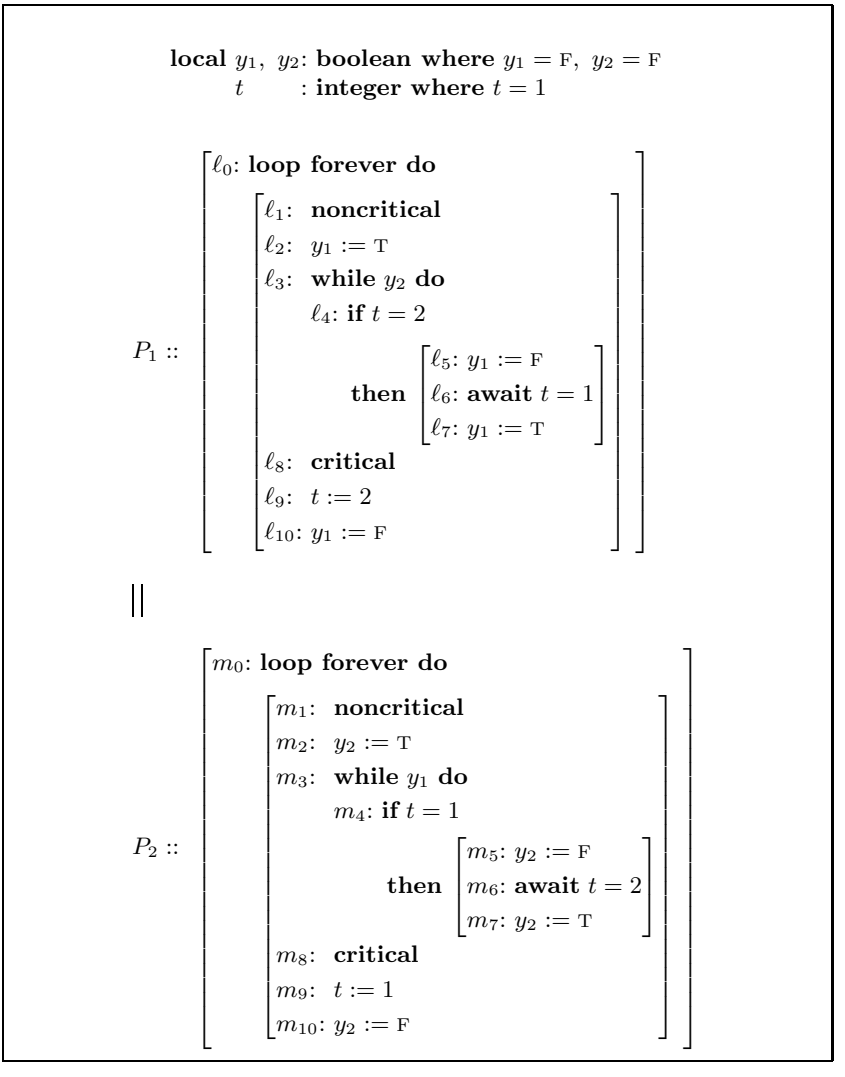


Fig. 17. Program MUX-DEK (Dekker's algorithm for mutual exclusion)

In **Problem 2**, the reader is requested to verify accessibility for a family of mutual exclusion algorithms, known as the *bakery algorithms*.

Example (Dekker's algorithm)

Dekker's algorithms for solving the mutual exclusion problem is presented in program MUX-DEK of Fig. 17.

In comparison to Peterson's algorithm, Dekker's algorithm has a relatively simple safety proof but rather elaborate proof of accessibility.

- *Invariants*

In Section 1.4 of the SAFETY book we derived the following invariants for program MUX-DEK:

$$\chi_1\colon \quad t = 1 \ \vee \ t = 2$$

$$\chi_2\colon \quad y_1 \ \leftrightarrow \ (at\_\ell_{3..5,8..10})$$

$$\chi_3\colon \quad y_2 \ \leftrightarrow \ (at\_m_{3..5,8..10})$$

These are the invariants we needed to prove the mutual exclusion property, i.e., the invariance of

$$\chi_4\colon \quad \neg(at\_\ell_{8..10} \ \wedge \ at\_m_{8..10}).$$

As we will see, additional invariants are needed for the support of the response property. We will develop them as they are needed.

- *Response*

The main response property of this algorithm is, of course, that of accessibility. It is stated by

$$\psi\colon \quad at\_\ell_2 \ \Rightarrow \quad at\_\ell_8.$$

We partition the proof of the accessibility property into three lemmas, proving respectively.

Lemma A. $at\_\ell_2 \ \Rightarrow \quad \Big((at\_\ell_4 \ \wedge \ t = 2) \ \vee \ (at\_\ell_{3..7} \ \wedge \ t = 1) \ \vee \ at\_\ell_8\Big)$

Lemma B. $at\_\ell_4 \ \wedge \ t = 2 \ \Rightarrow \quad (at\_\ell_{3..7} \ \wedge \ t = 1)$

Lemma C. $at\_\ell_{3..7} \ \wedge \ t = 1 \ \Rightarrow \quad at\_\ell_8.$

Obviously, the difficult part of the protocol is the loop at $\ell_{3..7}$. Being within this loop, P_1 is considered to have a higher priority when $t = 1$. Lemma A claims that if P_1 is just starting its journey towards the critical section, then it will either reach ℓ_4 with a lower priority, or get to $\ell_{3..7}$ with a higher priority, or reach ℓ_8. Lemma B claims that if P_1 is at ℓ_4 with a low priority it will stay within the loop and eventually gain a high priority. Lemma C shows that if P_1 is within this loop and has a higher priority, then it will eventually get to ℓ_8.

Clearly, by combining these three response properties, using the transitivity of response rule TRNS-R we obtain the required accessibility property.

■ *Proof of Lemma A*

The proof of the response property

$$at\_\ell_2 \ \Rightarrow \quad \Big((at\_\ell_4 \ \wedge \ t = 2) \ \vee \ (at\_\ell_{3..7} \ \wedge \ t = 1) \ \vee \ at\_\ell_8\Big)$$

is presented in the CHAIN diagram of Fig. 18.

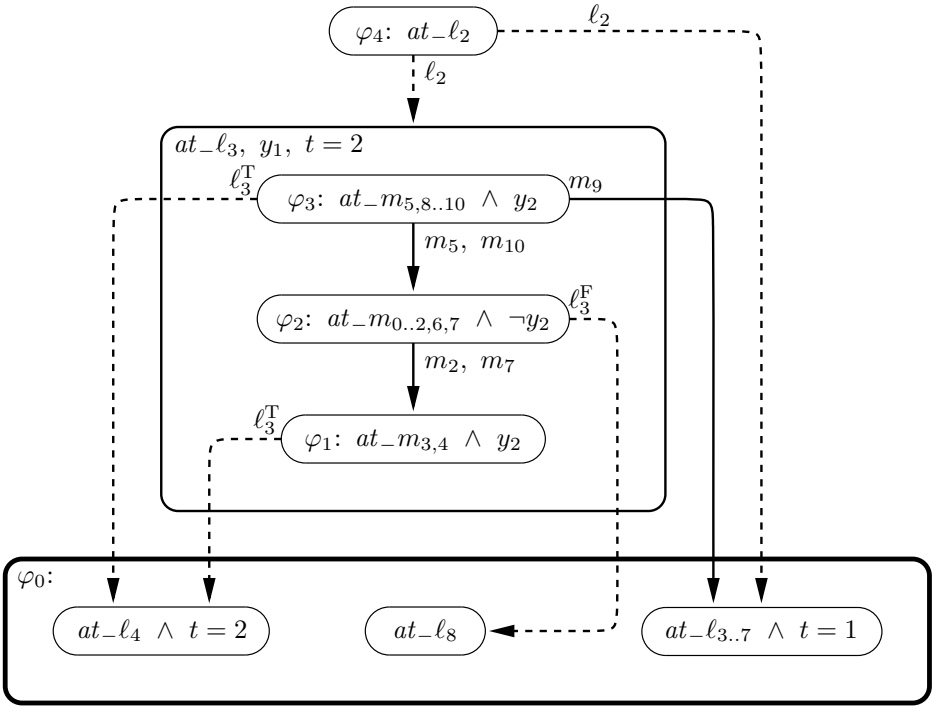


Fig. 18. CHAIN diagram for Lemma A

It is easy to follow P_1 from ℓ_2 to ℓ_3. If $t = 1$ on entry to ℓ_3, then we are already at the goal $at\_\ell_{3..7} \wedge t = 1$. Otherwise, we enter ℓ_3 with $t = 2$, setting y_1 to T. Here we examine the possible locations of P_2. Assertions φ_1, φ_2, and φ_3 cover all the possibilities. The possible motions of P_2 within these three assertions consist of taking m_9, setting t to 1, which raises the priority of P_1 and attains the goal $at\_\ell_{3..7} \wedge t = 1$. The other possible movements are from φ_3 to φ_2, and then to φ_1. Being at $m_{3,4}$ with $y_1 = \mathrm{T}$ and $t = 2$, P_2 cannot move elsewhere. Transition mode ℓ_3^{T} is enabled on φ_1 and φ_3 states, while mode ℓ_3^{F} is enabled on φ_2. Both are helpful since they lead to $at\_\ell_4 \wedge t = 2$ and at_ℓ_8, respectively.

■ *Proof of Lemma B*

The proof of the response property

$$at\_\ell_4\ \wedge\ t = 2\ \Rightarrow\qquad (at\_\ell_{3..7}\ \wedge\ t = 1)$$

is presented in the CHAIN diagram of Fig. 19.

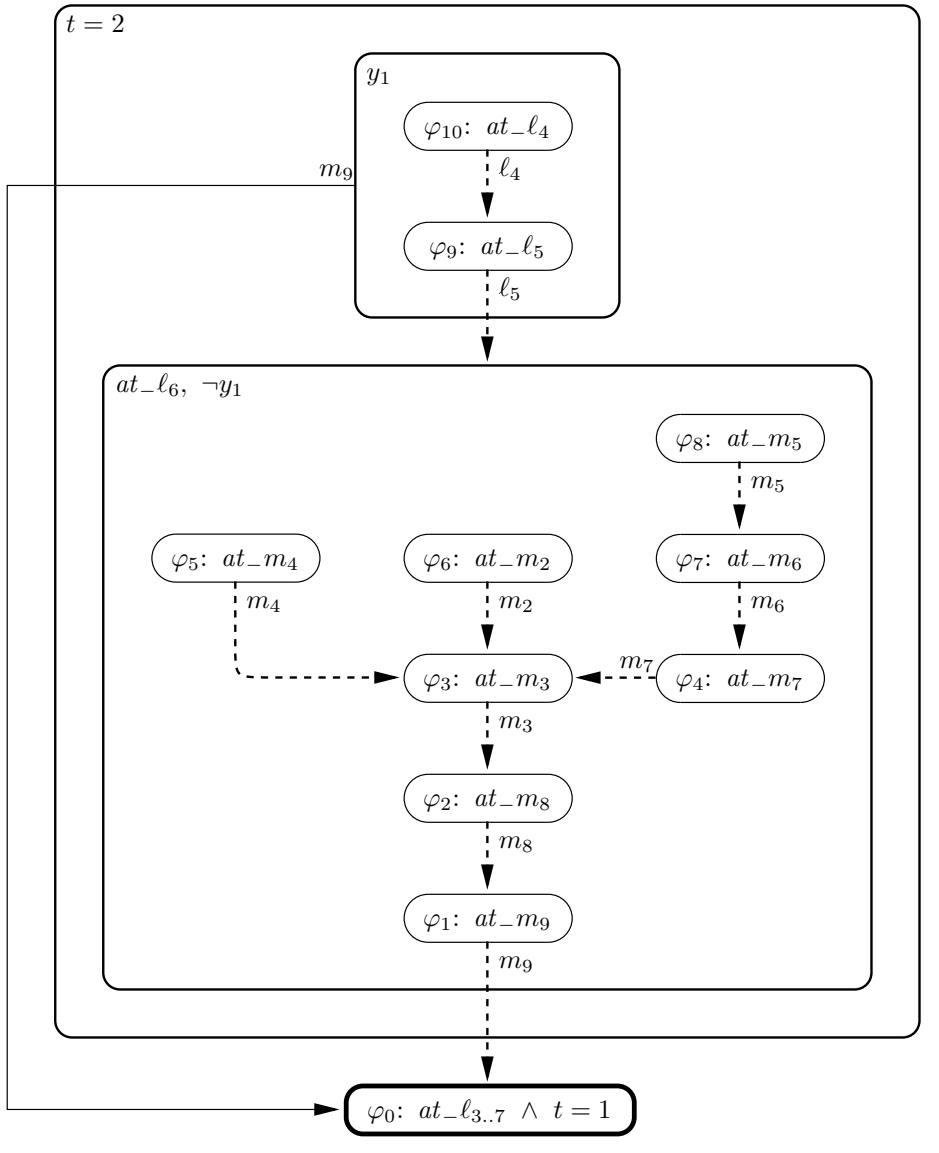


Fig. 19. CHAIN diagram for Lemma B

From ℓ_4, P_1 proceeds to ℓ_5 since $t = 2$, and then to ℓ_6 while resetting y_1 to F. While being at $\ell_{4,5}$, P_2 may still set t to 1 by performing m_9, which leads to the goal $at\_\ell_{3..7} \wedge t = 1$.

However, once P_1 enters ℓ_6, it stays at ℓ_6 waiting for t to change to 1. At that point we have to inspect where P_2 may currently be. We consider as possible locations of P_2 all of m_2–m_9, tracing their possible flow under the relatively stable situation of $t = 2$, $y_1 = \text{F}$. We see that all transitions are enabled and lead to m_9 which eventually sets t to 1 as required.

A tacit assumption made in this diagram is the exclusion of m_{10}, m_0, and m_1, as possible locations while P_1 is at ℓ_6 with $y_1 = \text{F}$ and $t = 2$. This assumption must hold for the program, if we believe Lemma B to be valid. Indeed, consider the situation that P_1 is waiting at ℓ_6 with $y_1 = \text{F}$ and $t = 2$, while P_2 is at m_1. Since P_2 is allowed to stay at the noncritical section forever, this would lead to a deadlock, denying accessibility from P_1.

We must therefore conclude that if the algorithm is correct, and guarantees accessibility to both processes, then the following assertion must be invariant

$$at\_\ell_6 \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}.$$

This invariance follows from the stronger invariant

$$\chi_5\text{:} \quad at\_\ell_{4..6} \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}$$

which we will prove.

By symmetry one can also require the invariance of

$$\chi_6\text{:} \quad at\_m_{4..6} \ \wedge \ t = 1 \ \rightarrow \ at\_\ell_{2..9}.$$

- *Proof of invariant* χ_5

Clearly,

$$\underbrace{\cdots \ \wedge \ at\_\ell_0 \ \wedge \ \cdots}_{\Theta} \ \rightarrow \ \underbrace{at\_\ell_{4..6} \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}}_{\chi_5}$$

holds.

Let us check the verification conditions for assertion χ_5, which are of the form

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ \neg y_1 \ \wedge \ \neg y_2 \ \wedge \ t = 1}_{\Theta} \ \wedge \ \underbrace{at\_\ell_{4..6} \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}}_{\chi_5} \ \rightarrow$$
$$\underbrace{at'\_\ell_{4..6} \ \wedge \ t' = 2 \ \rightarrow \ at'\_m_{2..9}}_{\chi'_5} .$$

There are three transitions that may potentially falsify assertion χ_5.

- Transition m_9

  Sets t to 1 which makes $t' = 2$ false and hence preserves the assertion.

- Transition ℓ_9

  Leads to at_ℓ_{10} which makes $at'\_\ell_{4..6}$ false.

- ℓ_3^{T} while $t = 2$

  This is possible only if $y_2 = \mathrm{T}$ which, by χ_3, implies $at\_m_{3..5} \vee at\_m_{8..10}$, and therefore $at\_m_{2..10}$. This almost gives us $at\_m_{2..9}$, with the exception of m_{10}. We thus need additional information that will exclude the possibility of P_2 being at m_{10} while $t = 2$.

Clearly, while entering m_{10} from m_9, P_2 sets t to 1. Can P_1 change it back to 2, while P_2 is still at m_{10}? The answer is no, because m_{10}, as we see in χ_4, is still a part of the critical section and is therefore exclusive of ℓ_9, the only statement capable of changing t to 2.

This suggests the invariant

$$\chi_7\colon \quad at\_m_{10} \;\rightarrow\; t = 1$$

and its symmetric counterpart

$$\chi_8\colon \quad at\_\ell_{10} \;\rightarrow\; t = 2.$$

To prove χ_7, we should inspect two transitions:

- Transition m_9

  Sets t to 1.

- ℓ_9 while at_m_{10}

  Impossible due to χ_4.

This establishes χ_7 and similarly χ_8. Having χ_7 we can use it to show that the last transition considered in the proof of χ_5, namely ℓ_3^{T} while $t = 2$, implies $at\_m_{2..9}$, which establishes χ_5.

- *Proof of Lemma C*

Lemma C states that if P_1 is within the waiting loop $\ell_{3..7}$ with higher priority, i.e., $t = 1$, then eventually it will reach ℓ_8. It is stated by

$$at\_\ell_{3..7} \;\wedge\; t = 1 \;\Rightarrow\; \qquad at\_\ell_8.$$

The proof is presented in the CHAIN diagram of Fig. 20.

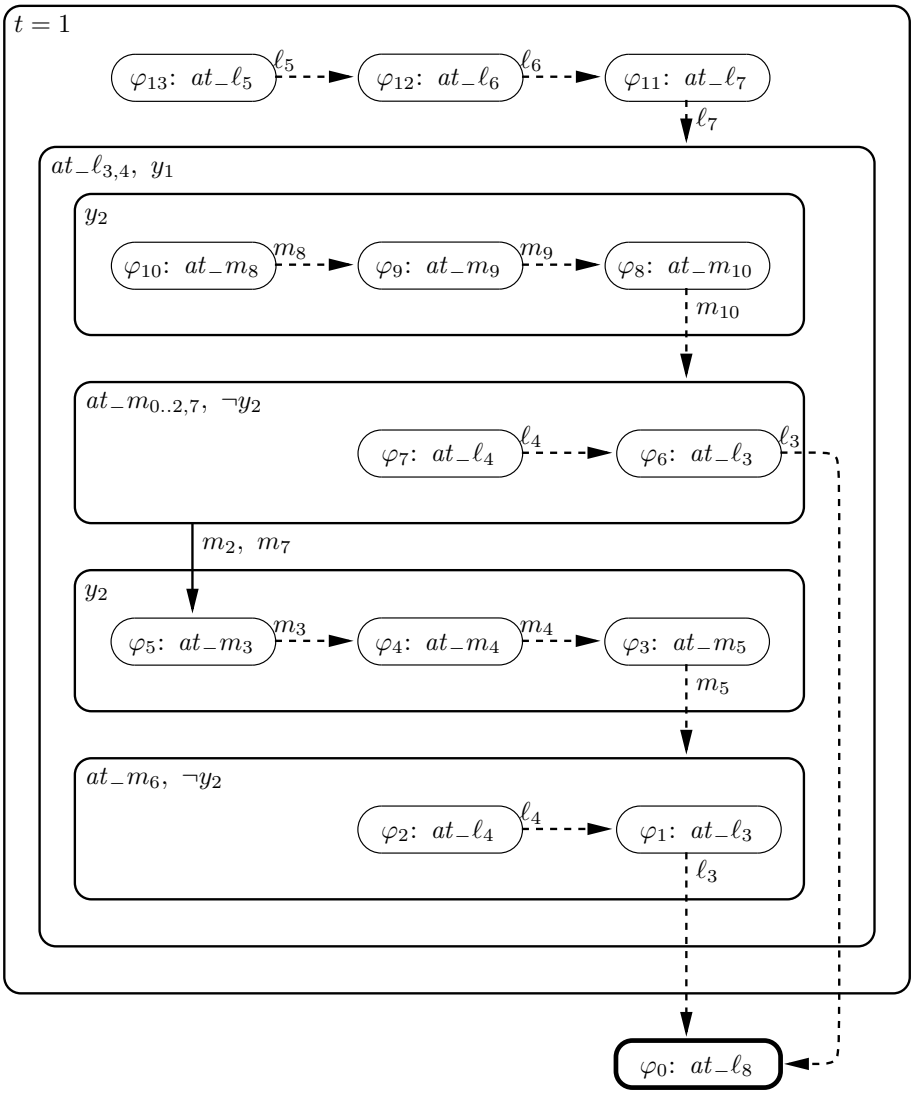


Fig. 20. CHAIN diagram for Lemma C

Note our efforts to minimize the number of assertions by grouping together situations with different control configurations, wherever possible. Thus for all the states where $y_1 = \text{T}$ and P_1 is either at ℓ_3 or at ℓ_4, we do not distinguish between these two possibilities, but partition the diagram according to the location of P_2. This is because, in this general situation, it is P_2 which is the helpful process and we have to trace its progress. On the other hand, when $y_2 = \text{F}$, P_1 becomes the helpful process and we start distinguishing between the cases of at_ℓ_3 and at_ℓ_4, while lumping together the locations of P_2 into two groups: $m_{0..2,7}$ and m_6. These two groups must be distinguished because it is possible (though not guaranteed) to exit the first group into a situation where $y_2 = \text{T}$, but it is impossible to exit m_6 into such a situation. This is because when P_2 is at m_6 with $t = 1$, it cannot progress until t is changed to 2. ▟

In **Problem 3**, the reader is requested to prove accessibility for two variants of Dekker's algorithm.

Case Splitting According to the Helpful Transitions

In the preceding examples, the main reason for using rule CHAIN-J with $m > 1$ intermediate assertions has been that the program requires m helpful steps to reach the goal. In most of these applications there always was a worst case computation that actually visited each of the assertions, starting with φ_m and proceeding through $\varphi_{m-1}, \varphi_{m-2}, \ldots$ up to $\varphi_0 = q$.

This is not the only motivation for using several intermediate assertions. Another good reason for wishing to partition the state space lying between p and q into several assertions is that different states in that space may require different helpful transitions for getting them closer to the goal.

Example (maximum)

Consider, for example, program MAX presented in Fig. 21. This program places in output variable z the maximum of inputs x and y. The program consists of two parallel statements that compare the values of x and y.

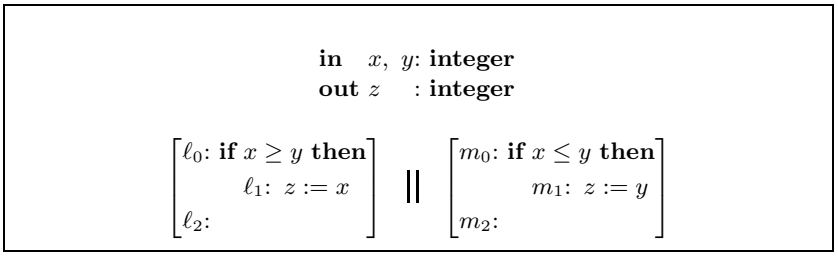


Fig. 21. Program MAX (maximum)

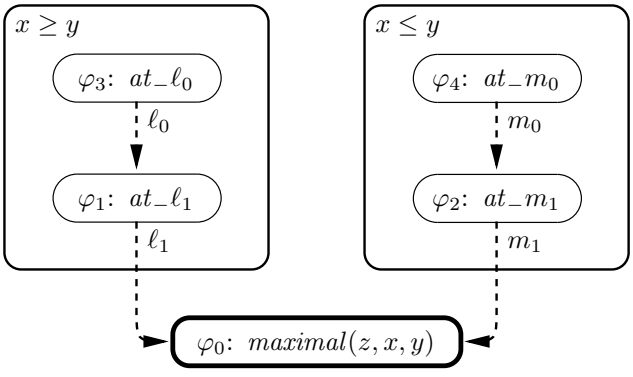


Fig. 22. CHAIN diagram for program MAX

The response statement we would like to prove for this program is

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0}_{p} \quad \Rightarrow \quad \underbrace{maximal(z,\ x,\ y)}_{q},$$

where $maximal(z, x, y)$ stands for the formula

$$maximal(z,\ x,\ y): \quad (z = x \ \vee \ z = y) \ \wedge \ z \geq x \ \wedge \ z \geq y,$$

claiming that z is the maximum of x and y.

Clearly, the goal of this response property is achieved in the helpful steps which are either ℓ_0 and ℓ_1 or m_0 and m_1. Perhaps one would expect a proof of this property by rule RESP-J that only uses one intermediate assertion φ.

However, no such proof exists. The reason for this is that we cannot identify a *single* transition that is helpful for all the states satisfying p: $at\_\ell_0 \wedge at\_m_0$. Clearly, for all states satisfying $p \wedge x \geq y$, ℓ_0 is the helpful transition, while for states satisfying $p \wedge x \leq y$, m_0 is the helpful transition. Consequently, we need at least *four* intermediate assertions in a proof of this property by rule CHAIN-J.

We choose the following assertions and helpful transitions

φ_4:	$at\_m_0 \ \wedge \ x \leq y$	τ_4:	m_0
φ_3:	$at\_\ell_0 \ \wedge \ x \geq y$	τ_3:	ℓ_0
φ_2:	$at\_m_1 \ \wedge \ x \leq y$	τ_2:	m_1
φ_1:	$at\_\ell_1 \ \wedge \ x \geq y$	τ_1:	ℓ_1
φ_0:	q.		

It is straightforward to verify that all the premises of rule CHAIN-J are satisfied by this choice.

In Fig. 22, we present a CHAIN diagram for the proof of the considered response property.

Assertions φ_3 and φ_4 partition (non exclusively) the situation $at\_\ell_0 \wedge at\_m_0$ into states for which ℓ_0 is helpful and has not been taken yet, and states for which m_0 is helpful and has not been taken yet.

It is not difficult to verify that taking ℓ_0 from a φ_3-state, as well as taking m_0 from a φ_4-state, leads to φ_1 and φ_2, respectively. Choosing φ_4 to rank above φ_3 is quite arbitrary. In particular, we do not have a computation that goes from φ_4-states to φ_3-states. Every computation follows either the φ_3, φ_1 route or the φ_4, φ_2 route. ▟

5 Well-Founded Rule

In rule CHAIN-J we treated each of the participating assertions $\varphi_0, \ldots, \varphi_m$ as separate entities, and made no attempt to find a uniform representation for φ_j as a single formula involving j. This approach is adequate for response properties which require a *bounded* number of steps for their achievement, e.g., at most five in the case of program MUX-PET1 (Fig. 14). The bound must be uniform and independent of the initial state.

There are many cases, however, in which no such bound can be given a priori. To deal with these cases, we must generalize the induction over a fixed finite subrange of the integers, such as $0, 1, \ldots, m$ in rule CHAIN-J, into an explicit induction over an arbitrary well-founded relation.

Well-Founded Domains

We define a *well-founded domain* $(\mathcal{A}, \succ)$ to consist of a set $\mathcal{A}$ and a *well-founded order* relation $\succ$ on $\mathcal{A}$. A binary relation $\succ$ is called an *order* if it is

- transitive: $a \succ b$ and $b \succ c$ imply $a \succ c$, and
- irreflexive: $a \succ a$ for no $a \in \mathcal{A}$.

The relation $\succ$ is called *well-founded* if there does not exist an infinitely descending sequence $a_0, a_1, \ldots$ of elements of $\mathcal{A}$ such that

$$a_0 \succ a_1 \succ \cdots .$$

A typical example of a well-founded domain is $(\mathbb{N}, >)$, where $\mathbb{N}$ are the natural numbers (including 0) and $>$ is the greater-than relation. Clearly, $>$ is well-founded over the natural numbers, because there cannot exist an infinitely descending sequence of natural numbers

$$n_0 > n_1 > n_2 > \ldots .$$

For an arbitrary order relation $\succ$ on $\mathcal{A}$, we define its *reflexive extension* $\succeq$ to hold between $a, a' \in \mathcal{A}$, written $a \succeq a'$, if either $a \succ a'$ or $a = a'$.

The Lexicographic Product

Given two well-founded domains, $(\mathcal{A}_1, \succ_1)$ and $(\mathcal{A}_2, \succ_2)$, we can form their *lexicographical product* $(\mathcal{A}, \succ)$, where

$\mathcal{A}$ is defined as $\mathcal{A}_1 \times \mathcal{A}_2$, i.e., the set of all pairs (a_1, a_2), such that $a_1 \in \mathcal{A}_1$ and $a_2 \in \mathcal{A}_2$.

$\succ$ is an order defined for $(a_1, a_2), (b_1, b_2) \in \mathcal{A}$ by

$$(a_1, a_2) \succ (b_1, b_2) \qquad \text{iff} \qquad a_1 \succ_1 b_1 \quad \text{or} \quad a_1 = b_1 \ \wedge \ a_2 \succ_2 b_2.$$

Thus, in comparing the two pairs (a_1, a_2) and (b_1, b_2), we first compare a_1 against b_1. If $a_1 \succ_1 b_1$, then this determines the relation between the pairs to be $(a_1, a_2) \succ (b_1, b_2)$. If $a_1 = b_1$, we compare a_2 with b_2, and the result of this comparison determines the relation between the pairs.

The order $\succ$ is called *lexicographic*, which implies that, as when searching in a dictionary, we locate the position of a word by checking the first letter first and only after locating the place where the first letter matches, do we continue matching the subsequent letters.

The importance of the lexicographic product follows from the following claim:

Claim. (lexicographic product)
If the domains $(\mathcal{A}_1, \succ_1)$ and $(\mathcal{A}_2, \succ_2)$ are well-founded, then so is their lexicographic product $(\mathcal{A}, \succ)$.

Clearly, by the above, the domain $(\mathbb{N}^2, \succ)$, where $\succ$ is the lexicographic order between pairs of natural numbers, is well-founded. This order is defined by

$$(n_1, n_2) \ \succ \ (m_1, m_2) \qquad \text{iff} \qquad n_1 > m_1 \quad \text{or} \quad n_1 = m_1 \ \wedge \ n_2 > m_2.$$

According to this definition

$$(10, 20) \ \succ \ (5, 15) \qquad (1, 0) \ \succ \ (0, 100) \qquad (1, 5) \ \succ \ (1, 3).$$

New well-founded domains can be constructed by taking lexicographic products of more than two well-founded domains. Applying this construction to the domain $(\mathbb{N}, >)$ of natural numbers, we obtain the domain $(\mathbb{N}^k, \succ)$, for $k \geq 2$, where $\succ$ is the lexicographic order between k-tuples of natural numbers. The order $\succ$ is defined by

$$(n_1, \ldots, n_k) \ \succ \ (m_1, \ldots, m_k) \qquad \text{iff} \qquad n_1 = m_1, \ \ldots, \ n_{i-1} = m_{i-1}, \ n_i > m_i$$
$$\text{for some } i, \ 1 \leq i \leq k.$$

For example, for $k = 3$

$$(7, \ 2, \ 1) \ \succ \ (7, \ 0, \ 45).$$

It is easy to show that the domain $(\mathbb{N}^k, \succ)$ is well-founded.

The Rule

Let $(\mathcal{A}, \succ)$ be a well-founded domain. As in rule CHAIN-J, we use several intermediate assertions $\varphi_1, \ldots, \varphi_m$ to describe the evolution from p to $q = \varphi_0$. Rule CHAIN-J uses the index of the assertion as a measure of the distance from the goal q. The rule presented here associates an explicit *ranking function* δ_i with each assertion φ_i, $i = 0, \ldots, k$. The function δ_i maps states into the set $\mathcal{A}$ and is intended to measure the distance of the current state to a state satisfying the goal q.

We refer to the value of δ_i in a φ_i-state as a *rank* of the state. The well-founded rule WELL for response properties is given in Fig. 23. Premise W1 states that every p-position satisfies one of $\varphi_0, \ldots, \varphi_m$. Premise W2 states that every φ_i-position with positive i and rank u is eventually followed by a position which satisfies some φ_j, with a rank lower than u.

For assertions p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m$: $\Sigma \mapsto \mathcal{A}$

W1. $p \rightarrow \bigvee_{i=0}^{m} \varphi_i$

W2. $\varphi_i \wedge \delta_i = u \Rightarrow \left(\bigvee_{j=0}^{m} (\varphi_j \wedge u \succ \delta_j) \right)$ for $i = 1, \ldots, m$

$p \Rightarrow q$

Fig. 23. Rule WELL (well-founded response)

Justification. It is straightforward to justify rule WELL. Consider a computation σ that satisfies premises W1, W2, and let t_1 be a position in σ which satisfies p. By W1, some φ_i is satisfied at t_1. If it is $\varphi_0 = q$, we are done. Otherwise, let φ_{i_1}, $i_1 > 0$, be the assertion holding at t_1 and let u_1 denote the rank of the state at position t_1. By W2, there exists a position t_2, $t_2 \geq t_1$, such that some φ_j holds at t_2 with a rank $u_2 \in \mathcal{A}$, such that $u_1 \succ u_2$. If $j = 0$, we are done. Otherwise, we proceed to locate a position $t_3 \geq t_2$.

In this way we construct a sequence of positions

$$t_1 \leq t_2 \leq t_3 \leq \ldots,$$

and a corresponding sequence of elements from $\mathcal{A}$ (ranks)

$$u_1 \succ u_2 \succ u_3 \succ \ldots,$$

such that either the sequence is of length k and $q = \varphi_0$ holds at the position t_k, or the sequence is infinite and some φ_j, $j > 0$, holds at each t_i with rank $\delta_j = u_i$ there. The later case is impossible since that would lead to an infinitely descending sequence of elements of $\mathcal{A}$, in contrast to the well-foundedness of $\succ$ over $\mathcal{A}$. It follows that for some $t_k \geq t_1$, $q = \varphi_0$ holds at t_k, which shows that q holds at t_1. ∎

Example (factorial)

Consider program FACT of Fig. 24. This program computes in z the factorial of a nonnegative integer x. We wish to prove for this program the response property

$$\underbrace{at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \quad \Rightarrow \quad \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{q=\varphi_0} \ .$$

in x: **integer where** $x \geq 0$
local y: **integer where** $y = x$
out z: **integer where** $z = 1$

ℓ_0: **while** $y > 0$ **do**
  ℓ_1: $(y, \ z) := (y - 1, \ z \cdot y)$
ℓ_2:

Fig. 24. Program FACT (factorial)

Intending to use rule WELL with $m = 1$, it only remains to choose the assertion φ_1, and the ranking functions δ_0 and δ_1. This necessitates the identification of a well-founded domain $(\mathcal{A}, \succ)$, where $\mathcal{A}$ serves as the range of δ_i. Obviously, φ_1 should describe the intermediate stage in the process of getting from p to q, and δ_1 should measure the distance of this intermediate stage from the goal $q = \varphi_0$. Premise W2 ensures that steps in the computation always bring us closer to the goal.

For program FACT, a good measure of the distance from termination is the value of y. This is because when we are at ℓ_0, there are y more iterations of the *while* loop before the program terminates. We therefore choose $(\mathbb{N}, >)$ as our well-founded domain and $|y|$ as the ranking function. Thus,

$$(\mathcal{A}, \ \succ) \ = \ (\mathbb{N}, \ >), \qquad \delta_0 \colon 0, \qquad \text{and} \qquad \delta_1 \colon \ |y| + 1.$$

The choice of $\delta_0 = 0$ is natural because, being at a φ_0-state, we are already at the goal, and the distance to the goal can therefore be taken as 0.

The intermediate assertion φ_1 should represent the progress the computation has made, so that when $y = 0$, we can infer that $z = x!$. Clearly, the way the

program operates is that it accumulates in z the product of the terms $x \cdot (x-1) \cdots$. In an intermediate stage, z contains the product $x \cdot (x-1) \cdots (y+1)$, which can also be expressed as $x!/y!$, provided $0 \leq y \leq x$.

We thus arrive at the intermediate assertion

$$\varphi: \quad at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y! \ .$$

It only remains to show that premises W1,W2 are satisfied by these choices.

- Premise W1

  This premise requires

  $$\underbrace{at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \ \rightarrow$$

  $$\underbrace{\cdots}_{\varphi_0} \ \vee \ \underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ .$$

  This implication is obviously valid.

- Premise W2

  This premise requires showing

  $$\underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ |y| + 1 = n \ \Rightarrow$$

  $$\left(\begin{array}{l} \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{\varphi_0} \ \wedge \ n > \underbrace{0}_{\delta_0} \\ \quad \vee \\ \underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ n > \underbrace{|y| + 1}_{\delta_1} \end{array} \right)$$

  Since $\varphi_1 \wedge |y|+1 = n$ implies that $n > 0$, it is sufficient to prove this implication for every $n > 0$. As φ_1 implies $y \geq 0$, we may replace $|y|$ by y.

  Case $n = 1$:

  For this value of n, we prove

  $$\underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ y + 1 = 1 \ \Rightarrow$$

  $$\left(\underbrace{at\_\ell_2 \ \wedge \ z = x!}_{\varphi_0} \ \wedge \ 1 > 0 \right) ,$$

  which simplifies to

  $$at\_\ell_0 \ \wedge \ z = x! \ \wedge \ y = 0 \ \Rightarrow \qquad (at\_\ell_2 \ \wedge \ z = x!) \ .$$

  This, of course, can be proven by a single application of rule RESP-J, observing that, under the situation described by the antecedent, only transition ℓ_0 is enabled, and taking it leads to $at\_\ell_2 \wedge z = x!$.

Case $n > 1$:

In this case, we will prove

$$\underbrace{at\_\ell_0 \ \wedge \ 0 \le y \le x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ y + 1 = n > 1 \ \Rightarrow$$

$$\left(\underbrace{at\_\ell_0 \ \wedge \ 0 \le y \le x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ n > y + 1 \right).$$

This can be proven by rule CHAIN-J using three assertions, ψ_0, ψ_1, and ψ_2. The top assertion ψ_2 corresponds to $\varphi_1 \wedge y+1 = n$. Assertion ψ_1 describes the intermediate state, after passing the test of the *while* statement, and being at ℓ_1. The final assertion ψ_0 implies $\varphi_1 \wedge y + 1 = n - 1 < n$, and describes the situation after performing the assignment ℓ_1 and arriving back at ℓ_0. The verification diagram in Fig. 25 describes this proof.

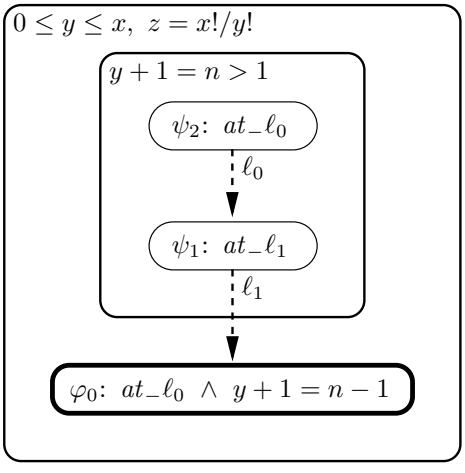


Fig. 25. Verification diagram for case $n > 1$

Thus, by treating separately the cases $n = 1$ and $n > 1$, we conclude that premise W2 holds for every $n \ge 1$. This establishes that the conclusion

$$\underbrace{at\_\ell_0 \ \wedge \ x \ge 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \ \Rightarrow \ \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{q}$$

of rule WELL is valid. ∎

A Rule with Nontemporal Premises

Rule WELL, used for proving response formulas, has as its premise W2, another response formula. This allows a recursive use of the rule, by which the temporal

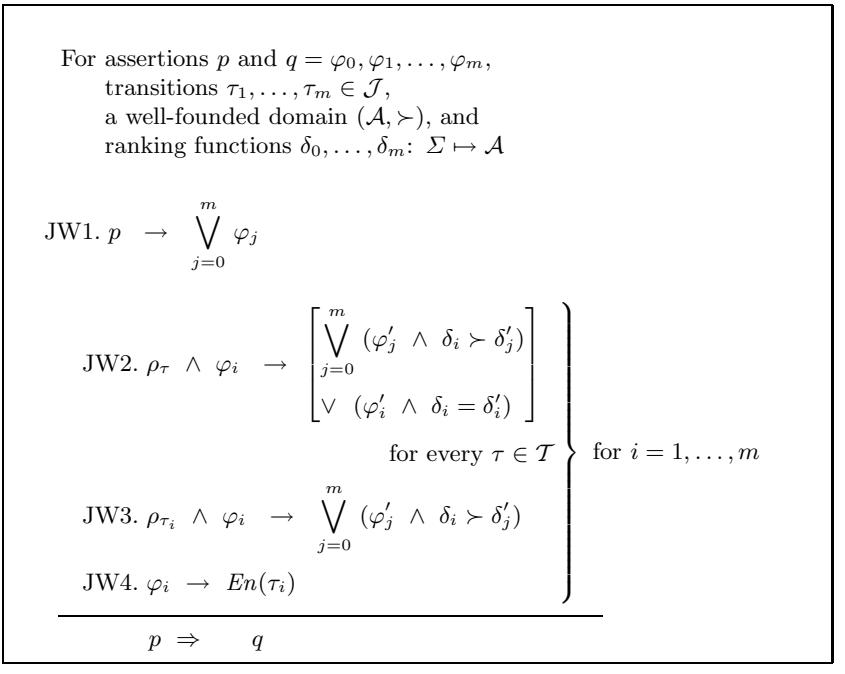


Fig. 26. Rule WELL-J (well-founded response under justice)

premise W2 is proved either by the simpler rule RESP-J, or by rule WELL again, only applied to simpler assertions. As a matter of fact, if we closely examine the proof of the previous example, we can identify there the use of those two options. For proving W2 for $n = 1$, we used rule RESP-J, since the response property for this case is accomplished in one step. On the other hand, the case of $n > 1$ accomplishes W2 in two steps, and we therefore had to use rule CHAIN-J.

However, in many cases, we do not need the recursive application of the rule, which means that premise W2 is proved directly by rule RESP-J. In these cases it is advantageous to replace the temporal premise W2 by the nontemporal premises of rule RESP-J, which are necessary for its derivation. This leads to a (combined) form of the rule in which all premises are nontemporal. Such a form is often more satisfactory because it explicitly manifests the power of the rule to derive temporal statements from nontemporal ones.

This leads to rule WELL-J (Fig. 26).

The rule requires finding auxiliary assertions φ_i and transitions $\tau_i, i = 1, \ldots, m$, a well-founded domain $(\mathcal{A}, \succ)$, and ranking functions δ_i: $\Sigma \mapsto \mathcal{A}$. Each assertion $\varphi_i, i > 1$, is associated with the transition τ_i that is helpful at positions satisfying φ_i, and with its own ranking function δ_i.

Premise JW1 requires that every p-position satisfies one of $\varphi_0, \ldots, \varphi_m$.

Premises JW2–JW4 impose three requirements for each $i = 1, \ldots, m$.

Premise JW2 requires that, taking any transition from a φ_i-position k, always leads to a successor position $k' = k + 1$, such that

- either some φ_j, $j = 0, \ldots, m$, holds at k' with a rank δ'_j lower than δ_i at k, or
- φ_i holds at k' with a rank δ'_i equal to the rank δ_i at k.

The main implication of premise JW2 is that if the situation has not improved in any noticeable way in going from k to k', i.e., the new rank still equals the old rank, at least we have not lost the identity of the helpful transition and the transition that was helpful in k is also helpful at k'.

Premise JW3 requires that transition τ_i, which is helpful for φ_i, always leads from a φ_i-position k to a next position which satisfies some φ_j and has a rank lower than that of k, i.e., $\delta_i \succ \delta'_j$.

Premise JW4 requires that the helpful transition τ_i is enabled at every φ_i-position.

Justification. To justify the rule, assume a computation such that p holds at position k, and no later position $i \geq k$, satisfies $q = \varphi_0$. By this assumption and JW1, some φ_j, $j > 0$, must hold at position k. Let φ_{i_1} be the formula holding at k, and denote the rank δ_{i_1} at k by u_1. By JW4, transition τ_{i_1} is enabled at position k.

Consider the transition τ taken at position k, leading into position $k + 1$. By JW2 and JW3, either position $k + 1$ has a lower rank u_2, $u_2 \prec u_1$, or it has the same rank, but then τ_{i_1} is still the helpful transition at $k+1$ and is enabled there. In the case that the rank is still u_1, we can continue the argument from $k + 1$ to $k + 2$, $k + 3$, etc. However, we cannot have all positions $i > k$ with the same rank. To see this, assume that all positions beyond k do have the same rank. By JW2 and JW4, this implies that τ_{i_1} is continuously helpful and enabled. By JW3, τ_{i_1} is not taken beyond k because taking it would have led to a state with a rank lower than u_1. Thus, our assumption that all positions beyond k have the same rank leads to the situation that τ_{i_1} is continuously enabled and not taken, violating the justice requirement for τ_{i_1}.

Thus, eventually, we must reach a position k_2, $k_2 > k$, with lower rank u_2, where $u_2 \prec u_1$. In a similar way we can establish the existence of a position $k_3 > k_2$, with rank u_3 where $u_3 \prec u_2$. Continuing in this manner, we construct an infinitely descending sequence $u_1 \succ u_2 \succ u_3 \succ \cdots$ of elements of $\mathcal{A}$. This is impossible, due to the well-foundedness of $\succ$ on $\mathcal{A}$.

We conclude that every p-position must be followed by a q-position, establishing the consequence of the rule. ∎

Note that since premise JW3 implies premise JW2 for $\tau = \tau_i$, it is sufficient to check premise JW2 only for $\tau \neq \tau_i$.

Rule CHAIN-J can be viewed as a special case of rule WELL-J which uses $\delta_i = i$, for $i = 0, \ldots, m$, as ranking functions. It is not difficult to see that the premises J2, J3, and J4 of rule CHAIN-J correspond precisely to premises JW2, JW3, and JW4 of rule WELL-J. The well-founded domain used in this special case is the finite segment $[0..m]$ of the natural numbers ordered by $>$.

Example (factorial)

We use rule WELL-J to prove that program FACT of Fig. 24 satisfies the response property of total correctness

$$\underbrace{at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \quad \Rightarrow \quad \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{q=\varphi_0} \ .$$

Obviously, except for the terminating state, execution of program FACT alternates between states satisfying at_ℓ_0 in which ℓ_0 is the helpful transition, and states satisfying at_ℓ_1 in which ℓ_1 is helpful.

Consequently, we take $m = 2$ and use the following intermediate assertions.

$$\varphi_2: \quad at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!$$

$$\varphi_1: \quad at\_\ell_1 \ \wedge \ 1 \leq y \leq x \ \wedge \ z = x!/y!$$

$$\varphi_0: \quad at\_\ell_2 \ \wedge \ z = x! \ .$$

Note that when control is at ℓ_1, y is required to be greater than or equal to 1.

It remains to determine the ranking functions δ_i, $i = 0, 1, 2$. Our previous analysis of the considered response property for program FACT (using rule WELL) identified $|y|$ as a good measure of progress over $(\mathbb{N}, >)$. Variable y keeps decreasing as the program gets closer to termination. Unfortunately, premise JW3 of rule WELL-J requires that δ decreases on *each* activation of a helpful transition. As we see, not every helpful transition causes $|y|$ to decrease. In particular, ℓ_0 does not change $|y|$. Consequently, we have to supplement $|y|$ by an additional component that will decrease when $|y|$ stays the same. This leads to the following choice:

$$\delta_2: \quad \big(|y|, \ 2\big)$$

$$\delta_1: \quad \big(|y|, \ 1\big)$$

$$\delta_0: \quad \big(0, \ 0\big) \ .$$

The corresponding well-founded domain is $\big(\mathbb{N} \times \{1, 2\}, \ \succ\big)$, where $\succ$ is the lexicographical order between pairs of integers.

In the previous proof of this property, we used the measure $|y| + 1$ to ensure that the rank decreases also on the transition from ℓ_0 to ℓ_2. Since the use of pairs guarantees such a decrease by a decreasing second component, we can omit the $+1$ increment and take the first component to be simply $|y|$.

We may view the ranking function δ_i: $(|y|, i)$ as consisting of a *major* and a *minor* measure of progress. Function $|y|$ measures large steps of progress, such as one full iteration of the loop at ℓ_0. The minor component i measures smaller steps of progress. Observe that transition ℓ_1 actually causes the minor measure i to increase from 1 to 2, but at the same time it decreases the major measure $|y|$.

Let us consider the premises of rule WELL-J. Since both φ_i's imply $y \geq 0$, we may replace $|y|$ by y.

- Premise JW1

We prove JW1 by showing

$$\underbrace{at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \ \rightarrow$$

$$\cdots \ \vee \ \underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_2},$$

which is obviously valid.

- Premises JW2, JW3 for $i = 2$

For $i = 2$ we will show

$$\rho_\tau \ \wedge \ \underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_2} \ \rightarrow$$

$$\left(\begin{array}{l} \underbrace{at'\_\ell_2 \ \wedge \ z = x!}_{\varphi'_0} \ \wedge \ \underbrace{(y,2)}_{\delta_2} \succ \underbrace{(y',0)}_{\delta'_0} \\ \qquad \vee \\ \underbrace{at'\_\ell_1 \ \wedge \ 1 \leq y' \leq x \ \wedge \ z' = x!/y'!}_{\varphi'_1} \ \wedge \ \underbrace{(y,2)}_{\delta_2} \succ \underbrace{(y',1)}_{\delta'_1} \end{array} \right),$$

for each transition $\tau \in \{\ell_0, \ell_1\}$, not necessarily the helpful one. Obviously, this will satisfy both JW2 and JW3. Since at_ℓ_0 implies that transition ℓ_1 is disabled, the left-hand side of the implication for $\tau = \ell_1$ is false and the implication is trivially true.

For $\tau = \ell_0$, we prove the implication by separately considering the cases $y = 0$ and $y \neq 0$.

Case $y = 0$:
In this case ρ_{ℓ_0} implies at'_ℓ_2, $y' = y = 0$, and $z' = z$. Since $z = x!/y!$ and $y = 0$ imply $z = x!$, it follows that the left-hand side implies $\varphi'_0 \wedge (0,2) \succ (0,0)$.

Case $y \neq 0$:
In this case φ_2 implies that $y > 0$, which together with ρ_{ℓ_0}, implies at'_ℓ_1, $y' = y$, and $z' = z$. Assertion φ_2 implies $y \leq x \wedge z = x!/y!$ which together with $y > 0$ establishes φ'_1. The rank decrease $(y,2) \succ (y,1)$ is obvious.

- Premises JW2, JW3 for $i = 1$

For $i = 1$ we will show

$$\rho_\tau \ \wedge \ \underbrace{at\_\ell_1 \ \wedge \ 1 \le y \le x \ \wedge \ z = x!/y!}_{\varphi_1} \ \rightarrow$$

$$\cdots \ \vee \ \Big(\underbrace{at'\_\ell_0 \ \wedge \ 0 \le y' \le x \ \wedge \ z' = x!/y'!}_{\varphi'_2} \ \wedge \ \underbrace{(y,1)}_{\delta_1} \succ \underbrace{(y',2)}_{\delta'_2} \Big) ,$$

for each transition $\tau \in \{\ell_0, \ell_1\}$, not necessarily the helpful one. Obviously, this will satisfy both JW2 and JW3. Since at_ℓ_1 implies that transition ℓ_0 is disabled, the left-hand side of the implication for $\tau = \ell_0$ is false and the implication is trivially true.

For $\tau = \ell_1$, we observe that ρ_{ℓ_1} implies at'_ℓ_0, $y' = y - 1$, and $z' = z \cdot y$. Substituting these expressions in the right-hand side of the implication reduces the conjunction to

$$0 \le y - 1 \le x \ \ \wedge \ \ z \cdot y = x!/(y-1)! \ \ \wedge \ \ (y,1) \succ (y-1,2),$$

all of which are either obviously valid or are implied by φ_1.

- Premises JW4

The helpful transitions for φ_1 and φ_2 are ℓ_1 and ℓ_0, with enabling conditions at_ℓ_1 and at_ℓ_0, respectively. Obviously, they satisfy

$$\underbrace{at\_\ell_1 \ \wedge \ \cdots}_{\varphi_1} \ \rightarrow \ \underbrace{at\_\ell_1}_{En(\tau_1)}$$

$$\underbrace{at\_\ell_0 \ \wedge \ \cdots}_{\varphi_2} \ \rightarrow \ \underbrace{at\_\ell_0}_{En(\tau_2)}$$

as required by premise JW4.

This establishes the four premises of rule WELL-J, proving the response property of total correctness for program FACT

$$at\_\ell_0 \ \wedge \ x \ge 0 \ \wedge \ y = x \ \wedge \ z = 1 \ \ \Rightarrow \ \ \ (at\_\ell_2 \ \wedge \ z = x!). \qquad \blacksquare$$

A Condensed Representation of Ranking Functions

Many of our proofs consider ranking functions that consist of lexicographic pairs of natural numbers, i.e.,

$$\delta \ = \ (d_1, d_2).$$

Such a function decreases over a transition even if d_2 increases, provided d_1 decreases at the same time. Lexicographic order implies that even a small decrease in d_1 outweighs an arbitrarily large increase in d_2. In some cases there exists a bound M, $M > 0$, which is larger than any possible increase in d_2. In these cases we may use the ranking function

$$\widehat{\delta} \ = \ M \cdot d_1 + d_2,$$

which ranges over $\mathbb{N}$, instead of the original $\delta = (d_1, d_2)$ which ranges over $\mathbb{N} \times \mathbb{N}$. We refer to $\widehat{\delta}$ as a *condensed representation* ranking function.

In **Problem 4** the reader is requested to prove that in such cases,

$$\widehat{\delta} \;=\; M \cdot d_1 + d_2 \;>\; \widehat{\delta}\,' \;=\; M \cdot d_1' + d_2' \qquad \text{iff} \qquad \delta\colon\ (d_1, d_2) \;\succ\; \delta'\colon\ (d_1', d_2').$$

For example, in the above proof of program FACT, we used the ranking functions

$$\delta_i\colon \quad (|y|,\ i).$$

Since the maximal increase in the value of i is 1 which is smaller than 2, we could have used instead the condensed ranking function

$$\widehat{\delta_i}\colon \quad 2 \cdot |y| \;+\; i.$$

Example (up down)

Consider program UP-DOWN presented in Fig. 27. This program can be viewed as an extension of program ANY-Y of Fig. 2. Process P_1 increments y, counting up in ℓ_0, ℓ_1, as long as $x = 0$. Once P_1 finds that x is different from 0, it proceeds to ℓ_2, ℓ_3, where y is decremented until it becomes 0. Process P_2's single action is to set x to 1. Obviously, due to justice, x will eventually be set to 1. However, one cannot predict the number of helpful steps required for P_1 to terminate. The longer P_2 waits before performing m_0, the higher the value y will attain on the move to ℓ_2. It is this value of y which determines the number of remaining steps to termination.

In fact, for every $n > 0$, we can construct a computation requiring more than $4n$ helpful steps to achieve $y = 0$. This computation allows P_1 to increase y up to n, and only then activates m_0. At least $2n$ more steps of P_1 are needed to decrement y back to 0.

Consequently, we need rule WELL-J to prove the response property

$$\underbrace{at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = y = 0}_{p = \Theta} \quad \Rightarrow \quad \underbrace{at\_\ell_4 \;\wedge\; at\_m_1}_{q}$$

for program UP-DOWN.

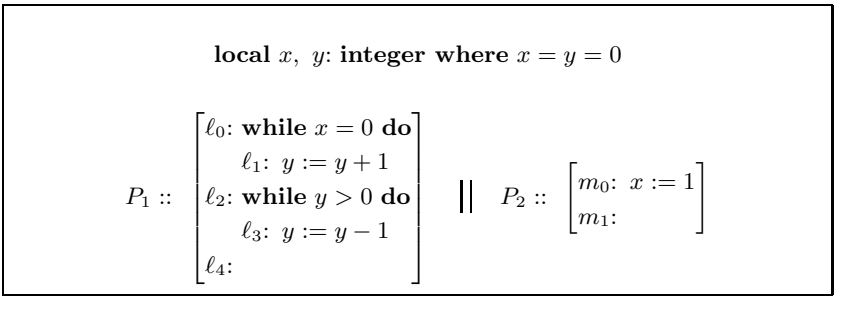


Fig. 27. Program UP-DOWN

In order to construct the intermediate assertions and the ranking functions δ_i, we observe that there are three distinct phases in the achievement of $at\_\ell_4 \wedge at\_m_1$. The first phase waits for P_2 to perform m_0. This phase terminates when m_0 is executed. In the second phase, P_1 senses that x has been set to 1 and moves to ℓ_2. In the third phase, P_1 is within $\ell_{2,3}$ and decrements y until y reaches 0 and P_1 moves to ℓ_4.

Consequently, it seems advisable to use the well-founded domain $(\mathbb{N}^3, \succ)$ of lexicographic triples (n_1, n_2, n_3), whose first element n_1 identifies the phase, and whose remaining elements, n_2 and n_3, identify progress within the phase. Recall that lexicographic ordering on triples of natural numbers is defined by

$$(n_1,\ n_2,\ n_3) \succ (m_1,\ m_2,\ m_3) \qquad \text{iff} \qquad \begin{cases} n_1 > m_1 & \text{or} \\ n_1 = m_1,\ n_2 > m_2 & \text{or} \\ n_1 = m_1,\ n_2 = m_2,\ n_3 > m_3. \end{cases}$$

Obviously, this ordering is a well-founded relation on $\mathbb{N}^3$.

Consider the remaining elements needed to measure progress within a phase. The first phase terminates after one helpful step, m_0. The second phase terminates in two helpful steps, ℓ_1 followed by ℓ_0. The last phase has y measuring coarse progress, and it takes two steps to decrement y, ℓ_2 followed by ℓ_3.

Consequently, we define the following assertions, helpful transitions and ranking functions,

$$\begin{array}{llllll}
\varphi_5\colon & at_\ell_{0,1} \wedge at_m_0 \wedge x = 0 \wedge y \geq 0 & \tau_5\colon & m_0 & \delta_5\colon & (2,\ 0,\ 0) \\
\varphi_4\colon & at_\ell_1 \ \wedge at_m_1 \wedge x = 1 \wedge y \geq 0 & \tau_4\colon & \ell_1 & \delta_4\colon & (1,\ 0,\ 1) \\
\varphi_3\colon & at_\ell_0 \ \wedge at_m_1 \wedge x = 1 \wedge y \geq 0 & \tau_3\colon & \ell_0 & \delta_3\colon & (1,\ 0,\ 0) \\
\varphi_2\colon & at_\ell_2 \ \wedge at_m_1 \wedge x = 1 \wedge y \geq 0 & \tau_2\colon & \ell_2 & \delta_2\colon & (0,\ |y|,\ 2) \\
\varphi_1\colon & at_\ell_3 \ \wedge at_m_1 \wedge x = 1 \wedge y > 0 & \tau_1\colon & \ell_3 & \delta_1\colon & (0,\ |y|,\ 1) \\
\varphi_0\colon & at_\ell_4 \ \wedge at_m_1 & & & \delta_0\colon & (0,\ 0,\ 0).
\end{array}$$

Note that progress within the second phase is measured by the third component, which moves from 1 to 0 on execution of ℓ_1. Progress within the third phase is measured by the pair $(y, 1 + at\_\ell_2)$ which decreases on execution of both ℓ_3 and ℓ_2.

Let us show that all premises of rule WELL-J are satisfied by these choices.

- Premise JW1

For this premise we have to show the implication

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = y = 0}_{p} \ \rightarrow$$

$$\cdots \ \vee \ \underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0 \ \wedge \ y \geq 0}_{\varphi_5},$$

which is obvious.

- Premise JW2 for φ_5

It is sufficient to show the following for each $\tau \neq m_0$

$$\rho_\tau \ \wedge \ \underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0 \ \wedge \ y \geq 0}_{\varphi_5} \ \rightarrow$$

$$\cdots \ \vee \ \Big(\underbrace{at'\_\ell_{0,1} \ \wedge \ at'\_m_0 \ \wedge \ x' = 0 \ \wedge \ y' \geq 0}_{\varphi'_5} \ \wedge \ \underbrace{(2,\ 0,\ 0)}_{\delta_5} \ = \ \underbrace{(2,\ 0,\ 0)}_{\delta'_5} \Big) .$$

The only transitions $\tau \neq m_0$ enabled on φ_5-states are ℓ_0 and ℓ_1. For each of them, ρ_τ implies $at'\_\ell_{0,1}$, $x' = x = 0$, and $y' \geq y \geq 0$. Consequently, the implication is valid.

- Premise JW3 for φ_5

We show

$$\rho_{m_0} \ \wedge \ \underbrace{at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0 \ \wedge \ y \geq 0}_{\varphi_5} \ \rightarrow$$

$$\left[\begin{array}{l} \underbrace{at'\_\ell_1 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_4} \ \wedge \ \underbrace{(2,\ 0,\ 0)}_{\delta_5} \succ \underbrace{(1,\ 0,\ 1)}_{\delta'_4} \\ \qquad \vee \\ \underbrace{at'\_\ell_0 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_3} \ \wedge \ \underbrace{(2,\ 0,\ 0)}_{\delta_5} \succ \underbrace{(1,\ 0,\ 0)}_{\delta'_3} \end{array} \right],$$

Clearly, ρ_{m_0} implies $x' = 1$, $y' = y$, at'_m_1 and $at'\_\ell_i = at\_\ell_i$ for $i = 0, 1$. By φ_5, either at_ℓ_0 or at_ℓ_1 holds. In the case that $at\_\ell_0 = \mathrm{T}$, the second disjunct is implied. In the case that $at\_\ell_1 = \mathrm{T}$, the first disjunct is implied. The decrease in rank is obvious in both cases.

- Premises JW2, JW3 for φ_4

Since ℓ_1 is the only transition enabled on φ_4-states, the following implication establishes both JW2 and JW3 for φ_4:

$$\rho_{\ell_1} \ \wedge \ \underbrace{at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y \geq 0}_{\varphi_4} \ \rightarrow$$

$$\cdots \ \vee \ \Big(\underbrace{at'\_\ell_0 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_3} \ \wedge \ \underbrace{(1,\ 0,\ 1)}_{\delta_4} \succ \underbrace{(1,\ 0,\ 0)}_{\delta'_3} \Big) .$$

Transition relation ρ_{ℓ_1} implies at'_ℓ_0, $at'\_m_1 = at\_m_1$, $x' = x$, and $y' \geq y$. By φ_4, it follows that φ'_3 holds, and the decrease in rank is obvious.

- Premises JW2, JW3 for φ_3

Since ℓ_0 is the only transition enabled on φ_3-states, the following implication establishes both JW2 and JW3 for φ_3

$$\rho_{\ell_0} \ \wedge \ \underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y \geq 0}_{\varphi_3} \ \rightarrow$$

$$\cdots \ \vee \ \Big(\underbrace{at'\_\ell_2 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_2} \ \wedge \ \underbrace{(1,\ 0,\ 0)}_{\delta_3} \succ \underbrace{(0,\ |y'|,\ 2)}_{\delta'_2} \Big)$$

Transition relation ρ_{ℓ_0} under $x = 1$ implies at'_ℓ_2 and $at'\_m_1 = at\_m_1$. It also implies $x' = x$ and $y' = y$. The rank decrease $(1, 0, 0) \succ (0, |y'|, 2)$ is obvious, since 1, the first component of the left-hand side, is larger than 0, the first component of the right-hand side.

- Premises JW2, JW3 for φ_2

Since ℓ_2 is the only transition enabled on φ_2-states, the following implication establishes both JW2 and JW3 for φ_2

$$\rho_{\ell_2} \ \wedge \ \underbrace{at\_\ell_2 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y \geq 0}_{\varphi_2} \ \rightarrow$$

$$\left(\begin{array}{l} \underbrace{at'\_\ell_4 \ \wedge \ at'\_m_1}_{\varphi'_0} \ \wedge \ \underbrace{(0,\ |y|,\ 2)}_{\delta_2} \succ \underbrace{(0,\ 0,\ 0)}_{\delta'_0} \\ \qquad \vee \\ \underbrace{at'\_\ell_3 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' > 0}_{\varphi'_1} \ \wedge \ \underbrace{(0,\ |y|,\ 2)}_{\delta_2} \succ \underbrace{(0,\ |y'|,\ 1)}_{\delta'_1} \end{array} \right).$$

We distinguish between two cases.

Case $y = 0$:
In this case, ρ_{ℓ_2} implies at'_ℓ_4, $y' = y = 0$, and $at'\_m_1 = at\_m_1$. Since φ_2 implies $at\_m_1 = \text{T}$, the left-hand side of this verification condition implies the right-hand side disjunct $\varphi'_0 \ \wedge \ (0, |y|, 2) \succ (0, 0, 0)$.

Case $y \neq 0$:
By φ_2, it follows that $y > 0$. In this case, ρ_{ℓ_2} implies at'_ℓ_3, $at'\_m_1 = at\_m_1$, $x' = x$, and $y' = y > 0$. Together with φ_2, these imply φ'_1. To show the rank decrease, we observe that $|y| = |y'|$ and $2 > 1$.

- Premises JW2, JW3 for φ_1

Since ℓ_3 is the only transition enabled on φ_1-states, the following implication establishes both JW2 and JW3 for φ_1

$$\rho_{\ell_3} \ \wedge \ \underbrace{at\_\ell_3 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y > 1}_{\varphi_1} \ \rightarrow$$

$$\cdots \ \vee \ \Big(\underbrace{at'\_\ell_2 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_2} \ \wedge \ \underbrace{(0,\ |y|,\ 1)}_{\delta_1} \succ \underbrace{(0,\ |y'|,\ 2)}_{\delta'_2} \Big).$$

Transition relation ρ_{ℓ_3} implies at'_ℓ_2, $at'\_m_1 = at\_m_1$, $x' = x$, and $y' = y - 1$. By the clause $y > 0$ in φ_1 we have $y' = y - 1 \geq 0$. The decrease in rank follows from $y > 0$ and $(0, |y|, 1) = (0, y, 1) \succ (0, y - 1, 2) = (0, |y'|, 2)$.

- Premise JW4

This premise requires showing the following implication for each $i = 1, \ldots, 5$.

$$\varphi_i \ \rightarrow \ En(\tau_i).$$

By inspecting φ_i for each $i = 1, \ldots, 5$, we see that this is indeed the case.

This concludes the proof. ∎

Persistence of the Helpful Transitions

Premise JW2 of rule WELL-J requires that, in the case that a transition does not attain a lower rank in the next state, it must maintain the rank and lead to a state that still satisfies φ_i, and therefore maintain τ_i as the helpful transition. We refer to this clause as a requirement for the persistence of helpful transitions. One may wonder how essential this requirement is, and whether it would be possible to relax this requirement. In **Problem 5**, we request the reader to consider a version of rule WELL-J in which premise JW2 has been relaxed to allow the helpful transition to change without rank decrease. The problem shows that the resulting rule is unsound.

6 Rank Diagrams

To represent by diagrams proofs of response properties that require the use of well-founded ranking, we have to add some more components to the labels of nodes.

A verification diagram is said to be a RANK *diagram* if its nodes are labeled by assertions $\varphi_0, \ldots, \varphi_m$, with φ_0 being the terminal node, and ranking functions $\delta_0, \ldots, \delta_m$, where each δ_i maps states into $\mathcal{A}$, and it satisfies the following requirement:

- Every node φ_i, $i > 0$, has a dashed edge departing from it. This identifies the transition labeling such an edge as *helpful* for assertion φ_i. All helpful transitions must be just.

Note that, unlike CHAIN diagrams, we allow node φ_i to be connected to φ_j for $j > i$.

Verification and Enabling Conditions for RANK Diagrams

Consider a nonterminal node labeled by assertion φ and ranking function δ, and let $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the τ-successors of φ and $\delta_1, \ldots, \delta_k$ be their respective ranking functions.

- If transition τ is unhelpful for φ, i.e., labels only single edges departing from the node, then we associate with φ and τ the following verification condition

$$\Big\{\varphi \wedge \delta = u\Big\} \ \tau \ \Big\{(\varphi \wedge u \succeq \delta) \ \vee \ (\varphi_1 \wedge u \succ \delta_1) \ \vee \ \cdots \ \vee \ (\varphi_k \wedge u \succ \delta_k)\Big\}.$$

- If τ is helpful for φ (labels dashed edges), we associate with φ and τ the following verification condition

$$\Big\{\varphi \wedge \delta = u\Big\} \ \tau \ \Big\{(\varphi_1 \wedge u \succ \delta_1) \ \vee \ \cdots \ \vee \ (\varphi_k \wedge u \succ \delta_k)\Big\}.$$

- For every nonterminal node φ and a transition τ labeling a dashed edge departing from φ, we require

$$\varphi \ \rightarrow \ En(\tau).$$

Note that in the case of an unhelpful transition, we allow a τ-successor with a rank equal to that of φ, provided it satisfies the same assertion φ.

Valid RANK Diagrams

A RANK diagram is said to be *valid over program* P (*P-valid* for short) if all the verification and enabling conditions associated with the diagram are P-state valid.

The consequences of having a valid RANK diagram are stated in the following claim.

Claim. (RANK diagrams)
A P-valid RANK diagram establishes that the response formula

$$\bigvee_{j=0}^{m} \varphi_j \ \Rightarrow \ \varphi_0$$

is P-valid.
If, in addition, we can establish the P-state validity of the following implications:

$$p \ \rightarrow \ \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \varphi_0 \ \rightarrow \ q$$

then, we can conclude the validity of

$$p \ \Rightarrow \ q.$$

Justification. It is not difficult to see that a valid RANK diagram establishes the premises of rule WELL-J with p: $\bigvee_{j=0}^{m} \varphi_j$, q: φ_0, and τ_i the transition helpful for φ_i being the transition labeling the dashed edge departing from φ_i in the diagram. This establishes the P-validity of

$$\bigvee_{j=0}^{m} \varphi_j \ \Rightarrow \ \varphi_0.$$

Given assertions p and q, satisfying the implications

$$p \rightarrow \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \varphi_0 \rightarrow q,$$

we can use rule MON-R of Fig. 3 to infer

$$p \Rightarrow q. \blacksquare$$

Example (factorial)

The diagram of Fig. 28 presents a valid RANK diagram that establishes total correctness for program FACT (Fig. 24).

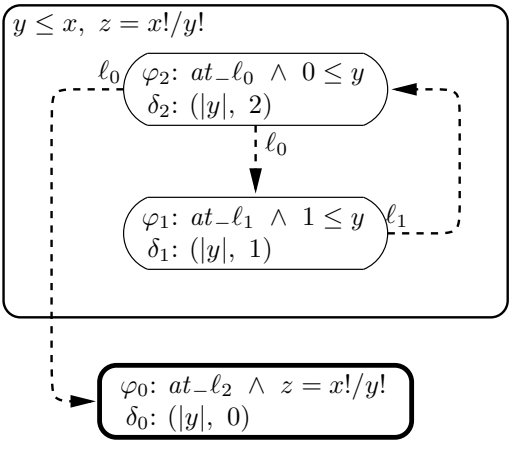


Fig. 28. RANK diagram for total correctness of program FACT

This diagram contains a connection from φ_1 to φ_2 which is disallowed in chain diagrams. However, the validity of the implied verification conditions ensures that, whenever a transition is taken from a φ_2-state s_2 to a φ_1-state s_1, the rank decreases, i.e., $\delta_2(s_2) \succ \delta_1(s_1)$. ∎

Distributing the Ranking Functions

To make RANK diagrams more readable, we introduce additional encapsulation conventions.

One of the useful conventions is that compound nodes may be labeled by a list of assertions. Such labeling indicates that the full assertion associated with a basic (non compound) node n_i is a conjunction of the assertion labeling the node itself and all the assertions labeling compound nodes that contain n_i.

Thus, while the label of node φ_2 in the diagram of Fig. 28 is $at\_\ell_0 \wedge 0 \le y$, the full assertion associated with this node is

$$at\_\ell_0 \ \wedge\ 0 \le y \ \wedge\ y \le x \ \wedge\ z = x!/y! \ .$$

We can view this representation as distribution of the full assertion into the part $at\_\ell_0 \wedge 0 \leq y$ labeling the node itself and the part $y \leq x \wedge z = x!/y!$ labeling the enclosing node, which is common to both φ_1 and φ_2.

In a similar way, we introduce a convention for distribution of ranking functions. The convention allows us to label a compound node by

$$\delta\colon\ f,$$

where f is some ranking function mapping states into a well-founded domain $\mathcal{A}$. In most of our examples, the domains are either $(\mathbb{N}, >)$ or lexicographic products of this domain.

Consider a basic node n_i labeled by assertion φ_i and local ranking function f_b. Assume that node n_i is contained in a nested sequence of compound nodes that are labeled by ranking labels $\delta\colon f_1, \ldots, \delta\colon f_m$, as we go from the outermost compound node towards n_i. This situation is depicted in Fig. 29. Then the full ranking function associated with the node φ_i is given by the tuple

$$\delta_i \ = \ (f_1,\ \ldots,\ f_m,\ f_b).$$

That is, we consider the outermost ranking f_1 to be the most significant component in δ_i, and the local ranking f_b to be the least significant component.

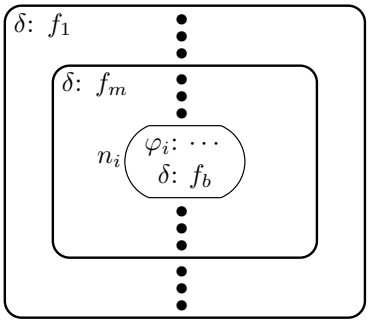


Fig. 29. Encapsulated sequence of nodes

Example (factorial)

In Fig. 30, we present a version of the RANK diagram of Fig. 28, in which a common component of the ranking function appears as a ranking label of the enclosing compound state.

The full ranking functions associated with the nodes in the RANK diagram of Fig. 30 are identical to those appearing in Fig. 28. ▟

Another rank distribution convention allows one to omit the local rank labeling a node φ_i altogether. This is interpreted as if the node were labeled with the ranking function $\delta\colon\ i$, where i is the index of the node (and the assertion labeling it). In Fig. 31, we present another version of the RANK diagram for program FACT, using this convention.

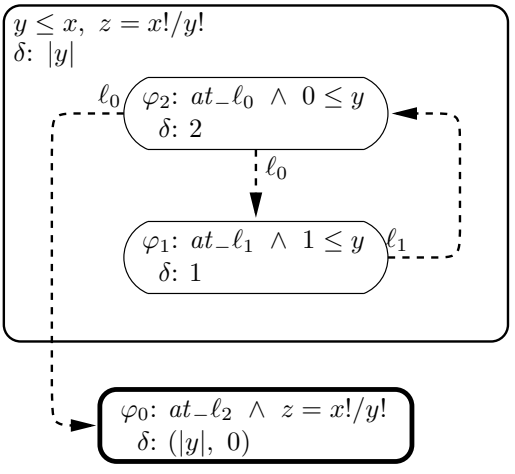


Fig. 30. RANK diagram with distributed ranking functions

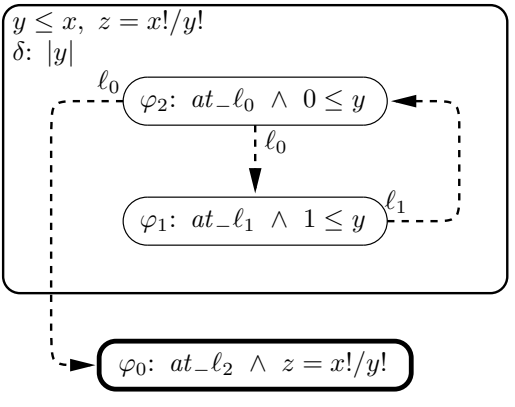


Fig. 31. RANK diagram with default local ranking

The full ranking functions associated with the nodes in this diagram are:

$$\delta_2\colon\ (|y|,\ 2), \qquad \delta_1\colon\ (|y|,\ 1), \qquad \text{and} \qquad \delta_0\colon\ 0.$$

This raises the question of how to compare lexicographic tuples of unequal lengths such as $\delta_2\colon\ (|y|,\ 2)$ and $\delta_0\colon\ 0$.

Since all our examples will be based on tuples of non-negative integers, we agree that the relation holding between $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_k)$ for $i < k$ is determined by lexicographically comparing $(a_1, \ldots, a_i, 0, \ldots, 0)$ to $(b_1, \ldots, b_i, b_{i+1}, \ldots, b_k)$. That is, we pad the shorter tuple by zeros on the right until it assumes the length of the longer tuple.

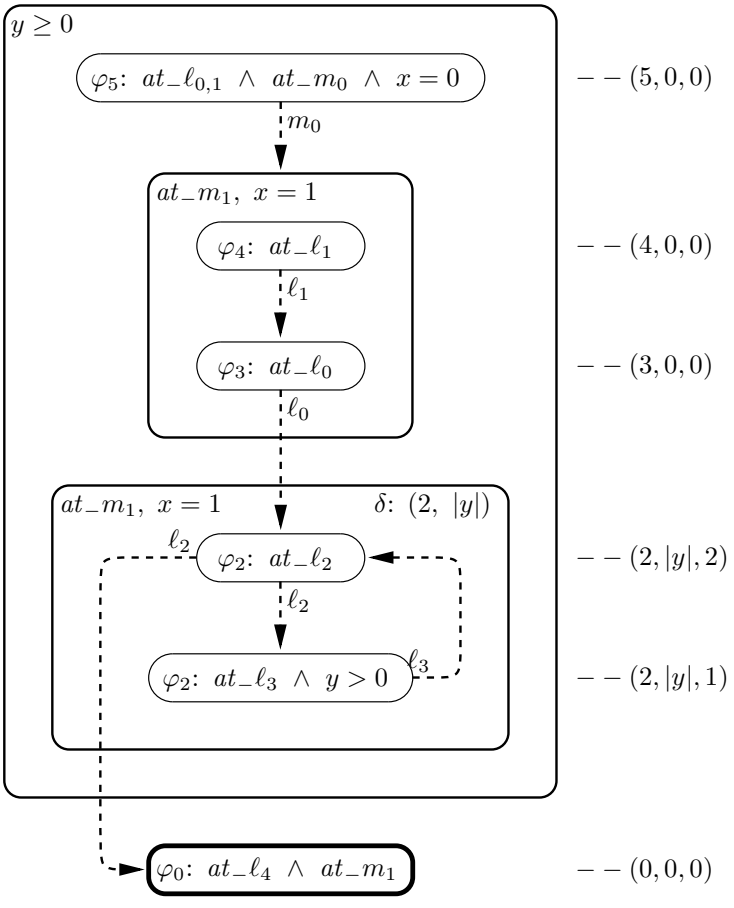


Fig. 32. RANK diagram for termination of UP-DOWN

According to this definition, $(|y|, 2) \succ 0$, since $(|y|, 2) \succ (0, 0)$.

Example (up-down)

In Fig. 32 we present a valid RANK diagram which implies, by monotonicity, the property of termination for program UP-DOWN (Fig. 27).

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = y = 0}_{p = \Theta} \quad \Rightarrow \quad \underbrace{at\_\ell_4 \ \wedge \ at\_m_1}_{\varphi_0} \ .$$

The diagram provides a detailed description of the progress of the computation from φ_5 to φ_0. It shows that progress from φ_5 to φ_2 is due to a CHAIN-like reasoning. Then, the progress from φ_2 and φ_1 to φ_0 requires a well-founded argument with the measure $|y|$ for coarse progress, and the index $j = 1, 2$ of φ_j for measuring fine progress.

The ranking functions appearing in this diagram are somewhat different from the ones used originally. When padded to the maximum length of 3, they are given by

δ_5: $(5,\ 0,\ 0)$

δ_4: $(4,\ 0,\ 0)$

δ_3: $(3,\ 0,\ 0)$

δ_2: $(2,\ |y|,\ 2)$

δ_1: $(2,\ |y|,\ 1)$

δ_0: $(0,\ 0,\ 0)$.

Note that the diagram contains a connection from φ_1 to φ_2. This is allowed because ℓ_3 decrements y and leads to a decrease in rank, stated by

$$(2,\ |y|,\ 1) \ \succ\ (2,\ |y-1|,\ 2).$$ ▟

In **Problems 6–9**, the reader is requested to prove total correctness of several programs.

7 Response with Past Subformulas

In this section we generalize the methods and proof rules presented in the preceding sections to handle response formulas $p \Rightarrow \quad q$, where p and q are past formulas.

The generalization is straightforward. It involves the following systematic modifications and replacements.

- Wherever a rule calls for one or more intermediate assertions, the past version of the rule requires finding past formulas.
- Each premise of the form $\varphi \rightarrow \psi$, for assertions φ and ψ, is replaced by an entailment $\widehat{\varphi} \rightarrow \widehat{\psi}$ for corresponding past formulas $\widehat{\varphi}$ and $\widehat{\psi}$.
- A verification condition $\{p\}\,\tau\,\{q\}$, for past formulas p and q and transition τ, is interpreted as the entailment $\rho_\tau \wedge p \Rightarrow q'$, where the primed version of a past formula is calculated as in Section 4.1 of the SAFETY book.

For example, in Fig. 33, we present the past version of rule WELL-J.

Similar past versions can be derived for rules RESP-J and CHAIN-J.

Example. Let us illustrate the use of the past version of rule WELL-J for proving the response property

$$0 \le n \le y \quad \Rightarrow \quad \big(y = n \ \wedge \quad (at\_\ell_0 \ \wedge \ y = n)\big)$$

holds for program UP-DOWN of Fig. 27.

This property states that any position i at which y is greater or equal to some $n \ge 0$, is followed by a position j at which $y = n$ and such that, at a preceding

For assertions p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J}$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m$: $\Sigma \mapsto \mathcal{A}$

JW1. $p \;\Rightarrow\; \bigvee_{j=0}^{m} \varphi_j$

JW2. $\rho_\tau \;\wedge\; \varphi_i \;\Rightarrow\; \left[\begin{array}{l} \bigvee_{j=0}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j) \\ \vee\; (\varphi'_i \;\wedge\; \delta_i = \delta'_i) \end{array}\right]$ for every $\tau \in \mathcal{T}$ } for $i = 1, \ldots, m$

JW3. $\rho_{\tau_i} \;\wedge\; \varphi_i \;\Rightarrow\; \bigvee_{j=0}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j)$

JW4. $\varphi_i \;\Rightarrow\; En(\tau_i)$

$p \;\Rightarrow\; q$

Fig. 33. Past version of rule WELL-J

position $k \leq j$, y equaled n while control was at ℓ_0. This property characterizes a feature of program UP-DOWN by which a computation that achieves $y \geq n$ at some state, has at least two occurrences of states in which $y = n$. One occurrence has control at ℓ_0 while the other occurrence has control at ℓ_2.

In our proof, we use the following invariants for program UP-DOWN

$$\chi_0\colon \quad y \geq 0$$

$$\chi_1\colon \quad (at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0) \quad \vee \quad (at\_\ell_{0..4} \;\wedge\; at\_m_1 \;\wedge\; x = 1)$$

$$\chi_2\colon \quad at\_\ell_4 \;\rightarrow\; y = 0$$

$$\chi_3\colon \quad y < n \;\vee\; (at\_\ell_0 \;\wedge\; y = n).$$

State invariants χ_0, χ_1, and χ_2 are derived and proven in the usual way.

Proving the Invariance of χ_3

Formula χ_3 is a past invariant, and can be proven by rule P-INV, taking $\varphi = \psi\colon\; y < n \vee (at\_\ell_0 \wedge y = n)$. Premise P1 of rule P-INV is trivial since $\varphi = \psi$. Premise P2 requires

$$\underbrace{\cdots at\_\ell_0 \;\wedge\; y = 0 \;\wedge\; \cdots}_{\Theta} \;\rightarrow\; \underbrace{y < n \;\vee\; (at\_\ell_0 \;\wedge\; y = n)}_{\varphi_0}\,.$$

As $n \geq 0$, we consider two cases. If $n > 0$ then $y = 0$ implies $y < n$. If $n = 0$ then $at\_\ell_0 \wedge y = 0$ implies $at\_\ell_0 \wedge y = n$.

Finally, premise P2 requires showing

$$\rho_\tau \ \wedge \ \underbrace{y < n \ \vee \ \ (at\_\ell_0 \ \wedge \ y = n)}_{\varphi} \ \Rightarrow$$
$$\underbrace{y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n) \ \vee \ \ (at\_\ell_0 \ \wedge \ y = n)}_{\varphi'} .$$

This can be shown by temporal instantiation of the implication

$$\rho_\tau \ \wedge \ (y < n \ \vee \ p) \ \Rightarrow \ \big(y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n) \ \vee \ p\big).$$

Obviously if $p = \mathrm{T}$ the implication is trivially valid. It therefore remains to show that the following holds:

$$\rho_\tau \ \wedge \ y < n \ \rightarrow \ y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n).$$

This implication can be potentially falsified only by a transition that can transform a state satisfying $y < n$ into a next state satisfying $\neg(y' < n)$, i.e., $y' \geq n$. The only candidate transition is ℓ_1. Therefore, we consider

$$\underbrace{\cdots \ \wedge \ at'\_\ell_0 \ \wedge \ y' = y + 1}_{\rho_\tau} \ \wedge \ y < n \ \rightarrow \ y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n).$$

As $y < n$, we consider two cases. If $y < n - 1$ then $y' = y + 1 < n$. If $y = n - 1$ then $at'\_\ell_0 \wedge y' = y + 1$ implies $at'\_\ell_0 \wedge y' = n$.

This concludes the proof of past invariant χ_3.

Proving the Response Formula

To prove the response formula

$$\underbrace{0 \leq n \leq y}_{p} \ \Rightarrow \ \ \underbrace{y = n \ \wedge \ \ (at\_\ell_0 \ \wedge \ y = n)}_{q},$$

we use the past version of rule WELL-J as presented in Fig. 33.

The choice of intermediate past formulas φ_1–φ_5, helpful transitions and ranking functions is presented in the verification diagram of Fig. 34.

Note that each of $\varphi_0, \ldots, \varphi_5$ is a past formula. For example, the full formula φ_4 is given by

$$\varphi_4: \quad at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y > n \geq 0 \ \wedge \ \ (at\_\ell_0 \ \wedge \ y = n).$$

Let us consider some of the premises required by rule WELL-J. Premise JW1 requires the following implication

$$0 \leq n \leq y \ \Rightarrow$$
$$\underbrace{(at\_\ell_0 \ \wedge \ y = n) \ \wedge \ \left(y = n \ \vee \ y > n \wedge \begin{pmatrix} at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0 \\ \vee \\ at\_\ell_{0..3} \ \wedge \ at\_m_1 \ \wedge \ x = 1 \end{pmatrix}\right)}_{\varphi_0 \ \vee \ \cdots \ \vee \ \varphi_5} .$$

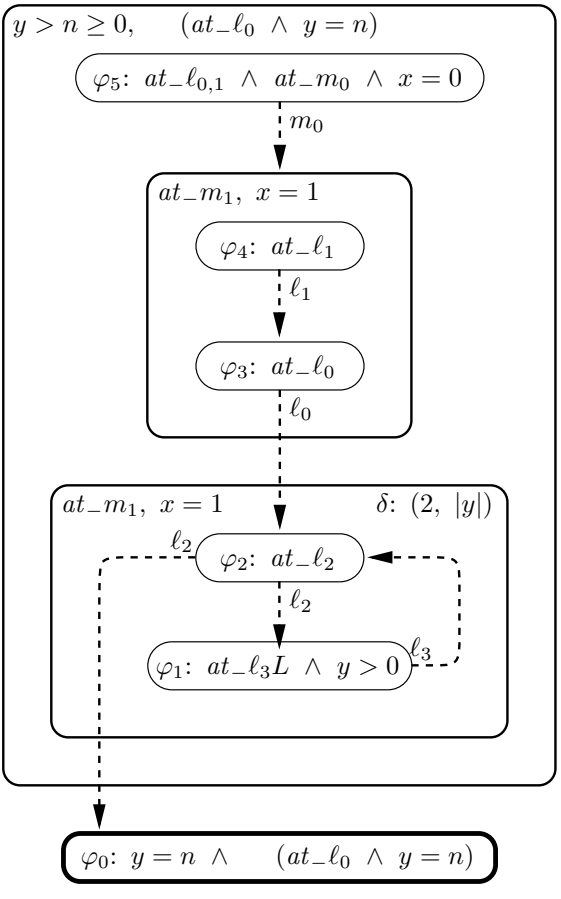


Fig. 34. RANK diagram for $(0 \leq n \leq y) \ \Rightarrow \quad \big(y = n \ \wedge \quad (at\_\ell_0 \ \wedge \ y = n)\big)$

It is not difficult to see that this entailment follows from invariants χ_0–χ_3.

Observe that each φ_i, $i = 0, \ldots, 5$ can be written in the form

$$\varphi_i \quad = \quad (at\_\ell_0 \ \wedge \ y = n) \ \wedge \ \widehat{\varphi}_i,$$

where $\widehat{\varphi}_i$ is a state formula.

Consequently, premise JW2 can be written as follows

$$\rho_\tau \ \wedge \ \widehat{\varphi}_i \ \wedge \quad (at\_\ell_0 \ \wedge \ y = n) \ \Rightarrow$$

$$\Big(\quad (at\_\ell_0 \ \wedge \ y = n)\Big)' \ \wedge \ \left(\widehat{\varphi}'_0 \ \vee \ \bigvee_{j=1}^{5} \ (\widehat{\varphi}'_j \ \wedge \ \delta_i \succ \delta'_j) \ \vee \ (\widehat{\varphi}_i \ \wedge \ \delta_i = \delta'_i)\right).$$

Since $(at\_\ell_0 \wedge y = n)$ entails $\big(\quad (at\_\ell_0 \wedge y = n)\big)'$, which expands to the disjunction

$$\big(\quad (at\_\ell_0 \ \wedge \ y = n)\big)' : \quad (at'\_\ell_0 \ \wedge \ y' = n) \ \vee \quad (at\_\ell_0 \ \wedge \ y = n),$$

it only remains to establish the following state entailment

$$\rho_\tau \ \wedge \ \widehat{\varphi}_i \quad \Rightarrow \quad \widehat{\varphi}'_0 \ \vee \ \bigvee_{j=1}^{5} (\widehat{\varphi}'_j \ \wedge \ \delta_i \succ \delta'_j) \ \vee \ (\widehat{\varphi}_i \ \wedge \ \delta_i = \delta'_i).$$

This entailment can be proven in a way similar to the proof of the verification conditions in the RANK diagram of Fig. 32, whose ranking functions are identical to those of Fig. 34.

The past conjunct $(at\_\ell_0 \ \wedge \ y = n)$ can be similarly factored out also for premise JW3.

This concludes the proof that property

$$0 \le n \le y \quad \Rightarrow \quad \big(n = y \ \wedge \quad (at\_\ell_0 \ \wedge \ y = n)\big)$$

for program UP-DOWN. ∎

8 Compositional Verification of Response Properties

Compositional verification is a method intended to reduce the complexity of verifying properties of large programs. The method infers properties of the whole system from properties of its components, which are proven separately for each component.

We apply compositional verification to programs that can be decomposed into several top-level processes, called *components*, which communicate by shared variables and such that every variable of the program can be modified by at most one of these components. A variable which is modified by component P_i is said to be *owned* by P_i.

Modular Computations

Let $P :: \big[\text{declarations}; \ \big[P_1 :: \ [\ell_0^1 \colon \ S_1] \| \cdots \| P_k :: \ [\ell_0^k \colon \ S_k]\big]\big]$ be a program, and P_i be a component of P. Denote by $V = \{\pi\} \cup Y$ the set of system variables of P, and let $Y_i \subseteq Y$ be the set of variables owned by P_i. Let L_i denote the set of locations of process P_i.

Assume that we have constructed the fair transition system (FTS) $S_P \colon \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}\rangle$ corresponding to P. We assume that the initial condition has the form

$$\Theta \colon \quad \pi \ = \ \{\ell_0^1, \ldots, \ell_0^k\} \ \wedge \ \bigwedge_{y \in Y} p_y(y),$$

where, for each $y \in Y$, $p_y(y)$ is an assertion constraining the initial values of variable y. Obviously, $p_y(y)$ is derived from one of the *where* clauses in the

declarations of variables in P. If there is no *where* clause constraining y, then $p_y(y) = \text{T}$. Let T_i denote the transitions of S_P associated with the statements of P_i.

Based on the FTS S_P and process P_i, we construct a new FTS $S^M_{P_i}$: $\langle V_i, \Theta_i, \mathcal{T}_i, \mathcal{J}_i, \mathcal{C}_i \rangle$ called the *modular* FTS *corresponding to* P_i. The FTS $S^M_{P_i}$ is intended to capture the possible behavior of process P_i in any context (not necessarily that of P) which respects the ownership of Y_i by P_i. That is, we are ready to consider any context whose only restriction is that it cannot modify any variable owned by P_i. The constituents of $S^M_{P_i}$ are given by:

- V_i: V
  The system variables of $S^M_{P_i}$ are identical to the system variables of the complete FTS S_P.
- Θ_i: $\big(\pi \cap L_i = \{\ell^i_0\}\big) \quad \wedge \quad \bigwedge_{y \in Y_i} p_y(y)$
  The initial condition of $S^M_{P_i}$ requires that, initially, the only L_i-location contained in π is ℓ^i_0 and all the variables owned by P_i satisfy their initial constraints as specified in the *where* clauses of the program declarations. Except for π, nothing is required by Θ_i concerning the system variables not owned by P_i.
- $\mathcal{T}_i = T_i \cup \{\tau_E\}$
  The transitions of $S^M_{P_i}$ include all transitions associated with statements of P_i (T_i) and a special *environment transition* τ_E. Transition τ_E is intended to represent the actions of an arbitrary context which respects the ownership of Y_i by P_i. For each $\tau \in T_i$, the transition relation associated with τ in program $S^M_{P_i}$ is ρ_τ, the transition relation associated with τ in the original program S_P. The transition relation for τ_E is given by

  $$\rho_E: \quad (\pi' \cap L_i = \pi \cap L_i) \quad \wedge \quad pres(Y_i).$$

  Note that $pres(Y_i) : \bigwedge_{y_i \in Y_i} y'_i = y_i$. This transition relation guarantees the preservation of the L_i-part of π and preservation of the values of all variables owned by P_i. The special treatment of π can be described by saying that, in addition to owning the variables in Y_i, P_i also owns the L_i-part of π (projection of π on L_i).
- $\mathcal{J}_i = \mathcal{J} \cap T_i$
  The just transitions of $S^M_{P_i}$ are the just transitions among T_i.
- $\mathcal{C}_i = \mathcal{C} \cap T_i$
  The compassionate transitions of $S^M_{P_i}$ are the compassionate transitions among T_i.

There is no need to include the idling transition τ_I in $\mathcal{T}_i$ because the effect of τ_I, a transition that changes no system variable, can be obtained as a special case of τ_E.

We refer to each computation of FTS $S^M_{P_i}$ as a *modular computation* of process P_i. As previously explained, any such computation represents a possible behavior of process P_i when put in an arbitrary context which is only required to respect the ownership rights of P_i.

Example (program KEEPING-UP)

Consider program KEEPING-UP presented in Fig. 35. Top-level process P_1 owns variable x and the $\ell_{0..2}$-part of π. We use $\ell_{0..2}$ as abbreviation for $\{\ell_0, \ell_1, \ell_2\}$. We can construct $S^M_{P_1}$, the modular FTS corresponding to process P_1 as follows:

$$\textbf{local } x,\ y : \textbf{integer where } x = y = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{await } x < y + 1 \\ \ell_2\text{: } x := x + 1 \end{array}\right] \end{array}\right]$$

$$\| \quad P_2 :: \left[\begin{array}{l} m_0\text{: } x := 1 \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{await } y < x + 1 \\ m_2\text{: } y := y + 1 \end{array}\right] \end{array}\right]$$

Fig. 35. Program KEEPING-UP

- V_1: $\{\pi, x, y\}$
- Θ_1: $\big(\pi \cap \ell_{0..2} = \{\ell_0\}\big) \wedge x = 0$
- $\mathcal{T}_1$: $\{\tau_{\ell_0}, \tau_{\ell_1}, \tau_{\ell_2}, \tau_E\}$

with the following transition relations (after some simplifications):

$$\begin{array}{l} \rho_{\ell_0}\text{: } move(\ell_0, \ell_1) \wedge pres(x, y) \\ \rho_{\ell_1}\text{: } move(\ell_1, \ell_2) \wedge x < y + 1 \wedge pres(x, y) \\ \rho_{\ell_2}\text{: } move(\ell_2, \ell_0) \wedge x' = x + 1 \wedge pres(y) \\ \rho_E\text{: } (\pi' \cap \ell_{0..2} = \pi \cap \ell_{0..2}) \wedge pres(x) \end{array}$$

In this relations, we used the following abbreviation:

$$move(\ell_i, \ell_j)\text{: } at\_\ell_i \wedge \pi' = \big(\pi - \{\ell_i\}\big) \cup \{\ell_j\}$$

- $\mathcal{J}_1$: $\{\tau_{\ell_0}, \tau_{\ell_1}, \tau_{\ell_2}\}$
- $\mathcal{C}_1$: $\emptyset$

The following is a modular computation of process P_1:

$$\begin{aligned}\widehat{\sigma}:\quad & \langle \pi{:}\{\ell_0, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_0} \langle \pi{:}\{\ell_1, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_1} \\ & \langle \pi{:}\{\ell_2, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\tau_E} \langle \pi{:}\{\ell_2, m_0\},\ x{:}0,\ y{:}-1\rangle \xrightarrow{\ell_2} \\ & \langle \pi{:}\{\ell_0, m_0\},\ x{:}1,\ y{:}-1\rangle \cdots .\end{aligned}$$

In a similar way, we can construct $S^M_{P_2}$, the modular FTS corresponding to process P_2. ∎

The following claim establishes a connection between computations of the entire program and modular computations of its processes.

> *Claim.* (computations of programs and modular computations)
> Every computation of a program is a modular computation of each of its top-level processes.
> Thus, the set of computations of the entire program is a subset of the set of modular computations of each of its top-level processes.

Example. Consider, for example, the following computation of program KEEPING-UP

$$\begin{aligned}\sigma:\quad & \langle \pi{:}\{\ell_0, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_0} \langle \pi{:}\{\ell_1, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_1} \\ & \langle \pi{:}\{\ell_2, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{m_0} \langle \pi{:}\{\ell_2, m_1\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_2} \\ & \langle \pi{:}\{\ell_0, m_1\},\ x{:}1,\ y{:}0\rangle \xrightarrow{m_1} \langle \pi{:}\{\ell_0, m_2\},\ x{:}1,\ y{:}0\rangle \xrightarrow{m_2} \\ & \langle \pi{:}\{\ell_0, m_0\},\ x{:}1,\ y{:}1\rangle \xrightarrow{\ell_0} \langle \pi{:}\{\ell_1, m_0\},\ x{:}1,\ y{:}1\rangle \cdots .\end{aligned}$$

Viewed as a modular computation of process P_1, this computation can be presented as:

$$\begin{aligned}\sigma_1:\quad & \langle \pi{:}\{\ell_0, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_0} \langle \pi{:}\{\ell_1, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_1} \\ & \langle \pi{:}\{\ell_2, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\tau_E} \langle \pi{:}\{\ell_2, m_1\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\ell_2} \\ & \langle \pi{:}\{\ell_0, m_1\},\ x{:}1,\ y{:}0\rangle \xrightarrow{\tau_E} \langle \pi{:}\{\ell_0, m_2\},\ x{:}1,\ y{:}0\rangle \xrightarrow{\tau_E} \\ & \langle \pi{:}\{\ell_0, m_0\},\ x{:}1,\ y{:}1\rangle \xrightarrow{\ell_0} \langle \pi{:}\{\ell_1, m_0\},\ x{:}1,\ y{:}1\rangle \cdots .\end{aligned}$$

Viewed as a modular computation of process P_2, this computation can be presented as:

$$\begin{aligned}\sigma_2:\quad & \langle \pi{:}\{\ell_0, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\tau_E} \langle \pi{:}\{\ell_1, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\tau_E} \\ & \langle \pi{:}\{\ell_2, m_0\},\ x{:}0,\ y{:}0\rangle \xrightarrow{m_0} \langle \pi{:}\{\ell_2, m_1\},\ x{:}0,\ y{:}0\rangle \xrightarrow{\tau_E} \\ & \langle \pi{:}\{\ell_0, m_1\},\ x{:}1,\ y{:}0\rangle \xrightarrow{m_1} \langle \pi{:}\{\ell_0, m_2\},\ x{:}1,\ y{:}0\rangle \xrightarrow{m_2} \\ & \langle \pi{:}\{\ell_0, m_0\},\ x{:}1,\ y{:}1\rangle \xrightarrow{\tau_E} \langle \pi{:}\{\ell_1, m_0\},\ x{:}1,\ y{:}1\rangle \cdots .\end{aligned}$$

This illustrates that a computation of a program is a modular computation of each of its top-level processes. ▟

The weak converse of Claim 8 is not true. There are modular computations of process P_i which do not correspond to computations of the entire program. This is illustrated by $\widehat{\sigma}$ the previously presented modular computation of process P_1 in program KEEPING-UP. This computation contains a state with $y = -1$ as a τ_E-successor of a state with $y = 0$. No such state can occur in a computation of KEEPING-UP. This shows that the definition of modular computations of process P_i allows more general contexts than the actual context provided by the program containing P_i. The actual context of P_1 within program KEEPING-UP is process P_2 which can never change y from a value of 0 to a value of -1.

On the other hand, the strong converse of Claim 8 is true. Let σ be a model (infinite state sequence) such that the interpretation of π is a subset of the locations of P. The valid converse of Claim 8 states that if σ is simultaneously a modular computation of every top-level process of P then σ is a computation of P. In **Problem 10**, we request the reader to prove this fact.

Modular Validity and a Basic Compositionality Rule

For a top-level process P_i within program P, we say that formula φ is *modularly valid over* P_i, denoted

$$P_i \ \ _m \ \varphi,$$

if φ holds over all modular computations of P_i. For example, the formula $(x \geq x^-)$, stating that x never decreases, is modularly valid over process P_1 of program KEEPING-UP (Fig. 42), while $(y \geq y^-)$ is modularly valid over process P_2 of the same program[3].

Rule COMP-B, presented in Fig. 36, infers the P-validity of a formula φ from the premise that φ is modularly valid over some top-level process of P.

For a program P, P_i a top-level process of P, and φ a temporal formula

$$\frac{P_i \ \ _m \ \varphi}{P \quad \varphi}$$

Fig. 36. Rule COMP-B (basic compositionality)

Soundness

Let P be a program and P_i be a top-level process of P. Assume that formula φ is modularly valid over P_i, i.e. $P_i \ \ _m \ \varphi$. This means that φ holds over all modular

[3] x^- and y^- denote the values of x and y in the preceding state.

computations of P_i. By Claim 8, every computation of P is also a modular computation of P_i. It follows that all computations of P satisfy φ and, hence, φ is P-valid. ∎

Rule COMP-B can be used to reduce the goal of establishing $P \quad \varphi$ into the subgoals of establishing several modular validities (not necessarily of the same formula φ. In **Problem 11** the reader is requested to establish this fact.

A Compositional Rule for Safety Properties

In theory, rule COMP-B is adequate for compositional verification of any temporal formula. In practice, however, its application often proves inconvenient and calls for additional temporal reasoning. Therefore, it is advantageous to derive more specific rules, each of which is tailored to deal with temporal formulas of particular classes.

In Fig. 37 we present rule COMP-S which can be used for compositional verification of safety formulas.

For P_i, a top-level process of program P, and past formulas χ, p,

$$\frac{\begin{array}{ll}\text{CS1.} & P \qquad \chi \\ \text{CS2.} & P_i \ {}_m \qquad \chi \ \Rightarrow \ p\end{array}}{P \qquad p}$$

Fig. 37. Rule COMP-S (compositional verification of safety properties)

Premise CS1 states that χ is an invariant of the entire program P. Premise CS2 states that the entailment $\chi \Rightarrow p$ is modularly valid over some top-level process P_i. From these two assumptions, the rule infers that p is an invariant of P.

Justification. Assume that premises CS1 and CS2 hold and let σ be a computation of program P. By premise CS1, formula χ holds at all positions of σ. Since every computation of P is also a modular computation of P_i, premise CS2 implies that the formula $\chi \rightarrow p$ holds at all positions of σ.

Consider an arbitrary position $j \geq 0$ of σ. By CS1, χ holds at all positions $k \leq j$ and, therefore χ holds at j. By CS2, $\chi \rightarrow p$ holds at j and, therefore, so does p.

We conclude that p holds at all positions of σ. ∎

Rule COMP-S is often used in an incremental style. As a first step we take $\chi = \text{T}$ and prove $P_i \ {}_m \quad p_1$. From this the rule infers

$$P \qquad p_1.$$

Next, we take $\chi = p_1$ and prove $P_i \;\; {}_m \;\; p_1 \Rightarrow p_2$. This leads to

$$P \qquad p_2,$$

which may be followed by additional steps.

The advantage of this proof pattern is that in each step we concentrate on proving a modular validity over a single process P_i. If P_i is only a small part of the program, each compositional verification step has to consider only a small fraction of the transitions in the complete program.

We illustrate the use of rule COMP-S on a simple example.

Example (program KEEPING-UP)

Consider program KEEPING-UP presented in Fig. 35. Process P_1 in this program repeatedly increments x, provided x does not exceed $y+1$. In a symmetric way, process P_2 repeatedly increments y, provided y does not exceed $x+1$.

We wish to prove for this program the invariance of the assertion $|x-y| \leq 1$, i.e.,

$$\underbrace{|x-y| \leq 1}_{p},$$

claiming that the difference between x and y never exceeds 1 in absolute value.

We prove this property by compositional verification, using rules INV-P and COMP-S. We first show the P-validities

$$P \qquad (x \geq x^-) \qquad \text{and} \qquad P \qquad (y \geq y^-),$$

and then the P-validities

$$P \qquad (x \leq y+1) \qquad \text{and} \qquad P \qquad (y \leq x+1).$$

The invariants $x \leq y+1$ and $y \leq x+1$ imply the desired P-validity

$$P \qquad (|x-y| \leq 1).$$

For more details of this proof, we refer the reader to Section 4.3 of the SAFETY book. ⌟

A Compositional Rule for Response Properties

Next, we present a rule that can support compositional verification of response properties and illustrate its use. This is rule COMP-R, presented in Fig. 38.

Justification. Assume that premises CR1 and CR2 hold and let σ be a computation of program P. By premise CR1, formula χ holds at all positions of σ. Since every computation of P is also a modular computation of P_i, premise CR2 implies that the formula $p \rightarrow \;\; (q \vee \neg\chi)$ holds at all positions of σ. Let j be a p-position of σ. By CR2, there exists a position $k \geq j$ such that either q holds at k or χ is false at k. The second alternative is impossible, due to CR1. We conclude that every p-position is followed by a q-position and, therefore, $p \Rightarrow \;\; q$ is valid over P. ⌟

For P_i, a top-level process of program P, and past formulas χ, p, and q,

$$\begin{array}{lll} \text{CR1.} & P & \Box\chi \\ \text{CR2.} & P_i & \Box_m\, p \Rightarrow \Diamond(q \vee \neg\chi) \\ \hline & P & p \Rightarrow \Diamond q \end{array}$$

Fig. 38. Rule COMP-R (compositional verification of response properties)

Example (program PING-PONG)

We illustrate the use of rule COMP-R for compositional verification of response properties on an example. In Fig. 39, we present program PING-PONG.

local $x,\ y,\ z$: **integer where** $x = y = z = 0$

$$P_1 :: \left[\begin{array}{l} \textbf{own out } x,\ z \\ \ell_0\text{: } x := 1 \\ \ell_1\text{: } \textbf{await } y > 0 \\ \ell_2\text{: } z := 1 \\ \ell_3\text{:} \end{array}\right] \quad || \quad P_2 :: \left[\begin{array}{l} \textbf{own out } y \\ m_0\text{: } \textbf{await } x > 0 \\ m_1\text{: } y := 1 \\ m_2\text{:} \end{array}\right]$$

Fig. 39. Program PING-PONG

The two processes of this program maintain a coordination protocol. The protocol starts by P_1 setting x to 1 at statement ℓ_0. This is sensed by P_2 at m_0, and is responded to by setting y to 1 at statement m_1. This is sensed by P_1 at ℓ_1, and is responded to by setting z to 1 at ℓ_2.

We wish to establish for this program the response property

$$\Theta \Rightarrow \Diamond(z = 1).$$

We start by proving, using rule P-INV, the modular invariance

$$P_1 \ \Box_m \ (x \geq x^-).$$

This is easy to prove since the local formula $x \geq x^-$ is inductive over the modular FTS corresponding to process P_1.

By rule COMP-B we can infer

$$P \ \Box\ (x \geq x^-).$$

In a similar way, we establish $P_2 \;\;_m \;\; (y \geq y^-)$, leading to

$$P \qquad (y \geq y^-).$$

Now, we use rule RESP-J to prove

$$P_1 \;\;_m \;\; \underbrace{\Theta}_{p} \;\Rightarrow\; \underbrace{x > 0}_{q}\ .$$

As the intermediate assertion and helpful transition, we take φ: at_ℓ_0 and τ_h: ℓ_0.

Using rule COMP-R with χ: T, we conclude

$$P \qquad \Theta \;\Rightarrow\; (x > 0).$$

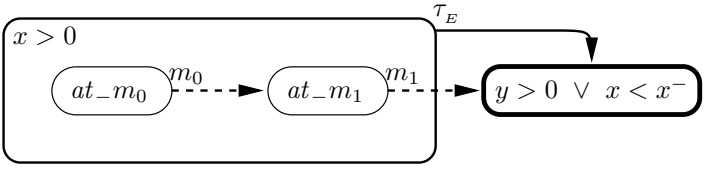
Next, we intend to establish

$$P_2 \;\;_m \;\; x > 0 \;\Rightarrow\; (y > 0 \;\vee\; x < x^-).$$

As a first step, we use rule INV to prove

$$P_2 \;\;_m \;\; \underbrace{at\_m_{0,1} \;\vee\; y > 0}_{\varphi_2}\ .$$

Having established the modular invariance of φ_2 over P_2, the following verification diagram proves that $x > 0 \Rightarrow \;\; (y > 0 \vee x < x^-)$ is modularly valid over P_2.



Note that the diagram allows the possibility that the environment changes x from a positive value to a non positive one. However, such a change leads to a position satisfying $x < x^-$.

Now, use rule COMP-R with χ: $x \geq x^-$, p: $x > 0$, and q: $y > 0$, to conclude

$$P \qquad x > 0 \;\Rightarrow\; (y > 0).$$

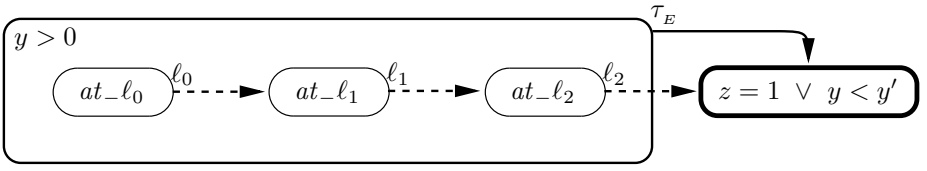
Next, we plan to establish

$$P_1 \;\;_m \;\; y > 0 \;\Rightarrow\; (z = 1 \;\vee\; y < y^-).$$

As a first step, we use rule INV to prove

$$P_1 \;\;_m \;\; (at\_\ell_{0..2} \;\vee\; z = 1).$$

Having established this modular invariant, the following verification diagram proves that $y > 0 \Rightarrow \;\; (z = 1 \vee y < y^-)$ is modularly valid over P_1.



Now, use rule COMP-R with χ: $y \geq y^-$, p: $y > 0$, and q: $z = 1$, to conclude

$$P \qquad y > 0 \ \Rightarrow \qquad (z = 1).$$

Thus, we have shown that the three response properties $\Theta \Rightarrow \quad (x > 0)$, $x > 0 \Rightarrow \quad (y > 0)$, and $y > 0 \Rightarrow \quad (z = 1)$ are all valid over program PING-PONG. Using rule TRNS-R, we conclude

$$\Theta \ \Rightarrow \qquad (z = 1). \ \blacksquare$$

9 Guarantee Properties

In this and the following section we consider methods for proving properties belonging to the *guarantee* and *obligation* classes. Our approach to these classes is to consider them as special cases of the response class, and to use response rules with some simplifications for their verification.

As defined in Section 0.5 of the SAFETY book, guarantee properties are properties that can be specified by a formula of the form

$$\qquad r$$

for some past formula r.

Clearly, guarantee properties are a special case of response properties, $p \Rightarrow \quad r$, where the antecedent p refers to the beginning of the computation. Consequently, an obvious rule for proving guarantee properties, is rule GUAR (Fig. 40).

For past formula r

$$\frac{\mathit{first} \ \wedge \ \Theta \ \Rightarrow \qquad r}{\qquad r}$$

Fig. 40. Rule GUAR (proving guarantee properties)

The rule requires as a premise a response property by which the initial condition of the program guarantees the eventual realization of r.

Example. Consider system INC2, presented in Fig. 41.

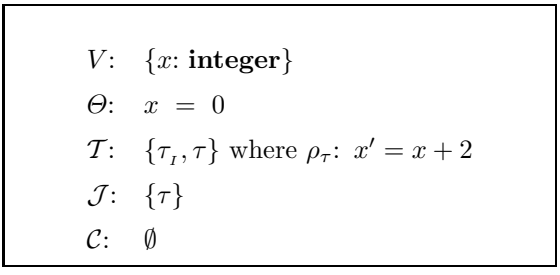


Fig. 41. System INC2

We wish to establish the guarantee property

$$\big(x \geq 20 \ \wedge \quad (x = 10)\big)$$

for system INC2, using rule GUAR. The premise of rule GUAR requires the response property

$$\overbrace{\mathit{first} \ \wedge \ \underbrace{x = 0}_{\Theta}}^{p} \ \Rightarrow \quad \Big(\underbrace{x \geq 20 \ \wedge \quad (x = 10)}_{r}\Big) \ .$$

This response property can be proven by rule WELL-J, using the intermediate past formula

$$\varphi\colon \quad \mathit{even}(x) \ \wedge \ 0 \leq x \leq 18 \ \wedge \ \big(x \geq 10 \ \rightarrow \quad (x = 10)\big),$$

the helpful transition τ, and the ranking function $\delta\colon |20 - x|$.

Let us establish premise JW1 of rule WELL-J. It requires showing

$$\underbrace{\cdots \ \wedge \ x = 0}_{p} \ \Rightarrow \ \cdots \ \vee \ \underbrace{\mathit{even}(x) \ \wedge \ 0 \leq x \leq 18 \ \wedge \ \big(x \geq 10 \ \rightarrow \quad (x = 10)\big)}_{\varphi} .$$

Clearly, $x = 0$ entails all three conjuncts comprising φ. Note that, in this case, we do not use the conjunct *first* which is part of p.

By rule GUAR, we conclude that property

$$\big(x \geq 20 \ \wedge \quad (x = 10)\big)$$

is valid over system INC2. ∎

The premise of rule GUAR contains *first* as part of the antecedent. Its purpose is to ensure that we only consider Θ at the beginning of the computation. As illustrated in the last example, in many cases we do not use this conjunct and simply prove $\Theta \Rightarrow \quad p$. There are, however, some cases in which this conjunct is necessary, as illustrated below.

Example. Consider the simple program

$$\textbf{local } x\text{: } \textbf{integer where } x = 1$$

$$\ell_0\text{: } \textbf{loop forever do } [\ell_1\text{: } \textbf{skip}; \ell_2\text{: } x := -x]$$

We consider the guarantee property

$$\psi\text{:} \qquad \big(at\_\ell_2 \ \wedge \quad (x = 1)\big).$$

This property states that every computation of the program contains a position j satisfying at_ℓ_2, and such that $x = 1$ at all positions $i \leq j$.

This property is certainly valid for the program. To prove it, we have to establish the response property

$$\mathit{first} \ \wedge \ at\_\ell_0 \ \wedge \ x = 1 \quad \Rightarrow \qquad \big(at\_\ell_2 \ \wedge \quad (x = 1)\big).$$

Note, however, that the *first* conjunct is essential in this case, since the formula

$$at\_\ell_0 \ \wedge \ x = 1 \quad \Rightarrow \qquad \big(at\_\ell_2 \ \wedge \quad (x = 1)\big)$$

is not valid over the program.

Consider position i in the computation, corresponding to the third visit to ℓ_2. At this position $x = 1$, but there exist earlier positions in which $x = -1$. Therefore, there exists no position later than i, at which $at\_\ell_2 \wedge \quad (x = 1)$ holds. It follows that the implication $at\_\ell_0 \wedge x = 1 \rightarrow \quad \big(at\_\ell_2 \wedge \quad (x = 1)\big)$ does not hold at i.

To prove that the full premise

$$\underbrace{\mathit{first} \ \wedge \ at\_\ell_0 \ \wedge \ x = 1}_{p} \quad \Rightarrow \qquad \Big(\underbrace{at\_\ell_2 \ \wedge \quad (x = 1)}_{r}\Big),$$

is valid, we may use rule CHAIN-J (Fig. 7) with the following intermediate past formulas and helpful transitions:

$$\begin{array}{rlll} \varphi_2\text{:} & at\_\ell_0 \ \wedge & (x = 1) & \tau_2\text{: } \ell_0 \\ \varphi_1\text{:} & at\_\ell_1 \ \wedge & (x = 1) & \tau_1\text{: } \ell_1 \\ \varphi_0 = r\text{:} & at\_\ell_2 \ \wedge & (x = 1) & \tau_0\text{: } \ell_2. \end{array}$$

Premise J1 requires showing

$$\underbrace{\mathit{first} \ \wedge \ at\_\ell_0 \ \wedge \ x = 1}_{p} \quad \Rightarrow \quad \cdots \quad \vee \quad \underbrace{at\_\ell_0 \ \wedge \quad (x = 1)}_{\varphi_2}.$$

Clearly, the antecedent implies at_ℓ_0. To see that it also implies $\quad (x = 1)$, we observe that under *first*, any past formula p is congruent to $(p)_0$, the initial version of p. Since the initial version of $\quad (x = 1)$ is

$$\big(\quad (x = 1)\big)_0 \ = \ (x = 1),$$

the right-hand side simplifies to $at\_\ell_0 \wedge x = 1$ which is entailed by the left-hand side.

Another premise that has to be checked is J3 for transition ℓ_0,

$$\rho_{\ell_0} \ \wedge \ \underbrace{at\_\ell_0 \ \wedge \ \ (x=1)}_{\varphi_2} \ \Rightarrow \ \cdots \ \vee \ \underbrace{at'\_\ell_1 \ \wedge \ x'=1 \ \wedge \ \ (x=1)}_{\varphi'_1} \ .$$

Since ρ_{ℓ_0} implies at'_ℓ_1 and $x'=x$, and $\ (x=1)$ implies $x=1$, the entailment is valid.

The rest of the premises are proven in a similar way. This establishes the validity of $\big(at\_\ell_2 \wedge \ \ (x=1)\big)$. ∎

Completeness of Rule GUAR

Rule GUAR is obviously sound, which means that the P-validity of the premise $\mathit{first} \wedge \Theta \Rightarrow \ \ r$ implies the P-validity of the conclusion $\ \ r$.

The rule is also *complete*, which means that the P-validity of the conclusion implies the P-validity of the premise. Consider σ, an arbitrary computation of program P. By the assumption that $\ \ r$ is P-valid, there exists a position k at which r holds. For σ to satisfy the premise we have to show that every position $i \geq 0$, satisfying $\mathit{first} \wedge \Theta$, is followed by a position j, $j \geq i$, satisfying r. Since 0 is the only position satisfying $\mathit{first} \wedge \Theta$, we can take j to be k.

Completeness of rule GUAR is important because it tells us that the rule is adequate for proving all P-valid guarantee formulas.

10 Obligation Properties

Before studying the class of obligation properties, we introduce a special class of response formulas.

Escape Formulas

Some response properties are naturally expressed by formulas of the form

$$p \ \Rightarrow \ \ q \ \vee \ \ r,$$

for past formulas p, q, and r.

This formula claims that, following a p-state, either q will hold forever or r eventually occurs. We may view such a formula as stating that, following p, q should hold continually unless we escape to a state that eventually leads to r. Consequently, we refer to formulas of this form as *escape formulas*.

To see that this formula specifies a response property, observe that it is equivalent to

$$\neg q \ \wedge \ (\neg r)\,\mathcal{W}\,(p \wedge \neg r) \ \Rightarrow \ \ r.$$

In this form, the formula states that every $\neg q$-position preceded by a p-position such that no r has occurred since, must be followed by an r-position.

While, in principle, it is possible to use the general response rules to establish escape formulas, it is more convenient to use a special rule, presented in Fig. 42.

For past formulas p, q, r, and φ

$$\begin{array}{ll} \text{E1.} & p \;\Rightarrow\; q \;\mathcal{W}\; \varphi \\ \text{E2.} & \varphi \;\Rightarrow\; r \\ \hline & p \;\Rightarrow\; q \;\vee\; r \end{array}$$

Fig. 42. Rule ESC (escape)

Rule ESC uses an auxiliary past formula φ. Premise E1 requires that, following a p-position, either q will hold forever or q will hold until an occurrence of φ. Premise E2 requires that every φ-position is followed by an r-position. Typically, we prove E1 by rule P-WAIT, a past version of rule WAIT (Fig. 3.3 of the SAFETY book), and E2 by appropriate response rules.

Example. Consider program MAY-HALT of Fig. 43. This trivial program has a nondeterministic choice at ℓ_1 between getting deadlocked at ℓ_2 or, taking the ℓ_1^a branch, proceeding to ℓ_3. Consequently, the program has some computations that reach ℓ_2 and stay there forever, and some computations that never halt.

ℓ_0: **loop forever do**

$$\left[\begin{array}{l} \ell_1: \left[\begin{array}{l} \ell_1^a\text{: } \textbf{skip} \\ \quad \textbf{or} \\ \ell_1^b\text{: } \textbf{skip};\ \ell_2\text{: } \textbf{halt} \end{array}\right] \\ \ell_3\text{: } \textbf{skip} \\ \ell_4\text{: } \textbf{skip} \end{array}\right]$$

Fig. 43. Program MAY-HALT (possible deadlock)

We use rule ESC to prove the property

$$\psi_1\colon \quad \underbrace{at\_\ell_{0,1}}_{p} \;\Rightarrow\; \underbrace{at\_\ell_{0..2}}_{q} \;\vee\; \underbrace{at\_\ell_4}_{r}$$

for this program.

Take

$$\varphi\colon \quad at\_\ell_3.$$

For past formulas q, r, and φ,

$$\begin{array}{ll} \text{O1.} & \mathit{first} \wedge \Theta \;\Rightarrow\; q \,\mathcal{W}\, \varphi \\ \text{O2.} & \varphi \;\Rightarrow\; \quad r \\ \hline & q \vee \quad r \end{array}$$

Fig. 44. Rule OBL (proving obligation properties)

Consider the two premises of rule ESC.

- Premise E1
  This premise requires
  $$\underbrace{at\_\ell_{0,1}}_{p} \;\Rightarrow\; \underbrace{at\_\ell_{0..2}}_{q} \;\mathcal{W}\; \underbrace{at\_\ell_3}_{\varphi} .$$
  It is straightforward to derive this property by rule WAIT.
- Premise E2
  $$\underbrace{at\_\ell_3}_{\varphi} \;\Rightarrow\; \quad \underbrace{at\_\ell_4}_{r} .$$
  A single application of rule RESP-J establishes this property.

This establishes the considered escape property. ∎

From Escape to Obligation

The (simple) obligation class includes all the properties that can be specified by a formula of the form

$$q \vee \quad r,$$

for past formulas q and r.

We observe that such a formula can be rewritten as

$$\underbrace{\mathit{first} \wedge \Theta}_{p} \;\Rightarrow\; \quad q \vee \quad r,$$

which represents it as a special case of an escape formula.

This observation inspires rule OBL (Fig. 44) for proving obligation properties.

Premise O1 requires that, from the beginning of the computation, q holds continuously and can be interrupted only at a position satisfying φ. Premise O2 states that φ guarantees an eventual r. Consequently, q can be interrupted only when r is guaranteed. It follows that $q \,\mathcal{W}\, (\quad r)$ is valid. By properties of the waiting-for operator, we may deduce $\quad q \vee \quad r$.

Example (incrementor-decrementor)

Consider Program INC-DEC presented in Fig. 45, which nondeterministically increments or decrements an integer variable y.

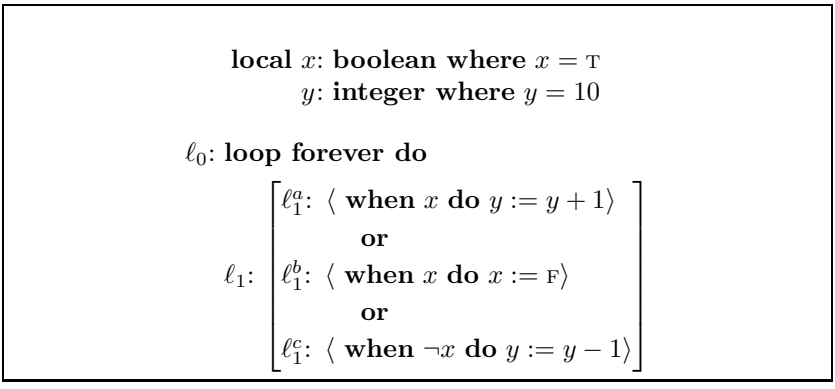


Fig. 45. Program INC-DEC (nondeterministic incrementor-decrementor)

We wish to prove for this program the obligation property

$$\underbrace{y \geq 10}_{q} \quad \vee \quad \underbrace{y = 0}_{r} \ .$$

Intending to apply rule OBL, it only remains to identify the intermediate formula φ. The main characterization of φ is that it describes the event whose occurrence guarantees the eventual realization of r.

Examining the program, we see that the first moment we realize that $y = 0$ is going to happen is when x becomes false. Consequently we take

$$\varphi: \quad \neg x \ \wedge \ y > 0.$$

The premises that have to be verified are as follows:

- Premise O1

To establish this premise it suffices to prove

$$\underbrace{\cdots \ \wedge \ \Theta}_{\widehat{p}} \quad \Rightarrow \quad \underbrace{y \geq 10}_{\widehat{q}=q} \quad \mathcal{W} \quad \underbrace{\neg x \ \wedge \ y > 0}_{\widehat{r}=\varphi} \ .$$

To prove this we use rule WAIT (Fig. 3.3 of the SAFETY book) with the intermediate assertion

$$\widehat{\varphi}: \quad x \ \wedge \ y \geq 10.$$

The three premises of rule WAIT require

$$\text{W1.} \qquad \underbrace{at\_\ell_0 \ \wedge \ x \ \wedge \ y = 10}_{\widehat{p}} \quad \rightarrow \quad \underbrace{x \ \wedge \ y \geq 10}_{\widehat{\varphi}} \ \vee \ \cdots \ ,$$

which is obviously state valid.

W2. $\underbrace{x \ \wedge \ y \geq 10}_{\widehat{\varphi}} \ \rightarrow \ \underbrace{y \geq 10}_{\widehat{q}}$,

which is trivially state valid.

W3. $\rho_\tau \ \wedge \ \underbrace{x \ \wedge \ y \geq 10}_{\widehat{\varphi}} \ \rightarrow \ \underbrace{x' \ \wedge \ y' \geq 10}_{\widehat{\varphi}'} \ \vee \ \underbrace{\neg x' \ \wedge \ y' > 0}_{\widehat{r}'}$,

for all transitions τ in the program. It is not difficult to see that all three requirements are valid.

- Premise O2

To establish this premise we have to prove

$$\underbrace{\neg x \ \wedge \ y > 0}_{\varphi} \ \Rightarrow \ \underbrace{y = 0}_{r} \ .$$

This is proven by the RANK verification diagram presented in Fig. 46.

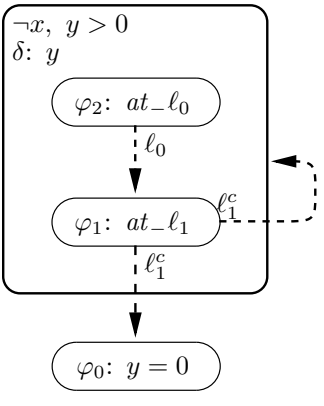


Fig. 46. Verification diagram for $\neg x \wedge y > 0 \Rightarrow \quad (y = 0)$

This concludes the proof of

$$(y \geq 10) \ \vee \quad (y = 0). \ \blacksquare$$

A general obligation property is a conjunction of the form

$$\bigwedge_{i=1}^{n} (\quad q_i \ \vee \quad r_i).$$

Consequently, to prove the validity of such a formula, it is sufficient (and necessary) to prove the validity of each conjunct, which is a simple obligation property.

Completeness of rule OBL

Rule OBL is *complete* for proving (simple) obligation properties. This means that, whenever $q \vee \quad r$ is P-valid, we can find a past formula φ, such that premises O1 and O2 are also P-valid. The choice we can always make is taking

$$\varphi: \quad r \ \vee \ (\neg q \ \wedge \quad \neg r).$$

We will show that, if the property $\quad q \vee \quad r$ is P-valid, then the two premises of rule OBL are also P-valid for this choice of φ.

- Premise O1

For premise O1 it suffices to show that

$$q \ \mathcal{W} \ \underbrace{r \ \vee \ (\neg q \ \wedge \quad \neg r)}_{\varphi}$$

is P-valid.

This formula states that either q holds forever, or it is interrupted by an r, or it is interrupted by a $\neg q$-position which is not preceded by any r-position. This formula is valid in general so, in particular, it holds over all computations.

- Premise O2

This premise requires

$$\underbrace{r \ \vee \ (\neg q \ \wedge \quad \neg r)}_{\varphi} \ \Rightarrow \quad r,$$

for which it is sufficient to show

$$\neg q \ \wedge \quad \neg r \ \Rightarrow \quad r.$$

Assume to the contrary, that there exists a computation σ and a position i such that $\neg q \wedge \quad \neg r$ holds at i, but r does not hold at any position $j \geq i$. Since $\neg q \wedge \quad r$ holds at i, r does not occur at any position $j < i$. Therefore σ does not satisfy $\quad r$. On the other hand, since $\neg q$ at i, σ also does not satisfy $\quad q$. This contradicts our assumption that $\quad q \vee \quad r$ is P-valid.

Consequently, premise O2 is also P-valid.

As we continuously remind the reader, the auxiliary formula φ constructed during a proof of completeness is not necessarily the one we recommend for actual use. In practice, we can almost always find better assertions.

Problems

Problem 1. (three values)

Prove accessibility for process P_1 of program MUX-VAL-3 of Fig. 47. This program uses the shared integer variables y_1 and y_2. Obviously, these variables can only assume one of the values $\{-1, 0, 1\}$.

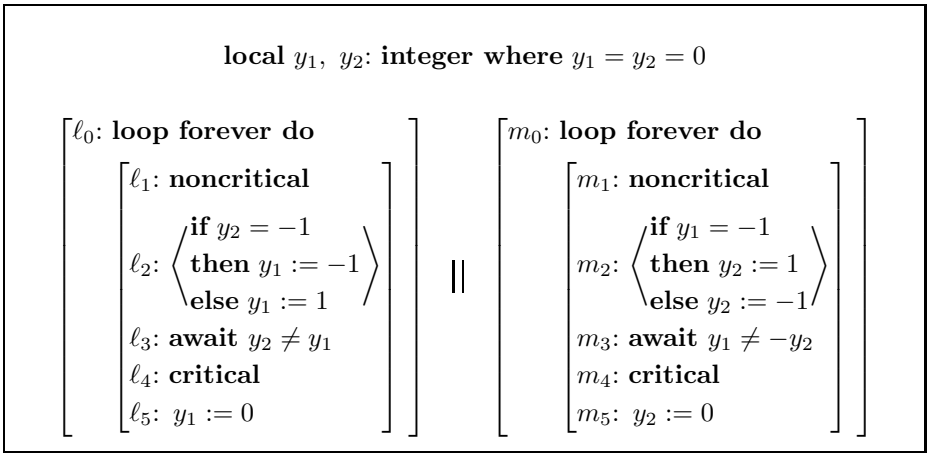


Fig. 47. Program MUX-VAL-3

Accessibility for P_1 is stated by the response formula

$$at\_\ell_2 \ \Rightarrow \qquad at\_\ell_4.$$

Problem 2. (bakery algorithms)

(a) Prove accessibility for process P_1 of program MUX-BAK-A of Fig. 48. Note that the two processes are not exactly symmetric due to the difference between statements ℓ_3^b and m_3^b.

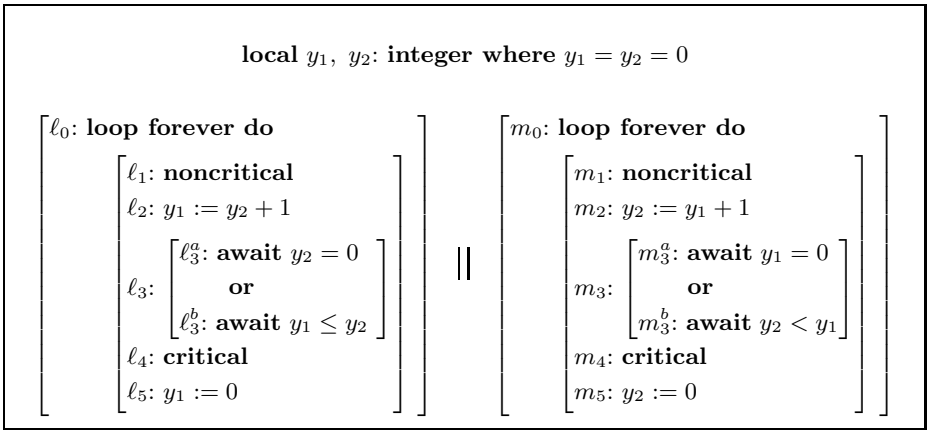


Fig. 48. Program MUX-BAK-A

The algorithm is called the *bakery* algorithm, since it is based on the idea that customers, as they enter, pick numbers which form an ascending sequence.

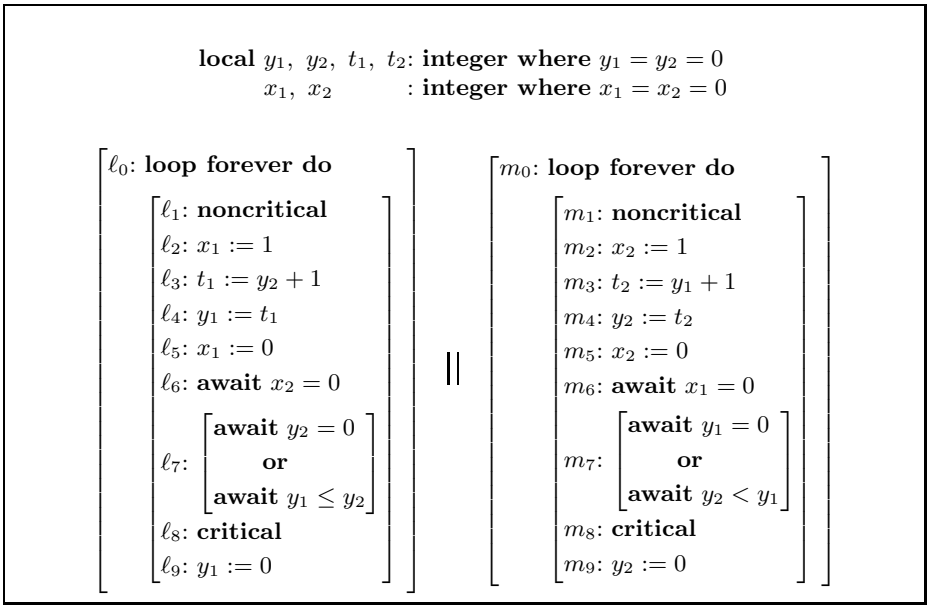


Fig. 49. Program MUX-BAK-C

Then, a customer with a lower number has higher priority in accessing its critical section. Statements ℓ_2 and m_2 ensure that the number assigned to y_i, $i = 1, 2$, is greater than the current value of y_j, $j \neq i$.

(b) Program MUX-BAK-A does not obey the LCR restriction. In particular, statements ℓ_2 and m_2 each contain two critical references: to y_1 and to y_2. To correct this situation, we propose program MUX-BAK-C of Fig. 49. This LCR-program contains two additional *await* statements that ensure that processes do not wait too long at locations ℓ_3 or m_3. Show that program MUX-BAK-C guarantees accessibility for process P_1.

Problem 3. (variants of Dekker's algorithm)

(a) Prove accessibility for process P_1 of program MUX-DEK-A of Fig. 50. That is, show that the response formula

$$at\_\ell_2 \ \Rightarrow \ \diamondsuit at\_\ell_7$$

is valid over MUX-DEK-A.

(b) Prove accessibility for process P_1 of program MUX-DEK-B of Fig. 51. That is, show that the response formula

$$at\_\ell_2 \ \Rightarrow \ \diamondsuit at\_\ell_7$$

is valid over MUX-DEK-B.

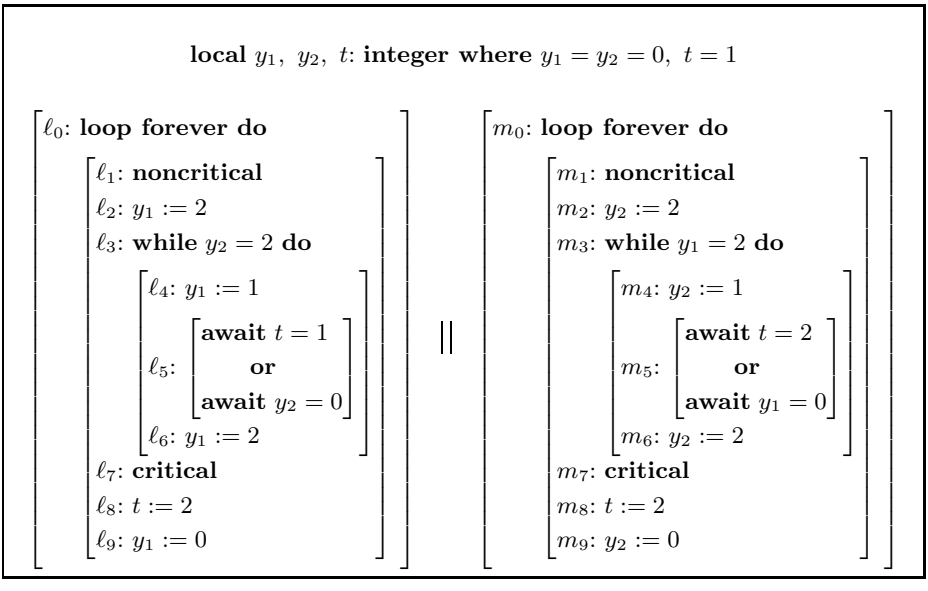


Fig. 50. Program MUX-DEK-A (a variant of Dekker's algorithm)

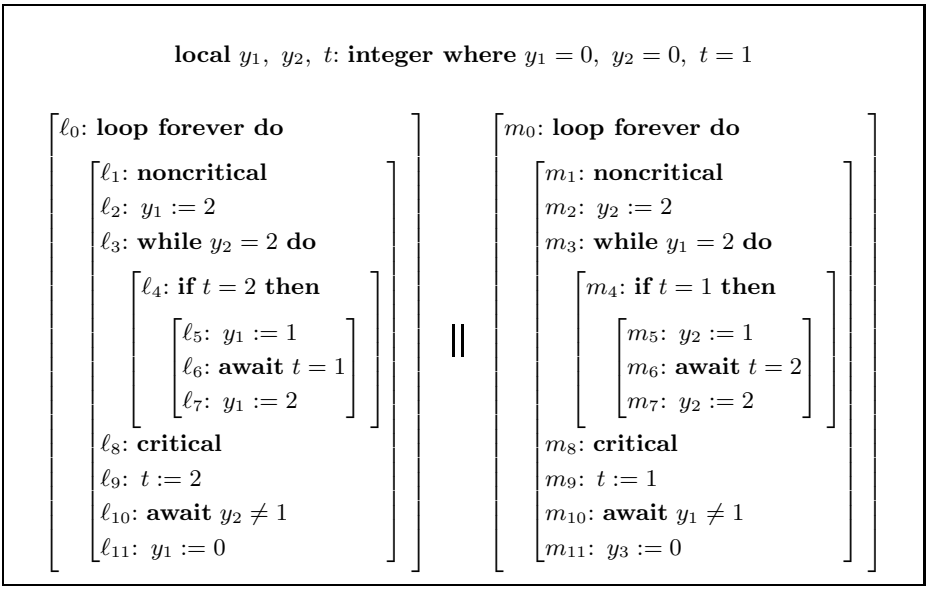


Fig. 51. Program MUX-DEK-B

Problem 4. (condensed form of ranking functions)

In the text, it was suggested that a ranking function $\delta = (d_1, d_2)$, where d_1 and $d_2 < M$ are natural numbers, can always be replaced by the ranking function $\widehat{\delta} = M \cdot d_1 + d_2$. Show that if both $d_2 < M$ and $d_2' < M$, then

$$\widehat{\delta} \;=\; M \cdot d_1 + d_2 > \widehat{\delta'} \;=\; M \cdot d_1' + d_2' \qquad \text{iff} \qquad \delta\colon\ (d_1, d_2) \;\succ\; \delta'\colon\ (d_1', d_2').$$

Problem 5. (rule with relaxed premise JW2)

Consider a version of rule WELL-J in which premise JW2 has been replaced by the weaker premise

$$\widehat{\text{JW2}}.\ \rho_\tau \;\wedge\; \varphi_i \;\;\rightarrow\;\; q' \;\vee\; \bigvee_{j=1}^{k} (\varphi_j' \;\wedge\; \delta_i \succeq \delta_j'),$$

where $\delta_i \succeq \delta_j'$ stands for $(\delta_i \succ \delta_j') \vee (\delta_i = \delta_j')$. This premise requires that either q is achieved by τ, or the rank does not increase and *some* assertion φ_j (not necessarily φ_i) holds after the transition.

Show that the resulting rule is unsound. That is, show a property that satisfies premises JW1, $\widehat{\text{JW2}}$, JW3, and JW4, over a given program, and yet is invalid. This will show that persistence of helpful transitions is essential.

Problem 6. (integer division)

Program IDIV of Fig. 52 accepts two positive integers in variables x and y and places in variable z their integer quotient $x\ div\ y$, and in variable w the remainder of their division $x\ mod\ y$. Prove total correctness of program IDIV, which can be specified by the response formula

$$\Theta \;\Rightarrow\qquad (at\_\ell_5 \;\wedge\; x = z \cdot y + w \;\wedge\; 0 \le w < y).$$

This formula states that every computation of program IDIV terminates (i.e., reaches the terminal location ℓ_5) with values of z, w satisfying $x = z \cdot y + w$ and $0 \le w < y$.

Problem 7. (greatest common divisor)

Program GCDM of Fig. 53 accepts two positive integers in variables x_1 and x_2. It computes in z the greatest common divisor (gcd) of x_1 and x_2, and in variables w_1 and w_2 two integers which express z as a linear combination of the inputs x_1 and x_2. Prove total correctness of program GCDM, which can be stated by the response formula

$$\Theta \;\Rightarrow\qquad \big(at\_\ell_6 \;\wedge\; z = gcd(x_1, x_2) \;\wedge\; z = w_1 \cdot x_1 + w_2 \cdot x_2\big).$$

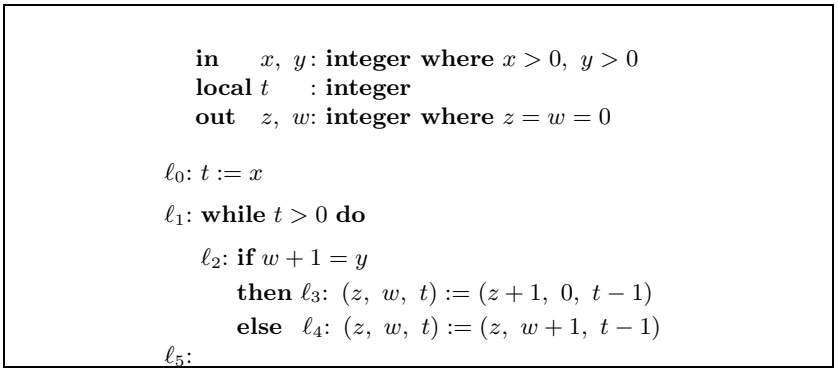


Fig. 52. Program IDIV (integer division)

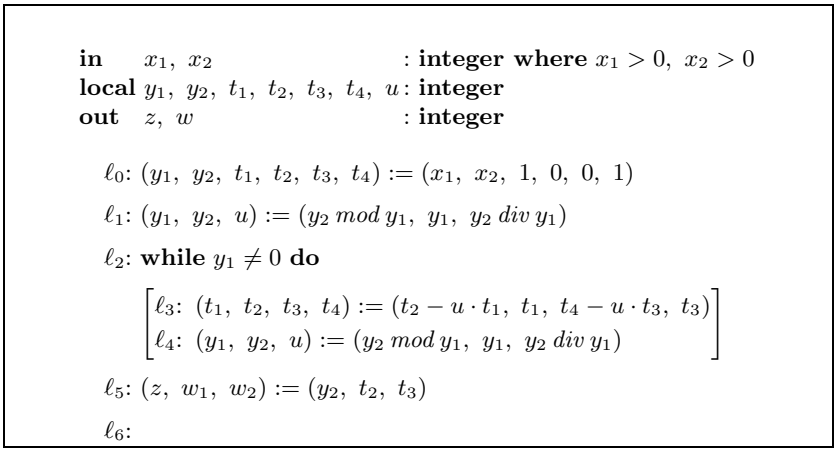


Fig. 53. Program GCDM (greatest common divisor with multipliers)

The program uses the operation *div* of integer division and the operation *mod* which computes the remainder of an integer division. In your proof you may use the following properties of the *gcd* function which hold for all nonzero integers m and n (possibly negative):

$$gcd(m,n) \;=\; gcd(m-n,n) \quad \text{for every } m \neq n$$
$$gcd(m,m) \;=\; |m|.$$

Problem 8. (computing the *gcd* and *lcm*)

Program GCDLCM of Fig. 54 accepts two positive integers in variables x_1 and x_2. It computes in variable z their greatest common divisor and in variable w their

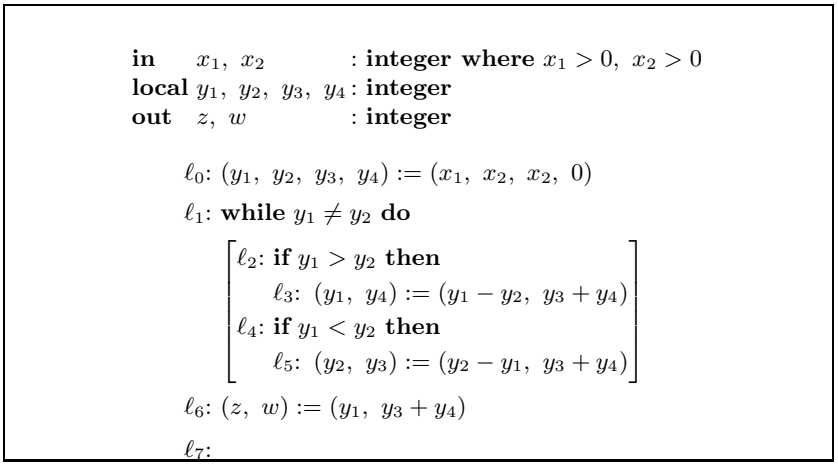


Fig. 54. Program GCDLCM (computing the *gcd* and *lcm*)

least common multiple. Prove total correctness of program GCDLCM, which can be stated by the response formula

$$\Theta \ \Rightarrow \quad \big(at\_\ell_7 \ \wedge \ z = gcd(x_1, x_2) \ \wedge \ w = lcm(x_1, x_2)\big).$$

In your proof you may use the properties of the *gcd* function listed in **Problem 7**, and the following property of the *lcm* function:

$$lcm(m, n) \ = \ m \cdot n / gcd(m, n).$$

Problem 9. (set partitioning)

Consider program EXCH presented in Fig. 55. The program accepts as input two sets of natural numbers S and T, whose initial values are S_0 and T_0, respectively.

Process P_1 repeatedly identifies and removes the maximal element in S and sends it to P_2 which places it in T. Symmetrically, P_2 identifies and removes the minimal element in T and sends it to P_1 which places it in S. The processes use the operations $\max(S)$ and $\min(T)$ which find, respectively, the maximal element in the set S and the minimal element in the set T. Show total correctness of program EXCH, which can be specified by the response formula

$$\Theta \ \Rightarrow \quad (at\_\ell_9 \ \wedge \ at\_m_7 \ \wedge \ |S| = |S_0| \ \wedge \ |T| = |T_0| \ \wedge \ S \leq T).$$

This formula states that the program terminates and on termination, sets S and T have preserved their initial sizes and that every element in S is smaller than or equal to every element in T.

Problem 10. (converse of Claim 8)

Let $P :: [P_1 :: S_1 \| \cdots \| P_k :: S_k]$ be a program whose top-level processes communicate by shared variables and such that every program variable is owned by one

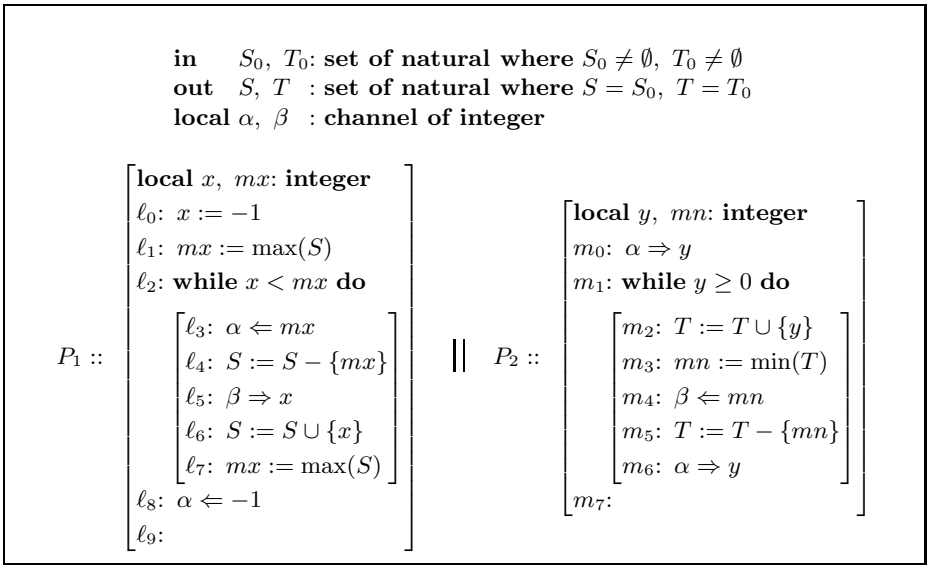


Fig. 55. Program EXCH (partitioning two sets)

of the top-level processes. Let σ be a model such that the interpretation of π is a subset of the locations of P and σ is simultaneously a modular computation of every P_i, $i = 1, \ldots, k$. Show that σ is a computation of P.

Problem 11. (completeness of rule COMP-B)

Let $P :: [P_1 :: S_1 \| \cdots \| P_k :: S_k]$ be a program whose top-level processes communicate by shared variables and such that every program variable is owned by one of the top-level processes. Let φ be a P-valid formula. Show that the P-validity of φ can be compositionally inferred from modular validities, using rule COMP-B and temporal reasoning. This establishes the completeness of rule COMP-B for compositional verification.

A solution to this problem can be organized as follows.

- For each top-level process P_i, $i = 1, \ldots, k$, construct a formula ψ_i capturing the temporal modular semantics of P_i. That is, a model σ satisfies ψ_i iff σ is a modular computation of P_i. Argue semantically that
  $P_i \ \stackrel{m}{=}\ \psi_i$,
  for each $i = 1, \ldots, k$.
- Use rule COMP-B and temporal reasoning to infer
  $P \quad \psi_1 \ \wedge \ \cdots \ \wedge \ \psi_k$.
- Argue semantically that a model σ satisfies $\psi_1 \wedge \cdots \wedge \psi_k$ iff σ is a computation of the entire program P. Therefore, if φ is P-valid, then the following general validity holds:
  $\psi_1 \ \wedge \ \cdots \ \wedge \ \psi_k \ \rightarrow \ \varphi$.
- Apply temporal reasoning to $P \quad \psi_1 \wedge \cdots \wedge \psi_k$ and $\quad \psi_1 \wedge \cdots \wedge \psi_k \rightarrow \varphi$ to infer
  $P \quad \varphi$.

The Arrow of Time through the Lens of Computing*

Krishna V. Palem

Kenneth and Audrey Kennedy Professor of Computing
Rice University
Houston, Texas, USA
Director, NTU-Rice Institute on Sustainable and Applied Infodynamics
Nanyang Technological University
Singapore
palem@rice.edu

Abstract. The concepts of temporal logic were introduced by Amir Pnueli [1] into the realm of computer science in general and programs in particular, to great effect. Given a program specification, a crucial element of reasoning through temporal logic is our ability to assert that one program event occurs *before* or *after* the other, with an order intuitively rooted in our notion of "time". In the realm of temporal logic, such assertions are abstracted as pure mathematical facts. An alternative is to consider the physical realization by executing the specified program through, for example, a microprocessor-based system. In such a case, a mechanism is used to ensure that the desired temporal relationships from the program specification are obeyed, and quite often this mechanism takes the form of a clock. In physical instantiations however clocks and similar mechanisms have an associated energy cost. They are guided by the laws of physics in general and thermodynamics in particular, with which the arrow of time and the associated *irreversibility* are intimately intertwined. Viewed through this lens, a key question arises of whether *the need for ensuring that the temporal norms needed for program correctness accrue an inevitable energy cost.* In this paper, I sketch some of the intricacies underlying this question. I will hint at the subtle interactions between models of computing, time as it is represented in them, and the associated thermodynamic cost. In his early work, Amir relied as much on the philosophy of reasoning about time [2–4] as on the technical intricacies of mathematical logic. In recognition of the richness of his intellectual endeavor, I have developed this exposition in a philosophical style mimicking that of the ancient greek philosopher Zeno [5, 6].

* This work was supported in part by a visit to the Nanyang Technological University as a Canon Visiting Professor, and to the California Institute of Technology as a Moore Distinguished Scholar, 2006-07.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 362–369, 2010.

1 And The Race Begins...

It is dawn and the fabled runner Achilles and the tortoise[1] are discussing the terms of a daylong race.

Tortoise. O great Achilles, you run faster than the wind. How am I to know how many laps you have finished for I, a slow tortoise, will not be there to see you cross the finish line of a lap?

Achilles. But Tortoise, I am honest. I will tell you what the count is!

Tortoise. Indeed Achilles, your reputation for honesty is great. But given your speed, how will you keep track of the count? Won't the effort of counting be costly?

Achilles. That is easy tortoise. I will simply remember the count.

Tortoise. O Achilles, indeed you will know the count but how will I know it? If at noon, I know how may laps you have run and I know where I stand in the race, then I may decide to run some more or to take a nap.

Achilles. Tortoise, you are making this difficult. How can I give you my count without waiting for you? If I must wait for you I can no longer run like the wind.

Tortoise. Agreed great sir. It would slow you down if you had to give me the count all the time, but you are also known for your cleverness. Can you not think of another way?

Achilles. If I cannot wait for you and you cannot see me *each time* I finish a lap, we need help from someone who has the power to see me effortlessly and tell you each time I finish a lap.

Tortoise. Indeed, that seems necessary. But you are so fast. Who in all the world will be able to keep up with you?

Achilles. Ah! Therein lies the answer. When no one in this world can help, we must look to the heavens.

Achilles prays to the gods.

Achilles. O gods in heaven, who among you can watch me run and let the tortoise know how many laps I have completed?

Gods. Achilles, you run faster than the wind. Zeus is the only god who can help you.

Achilles. O great Zeus, I am planning a race with the tortoise and need your help. I run so fast. How can the tortoise know how many laps I have completed. You are omnipotent. Can you help us?

Since Achilles is one of his favorites, Zeus acquiesces.

Zeus. Achilles, you are the greatest runner in the world. It is foolish for the tortoise to race you. What chance does the tortoise have? Why are you wasting *time*?

Achilles. Thank you great Zeus for your kindness. I have never raced the tortoise. How can I know who will win?

Zeus. Very well, why don't you simply remember how many laps you have completed and tell the tortoise?

[1] The plot is an adaptation of one of Zeno's paradoxes[5, 6].

Achilles. I proposed this to the tortoise great Zeus, but the tortoise is clever and said: "O Achilles, indeed you know what the count is, but how will I know? If at noon, I know how may laps you have run, I can decide whether to run some more or to take a nap."

Zeus. What a clever tortoise Achilles. There must be a simple solution where you can let the tortoise know how many laps you have run without waiting.

Achilles. It is for this that I pray for your help O great Zeus.

Zeus. Achilles, do you need me, a god, to help with this? Can you not let the tortoise know through an earthly solution?

Zeus turns to his wise daughter and advisor Athena.

Zeus. Dear daughter Athena, Achilles and the tortoise are planning a race and the tortoise wants Achilles to keep track of the number of laps that he has run and to tell him the number at noon. Achilles cannot do this without waiting for the tortoise. Yet Achilles runs faster than the wind. He needs someone to keep track of him.

Athena. O Father! This seems a petty problem. It can easily be resolved on Earth. Why are you bothered with this?

Zeus. As none of the beings on Earth can keep pace with Achilles, he has come to me. But alas, I am very busy in a feud with Poseidon. I cannot devote any *effort* to this. Yet Achilles is one of my favorites and therefore, I would like your help in finding a solution.

At this point, Achilles wonders why Zeus is being so difficult and thinking of ways out that even the tortoise can foil!

Achilles. (to the tortoise) Tortoise, I have gone all the way to heaven. There seems to be no way of letting you know how many laps I have completed. Is there a way of my simply keeping track of this in my mind and letting you know my mind?

Tortoise. Ah, I have been thinking about this O Achilles. There is a problem even if I can read your mind.

Achilles. What is the problem Tortoise?

Tortoise. How will I know–because of how fast you are running–that you will take the time to make certain of writing down the number of rounds in order?

Achilles. What do you mean?

Tortoise. When you are thinking that you have completed three laps, how will I know you have completed one lap, two laps and then three laps? Maybe you will think of the number 3 before you think of the number 2?

Achilles. Ah, that is easy. Each time I finish a lap, I will take a *pebble* in my mind and move it to another location.

Tortoise. Oh Achilles, who am I to fault your keen thinking? But how do you know that you will move the pebble only *after* you have completed the lap and *not before*? If I read your mind before you had completed the lap, and yet you had already moved the pebble, the count would be wrong.

Achilles. Tortoise, you are more clever than I thought. That is easy. All I need to do is to set another pebble to move the first pebble.

Tortoise. Oh Achilles, you are truly cunning. I am only a simple tortoise, and when I read your mind, I might get confused by the pebbles. How will I know the difference between the pebble that tells you what the count is and the pebble that allows you to move the previous pebble? I am getting confused even with the idea.

Achilles. Easily solved Tortoise. I will move red pebbles to tell me that I have finished a lap and blue pebbles to tell me when to move the red pebbles.

Tortoise. Well Achilles, how am I to make sure you will set the blue pebbles at the right instant?

Achilles. This can be solved by setting a yellow pebble each time I have to set a blue pebble, but....

The tortoise smiles at Achilles.

Meanwhile, in heaven...

Zeus. O Athena, poor Achilles is in trouble. The tortoise is running circles around him.

Athena. Yes Zeus, if Achilles continues, all the colors in earth and heaven will not be enough to help him. Maybe we can give him a hint of a solution?

Zeus. (to Achilles) Achilles, the tortoise is correct. You will not have enough colors in earth and heaven if you continue with this. We in heaven think you can try something simple. How about reusing the colored pebbles by using each color to tell you when you are ready to move the other?

An excited Achilles runs to find the tortoise and explained the new solution from Zeus.

Tortoise. That sounds wonderful Achilles. Let me think about it after I finish my nap.

The tortoise and Achilles bump into each other in a couple of days. An impatient Achilles wants to start the race the next morning.

Tortoise. Achilles, the great god Zeus had a truly interesting idea. I am only a poor mortal tortoise. It is my ignorance that is causing me confusion. Please forgive me if I am delaying things due to my own weakness.

Achilles. Tortoise, what is the problem? I thought that we solved the problem with the help of the great god Zeus.

Tortoise. O Achilles, I must plod through at my own pace. You say that a red pebble will be used to let you know that you have crossed a lap. Next, you say a blue pebble will be used to let you know when the red pebble is active. Suppose we have a red pebble and a blue pebble. Now, each time I use a red pebble to mark a lap as completed. Is that correct?

Achilles. I am glad you understand so well Tortoise. You are not plodding at all. In fact, you understood the idea immediately.

Tortoise. I am honored that you think so Achilles. Next, by using the cleverness of reusing colors, you say we can use a red pebble a second time, is that correct?

Achilles. Absolutely correct Tortoise. You are quick.

Tortoise. Thank you again O Achilles. I was merely following what you told me. Unfortunately, now I am stuck.

Achilles. Tortoise, you have been so quick so far. How may I help you? I am ready to start our race.

Tortoise. If the red pebble is reused to activate a blue pebble, how do you know that the same red pebble, which must be active already, does not allow you to claim that you have completed a lap? I, a mere tortoise, cannot tell the difference between an active red pebble that allows you to mark a new lap as being completed on the one hand, and the red pebble that is used to activate a blue pebble on the other?

Achilles. Tortoise, your confusion is giving me a headache. Are you saying I cannot distinguish the purpose of a red pebble between its two uses; on the one hand the direct use of allowing me to mark that I have completed a lap and on the other the need to activate a blue pebble by reusing it, as the great Zeus suggested?

Tortoise. That is indeed my confusion O Achilles!

Achilles. Tortoise, this is proving to be harder than running the race. Let me consult Zeus again.

Achilles prays to Zeus again and asks for his help. Zeus is awakened from his siesta and hurls a thunderbolt to express his displeasure.

Achilles. (to Zeus) O great Zeus, I regret interrupting you. We have been foiled by the tortoise again. It looks as though I still need far too many colors again. Please help me find a way to inform the tortoise of my progress, so I may prove once and for all that I am the fastest in the world.

Zeus. Achilles, I am divine. Even if I want to help you, how can I communicate this with a mere tortoise?

Achilles. O Zeus, thank you. The thunderbolt you hurled was seen and heard all over the world. Could you not simply hurl one each time I complete a lap? I am sure the tortoise can both see the lightning and hear the thunder.

Zeus approaches Athena one more time.

Zeus. (to Athena) The tortoise is indeed clever. It seems as though we are back where we started. Alas, I might have to get involved for Achilles' sake.

Athena. Achilles is so fast that as omnipotent as you are, hurling thunderbolts will be a distraction. How about making sure that he knows your effort is going to cost him? This might make him run a little more slowly.

Zeus. Agreed Athena. What do you suggest?

Athena. Why don't you charge Achilles a gold coin for each lap where he would like a thunderbolt to be thrown? He is so fast, he can slow down and still beat the tortoise, but he will think twice about running so fast. That should limit your distraction from the contest with Poseidon.

Zeus. Athena, that is a good idea but Achilles, the great runner that he is, is not rich. We might be giving him too much of a handicap, even in a contest against a tortoise. Perhaps we can help him a little more.

Athena. Well Zeus, since Achilles indeed runs faster than the wind, let me suggest something else. Since each lap costs Achilles a gold coin if he runs in the forward direction, let us allow him to reclaim his coins by running backward.

Zeus. That is an interesting idea. Can you elaborate O wise Athena?

Athena. I suggest charging Achilles a gold coin for each lap he begins in the usual (forward direction), and charging a silver each time he chooses to start a lap backwards. Now, each time you charge Achilles a silver coin, return his gold coin if you still have one and when you charge him a gold coin, give his silver one back.

Zeus. We need to let Achilles know that silver is as dear as gold, and it is the number of coins that matters.

Athena. Then, I think the simplest will be to hurl the thunderbolt whenever Achilles has reclaimed all of his coins!

And thus started the race between Achilles and the tortoise and one wonders what the outcome was!

2 Personal Remarks

In wrestling with the question of a contribution that is worthy of Amir's memory and one that has a connection to his thinking and work, I ended up developing a topic that has its roots in a conversation I had with him in 2005 when we met for lunch at the Dan David hotel in Tel Aviv. But a brief digression first. I met Amir in 1990 while we shared an office as we were both visiting Stanford University at the same time. We would walk down together to have lunch and thus started our interaction and discussion on the myriad ways in which time manifests itself in the sciences. In keeping with our style of interaction since, these discussions continued as friendly arguments both in Rehovot and in New York and led to two co-authored papers [7, 8]. The core of our arguments revolved around my push to be vigilant about the *cost* incurred in enforcing specified temporal constraints, so that the resulting program execution is correct. Our early discussions and work used traditional time-complexity as the cost-measure. Returning to that afternoon in Tel Aviv, our discussion was wide-ranging: spanning the time spent by his parents in China enroute to Israel, the importance of how theoretical methods have to be rooted in practice, and finally, the physical versus the logical characterization of time. Since I was very much in the thick of working on the energy cost and advantages associated with probabilistic or randomized approaches to hardware design [9, 10] (since anointed with the "daunting moniker" [11] probabilistic CMOS or PCMOS), our discussions naturally took us to the arrow of time [12, 13] and some of my earlier encounters with what I considered the definitive work on this topic by Prigogine [14, 15] (see Dieter Zeh for a comprehensive exposition [16]).

Since then, the question remained on the proverbial backburner till 2006-7 when I started my visit to Caltech, where hardware and the associated cost of enforcing temporal order have been a topic of enquiry with a sustained history [17]. I thus returned to the question and started thinking of it in hardware terms, notably, reviewing the (energy) cost associated with mechanisms that enforce synchrony through implementations of sequencing mechanisms such as a clock. As I began thinking of this a little more, it seemed that the hardware insights had, perhaps not surprisingly in retrospect, an essential connection to the

thermodynamic question of symmetry breaking. Specifically, it looked as though the logical structure of Prigogine's argument [14] applied more broadly and is related to enforcing temporal relationships germane to program specifications as well.

At this point, the line of enquiry branched into two paths. One of these involved understanding whether some form of symmetry breaking is manifest in ensuring program order when we consider more abstract models of computing like Turing machines, which are meant to be divorced from the detailed realities of hardware design. A reason for considering this direction is to see how these concepts may depend on the specifics of hardware design methods, as opposed to models of computing meant to capture the essence of hardware more abstractly; in this context, suitable adaptations of models based on communication complexity [18] and the framework of zero-knowledge-proofs [19] (or ZKP) seem to be well-suited.

A second path naturally led to the relationship between all of the above issues and the thermodynamic cost of computing associated with *erasure* following Landauer [20], and the reversible approaches, such as Fredkin gates [22], Bennett's reversible models [21] and the work of Fredkin-Toffoli [22] (see Leff and Rex [23] for an impressive compendium of work in this area). The results of these excursions are captured through the anecdotal development in this paper,[2] which I dedicate to Amir Pneuli, a computer scientist of historic stature, a philosopher, and someone I am truly fortunate to have known as a friend.

Acknowledgements. The author wishes to thank his advisees Kirthi Krishna Muntimadugu, Scott Novich, and Jade Boyd, Rice University's Science Editor for helpful editorial support.

References

1. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pp. 46–67 (1977)
2. Prior, A.: Time and modality. Oxford University Press, US (1957)
3. Prior, A.: The syntax of time-distinctions. Franciscan studies 18(2), 105–120 (1958)
4. Rescher, N., Urquhart, A.: Temporal logic. Springer, New York (1971)
5. Huggett, N.: Zenoś paradoxes. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy, Fall edn. (2008)
6. Hodgson, S.H.: The achilles and tortoise: A dialogue. Mind 5(19), 386–390 (1880)
7. Leung, A., Palem, K.V., Pnueli, A.: TimeC: A time constraint language for ILP processor compilation. Constraints 7(2), 75–115 (2002)
8. Leung, A., Palem, K.V., Pnueli, A.: Scheduling time-constrained instructions on pipelined processors. ACM Trans. Program. Lang. Syst. 23(1), 73–103 (2001)
9. Palem, K.V.: Computational proof as experiment: Probabilistic algorithms from a thermodynamic perspective. In: Proceedings of The Intl. Symposium on Veri cation (Theory and Practice), pp. 524–547 (2003)

[2] A technically complete development is the subject of a forthcoming technical report [24].

10. Palem, K.V.: Energy aware computing through probabilistic switching: A study of limits. IEEE Trans. Comput. 54(9), 1123–1137 (2005)
11. Jonietz, E.: Probabilistic chips. In: Special Report: 10 Emerging Technologies 2008, Technology Review, Published by MIT (2008)
12. Eddington, A.: The Nature of the Physical World: Gifford Lectures, 1927. The Macmillan Company, Basingstoke (1929)
13. Reichenbach, H., Reichenbach, M.: The direction of time. Univ of California Pr., Berkeley (1991)
14. Prigogine, I.: Dynamical roots of time symmetry breaking. Philosophical Transactions: Mathematical, Physical and Engineering Sciences 360(1792), 299–301 (2002)
15. Prigogine, I.: Irreversibility as a symmetry-breaking process(thermodynamics). Nature 246, 67–71 (1973)
16. Zeh, H.D.: The physical basis of the direction of time. Springer, Berlin (1989)
17. Davis, A., Nowick, S.M.: An introduction to asynchronous circuit design. Computer Science Department, University of Utah, UUCS-97-013 (1997)
18. Yao, A.C.C.: Some complexity questions related to distributive computing (preliminary report). In: STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing, pp. 209–213 (1979)
19. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing, pp. 291–304 (1985)
20. Landauer, R.: Irreversibility and heat generation in the computing process. IBM J. Res. Dev. 5(3), 183–191 (1961)
21. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. 17(6), 525–532 (1973)
22. Fredkin, E., Toffoli, T.: Conservative logic. International Journal of Theoretical Physics 21(3), 219–253 (1982)
23. Leff, H., Rex, A.: Maxwell's Demon 2: entropy, classical and quantum information, computing. Inst. of Physics Pub. Inc. (2003)
24. Palem, K.V.: The arrow of time through the lens of programming. Department of Computer Science, Rice University, TR10-09 (2010)

What Is in a Step: New Perspectives on a Classical Question*

Willem-Paul de Roever[1], Gerald Lüttgen[2], and Michael Mendler[2]

[1] Institute of Computer Science and Applied Mathematics,
Christian-Albrechts-University of Kiel, Germany
wpr@informatik.uni-kiel.de
[2] Software Technologies and Informatics Theory Research Groups,
Otto-Friedrich-University of Bamberg, Germany
gerald.luettgen@swt-bamberg.de, michael.mendler@uni-bamberg.de

Abstract. In their seminal 1991 paper "What is in a Step: On the Semantics of Statecharts", Pnueli and Shalev showed how, in the presence of global consistency and while observing causality, the synchronous language Statecharts can be given coinciding operational and declarative step semantics. Over the past decade, this semantics has been supplemented with order-theoretic, denotational, axiomatic and game-theoretic characterisations, thus revealing itself as a rather canonical interpretation of the synchrony hypothesis.

In this paper, we survey these characterisations and use them to emphasise the close but not widely known relations of Statecharts to the synchronous language Esterel and to the field of logic programming. Additionally, we highlight some early reminiscences on Amir Pnueli's contributions to characterise the semantics of Statecharts.

1 Introduction

One of the many contributions of Amir Pnueli to the field of Computer Science is in the semantics of *Statecharts* [27, 28, 25, 51, 15]. Statecharts is a popular language for specifying and developing reactive systems, which was invented by Harel in the early 1980s [22]. It extends Mealy machines with concepts of (i) hierarchy, so that a state may have sub-states; (ii) concurrency, thus allowing states to have simultaneously active sub-states that may communicate via the broadcasting of events; (iii) priority, such that one may express that certain events have priority over others. The novelty at the time was that Statecharts is a visual and executable language that it easily understood by software and systems engineers. It is one of the earliest examples of its kind that embraces model-driven software engineering [23]. Within a decade of its inception, already two dozen variants of Statecharts existed [5]. Some of today's widely used variants are the original *STATEMATE* [25], Matlab/Simulink's *Stateflow* [47], and the UML state-machine dialect *Rhapsody* [24].

* The initial ideas leading to this paper were developed by the authors during the Dagstuhl Seminar 09481 (Synchron 2009) in November 2009.

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 370–399, 2010.

Towards a semantics. Statecharts belongs to the family of *synchronous* languages which also includes, e.g., Esterel, Signal, Lustre and Argos [6]. Their semantics is based on a cycle-based reaction whereby the events input by the system's environment are sampled first and potentially cause the firing of transitions that may produce new events. The generated events are output to the environment when the reaction cycle ends. The *synchrony hypothesis* [7], which is adopted by all synchronous languages, ensures that this potentially complex, non-atomic reaction is bundled into an atomic *step*. The hypothesis is justified in practice by the fact that reactions can typically be computed much quicker than it takes for new events to arrive from the system's environment.

Obviously, this concept of a step-based reaction still offers several choices as to what exactly constitutes a step [31, 32, 44, 16, 17]. One important choice is whether generated events may be sensed only in the next step, or already in the current step and may thus trigger the firing of further transitions. The first option was adopted by Harel et al. in the "official" but non-compositional semantics of Statecharts as is implemented in STATEMATE [25, 26, 29]. STATEMATE steps are typically run to completion via its so-called super-step semantics for which Damm, Josko, Hungar and Pnueli have later proposed a compositional variant [15] that supports modular system verification [8].

The second option was investigated by Harel, Pnueli, Schmidt and Sherman [28], where a step involves a causal chain of firing transitions. Here, a transition fires if the positive events in its trigger are present (i.e., if they are offered by the system environment or have been generated by a transition that has fired previously in the same step) and if its negative trigger events are absent (i.e., if they are not present). When it fires, the transition may, as part of its action, broadcast new events which, by the principle of *causality*, may trigger further transitions. Only when this chain reaction of firing transitions comes to a halt is a step complete and becomes, according to the synchrony hypothesis, an atomic entity. Unsurprisingly, the semantics of [28] is not compositional since bundling transitions into an atomic step implies forgetting about the transitions' causal justification [32]. This shortcoming has later been remedied in a fully-abstract fashion by Huizing, Gerth and de Roever [33]. In addition, the semantics of [28] is not *globally consistent* as it permits an event to be both present and absent within a step: an event that occurs negatively in the trigger of one firing transition may be generated by a transition that fires later within the same step.

Pnueli and Shalev's contribution. In the words of Pnueli and Shalev, "a proven sign of healthy and robust understanding of the meaning of a programming or a specification language is the possession of both an operational and declarative semantics, which are consistent with one another" [51]. They showed in their seminal paper (cf. Sec. 2) that adding global consistency is the key to achieving this ambitious goal for Statecharts, and this meant to move away from the semantics of [28]. Their operational semantics for Statecharts relies on an iterative *fixed point construction* over a non-monotonic enabledness function for transitions. This construction ensures causality but involves backtracking as soon as

a global inconsistency is encountered; in the extreme, this may imply that a Statechart does not possess any step under the considered input by the environment. Pnueli and Shalev's declarative semantics for Statecharts then identifies the desired fixed points of the enabledness function via a notion of *separability*.

Levi later developed a compositional proof system for the Pnueli-Shalev step semantics in terms of the modal μ-calculus [36] which facilitates the modular verification of Statecharts. A variant of Pnueli-Shalev semantics which disables transitions that may introduce global inconsistency was presented by Maggiolo-Schettini, Peron and Tini in [43]. This semantics was also used in an early axiomatisation of Statecharts by Hooman, Ramesh and de Roever [30].

This paper. Whereas, to the best of our knowledge, the Pnueli-Shalev step semantics has never been implemented in a Statecharts tool, its mathematical elegance has attracted attention by the concurrency theory community. Over the past decade, the semantics has been supplemented with order-theoretic, denotational, algebraic and game-theoretic perspectives, thus further testifying to its robustness (cf. Sec. 3). The *order-theoretic semantics* of Levi [36] fixes the lacking compositionality of the Pnueli-Shalev step semantics by encoding the causality relation between a step's firing transitions via an irreflexive ordering relation. The *denotational semantics* [41] also addresses the compositionality problem and does so in a fully-abstract way. It is based on an intuitionistic logic interpretation of steps, which appreciates the possibility that an event may neither be present nor absent in a step, but that it may be introduced by the system's environment in the middle of a step. The *algebraic semantics* [40] characterises this fully-abstract denotational semantics in terms of equations, thus leading to a step algebra. Finally, the *game-theoretic semantics* [1] interprets Pnueli-Shalev steps via winning strategies in a 2-player maze game.

Other than revealing the Pnueli-Shalev semantics as a rather canonical interpretation of Statecharts steps, the characterisations mentioned above have opened the door for a mathematically exact comparison of Statecharts steps to *Esterel reactions* and for relating Statecharts to *logic programming* (cf. Sec. 4). Esterel is another popular synchronous language that was devised by Berry independently of, but at the same time as, Statecharts [52]. Its semantics differs from the one proposed by Pnueli and Shalev only by the interpretation of negative events. While one may speculate in Statecharts for an event to be absent, the absence of events in Esterel must be proved constructively, which is key to the language's determinism [42]. Negation is also widely studied in the field of logic programming where *stable negation* [49] corresponds to Pnueli and Shalev's reading of negative events in the presence of global consistency.

The aims of this paper are to (i) survey these additional perspectives on the Pnueli-Shalev semantics; (ii) highlight the semantic relationship between Statecharts and Esterel, and between Statecharts and logic programming; (iii) discuss Pnueli and Shalev's results in the light of related work. This offers new insights into *the* classical question in the Statecharts literature: *"What is in a step?"* Last, but not least, the first author recalls some reminiscences on Amir Pnueli's contributions to characterise the semantics of Statecharts (cf. Sec. 5).

2 Pnueli and Shalev's Interpretation of Statecharts

This section provides a brief introduction to Statecharts, and recalls the step semantics presented by Pnueli and Shalev in [51].

2.1 Introduction to Statecharts

A Statechart may best be understood as a hierarchical, concurrent Mealy machine, where *basic* states may be hierarchically refined by injecting other Statecharts. This creates composite states of two possible sorts, which are called *and*-states and *or*-states, respectively. Whereas and-states permit parallel decompositions of states, or-states allow for sequential decompositions. Consequently, an and-state is *active* if all its sub-states are active, and an or-state is active if exactly one of its sub-states is. At any given point during execution, the set of active states is referred to as a *configuration*.

A Statecharts step is defined relative to a configuration C and a set E of events that are given to the system by its environment. Key to a step are transitions t, each of which is labelled by two sets of events: a *trigger*, $\mathsf{trg}(t)$, and an *action*, $\mathsf{act}(t)$. The trigger $\mathsf{trg}(t) = P, \overline{N}$ splits into a set of *positive* events $P \subseteq \Pi$ and *negative* events $\overline{N} \subseteq \overline{\Pi}$, taken from a universe Π of events and their negative counterparts in $\overline{\Pi} =_{\mathrm{df}} \{\overline{e} : e \in \Pi\}$, respectively. For convenience, we define $\overline{\overline{e}} =_{\mathrm{df}} e$. Intuitively, t is *enabled* and thus fires if the set $E \subseteq \Pi$ is such that all events of P but none of N are in E, i.e., if $P \subseteq E$ and $N \cap E = \emptyset$. The effect of firing t is the generation of all events in the action of t, where a transition's action $\mathsf{act}(t) \subseteq \Pi$ consists of positive events only.

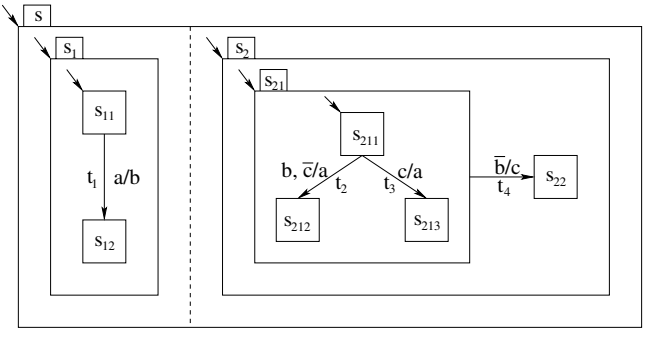


Fig. 1. Example Statechart

We illustrate the Statecharts language by means of a small example. Consider the Statechart depicted in Fig. 1 with and-state s, or-states s_1, s_2 and s_{21}, and *basic*-states s_{11}, s_{12}, s_{211}, s_{212}, s_{213} and s_{22}. Further assume that all components are in their initial states marked by the small unlabelled arrows, so that

the initial configuration C_1 is $\{s, s_1, s_{11}, s_2, s_{21}, s_{211}\}$. If, in this configuration, the environment offers event c, then transitions t_3 and t_4 are enabled. Since they are both placed within the same or-state s_2, only one of them may fire. In a Statecharts dialect that does not give transitions an implicit priority along the state hierarchy, the choice between t_3 and t_4 is *nondeterministic*. Thus, t_3 may fire, generate event a, and change state to s_{213}. Again depending on the Statecharts dialect, this generated event may or may not trigger transition t_1 in the parallel, or orthogonal, state s_1 within the same reaction cycle, i.e., within the same *step*. Hence, the question arises which transitions leaving states in C_1, which we denote by $\mathsf{trans}(C_1)$, may fire together to form a step.

2.2 The Pnueli-Shalev Step Semantics

As stated in the introduction, Pnueli and Shalev defined coinciding operational and declarative semantics of Statecharts configurations in their paper [51]. Given a configuration C, a step in the sense of Pnueli and Shalev comprises a *maximal*, globally consistent and causal, set of transitions in $\mathsf{trans}(C)$, which are mutually *orthogonal*, i.e., "consistent" in Statecharts terminology, and *triggered* by the events offered by the environment or produced by the firing of other transitions in the step.

Transition t is *consistent* with set T of transitions, in signs $t{\in}\mathsf{consistent}(C,T)$, if t is not in the same "parallel component" as any $t' \in T \setminus \{t\}$. Formally,

$$\mathsf{consistent}(C,T) =_{\mathrm{df}} \{t \in \mathsf{trans}(C) \mid \forall t' \in T.\ t \triangle_C t'\}\,,$$

where $t \triangle_C t'$ if (i) $t = t'$ or (ii) t and t' are in different substates of an enclosing and-state. Further, transition t is *triggered* by a set E of events, in signs $t \in \mathsf{triggered}(C,E)$, if the positive but not the negative trigger events of t are in E:

$$\mathsf{triggered}(C,E) =_{\mathrm{df}} \{t \in \mathsf{trans}(C) \mid \mathsf{trg}(t) \cap \mathit{\Pi} \subseteq E,\ \overline{(\mathsf{trg}(t) \cap \overline{\mathit{\Pi}})} \cap E = \emptyset\}\,.$$

Finally, transition t is *enabled* in C with respect to set E of events and set T of transitions, if $t \in \mathsf{enabled}(C,E,T)$ where

$$\mathsf{enabled}(C,E,T) =_{\mathrm{df}} \mathsf{consistent}(C,T) \cap \mathsf{triggered}(C, E \cup \bigcup_{t \in T} \mathsf{act}(t))\,.$$

Assuming the transitions in T are known to fire, $\mathsf{enabled}(C,E,T)$ determines the set of all transitions of C that are enabled by the environment events in E and, since generated events are sensed within the same step, the actions of T. In the following, we write $\mathsf{act}(T)$ for the actions $\bigcup_{t \in T} \mathsf{act}(t)$.

Operational semantics. Using this enabledness function $\mathsf{enabled}$, we may now present the *step-construction procedure* of [51] which operationally determines Statecharts steps relative to a configuration C and a set E of environment events:

```
procedure step–construction(C, E);
  var T := ∅;
  while T ⊂ enabled(C, E, T) do
    choose t ∈ enabled(C, E, T) \ T;
    T := T ∪ {t}
  od;
  if T = enabled(C, E, T) then return T
  else report failure
end step–construction.
```

This step-construction procedure computes sets T of transitions that can fire together in a step. Returning to our example, i.e., the Statechart depicted in Fig. 1, we have $\mathsf{enabled}(C, \{c\}, \emptyset) = \{t_3, t_4\}$. Therefore, *step–construction*$(C_1, \{c\})$ may choose transition t_3 in its first iteration and, since $\mathsf{enabled}(C, \{c\}, \{t_3\}) = \{t_1, t_3\}$, transition t_1 in its second iteration, before reaching a fixed point and returning $\{t_1, t_3\}$. Due to the presence of statement choose, the procedure may introduce nondeterminism. Indeed, *step–construction*$(C_1, \{c\})$ may also return $\{t_4\}$ when choosing t_4 instead of t_3 in the first iteration.

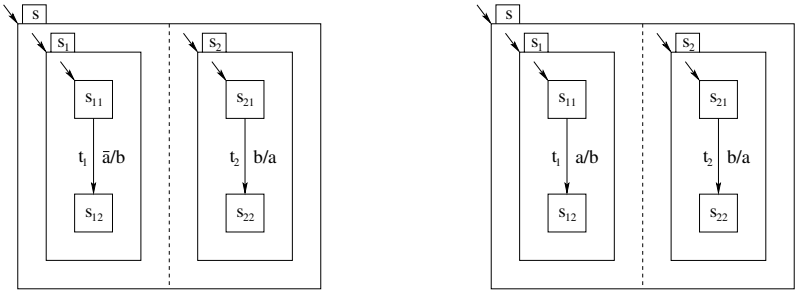


Fig. 2. Further example Statecharts

When the procedure reports a failure as the result of detecting an inconsistency, i.e., there exists a $t \in T$ such that $t \notin \mathsf{enabled}(C, E, T)$, it may backtrack and possibly make a different choice at statement choose. In particular, the procedure may only report failures and not produce any step. To see this, consider the Statechart shown on the left in Fig. 2 in its initial configuration C_2, and assume the empty environment. In its first iteration, *step–construction*$(C_2, \emptyset)$ picks the only enabled transition t_1, and then the other transition t_2 in the second iteration. At this point $T = \{t_1, t_2\}$ but $\mathsf{enabled}(C_2, \emptyset, T) = \{t_2\}$, and a failure is reported. No backtracking is possible since there have not been any nontrivial choice points along the computation. Hence, the step-construction procedure does not produce any step. This situation is to be distinguished from an empty response $T = \emptyset$, which is exhibited by the Statechart depicted on the

right in Fig. 2 in its initial configuration C_3 and for the empty environment, since $\mathsf{enabled}(C_3, \emptyset, \emptyset) = \emptyset$.

Following Pnueli and Shalev's terminology, a set T of transitions is called *constructible* for a given configuration C and a set E of environment events, if it can be obtained as a result of successfully executing procedure *step-construction.* For each constructible set T, set $A =_{\mathrm{df}} E \cup \mathsf{act}(T) \subseteq \Pi$ is called the *(step) response* of C for E.

Declarative semantics. Pnueli and Shalev also provided an equivalent *declarative* definition of their operational step semantics. Given a configuration C and a set E of environment events, a set T of transitions is called *separable* for C and E if there exists a proper subset $T' \subsetneq T$ such that $\mathsf{enabled}(C, E, T') \cap (T \setminus T') = \emptyset$. Further, T is *admissible* for C and E if T is inseparable for C and E and $T = \mathsf{enabled}(C, E, T)$. This declarative semantics is thus a fixed-point semantics. Observe that, in the absence of negative events, function $\mathsf{enabled}(C, E, \cdot)$ is monotonic, and then the uniquely defined inseparable fixed point coincides with the least fixed point. However, since function $\mathsf{enabled}(C, E, \cdot)$ is in general non-monotonic when transitions with negative trigger events are involved, a unique least fixed point may not exist. In this case, the notion of inseparability chooses distinguished fixed points that reflect causality. Indeed, a separable set of transitions points to a break in the causality chain when firing these transitions.

We illustrate the notion of inseparability by returning to the above examples. For the Statechart depicted in Fig. 1, $\{a, b, c\}$ is a step response of initial configuration C_1 for environment $E =_{\mathrm{df}} \{c\}$. Firstly, as seen above, $T =_{\mathrm{df}} \{t_1, t_3\}$ is a fixed point of $\mathsf{enabled}(C_1, E, T)$. Secondly, it is inseparable for C_1 and E since, for $T' = \emptyset$, we have $\mathsf{enabled}(C_1, E, T') \cap (T \setminus T') = \{t_3, t_4\} \cap T = \{t_3\} \neq \emptyset$, and similarly for the other proper subsets $T' \subsetneq T$. For the initial configurations C_2 and C_3 of the two Statecharts of Fig. 2, $\{a, b\}$ is not a step response for the empty environment. For the Statechart on the left in the figure, $T =_{\mathrm{df}} \{t_1, t_2\}$ is not a fixed point of function $\mathsf{enabled}$ since $\mathsf{enabled}(C_2, \emptyset, T) = \{t_2\} \neq T$. For the Statechart on the right, T is separable; consider $T' = \emptyset \subsetneq T$, for which $\mathsf{enabled}(C_3, \emptyset, T') \cap (T \setminus T') = \emptyset \cap T = \emptyset$.

Main result. We can now state the main result of Pnueli and Shalev's paper [51]:

Theorem 1 (Pnueli and Shalev). *For all configurations C and event sets $E \subseteq \Pi$, a set T of transitions is admissible for C and E if and only if T is constructible for C and E.*

Such a theorem can also be proved for the step semantics of Maggiolo-Schettini, Peron and Tini [43]. Their semantics uses a modified function $\mathsf{enabled}(C, E, T)$ in which a transition t is *not* enabled if its firing would generate an event whose absence is assumed in T, i.e., if $\mathsf{act}(t) \cap \bigcup_{t' \in T} \overline{\mathsf{trg}(t')} \neq \emptyset$. For example, the Statechart in Fig. 2 on the left, which did not have any response for $E = \emptyset$, now has response $\{b\}$. This response is obtained by t_1 firing on the basis of a being absent which then immediately disables t_2. One shows that $\{t_1\}$ is an inseparable with $\mathsf{enabled}(C_2, \emptyset, \{t_1\}) = \{t_1\}$.

3 Developments and New Perspectives

This section surveys four characterisations of the Pnueli-Shalev step semantics which have been developed within the past decade: an *order-theoretic* semantics that encodes causality via an irreflexive ordering relation [36]; a *denotational* semantics that is based on intuitionistic logic [41]; an *algebraic* semantics that specialises axioms of this logic to Statecharts steps [40]; and a *game-theoretic* semantics [1]. In contrast to Pnueli and Shalev's operational and declarative semantics, all four semantics presented here are compositional, and the denotational and algebraic semantics are fully-abstract.

3.1 Configuration Syntax

This paper focuses on the semantics of single Statecharts steps, since the semantics across steps is clear and well understood. It will therefore be convenient to reduce the Statecharts notation to the bare essentials and identify a Statecharts configuration with its set of leaving transitions, to which we — by abuse of terminology — also refer as *configuration*. We formalise configurations using the following, simple syntax, where $I \subseteq \Pi \cup \overline{\Pi}$ and $A \subseteq \Pi$:

$$C \quad ::= \quad 0 \mid I/A \mid C \| C \; .$$

Intuitively, 0 stands for the configuration with the empty behaviour. Configuration I/A encodes a transition t with $\mathsf{trg}(t) = I$ and $\mathsf{act}(t) = A$. When triggered, transition t fires and generates the events in A. Transitions I/A with empty trigger, i.e., $I = \emptyset$, are simply written as A below. If we wish to emphasise that trigger I consists of the positive events $P \subseteq \Pi$ and the negative events $\overline{N} \subseteq \overline{\Pi}$, i.e., $I = P \cup \overline{N}$, then we denote transition I/A by $P, \overline{N}/A$. Finally, configuration $C_1 \| C_2$ describes the parallel composition of configurations C_1 and C_2. Observe that 0 coincides semantically with a transition with empty action; nevertheless, it seems natural to include 0. Using this syntax, we may encode the initial configuration C_1 of our example Statechart of Fig. 1 as

$$a/b \parallel b, \overline{c}, \overline{e_3}, \overline{e_4}/a, e_2 \parallel c, \overline{e_2}, \overline{e_4}/a, e_3 \parallel \overline{b}, \overline{e_2}, \overline{e_3}/c, e_4 \; .$$

Here, the e_i are distinguished events not occurring in the triggers or actions of the Statechart's transitions. These events allow us to encode *nondeterministic choice*, including *state hierarchy*, via parallel composition and event negation, although one can do without them as is shown in [41]. Assuming the environment injects event c, Pnueli and Shalev's step-construction procedure may first fire transition t_3 and then t_1 within a single step from configuration C_1, thereby reaching configuration $\{s, s_1, s_{12}, s_2, s_{21}, s_{213}\}$. This latter configuration is represented by $0 \parallel \overline{b}/c$ in our syntax.

For simplicity, the following exposition focuses on Statecharts configurations with respect to the empty environment only. This is not a restriction, however, since considering the steps of a configuration C relative to a set $E \subseteq \Pi$ of environment events is equivalent to considering the steps of the configuration $C \| E$ relative to the empty set of environment events.

3.2 Order-Theoretic Perspective

The first results for turning Pnueli and Shalev's step construction into a compositional semantics for Statecharts were obtained by Uselton and Smolka [56], Levi [36], and Maggiolo-Schettini, Peron and Tini [43, 44]. They observed that Statecharts may be viewed as process terms in the style of process algebra, whose semantics is given by a compositional translation into labelled transition systems. Each transition represents a step of a configuration decorated with an action label specifying the synchronous interaction with the environment. It turned out that for this structured operational semantics to work, labels must be order-relational structures as opposed to simple first-order events, in order to encode sufficient causal information. In this section we recall the basic elements of this order-theoretic approach, following essentially the exposition of Levi in [36], albeit in a simplified form.

The set of *(causality) labels* $\Sigma(\Pi)$, or *basic actions* in the terminology of Levi, is the set of pairs $(\ell, \prec)$. Here, $\ell \subseteq \Pi \cup \overline{\Pi}$ is a *consistent* subset of positive or negative events, i.e., $\ell \cap \overline{\ell} = \emptyset$, and $A \prec B$ is an irreflexive and transitive causality ordering on subsets $A, B \subseteq \ell$, where $B = \emptyset$ or $B = \{b\}$ for $b \in \Pi$. Irreflexivity means that $A \prec \{b\}$ implies $b \notin A$, and transitivity requires that, if $A \prec \{b\}$ and $b \in B \prec C$, then $(B \setminus \{b\}) \cup A \prec C$.

Causality labels represent globally consistent and causally closed interactions that are composed from Statecharts transitions. Specifically, every transition $t \in \mathsf{trans}(C)$ leaving a configuration C induces a causality label $lab(t) =_{\mathrm{df}} (\ell_t, \prec_t)$, where $\ell_t =_{\mathrm{df}} \mathsf{trg}(t) \cup \mathsf{act}(t)$ and $\prec_t =_{\mathrm{df}} \{\mathsf{trg}(t) \prec_t \{e'\} : e' \in \mathsf{act}(t)\}$. It is assumed without loss of generality that transitions are nontrivial in the sense that $\mathsf{trg}(t) \cap \mathsf{act}(t) = \emptyset$ and, for no $e \in \Pi$, both $e, \overline{e} \in \mathsf{trg}(t) \cup \mathsf{act}(t)$. Then, ℓ_t is consistent, $\prec_t$ is irreflexive and, trivially, transitive. For instance, the transitions $t_1 =_{\mathrm{df}} a/b$ and $t_2 =_{\mathrm{df}} b, \overline{c}/d$ correspond to the labels $\sigma_i =_{\mathrm{df}} lab(t_i) = (\ell_i, \prec_i)$ with $\ell_1 = \{a, b\}$, $\{a\} \prec_1 \{b\}$, and $\ell_2 = \{b, \overline{c}, d\}$ with $\{b, \overline{c}\} \prec_2 \{d\}$. The joint execution of t_1 and t_2 would be the label $\sigma_3 =_{\mathrm{df}} (\ell_3, \prec_3)$ such that $\ell_3 = \{a, b, \overline{c}, d\}$ with causalities $\{a\} \prec_3 \{b\}$, $\{b, \overline{c}\} \prec_3 \{d\}$ and $\{a, \overline{c}\} \prec_3 \{d\}$. Here, the last pair arises from the combined reaction of t_1 triggering t_2; its presence is enforced by transitivity of $\prec_3$.

As causality labels are compositional generalisations of transitions, each $\sigma = (\ell, \prec) \in \Sigma(\Pi)$ has an associated set of *trigger* and *action events*, viz., $\mathsf{trg}(\sigma) =_{\mathrm{df}} \{e \in \ell \mid \neg \exists C \subseteq \ell.\, C \prec \{e\}\}$ and $\mathsf{act}(\sigma) =_{\mathrm{df}} \ell \setminus \mathsf{trg}(\sigma)$. Thus, a transition $t = I/A$ has $\mathsf{trg}(lab(t)) = I$ and $\mathsf{act}(lab(t)) = A$ as expected. For the label σ_3 from above, we get $\mathsf{trg}(\sigma_3) = \{a, \overline{c}\}$ and $\mathsf{act}(\sigma_3) = \{b, d\}$, which are the same trigger and action as in $lab(t_4)$, for $t_4 =_{\mathrm{df}} a, \overline{c}/b, d$. However, the latter does not express the causality contained in σ_3, viz., that event b is a consequence of a alone, while d depends on both a and $\overline{c}$. It is this extra causality information which makes labels compositional: σ_3 is the combined execution of t_1 and t_2 as opposed to t_4 which is a single atomic transiton.

Labels, like transition sets, can be enabled or disabled by the environment. A consistent $\ell \subseteq \Pi \cup \overline{\Pi}$ *enables* an action σ if $\mathsf{trg}(\sigma) \cap \Pi \subseteq \ell$ and $\overline{\mathsf{trg}(\sigma)} \cap \ell = \emptyset$. It *disables* σ if $\overline{\mathsf{trg}(\sigma)} \cap \ell \neq \emptyset$. For consistent and *complete* (or *binary*) ℓ, i.e.,

$\Pi \subseteq \ell \cup \overline{\ell}$, both notions are complementary. Note that if $\emptyset$ enables σ then no trigger is needed to execute σ.

Next we define the operation of parallel composition between causality labels $\sigma_1 = (\ell_1, \prec_1)$ and $\sigma_2 = (\ell_2, \prec_2)$ to form the full causal and concurrent closure of all interactions coded in two orderings. Due to nondeterminism, the composition $\sigma_1 \times \sigma_2$ does not yield a single causality label but rather a set of them. They are obtained as the maximal irreflexive and transitive sub-orderings of the transitive closure $(\prec_1 \cup \prec_2)^+$. Here, the transitive closure of $\prec_1 \cup \prec_2$ is the smallest relation $\prec$ with $\prec_1 \cup \prec_2 \subseteq \prec$ such that, if $A \prec \{b\}$ and $b \in B \prec C$, then $(B \setminus \{b\}) \cup A \prec C$. Now, $(\ell, \prec) \in \sigma_1 \times \sigma_2$ if (i) $\ell = \ell_1 \cup \ell_2$, (ii) $(\ell, \prec)$ is a causality label, and (iii) $\prec$ is maximal in $(\prec_1 \cup \prec_2)^+$.

For example, we have $lab(t_1) \times lab(t_2) = \{\sigma_3\}$, where t_1, t_2 and σ_3 are as before, which confirms formally that σ_3 is the composition of t_1 and t_2. Note that Cond. (ii) implies that $\ell_1 \cup \ell_2$ must be consistent. Hence, $lab(\overline{a}/b) \times lab(b/a) = \emptyset$ which reflects the fact that both transitions can never be part of the same step due to global consistency. Cond. (iii) resolves cyclic dependencies: Consider actions $lab(a/b) = (\{a, b\}, \prec_1)$, $lab(b/c) = (\{b, c\}, \prec_2)$ and $lab(c/a) = (\{c, a\}, \prec_3)$, which are consistent but their combined transitive closure $(\prec_1 \cup \prec_2 \cup \prec_3)^+$ has reflexive cycles $\{e\} \prec \{e\}$, for $e \in \{a, b, c\}$. The maximal irreflexive and transitive sub-orderings are given by $\sigma_4 =_{\text{df}} (\ell, \{a\} \prec \{b\} \prec \{c\}, \{a\} \prec \{c\})$, $\sigma_5 =_{\text{df}} (\ell, \{b\} \prec \{c\} \prec \{a\}, \{b\} \prec \{a\})$, $\sigma_6 =_{\text{df}} (\ell, \{c\} \prec \{b\} \prec \{a\}, \{c\} \prec \{a\})$, where $\ell = \{a, b, c\}$. Then, $lab(a/b) \times lab(b/c) \times lab(c/a) = \{\sigma_4, \sigma_5, \sigma_6\}$ which describes the three ways in which transitions a/b, b/c and c/a can partake in the same step. They show that the environment needs to provide at least one of the triggers $\mathsf{trg}(\sigma_4) = \{a\}$, $\mathsf{trg}(\sigma_5) = \{b\}$ or $\mathsf{trg}(\sigma_6) = \{c\}$ to generate the combined action $\mathsf{act}(\sigma_4) = \{b, c\}$, $\mathsf{act}(\sigma_5) = \{c, a\}$ or $\mathsf{act}(\sigma_6) = \{a, b\}$, respectively.

We can now define the initial causality labels of a configuration C presented as a one-step reaction relation $C \mapsto \sigma$, for $\sigma \in \Sigma(\Pi)$, by induction on C:

- $0 \mapsto (\ell, \emptyset)$ for all binary $\ell \subseteq \Pi \cup \overline{\Pi}$;
- $t \mapsto lab(t)$, and $t \mapsto (\ell, \emptyset)$ for all binary $\ell \subseteq \Pi \cup \overline{\Pi}$ which disable $lab(t)$.
- $C_1 \mapsto \sigma_1$ and $C_2 \mapsto \sigma_2$ implies $C_1 \parallel C_2 \mapsto \sigma$ for all $\sigma \in \sigma_1 \times \sigma_2$.

Observe that transitions not only generate *active* steps $t \mapsto lab(t)$ but also *passive*, or *idle*, steps $t \mapsto (\ell, \emptyset)$ with $\mathsf{trg}(lab(t)) \cap \ell \neq \emptyset$ in which they are disabled. This resolves conflicting choices and introduces internal nondeterminism. For example, although $lab(\overline{a}/b) \times lab(\overline{b}/a) = \emptyset$, there are active and passive steps $\overline{a}/b \mapsto lab(\overline{a}/b)$ and $\overline{b}/a \mapsto (\{\overline{a}, b\}, \emptyset)$, respectively, which combine $lab(\overline{a}/b) \times (\{\overline{a}, b\}, \emptyset) = \{lab(\overline{a}/b)\}$. Symmetrically, there is a passive step $\overline{a}/b \mapsto (\{a, \overline{b}\}, \emptyset)$ and active step $\overline{b}/a \mapsto lab(\overline{b}/a)$ giving $(\{a, \overline{b}\}, \emptyset) \times lab(\overline{b}/a) = \{lab(\overline{b}/a)\}$.

The following theorem is a key result of Levi [36]:

Theorem 2 (Correctness and Completeness). *If C is a configuration and $A \subseteq \Pi$, then A is a Pnueli-Shalev step response of C if and only if there exists a causality label σ with $C \mapsto \sigma$ such that $\emptyset$ enables σ and $A = \mathsf{act}(\sigma)$.*

Levi defines the labelled transition system across all steps $C_i \overset{\sigma;\kappa}{\mapsto} C_{i+1}$ of a Statechart, compositionally in the full syntax including choice and hierarchy.

The additional flag $\kappa \in \{\overline{\varepsilon}, \varepsilon\}$ in Levi's label indicates if the step is idle or non-idle. These flags are needed for compositionality with respect to choice, which is not part of the syntax considered here. Without the flag both 0 and $a/\emptyset$, say, would have the same initial labels, viz. $(\{a\}, \emptyset)$ and $(\{\overline{a}\}, \emptyset)$, and thus be semantically indistinguishable. However, they induce different behaviour in the context $(\cdot + \emptyset/b) \,\|\, \emptyset/a$: The configuration $(0 + \emptyset/b) \,\|\, \emptyset/a$ must always produce events $\{a, b\}$, whereas in $(a/\emptyset + \emptyset/b) \,\|\, \emptyset/a$ the transition $\emptyset/b$ can be preempted by $a/\emptyset$ making a step on its own triggered by the parallel transition $\emptyset/a$ in the context. Hence, $(a/\emptyset + \emptyset/b) \,\|\, \emptyset/a$ not only has the response $\{a, b\}$ but also $\{a\}$. Levi's flags avoid the confusion between 0 and $a/\emptyset$ since the initial step $0 \mapsto (\{a\}, \emptyset) : \overline{\varepsilon}$ of the former is idle while the intial step $a/\emptyset \mapsto (\{a\}, \emptyset) : \varepsilon$ of the latter is non-idle.

Further, Levi presents a compositional μ-calculus verification system for these labelled transition systems [36]. However, no congruence and full-abstraction results are proven. In the work of Maggiolo-Schettini, Peron, Tini [43, 44], a similar order-theoretic refinement for Pnueli-Shalev semantics, as well as for the modified semantics mentioned in Sec. 2.2, is developed, together with congruence results for several behavioural preorders. It has been shown by Lüttgen, von der Beeck and Cleaveland [38] that the two levels of the order-theoretic semantics, i.e., configurations and causality labels $\Sigma(\Pi)$, can also be flattened into a single labelled transition system with first-order labels in which special clock transitions mark the beginning and end of a step.

3.3 Denotational Perspective

While Pnueli and Shalev's declarative step semantics corresponds to their operational step semantics, it is not denotational because it lacks compositionality as an interaction with the environment is only allowed at the beginning of a step but not during a step. The denotational perspective presented in this section does away with this shortcoming.

Interaction steps. The idea is to read a configuration C of a Statechart as a specification of a set of *interaction steps* between the Statechart and all its possible environments. This set is nonempty since one may always construct an environment that disables those transitions in C that would cause a global inconsistency and, thus, failure in the sense of Pnueli and Shalev. Formally, an interaction step is a monotonic sequence $M = (M_0, M_1, \ldots, M_n)$ of reactions $M_i \subseteq \Pi$, where $M_{i-1} \subsetneq M_i$ for all i. Each reaction contains events representing both environmental input and the Statechart's response. Intuitively, by the requirement for monotonicity, such a sequence extends the communication potential between the Statechart and its environment, until this potential is exhausted.

An interaction step is best understood as a separation of a Pnueli-Shalev step response M_n. Each M_i extends M_{i-1} by new environmental stimuli plus the Statechart's response to these. Here, responses are computed according to Pnueli and Shalev, except that events not contained in M_n are assumed to be absent in M_i. In this way, global consistency is interpreted as a logical specification

of the full interaction step M and not only relative to a single reaction M_i. In other words, each interaction step separates a Pnueli-Shalev step response into causally closed sets of events. Each passage from M_{i-1} to M_i represents a non-causal "step" triggered by the environment. This creates a separation between M_{i-1} and M_i in the spirit of Pnueli and Shalev: as all events generated by the transitions enabled under M_{i-1} are contained in M_{i-1}, their intersection with $M_i \setminus M_{i-1}$ is empty.

Interpreting configurations, logically. Transitions $P, \overline{N}/A$ of the considered configuration C are interpreted on interaction steps $M = (M_0, M_1, \ldots, M_n)$ as follows. For each M_i, either (1) all events in A are also in M_i, or (2a) one or more events in A are not in M_i and $P \not\subseteq M_i$, or (2b) one or more events in A are not in M_i, and some event $e \in N$ is in M_j for some $i \leq j \leq n$. Intuitively, case (1) corresponds to the situation in which the transition is enabled and thus fires, or where the environment ensures that all events of the transition's action are provided. Cases (2a) and (2b) correspond to the situation where the transition is not enabled since not all positive trigger events are present (2a), or not all negative trigger events are absent because they are provided later in the sequence (2b). Case (2b) ensures that, as desired above, global consistency is enforced over the whole interaction step M.

Remarkably, this interpretation corresponds exactly to the one of intuitionistic logic [14] when reading negative events $\overline{e}$ as $\neg e$, and transition slashes "/" as logical implication. The composition of events in triggers or actions, as well as parallel composition "$\|$" on configurations, may simply be understood as conjunction, and our interaction steps M are nothing but linear Kripke structures. This correspondence with propositional intuitionistic logic over linear Kripke structures leads us to a general semantic relation $\models$, namely the logical satisfaction relation. Formally, an interaction step $M = (M_0, M_1, \ldots, M_n)$ *satisfies* configuration C, in signs $M \models C$, if $M, i \models C$ for all $0 \leq i \leq n$, where

$$
\begin{array}{ll}
M, i \models 0 & \textit{always} \\
M, i \models I/A & \textit{if } (I \cap \Pi \subseteq M_i \textit{ and } \overline{(I \cap \overline{\Pi})} \cap M_n = \emptyset) \textit{ implies } A \subseteq M_i \\
M, i \models C_1 \| C_2 & \textit{if } M, i \models C_1 \text{ and } M, i \models C_2\,.
\end{array}
$$

If $M \models C$ we also say that M is an *(interaction) model* of C. The above definition is a shaved version of the standard semantics of propositional intuitionistic logic [14]. Configuration 0 is identified with *true* and, if $I = \emptyset$ for a transition I/A, the semantics of I/A reduces to $A \subseteq M_i$. Now we have $M \models C$ if and only if C is valid in the linear Kripke structure M. Note that for interaction steps of length one, the notions of interaction model and classical model coincide, and we simply write M_1 for (M_1).

Response models. The step responses of a configuration C in the sense of Pnueli and Shalev are now exactly those interaction models M of C of length one, called *response models*, that are *not* suffixes of interaction models $N{=}(N_0, \ldots, N_m, M)$ of C with $m \geq 0$. If such a singleton interaction model was suffix of a longer interaction model, then — according to the argumentation above — the reaction

would be separable and hence not causal. Thus, we have the following theorem which is proved in [41]:

Theorem 3 (Correctness and Completeness). *If C is a configuration and $M \subseteq \Pi$, then M is a Pnueli-Shalev step response of C if and only if M is a response model of C.*

We illustrate our notion of response model by means of a few examples:

- Firstly, consider the configuration $\overline{a}/b$ which exhibits the Pnueli-Shalev step response $\{b\}$ for the empty environment. Indeed, $\{b\}$ is a response model, i.e., a model and not a suffix of a longer interaction model. The only possibility would be the interaction step $(\emptyset, \{b\})$, but this is not an interaction model since $(\emptyset, \{b\}), 0 \not\models \overline{a}/b$: by definition, we have to consider $\emptyset \subseteq \emptyset$ and $\{a\} \cap \{b\} = \emptyset$ implies $\{b\} \subseteq \emptyset$, and this implication is false because $b \notin \emptyset$.
- Secondly, configuration $C_2 =_{\mathrm{df}} \overline{a}/b \,\|\, b/a$ has no response model. Although $\{a, b\}$ is a classical model of C_2, it may be left-extended to the interaction model $(\emptyset, \{a, b\})$. Note in particular that $(\emptyset, \{a, b\}), 0 \models \overline{a}/b$: by definition, we have to consider $\emptyset \subseteq \emptyset$ and $\{a\} \cap \{a, b\} = \emptyset$ implies $\{b\} \subseteq \emptyset$, and this implication trivially holds. In other words, event a is absent at position 0 of the interaction step $(\emptyset, \{a, b\})$ since it is added later in the step, namely at position 1, and thus is *not* absent.
- Thirdly, consider configuration $C_3 =_{\mathrm{df}} a/b \,\|\, b/a$ with its Pnueli-Shalev step response $\emptyset$. It is easy to see that $\emptyset$ is trivially a response model. In contrast, the set $\{a, b\}$ — while being a classical model of C_3 — is not a response model since the suffix extension $(\emptyset, \{a, b\})$ is an interaction model of C_3.
- Fourthly, configuration $\overline{a}/b \,\|\, \overline{b}/a$ offers two response models, namely $\{a\}$ and $\{b\}$, which are exactly the configuration's Pnueli-Shalev step responses. As in the example regarding configuration C_2 above, neither response model can be left-extended to an interaction model of length greater than one.

Full abstraction. The interaction models of a configuration C encode all possible interactions of C with all its environments and nothing more. Firstly, any differences between the interaction models of C are differences in the interactions of C with its environments and thus can be observed. Secondly, any observable difference in the interaction of C with its environments should imply a difference in the interaction models, and this holds by the very construction of interaction models. Therefore, the above interaction step semantics provides the desired compositional and fully abstract semantics for Pnueli-Shalev steps:

Theorem 4 (Compositionality and Full Abstraction). *Let C_1, C_2 be configurations. Then, C_1 and C_2 have the same interaction models if and only if, for all configurations C_3, the parallel configurations $C_1 \| C_3$ and $C_2 \| C_3$ have the same Pnueli-Shalev step responses.*

The proof of this theorem can be found in [41], where interaction steps are called *sequence structures* and where interaction models are referred to as *sequence models*. Most notably, the proof shows that it is sufficient to consider interaction

models of lengths 1 and 2 only. This leads to a strategy for implementation, e.g., via encoding such interaction models using *binary decision diagrams* [13]. Finally, it should be remarked that the denotational approach has been generalised from single-step configurations to a full Statecharts language in [39].

3.4 Algebraic Perspective

We now turn to characterising the Pnueli-Shalev step semantics, or more precisely the largest congruence contained in equality on step responses, in terms of axioms. These are derived from general axioms of propositional intuitionistic formulas over linear Kripke models. Thus, the algebraic characterisation presented here is closely related to the above denotational characterisation.

Table 1. Axiom system for the Pnueli-Shalev step semantics

(A1)	$C_1 \parallel C_2$	$=$	$C_2 \parallel C_1$	
(A2)	$(C_1 \parallel C_2) \parallel C_3$	$=$	$C_1 \parallel (C_2 \parallel C_3)$	
(A3)	$C \parallel C$	$=$	C	
(A4)	$C \parallel 0$	$=$	C	
(B1)	$P, I/P$	$=$	0	
(B2)	$I/A \parallel I/B$	$=$	$I/(A \cup B)$	
(B3)	I/A	$=$	$I/A \parallel I, J/A$	
(B4)	$I/A \parallel A, J/B$	$=$	$I/A \parallel A, J/B \parallel I, J/B$	
(B5)	$P, \overline{N}/A$	$=$	0	*if* $P \cap N \neq \emptyset$
(C1)	$P, \overline{N}/A$	$=$	$P, \overline{N}/A, B$	*if* $N \cap A \neq \emptyset$
(C2)	$P, \overline{N}/A$	$=$	$P, e, \overline{N}/A \parallel P, \overline{N}, \overline{e}/A$	*if* $N \cap A \neq \emptyset$
(C3)	$I, \overline{N}/B \parallel P, \overline{N}/A$	$=$	$\{I, \overline{N}, \overline{e}/B : e \in P\} \parallel P, \overline{N}/A,$	*if* $N \cap A \neq \emptyset$ *and* $P \neq \emptyset$

Our axioms system is displayed in Table 1, where $A, B, N, P \subseteq \Pi$, $I, J \subseteq \Pi \cup \overline{\Pi}$ and $e \in \Pi$, and where C, C_1, C_2, C_3 are configurations. Axioms (A1)–(A4) are fairly natural, and we thus concentrate on explaining the remaining, more interesting axioms. Axiom (B1) describes that, if the firing of a transition merely reproduces in its action some of the events required by its trigger, then we might just as well not fire the transition at all. As a special case, $I/\emptyset = 0$. Axiom (B2) encodes that two transitions with the same trigger will always fire together and produce the events in both of their actions. Axiom (B3) states that, by adding in parallel to a transition I/A a transition $I, J/A$ with the same action A but additional trigger events J, the behaviour remains unchanged. Logically speaking, "guarding" via a trigger is a *weakening* operation.

Axiom (B4) is a version of the *cut* rule known from logic and reflects the chain-reaction character of firing transitions. The left-hand side $I/A \parallel A, J/B$ represents a situation in which there is a transition $A, J/B$ that is waiting, among other preconditions J, for the events in A that will be produced when transition I/A fires. Hence, it is safe to add transition $I, J/B$ to the right-hand side. Axiom (B5) deals with inconsistencies in triggers. If an action A is guarded

by a trigger $P, \overline{N}$ in which some event is required to be both present and absent, i.e., $P \cap N \neq \emptyset$, then this transition will never become enabled and is thus equivalent to 0.

The remaining Axioms (C1)–(C3) are concerned with conflicts between the trigger and action of a transition. They axiomatise the effect of transitions that produce a failure under certain trigger conditions. More precisely, these axioms involve a transition $P, \overline{N}/A$ with $N \cap A \neq \emptyset$, whose firing leads to a global inconsistency. Such a transition rejects the completion of all steps in which its trigger $P, \overline{N}$ is true. Thus, since $P, \overline{N}/A$ can never fire in a consistent way, the step construction cannot terminate in a situation in which trigger $P, \overline{N}$ holds true. In other words, whenever all events in P have become present, the step construction must continue until at least one event in N is present in order to inactivate the transition. If this does not happen, the step construction fails. Axioms (C1)–(C3) formalise three different consequences of this.

Axiom (C1) reflects the fact that, since $P, \overline{N}/A$ can never contribute to a completed step if $N \cap A \neq \emptyset$, we may add arbitrary other events B to its action, without changing its behaviour. Logically, this axiom corresponds to the laws $e \wedge \neg e \equiv \mathit{false}$ and $\mathit{false} \supset B \equiv \mathit{true}$, for any B. Axiom (C2) offers a second way of reading the inconsistency between triggers and actions. Since at completion time any event e is either present or absent, the same rejection that $P, \overline{N}/A$ produces can be achieved by $P, \overline{N}, e/A \parallel P, \overline{N}, \overline{e}/A$. This is because if e is present at completion time, then $P, \overline{N}, e/A$ raises the failure; if e is absent, then $P, \overline{N}, \overline{e}/A$ does the job. This is essentially the law $\neg e \wedge \neg\neg e \equiv \mathit{false}$ in logic. It is important to observe that the side condition $N \cap A \neq \emptyset$ is necessary: For example, $\emptyset/A$ is different from $e/A \parallel \overline{e}/A$ because in a parallel context A/e the latter fails (no step) while the former has the response $A \cup \{e\}$.

Finally, consider Axiom (C3). Instead of saying that $P, \overline{N}/A$ generates a failure if all events in P are present and all events in N are absent, we might say that, if all events in N are absent, then at least one of the events in P must be absent, provided the step under consideration is to be completed without failure. But then any parallel component of the form $I, \overline{N}/B$ can be replaced by the parallel composition $\|\{I, \overline{N}, \overline{e}/B : e \in P\}$. The reason is that, if $I, \overline{N}/B$ fires at all in the presence of transition $P, \overline{N}/A$, then at least one of the weaker transitions $I, \overline{N}, \overline{e}/C$ will be able to fire at some point, depending on which of the events in $P \neq \emptyset$ it is that will be absent to avoid failure. Again there is a logic equivalent for this, namely the law $\neg(p_1 \wedge p_2) \equiv \neg p_1 \vee \neg p_2$ that holds for *linear* Kripke structures. Last, but not least, it is important to note that configuration $P, \overline{N}/A$, for $N \cap A \neq \emptyset$, is not the same as configuration 0, since the former inevitably produces a failure if its trigger is true, while 0 does not respond at all.

Theorem 5 (Correctness and Completeness). *$C_1 = C_2$ can be derived from the axioms of Table 1 via standard equational reasoning if and only if, for all interaction steps M, $M \models C_1$ iff $M \models C_2$.*

A proof of this theorem be found in [40]. In that paper, a more general syntax for configurations has been employed in which transition actions may be arbitrary

configurations. Last, but not least, it should be remarked that Axioms (B3) and (C1)–(C3) are unsound for Maggiolo-Schettini, Peron and Tini's variant of the Pnueli-Shalev step semantics [43].

3.5 Game-Theoretic Perspective

During the 1990s, a promising alternative to the traditional operational and denotational semantics of programming languages emerged. Game-theoretic models, which had long been used in descriptive set theory, economics and engineering control theory, were identified as a surprisingly powerful setting for dealing with system-environment interactions in a compositional fashion. For example, in the semantics of discrete reactive systems, games were applied to capture notions of refinement sensitive to input/output causality [3]. More specifically on the topic of this paper, it has been demonstrated that 2-player positional games provide a natural way of characterising different step semantics in synchronous programming. Game theory handles cyclic causal dependencies of non-monotonic behaviours by accounting for the system and environment dichotomy through the binary polarity of player and opponent. The swapping of roles gives constructive meaning to negation, and different forms of winning conditions generate different response semantics with varying degrees of constructiveness [2, 1].

In the following let us recall the main result from [1] as it applies to the Pnueli-Shalev semantics. To this end we first introduce the notion of a *maze* as the game equivalent of a configuration. A maze is a labelled transition system $M = (\mathsf{S}_\iota, \mathsf{S}_\tau, \xrightarrow{\iota}, \xrightarrow{\tau})$ consisting of disjoint sets of *visible rooms* S_ι and *secret rooms* S_τ, together with accessibility relations $\xrightarrow{\gamma} \subseteq \mathsf{S} \times \mathsf{S}$ between rooms $\mathsf{S} = \mathsf{S}_\iota \cup \mathsf{S}_\tau$ with two possible labels $\gamma \in \{\iota, \tau\}$. The transitions represent valid moves or *corridors*; a transition $m \xrightarrow{\iota} m'$ corresponds to a visible corridor connecting room m with m', whereas $m \xrightarrow{\tau} m'$ is a secret corridor. Designating a room or corridor as secret makes it unobservable, i.e., abstracts from it semantically.

A maze M acts as the game board on which two *players* A and B compete with each other to conquer rooms by taking alternate turns in moving along the corridors. When the play enters a room m in which player A receives the turn, then m becomes part of A's territory. If A now moves to some connected room m' through a visible corridor $m \xrightarrow{\iota} m'$, then A must hand over to B who then plays from m'. On the other hand, if A moves along a secret corridor $m \xrightarrow{\tau} m'$, then A keeps their turn and continues to play from m'. Room m may later be revisited in the play and, depending on who has the turn then, m may either fall to the other player B, or possession of m is perpetuated by A. We assume that the players use positional and consistent strategies. A *strategy* is a function that determines the next move of a player at every stage of a play in which they receive the turn. A strategy is *positional* if the decision only depends on the room from which the move is made, and not on the history of the play. This implies that every time a player receives the turn in a given room, they will take the same corridor out of it. A strategy is called *consistent* if all the positions ever occupied by a player are never lost to the opponent, and also if the player never enters a room left to the opponent. A consistent strategy keeps

player A safely within a region $R_A \subseteq \mathsf{S}$, while at the same time it ensures that the opponent is confined to a region $R_B \subseteq \mathsf{S}$ from which they cannot escape.

In general, the objective of the game is that of defending regions (R_A, R_B), called *front lines*, according to a given *winning condition*. The winning condition that we are interested in here is *reactiveness*. A strategy is *reactive* if the player always eventually hands over to the opponent to make them appear in a visible room or get stuck in a secret room. We say that A *defends* front line (R_A, R_B) if A has a positional and consistent reactive strategy for all plays starting from R_A with A as the first player, and from R_B where B is the first to move. Reactive strategies permit infinite plays but require the player to be reactive in the sense that they are never embarrassed about a move when challenged and always generate a proper response (i.e., hand over to the opponent in a visible room) in finite time, though we do not insist that the player can stop the opponent from ever challenging again. In analogy with evaluation strategies in functional programming such defensible front lines are called *lazy* [1].

With every configuration C we associate a maze M_C such that the events Π correspond to the visible rooms S_ι and transitions $\mathsf{trans}(C)$ to secret rooms S_τ. The two sets (R_A, R_B) of a front line for M_C constitute a possible reactive response of C such that R_A and R_B will contain events that are present and absent, respectively. It turns out that the maximal lazy front lines of M_C are essentially the synchronous step responses of C as conceived by Pnueli and Shalev.

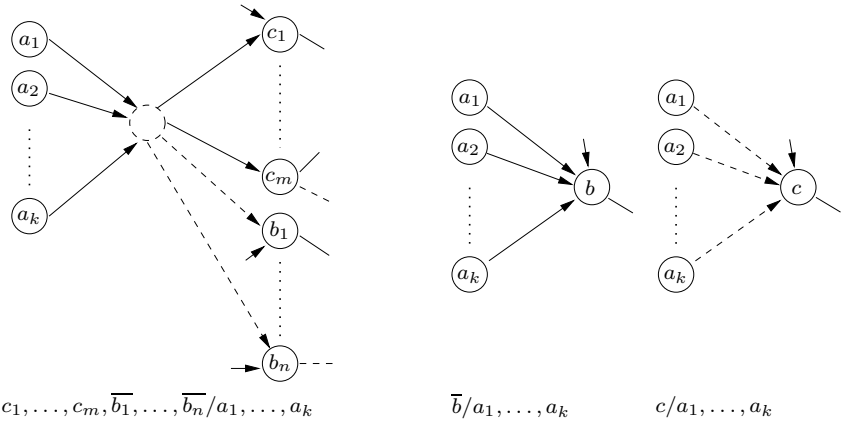


Fig. 3. Coding of transitions

The maze M_C is obtained by observing that a transition $t = P, \overline{N}/A$ of C expresses the fact that action $a \in A$ is caused to be in R_A (i.e., present) if, for all $c \in P$, c is in R_A *and*, for all $\overline{b} \in \overline{N}$, b is in R_B. This conjunction can be modelled canonically by considering the transition t as an intermediate (secret) room and by adding (i) a visible corridor between each $a \in A$ and t; (ii) a visible corridor between t and each $c \in R_A$; and (iii) a secret corridor between t and each $b \in N$ as seen in Fig. 3 on the left. The graphical convention used here is that

visible rooms/corridors are drawn with solid lines and secret rooms/corridors with dashed lines. When all transitions $t \in C$ have been represented in this way, they form a maze M_C of secret rooms connected through events Π as the visible rooms. Clearly, this translation is compositional where $C_1 \| C_2$ is the set-theoretic union of the mazes M_{C_1} and M_{C_2}. Some simplifications are possible, e.g., a transition like $c/a_1, \ldots, a_k$ with only one trigger may be coded without the intermediate room as a bundle of secret corridors from a_i to c. Similarly, a transition $\overline{b}/a_1, \ldots, a_k$ is simply a bunch of visible corridors from a_i to b. This is illustrated in Fig. 3 on the right.

Theorem 6 (Correctness and Completeness). *Let C be a configuration and M_C be the maze associated with C. Then, $A \subseteq \Pi$ is a Pnueli-Shalev step response of C if and only if there exists a lazy front line $(R_A, \mathsf{S} \setminus R_A)$ in M_C such that $A = R_A \cap \Pi$.*

The proof of this theorem can be found in [1]. Note how the game model accommodates both the failure and nondeterminism of step responses. Depending on M_C, it may happen that there is no strategy to avoid a (visible) room m being visited by both players infinitely often. This corresponds to Pnueli and Shalev's step-construction procedure returning a failure. Also, a room m may occur in two different lazy front lines, which yields nondeterministic behaviour.

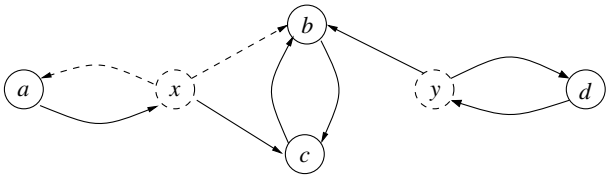


Fig. 4. The maze M_C for component $C = \overline{c}/b \,\|\, \overline{b}/c \,\|\, c, \overline{a}, \overline{b}/a \,\|\, b, d/d$ with maximal lazy front lines $(\{b, x, y\}, \{a, c, d\})$ and $(\{c, y\}, \{b, d\})$

In general, the responses of a configuration C are the maximal lazy front lines in the maze M_C. Every *binary* front line, i.e., a front line (R_A, R_B) with $R_A \cup R_B = \mathsf{S}$, is trivially maximal. Yet, maximal lazy front lines need neither be uniquely defined nor two-valued. For the maze M_C in Fig. 4 we find that there are two maximal lazy front lines $(\{b, x, y\}, \{a, c, d\})$ and $(\{c, y\}, \{b, d\})$. Of those only the former is binary and thus a Pnueli-Shalev step response according to Thm. 6. Consider $R_A = \{b, x, y\}$ and $R_B = \{a, c, d\}$ first. From all rooms in R_A player A has a strategy to make the opponent take the turn in one of the visible rooms R_B. E.g., from x we move secretly to b keeping the turn and then continue visibly to c where the opponent must continue. Also, from R_B the opponent must immediately hand back to A in a room of R_A with the first move. This keeps A consistently in R_A and B in R_B and makes B always eventually take the turn in a visible room. Similarly, one shows that the front line $R_A = \{c, y\}$ and $R_B = \{b, d\}$ is defensible. This, too, is a maximal front line because none

of the remaining rooms a or x can be defended consistently as part of R_A or of R_B. Indeed, the Pnueli-Shalev semantics eliminates this non-binary solution $(\{c, y\}, \{b, d\})$ by backtracking so that configuration C is deterministic.

4 Related Work, Esterel and Logic Programming

This section discusses the Pnueli-Shalev step semantics in the light of the rich volume of related work which was either triggered by or performed orthogonally to the research in Statecharts and its semantics. In particular, this section compares the Pnueli-Shalev step semantics to the constructive semantics of the synchronous language *Esterel* [7, 52], with the aim of highlighting the close semantic relationship between Statecharts and Esterel. We also relate the Pnueli-Shalev step semantics to the so-called stable models of *logic programming* [49], a field in which the interpretation of negation plays a prominent role, too.

4.1 Related Work

Defining a synchronous step response involves the incremental firing of transitions which may both trigger and inhibit each other via broadcasting events. The deterministic stabilisation of this micro-scheduling process is highly non-trivial in the presence of cyclic dependencies and negative trigger conditions. Pnueli and Shalev's approach is one of many conceivable ways of defining a consistent scheduling strategy. We set the scene here with a brief survey of work on synchronous step semantics based on how event absence is treated in the step construction. We do not consider semantics that evade the consistency problem by banning negation such as *Modecharts* [34] or *UML state machines* and their derivatives [55].

The first take on the problem was the view underlying the first formal Statecharts semantics [28]. It does not consider the constraint of global consistency so that the absence of an event remains a local, or a transient, condition which may be overridden within the same step.

The second take is to break causality cycles systematically, for which we can identify two strands. One option is to delay the broadcast of events into the next step, so as to avoid *instantaneous* broadcast. This is the approach adopted in Leveson et al's *RSML* [35] and the step semantics of *STATEMATE* [26]. The other option to break cycles by default may be subsumed under Boussinot's slogan "no instantaneous reaction to [event] absence," according to which a negative trigger event tests for absence in the *previous* step rather than the current one. This interpretation has gained some importance in the synchronous programming community. Examples are Boussinot's *Reactive-C* [9], Boussinot and de Simone's synchronous reactive calculus *SL* [12], Mandel and Pouzet's functional reactive programming language *ReactiveML* [45], and Boussinot and Dabrowski's *FunLoft* [11] which is a globally asynchronous, locally synchronous model of multi-threading. The idea is also applied in logic programming, specifically in Saraswat, Jagadeesan and Gupta's language *tcc* for timed concurrent constraint programming [53].

The third take permits both instantaneous reaction to absence and instantaneous event propagation under the constraint of *global consistency*. All these step semantics construct maximal causally-closed and consistent sets of transitions. There are surprisingly many strategies for doing this which have found their applications. One important split arises in the operational model of step construction from the question of who is responsible for event absence: the system or the environment. The system view of absence underlies the work of Maggiolo-Schettini, Peron and Tini [43] and of Lüttgen, von der Beeck and Cleaveland [37], as discussed above. Dual to this view is the environment view in which absence is defined externally and thus is not determined until the step is complete and closed off against the environment. This is logically the most tricky scenario as it involves constructive anticipation and forces one to deal with non-causal programs, i.e., potential failure and deadlock behaviour. A rather useful systematics for this class of semantics has been introduced by Boussinot in his *Sugarcubes* report [10]. Boussinot's classification is based on a *potential function* π which is used at stage i of the step construction to speculate about which events $\pi(i) \subseteq \pi$ may potentially be broadcast later. By complement, all other events $\Pi \setminus \pi(i)$ are deemed absent at stage i. If $\sigma(i)$ is the set of events that have been broadcast by stage i, then a transition t is triggered if $\mathsf{trg}(t) \cap \Pi \subseteq \sigma(i)$ and $\overline{\mathsf{trg}(t)} \cap \pi(i) = \emptyset$. If π is *correct* — i.e., it contains all events that are eventually broadcast: $\sigma(j) \subseteq \pi(i)$ for $i \leq j$ —, then one does not get an inconsistency failure and does not need to backtrack. Boussinot shows that every correct potential function leads to a deterministic but possibly deadlocking step [10].

The most prominent representatives in this category are Pnueli and Shalev's semantics [51], Philipps and Scholz' μCharts variation [50] of it, and Berry's constructive semantics for Esterel [52] discussed in detail below, which corresponds to a correct potential function. The Pnueli-Shalev step semantics is obtained for the trivial potential function $\pi(i) = \sigma(i)$, which permits full speculation so that all events not currently broadcast can be taken to trigger absences. Such π is not correct and, consequently, one has failure and nondeterminism. However, the scheduling cannot deadlock.

All approaches to global consistency reported so far are *operational*, in the sense that they can be implemented by some form of scheduling. This is different from the *logical* approach described in Sec. 3.3, which employs intuitionistic logic for interpreting negative events. Of course, we can also apply classical logic; a configuration C then induces Boolean equations over events which describe the necessary and sufficient conditions for each event to be present in a step of C. Each classical solution is called a *logically coherent* step. A program is *logically correct* if all its configurations have exactly one logically coherent step under every input stimulus. This is the *logical behavioural semantics* [52] that is applied in the visual language *Argos* [46], one of the early synchronous languages developed by Maraninchi around 1991. Argos is well-known for the invention of a fully semantical and thus compositional version of inter-level transitions. There are, of course, many other truth-value interpretations of configurations such as (a) the Kleene-style ternary interpretation [19] which is related to Esterel's

constructive semantics discussed in Sec. 4.2, and (b) the various models of normal logic programming mentioned in Sec. 4.3.

4.2 Relation to Esterel

Esterel is a textual, imperative language for specifying the behaviour of reactive systems, which has been developed by Berry and colleagues since the early 1980s [7, 52], concurrently to and independently of Harel's Statecharts. A visual version of Esterel is André's *SyncCharts* [4] which is implemented as *Safe State Machines* in the embedded-software development tool *SCADE* [18]. Similar to Statecharts, Esterel provides primitives for decomposing reactions sequentially and concurrently, where concurrent reactions may involve a complex exchange of events. In Esterel terminology, one speaks of the *emission of signals* rather than the generation of events or the firing of transitions.

Like the semantics of Statecharts, the semantics of Esterel is designed around the concept of a step, called an *instant*, and it also supports the principles of synchrony and causality. Unlike the Pnueli-Shalev semantics of Statecharts, however, those Esterel programs for which the step construction does not complete, are rejected by the Esterel compiler. As a further distinction from Statecharts steps, Esterel instants are guaranteed to be deterministic. Esterel's semantics has significantly evolved over the years. In [52], Berry describes a much improved version that is founded on the idea of *constructiveness* and that encodes the principle of causality in a precise way, and not in an approximative way as earlier Esterel versions did. He also establishes the coincidence of three constructive styles of Esterel semantics — a *behavioural* semantics, an *operational* semantics, and a *circuit* semantics —, thereby testifying to the mathematical elegance and robustness of Esterel.

Table 2. The *Must* and *Cannot* functions for computing Esterel instants

$$
\begin{array}{|rl|}
\hline
\mathit{Must}(0,S) & =_{\mathrm{df}} \emptyset \\
\mathit{Must}(I/A,S) & =_{\mathrm{df}} \begin{cases} A & \text{if } I \subseteq S \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{Must}(C_1 \| C_2, S) & =_{\mathrm{df}} \mathit{Must}(C_1,S) \cup \mathit{Must}(C_2,S) \\
\hline
\mathit{Cannot}(0,S) & =_{\mathrm{df}} \overline{\Pi} \\
\mathit{Cannot}(I/A,S) & =_{\mathrm{df}} \begin{cases} \overline{\Pi \setminus A} & \text{if } I \cap \overline{S} = \emptyset \\ \overline{\Pi} & \text{otherwise} \end{cases} \\
\mathit{Cannot}(C_1 \| C_2, S) & =_{\mathrm{df}} \mathit{Cannot}(C_1,S) \cap \mathit{Cannot}(C_2,S) \\
\hline
\end{array}
$$

The behavioural semantics of Esterel is declarative and based on computing the fixed point of a reaction function that is the analogue of Pnueli and Shalev's *enabled* function. As for Statecharts events, Esterel signals may be present or absent. While the presence of signals in Esterel is always derived from *emit*

statements explicitly contained in the program text, the absence of a signal is inferred indirectly from the absence of emit statements. Esterel's reaction function collects all those signals e as being present that *must* be emitted under the assumption that certain signals are asserted by the system environment or emitted earlier within the instant. However, in addition and unlike Pnueli and Shalev's *enabled* function, Esterel's reaction function also records signals e that *cannot* be emitted as being absent. Hence, both the presence and the absence of signals must be shown constructively in Esterel; in contrast to the Pnueli-Shalev step semantics, the absence of a signal is not inferred by speculation.

To be more precise, we define the semantics of Esterel instants for our configuration syntax [52]. As indicated above, Esterel's reaction function operates on sets $S \subseteq \Pi \cup \overline{\Pi}$ coding explicit presence and absence statuses of signals. These are determined by two functions, $\mathit{Must}(C, \cdot)$ and $\mathit{Cannot}(C, \cdot)$, each of which takes a set of consistent signal statuses and returns a set of positive or negative signal statuses, respectively, for a given configuration C. The formal definition of both functions is displayed in Table 2. Here, a set $S \subseteq \Pi \cup \overline{\Pi}$ is called consistent, if S does not contain both $e, \overline{e}$ for any $e \in \Pi$. The Esterel reaction function $\mathit{esterel}(C, \cdot)$ for configuration C is now defined as $\mathit{esterel}(C, S) =_{\mathrm{df}} \mathit{Must}(C, S) \cup \mathit{Cannot}(C, S)$, which is monotonic in S and preserves consistency. The Esterel semantics of C is then the least fixed point of $\mathit{esterel}(C, \cdot)$.

As an example, consider the configuration $C =_{\mathrm{df}} \overline{a}/b \,||\, \overline{b}/a$. According to Esterel's semantics, the absence of neither signal a nor b can be inferred since either signal may potentially be emitted; formally, $\mathit{Must}(C, \emptyset) = \mathit{Cannot}(C, \emptyset) = \emptyset$ and indeed $\emptyset$ is the least fixed point of $\mathit{esterel}(C, \cdot)$. Hence, the Esterel compiler cannot determine the status of signals a and b and thus rejects C as not being causal. In contrast, Pnueli and Shalev's step-construction procedure may initially assume that a is absent, or alternatively that b is absent, and thus infer two possible steps: step $\{b\}$ in the former case and step $\{a\}$ in the latter case. As suggested by this example, it is the constructive treatment of negation in Esterel that ensures the determinism of Esterel instants. Technically, this constructiveness ensures that Esterel's reaction function $\mathit{esterel}(C, \cdot)$ is monotonic, which is not the case for Pnueli and Shalev's *enabled* function. Thus, the least fixed point of $\mathit{esterel}(C, \cdot)$ is guaranteed to exist. The following theorem, which is proved in [42], relates the *least* fixed point property of Esterel to inseparability in the sense of Pnueli and Shalev. Recall here that inseparability reflects causality, i.e., a separable set of signal statuses points to a break in the causality chain when emitting signals.

Theorem 7 (Inseparability in Esterel). *Let C be a configuration. Then, S is the least fixed point of $\mathit{esterel}(C, \cdot)$ if and only if S is a fixed point of $\mathit{esterel}(C, \cdot)$ and inseparable for C.*

Esterel programs describing an instant may also be given a denotational semantics in terms of response models similar to Sec. 3.3, since Esterel instants are constructive in the sense of intuitionistic logics. This can be achieved by reading the behavioural Esterel reaction function as a formula in intuitionistic logic over signal statuses. Details of this denotational, model-theoretic approach can be

found in [42]. In a similar spirit can the game-theoretic approach to the Pnueli-Shalev step semantics presented in Sec. 3.5 be adapted to Esterel instants, as is shown in [2, 1]. Formally, it can be proved that every constructive Esterel instant of a configuration C is also a Pnueli-Shalev step response of C. Interestingly, the configuration of Fig. 4 that we have found to only have a single Pnueli-Shalev step response is non-constructive under Esterel's semantics. This means that Pnueli-Shalev steps are more liberal than Esterel instants even on deterministic behaviours.

4.3 Relation to Logic Programming

The simplest declarative view of Statecharts configurations is to consider each transition as a logical implication between atomic propositions stating the presence or absence of events within a synchronous step. For instance, $a, \overline{b}/c$ states that "*whenever a is present and b is absent then c is present.*" In logic syntax we would write $(a \wedge \neg b) \supset c$, as suggested in Sec. 3.3. In this way, a configuration C turns into a set of propositional Horn clauses with negative atoms, or a logic program in which all atoms are ground. While negation is not part of standard definite Horn clause programming, it is a central feature of *normal logic programming* (NLP) which permits negative literals in clause bodies and queries. It is thus natural to relate the Pnueli-Shalev step semantics of Statecharts with constructive interpretations of negation in logic programs.

Not surprisingly, NLP exhibits problems of compositionality and full-abstraction very similar to those that have hampered the development of Statecharts semantics. The gap between the declarative, model-theoretic semantics and the operational semantics is even bigger in NLP. Specifically, if the operational model of NLP is based on a strong sequential execution model, then the order in which clauses and literals are executed is constrained. The standard operational model of *negation-as-finite-failure* (NF) is based on *SLDNF resolution*. This is, essentially, a top-down, depth-first search in which all clauses and propositions are evaluated according to a deterministic rule selection strategy. For instance, under strict left-to-right selection, the program $a/a \parallel a, b/c \parallel \overline{c}/d$ loops for query $d?$. It needs to resolve atom $c?$ due to the third clause and then atom $a?$ as the first condition of $a, b/c$. In this process, however, the search gets caught in the looping clause a/a. On the other hand, if the first clause's body is commuted to $a/a \parallel b, a/c \parallel \overline{c}/d$, then the query $c?$ has finite failure, and $d?$ evaluates to true. Clearly, such intensional features of clause scheduling are difficult to capture by compositional model-theoretic or domain-theoretic techniques. Note that the step semantics of both Pnueli-Shalev and Esterel are better behaved, because of their implicit concurrent evaluation which makes trigger conjunction and parallel composition commutative. In both semantics, the program $a/a \parallel a, b/c \parallel \overline{c}/d$ generates a single step with a and b absent and d present.

Despite the problems with the standard operational SLDNF semantics, various types of declarative models based on three-valued and many-valued interpretations have been developed in the literature to approximate SLDNF for certain classes of NLP programs. We refer the reader to [54, 20] for a detailed survey of

the results. It has been observed in [1] that Pnueli and Shalev's interpretation of steps coincides exactly with the so-called *stable models* introduced by Gelfond and Lifschitz [21]. Consider configuration C as a propositional logic program. Given a set of events $E \subseteq \Pi$, let C_E be the program in which (i) all transitions with negative triggers in E are removed, i.e., we drop from C all $P, \overline{N}/A$ with $N \cap E \neq \emptyset$; and (ii) all remaining transitions are relieved from any negative events, i.e., every $P, \overline{N}/A$ with $N \cap E = \emptyset$ is simplified to P/A. The pruned program C_E has no negations, and thus it has a unique minimal classical model M. A classical model of C_E is a set $M \subseteq \Pi$ making all transitions/clauses of C_E true, i.e., for all P/A from C_E for which $P \subseteq M$ we have $A \subseteq M$. A set $M \subseteq \Pi$ is called a *stable model* of C if M is the minimal classical model of C_M. It has been shown in [21, 49] that stable models yield a more general semantics which consistently interprets a wider class of NLP programs than SLDNF.

Theorem 8 (Correctness and Completeness). *$M \subseteq \Pi$ is a stable model of configuration C if and only if M is a Pnueli-Shalev step response of C.*

The proof of this theorem is straightforward via the denotational characterisation theorem (Thm. 3), together with the observation that $(M_0, M_1, \ldots, M_n) \models C$ in the sense of Sec. 3.3 if and only if all M_i are classical models of C_{M_n}, i.e., $M_i \models C_{M_n}$. In one direction suppose that M is the minimal classical model of C_M, i.e., $M \models C_M$ and thus $M \models C$. For every $M' \subsetneq M$ with $(M', M) \models C$ we would have $M' \models C_M$, thus contradicting that M was assumed to be minimal. Hence, M is a response model of C. Vice versa, suppose M is a response model of C. Then, $M \models C$ and thus $M \models C_M$. Further, for any other classical model $M' \subsetneq M$ of C_M, we would have $(M', M) \models C$. However, since M is a response model of C, this is impossible. This proves that M is a minimal model of C_M.

It is interesting to note that, while Pnueli and Shalev's notion of synchronous steps has not had much impact on synchronous programming tools, stable models have gained practical importance for NLP as the semantical underpinning of *answer set programming* [48]. From a wider perspective, therefore, it is fair to say that Pnueli-Shalev steps have indeed been implemented successfully in software engineering, albeit in a different domain. In addition, the theoretical results obtained around the Pnueli-Shalev semantics have ramifications in NLP. For instance, Thm. 4 of Sec. 3.3 implies that the standard intuitionistic semantics of logic provides a compositional and fully-abstract semantics for ground NLP programs under the stable interpretation.

5 Reminiscences on Amir Pnueli's First Contributions to the Semantics of Statecharts (by Willem de Roever)

The first time Amir Pnueli mentioned Statecharts to me was in 1984 during a summer school in La Colle sur Loup, North of Nice, in France. He also mentioned that he had invented the term "reactive systems" together with David Harel, during a joint air-plane flight, for the type of systems he was trying to characterise. I was immediately enthused by the concept of Statecharts: a clear

pictorial specification that could be executed, with all the operators one needed, instead of using those cumbersome algebraic notations we were wrestling with! This was what we needed to make formal specification accessible to a much larger community of users, I thought.

When, through the Esprit funding programme of Basic Research of the European Community which was launched in 1985, the opportunity presented itself to cooperate with Amir on the semantics and proof theory of Statecharts within a European project (Descartes), we immediately grasped it and started visiting each other accompanied by our teams.

5.1 Visit to the Weizmann Institute in the Mid 1980s

I recall a visit to Amir in 1986 at the Weizmann Institute, accompanied by Rob Gerth, Cees Huizing, Ton Kalker and Ruurd Kuiper, in order to attend one of AdCad's first schools on Statecharts. We listened to the members of the new AdCad company which David Harel, Amir and Haggi and Ido Lachover had founded to commercially develop the STATEMATE system, enabling the execution of Statecharts and Activity Charts.

We had a great time! At the weekends we were taken on outings to Mitzpe Ramon, the remains of a large crater in the Negev, not far from Bersheva, and by Haggi Lachover to a cave where some remains of the Neanderthal man were discovered. And during the week we had these brilliant expositions of Statecharts and their semantics. For Amir and David had recognized very early that devising the "right" semantics for Statecharts would be a really challenging problem for us semanticists!

The discussions centered on ... 5 different semantics for Statecharts. Full Stop! This is amazing! We, computer scientists, are accustomed, indeed, to a range of semantics for programming languages and concepts, culminating in the "best" semantics, the so-called fully abstract one, which doesn't introduce any unobservable differences. But for Statecharts there seemed to exist widely different semantics, which, in a sense, contradicted each other. Of course we didn't truly believe this at first, and, helped by the probing minds of Rob Gerth and Cees Huizing [32], we obtained criteria on which to judge the semantics:

- **Responsiveness**, which guarantees an instantaneous response to a request for reaction, as dictated by Gérard Berry's synchrony hypothesis [7].
- **Modularity**, which consists of two properties: (1) The composition of two reactive systems is defined on the basis of their observable behaviours; there exist no additional inner details of the execution which can only be seen by the other system. (2) When an event is generated, it is broadcast all around the system and is immediately available to everyone.
- **Causality**, i.e., for every event that is generated, there is a causal chain of events that leads to that event.

To us, whether meeting at the Weizmann Institute with Amir and David or working at the EUT in Eindhoven, these were the three criteria which a reasonable semantics for Statecharts should satisfy. But somehow it turned out to be very difficult to meet these criteria simultaneously. And that explains why Amir introduced 5 different semantics for Statecharts.

5.2 The 5 Different Semantics

For instance, there was **semantics A** [26] adopted in the STATEMATE system, in which the events that are generated as a reaction to some input can only be sensed in the step following that input, i.e., semantics A was not responsive. Certainly we should be able to do better than that!

This led to **semantics B** [28] which was responsive, but required the introduction of the notion of *micro-steps*: every observable action, i.e., every *macro-step*, was divided into an arbitrary finite number of micro-steps. Of this one, Rob Gerth, Cees Huizing and I developed a fully-abstract version [33]. The problem with this semantics is that if you take micro-steps in a different order, one may get a different observable result. So, semantics B turned out to be too subtle and too nondeterministic to be of practical use.

This led to **semantics C** [51], also known as the *Pnueli-Shalev semantics*, which overcomes this problem by demanding *global consistency* of every micro-step. Relative to this semantics, Jozef Hooman, Ramesh and I [30] developed a sound and complete compositional Hoare logic for Statecharts, and Francesca Levi [36] a sound and complete compositional proof system for checking μ-calculus properties of Pnueli-Shalev Statecharts. However, semantics C does not fully solve the problem of *modularity*, i.e., the behaviour of a process cannot be explained in terms of macro-steps only.

This led to **semantics D** in which all events that are generated during some macro-step are considered as if they were present right from the start of the step, no matter at which particular micro-step they were generated. As a consequence, the macro-behaviour of a process suffices to describe its interactions with other processes. Early versions of the languages Esterel, Lustre and Argos follow this approach. The advantage of semantics C over D, however, is that the first respects causality: each reaction can be traced back to an input from the environment via a chain of reactions, each causing the next one. In semantics D, however, it is possible that reactions trigger themselves! I.e., there is a problem with *causality*.

And then there is **semantics E**, modeling the current implementation of Statecharts in STATEMATE, which is an "acceleration" of semantics A. Events are generated at the next step, but before the reaction of the system has completely died out no input from the environment is possible.

Fig. 5, taken from [32], shows how each version of the semantics is an attempt to improve upon the other one. The discovery of Rob Gerth and Cees Huizing in 1988 that no semantics for reactive systems can be responsive, modular and causal at the same time, contributed a lot to the clarification of our many

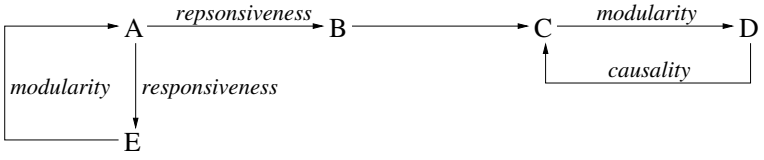


Fig. 5. Overview of the relationships among semantics A–E [32]

discussions with Amir on the semantics of Statecharts: *there exists no best semantics for them!* This helps us to explain the situation so aptly described by:

- Michael von der Beeck in [5], in which he lists more than 20 different semantics for Statecharts published by 1994;
- Andrea Maggiolo-Schettini, Adriano Peron and Simone Tini in [44], who employ some variants of Statecharts' step semantics using SOS semantics to study (pre-)congruence properties of their preorders and equivalences;
- Rick Eshuis in [16], who identifies a set of constraints ensuring that Pnueli-Shalev, STATEMATE and UML semantics coincide, if observations are restricted to linear, stuttering-closed and separable properties;
- Sharam Esmaeilsabzali, Nancy A. Day and Joanne M. Atlee in [17], who address the following two problems for Statecharts and for a large number of other languages that subscribe to the synchrony hypothesis: (1) When should one choose which semantic variant? (2) How can different semantic variants be compared, and on the basis of which criteria?

Thus one observes that, within a time span of 25 years, the focus of providing semantics for the concept of Statecharts and related languages has shifted from looking for one ideal "best" semantics to the realization that such a quest is hopeless. Instead, the insight has been gained that one either should look for a set of restrictions on the environment and one's observations such that these semantic differences disappear; or one accepts that different applications require different specification mechanisms, assisted by a catalogue of possible criteria one might want to be met.

References

[1] Aguado, J., Mendler, M.: Constructive semantics for instantaneous reactions. Theoretical Computer Science (to appear, 2010); A preliminary version of this article is available as Tech. Rep. 63/2005, Univ. of Bamberg (2005)
[2] Aguado, J., Mendler, M., Lüttgen, G.: A-maze-ing Esterel. In: SLAP '03. ENTCS, vol. 88, pp. 21–37 (2004)
[3] de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE '01, pp. 109–120. ACM Press, New York (2001)
[4] André, C.: Computing SyncCharts reactions. In: SLAP '03. ENTCS, vol. 88, pp. 3–19 (2004)

[5] von der Beeck, M.: A comparison of Statecharts variants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 128–148. Springer, Heidelberg (1994)
[6] Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. of the IEEE 91(1), 64–83 (2003)
[7] Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
[8] Bienmüller, T., Damm, W., Wittke, H.: The STATEMATE verification environment – Making it real. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 561–567. Springer, Heidelberg (2000)
[9] Boussinot, F.: Reactive C: An extension of C to program reactive systems. Software – Practice and Experience 21(4), 401–428 (1991)
[10] Boussinot, F.: SugarCubes implementation of causality. Technical Report RR-3487, INRIA (1998)
[11] Boussinot, F., Dabrowski, F.: Safe reactive programming: The FunLoft proposal. In: MULTIPROG: Programmability Issues for Multi-Core Computers. Informal workshop proceedings (2008)
[12] Boussinot, F., de Simone, R.: The SL synchronous language. IEEE Trans. on Software Engineering 22(4), 256–266 (1996)
[13] Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comp. 35(8), 677–691 (1986)
[14] van Dalen, D.: Intuitionistic logic. In: van Dalen, D. (ed.) Handbook of Philosophical Logic, ch. 4, vol. III, pp. 225–339. Reidel, Dordrechtz (1986)
[15] Damm, W., Josko, B., Hungar, H., Pnueli, A.: A compositional real-time semantics of STATEMATE designs. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 186–238. Springer, Heidelberg (1998)
[16] Eshuis, R.: Reconciling Statechart semantics. Science of Computer Programming 74(3), 65–99 (2009)
[17] Esmaeilsabzali, S., Day, N.A., Atlee, J.M.: Big-step semantics. Technical Report CS-2009-05, Univ. of Waterloo (2009)
[18] Esterel Technologies. SCADE Suite, `http://www.esterel-technologies.com`
[19] Fitting, M.: A Kripke-Kleene semantics for logic programs. Logic Programming 2(4), 295–312 (1985)
[20] Fitting, M.C.: Bilattices and the semantics of logic programming. Journal of Logic Programming 11(2), 91–116 (1991)
[21] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080 (1988)
[22] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8, 231–274 (1987)
[23] Harel, D.: Statecharts in the making: A personal account. In: History of Programming Languages, pp. 5-1–5-43. ACM Press, New York (2007)
[24] Harel, D., Gery, E.: Executable object modeling with Statecharts. IEEE Computer, 31–42 (July 1997)
[25] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtul-Trauring, A., Trakhtenbrot, M.: STATEMATE: A working environment for the development of complex reactive systems. IEEE Trans. on Software Engineering 16(4), 403–414 (1990)
[26] Harel, D., Naamad, A.: The STATEMATE semantics of Statecharts. ACM Trans. on Software Engineering Methodology 5(4), 293–333 (1996)

[27] Harel, D., Pnueli, A.: On the development of reactive systems. In: Logics and Models of Concurrent Systems. NATO ASI Series, vol. F-13, pp. 477–498. Springer, Heidelberg (1985)
[28] Harel, D., Pnueli, A., Schmidt, J.P., Sherman, R.: On the formal semantics of Statecharts. In: LICS '87, pp. 54–64. IEEE Computer Society Press, Los Alamitos (1987)
[29] Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw Hill, New York (1998)
[30] Hooman, J.J.M., Ramesh, S., de Roever, W.-P.: A compositional axiomatization of Statecharts. Theoretical Computer Science 101, 289–335 (1992)
[31] Huizing, C., de Roever, W.-P.: Introduction to design choices in the semantics of Statecharts. Information Processing Letters 37, 205–213 (1991)
[32] Huizing, C., Gerth, R.: Semantics of reactive systems in abstract time. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 291–314. Springer, Heidelberg (1992)
[33] Huizing, C., Gerth, R., de Roever, W.-P.: Modeling Statecharts behavior in a fully abstract way. In: Dauchet, M., Nivat, M. (eds.) CAAP 1988. LNCS, vol. 299, pp. 271–294. Springer, Heidelberg (1988)
[34] Jahanian, F., Mok, A.K.: Modechart: A specification language for real-time systems. IEEE Trans. on Software Engineering 20(12), 933–947 (1994)
[35] Leveson, N.G., Heimdahl, M., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Trans. on Software Engineering 20(9), 684–707 (1994)
[36] Levi, F.: A compositional μ-calculus proof system for Statecharts processes. Theoretical Computer Science 216(1-2), 271–311 (1999)
[37] Lüttgen, G., von der Beeck, M., Cleaveland, R.: Statecharts via process algebra. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 399–414. Springer, Heidelberg (1999)
[38] Lüttgen, G., von der Beeck, M., Cleaveland, R.: A compositional approach to Statecharts semantics. In: FSE 2000, ACM Software Engineering Notes, pp. 120–129 (2000)
[39] Lüttgen, G., Mendler, M.: Statecharts: From visual syntax to model-theoretic semantics. In: Integrating Diagrammatic and Formal Specification Techniques, pp. 615–621. Austrian Computer Society (2001)
[40] Lüttgen, G., Mendler, M.: Axiomatizing an algebra of step reactions for synchronous languages. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 163–174. Springer, Heidelberg (2002)
[41] Lüttgen, G., Mendler, M.: The intuitionism behind Statecharts steps. ACM Trans. on Computational Logic 3(1), 1–41 (2002)
[42] Lüttgen, G., Mendler, M.: Towards a model-theory for Esterel. In: SLAP '02. ENTCS, vol. 65(5) (2002)
[43] Maggiolo-Schettini, A., Peron, A., Tini, S.: Equivalences of Statecharts. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 687–702. Springer, Heidelberg (1996)
[44] Maggiolo-Schettini, A., Peron, A., Tini, S.: A comparison of Statecharts step semantics. Theoretical Computer Science 290(1), 465–498 (2003)
[45] Mandel, L., Pouzet, M.: ReactiveML: A reactive extension to ML. In: PPDP '05. ACM Press, New York (2005)
[46] Maraninchi, F., Rémond, Y.: Argos: An automaton-based synchronous language. Comput. Lang. 27(1/3), 61–92 (2001)

[47] The Mathworks. Stateflow user's guide, http://www.mathworks.com
[48] Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A system for answer set programming. In: Workshop on Non-Monotonic Reasoning, Breckenridge, Colorado, USA (April 2000)
[49] Pearce, D.: From here to there: Stable negation in logic programming. In: Gabbay, D.M., Wansig, H. (eds.) What is Negation?, pp. 161–181. Kluwer, Dordrecht (1999)
[50] Phillips, J., Scholz, P.: Compositional specification of embedded systems with Statecharts. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 637–651. Springer, Heidelberg (1997)
[51] Pnueli, A., Shalev, M.: What is in a step: On the semantics of Statecharts. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 244–264. Springer, Heidelberg (1991)
[52] Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling ESTEREL. Springer, Heidelberg (2007)
[53] Saraswat, V.A., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: LICS '94, pp. 71–80. IEEE Computer Society Press, Los Alamitos (1994)
[54] Shepherdson, J.C.: Logics for negation as failure. In: Logic from Computer Science, pp. 521–583. Springer, Heidelberg (1991)
[55] Taleghani, A., Atlee, J.M.: Semantic variations among UML State Machines. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 245–259. Springer, Heidelberg (2006)
[56] Uselton, A.C., Smolka, S.A.: A compositional semantics for Statecharts using labeled transition systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 2–17. Springer, Heidelberg (1994)

Author Index