Lattice structures for bisimilar Probabilistic Automata

Johann Schuster
University of the Federal Armed Forces Munich
Neubiberg, Germany
johann.schuster@unibw.de

Markus Siegle
University of the Federal Armed Forces Munich
Neubiberg, Germany
markus.siegle@unibw.de

Abstract The paper shows that there is a deep structure on certain sets of bisimilar Probabilistic Automata (PA). The key prerequisite for these structures is a notion of compactness of PA. It is shown that compact bisimilar PA form lattices. These results are then used in order to establish normal forms (in the sense of [4]) not only for finite automata, but also for infinite automata, as long as they are compact.

1 Introduction

Probabilistic automata (PA) [11, 12, 8] are a powerful and popular modelling formalism, since they allow one to reason about the behaviour of systems which feature both randomness and nondeterminism. For probabilistic automata (PA), several notions of simulations and bisimulations have been defined in the literature [11, 10]. Bisimulation relations – in general, but also in particular for probabilistic automata – are employed for characterising equivalent behaviour. Thus they may serve as the basis for checking whether two systems are equivalent in some sense. As a straightforward consequence, bisimulation relations are also very valuable for reducing the size of a system, by replacing it with an equivalent but smaller one. Hereby the goal is to find the smallest possible bisimilar system, i.e. the minimal one. We concentrate on two notions of bisimulation for PA: strong probabilistic bisimulation and weak probabilistic bisimulation. For automata with finite sets of states and transitions, both are known to be decidable in polynomial time [2, 6].

Recently, the question of calculating minimal canonical forms (i.e. normal forms) of probabilistic automata has been tackled (again, for automata with finite sets of states and transitions) [4], where it turned out that this problem can also be solved in polynomial time. In the present paper, we go one step further: We show that finiteness is not required for defining minimal canonical forms. We point out that also for automata with countably infinite ("countable" for short) state space, countable set of actions, and possibly uncountable set of transitions, there are notions of normal forms. However, we show that, in contrast to the finite case, normal forms do not always exist. It rather turns out that an auxiliary condition, namely compactness [3], is crucial for the existence of normal forms.

In the context of PA, the distributions reached by probabilistic schedulers form convex sets. The first ones to use this observation were Segala and Cattani who developed a decision algorithm out of this fact [2]. In this paper, we combine the strongly geometric ideas of [2] with the ideas of compact automata of [3]. This enables us to extend the recent results of [4] to a class of PA with countable state space. In this way we may use geometrical ideas to show that normal forms of PA arise naturally as some "generating points" – in a strong or weak sense – of convex sets. However, we also show (by a counterexample) that in general one cannot expect normal forms for arbitrary PA with countable state space. For normal forms wrt. strong bisimulation, we may directly use a classical result from functional analysis, the theorem of Krein-Milman [7], that directly extends the ideas of [2] to the case of countable state spaces. For weak bisimulation, we extend the results of [4] by adding some compactness assumptions.

L. Clemente and L. Holik (Eds.): 15th International Workshop on
Verification of Infinite-State Systems (INFINITY'13)
EPTCS 140, 2014, pp. 1–15, doi:10.4204/EPTCS.140.1

This paper is, to the best of our knowledge, the first approach to derive lattice structures on bisimilar objects. Technically, we work with partially ordered sets of bisimilar PA on which lattice structures are established, such that the normal form corresponds to the bottom element of the lattice. We show that there are unique bottom elements in the lattices we define. We work on quotients of PA, but do not address the question of how to find such quotients for arbitrary automata. For the countably infinite case, ideas from [1, 5] could be used to calculate quotients, but we do not investigate this further. However, even if it is hard to find quotients for infinite state systems, we feel that the lattice structures we establish are interesting just from an abstract point of view.

The paper is organised as follows: Sec. 2 recalls some basic facts on preorders, lattices and probabilistic automata, compact sets and extreme points. It also defines the notion of compact PA which is essential for our paper. Operations on bisimilar quotients and sets of quotients (intersection, union, rescaling) are defined in Sec. 3. The core of the paper consists of the results on lattice structures given in Sec. 4. Some illustrating examples are provided in Sec. 5, and Sec. 6 concludes the paper.

2 Preliminaries

Definition 1. *The disjoint union of two sets S_1 and S_2 is defined as $S_1 \dot\cup S_2 := \cup_{i\in\{1,2\}}\{(x,i)|x \in S_i\}$.*

The disjoint union is defined up to isomorphism, which implies commutativity $S_1 \dot\cup S_2 = S_2 \dot\cup S_1$. There are canonical embeddings $S_1 \to S_1 \dot\cup S_2$, $x \mapsto (x,1)$ and $S_2 \to S_1 \dot\cup S_2$, $x \mapsto (x,2)$.

2.1 Partial orders and lattices

Definition 2. *A* partial order *is a binary relation $\leq$ over a set S which is* antisymmetric*,* transitive*, and either* reflexive *or* irreflexive*, i.e., for all a, b, and c in S, we have that:*

- *If $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry).*
- *If $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).*
- *Either: $a \leq a$ (reflexivity) for all $a \in S$, or: $a \not\leq a$ (irreflexivity) for all $a \in S$.*

A set with a partial order is called a partially ordered set *(also called a poset). For a poset we write $a < b$ iff $a \leq b$ and $a \neq b$ (and similar $a > b$).*

Definition 3. *Let $(S,\leq)$ be a poset, and let a and b be two elements in S. An element c of S is the* meet *(or* greatest lower bound *or* infimum*) of a and b, if the following two conditions are satisfied:*

- *$c \leq a$ and $c \leq b$ (i.e., c is a* lower bound *of a and b).*
- *For any $d \in S$, such that $d \leq a$ and $d \leq b$, we have $d \leq c$ (i.e., c is greater than or equal to any other lower bound of a and b).*

An element c of S is the join *(or* least upper bound *or* supremum*) of a and b, if the following two conditions are satisfied:*

- *$a \leq c$ and $b \leq c$ (i.e., c is a* upper bound *of a and b).*
- *For any $d \in S$, such that $a \leq d$ and $b \leq d$, we have $c \leq d$ (i.e., c is smaller than or equal to any other upper bound of a and b).*

Remark 1. *If there is a meet (join) of a and b, then indeed it is unique, since if both c and c' are greatest lower bounds (least upper bounds) of a and b, then $c \leq c'$ and $c' \leq c$, whence indeed $c' = c$.*

Definition 4. *A poset $(L,\leq)$ is a* lattice *if it satisfies the following two axioms.*

- *(Existence of binary meets) For any two elements a and b of L, the set $\{a,b\}$ has a meet: $a \wedge b$ (also known as the greatest lower bound, or the infimum).*
- *(Existence of binary joins) For any two elements a and b of L, the set $\{a,b\}$ has a join: $a \vee b$ (also known as the least upper bound, or the supremum).*

A lattice is called bounded*, if it has a least and a greatest element, i.e. elements l,g such that for all $x \in L : x \leq g$ and $l \leq x$. We will also write $\bot(L) = l$, $\top(L) = g$. A lattice is called* complete*, if meet and join exist for* all *subsets $A \subseteq L$.*

We will use the following simple but elementary lemma:

Lemma 1 (descending chain condition (DCC))**.** *Let $(S,\leq)$ be a poset, then the following statements are equivalent:*

- *Every nonempty subset $A \subseteq S$ contains an element minimal in A*
- *S contains no infinite descending chain $a_0 > a_1 > a_2 > \ldots$*

2.2 Probabilistic Automata

First we define the notion of discrete subdistribution and related terms and notations:

Definition 5 ((Sub-)distributions)**.** *Let S be a countable set. A mapping $\mu : S \to [0,1]$ is called (discrete)* subdistribution*, if $\sum_{s \in S} \mu(s) \leq 1$. As usual we write $\mu(S')$ for $\sum_{s \in S'} \mu(s)$. The* support *of μ is defined as $Supp(\mu) := \{s \in S | \mu(s) > 0\}$. The empty subdistribution $\mu_\emptyset$ is defined by $Supp(\mu_\emptyset) = \emptyset$. The* size *of μ is defined as $|\mu| := \mu(S)$. A subdistribution μ is called* distribution *if $|\mu| = 1$. The sets $Dist(S)$ and $Subdist(S)$ denote distributions and subdistributions defined over the set S. Let $\Delta_s \in Dist(S)$ denote the* Dirac *distribution on s, i.e. $\Delta_s(s) = 1$. For two subdistributions μ, μ' the sum $\mu'' := \mu \oplus \mu'$ is defined as $\mu''(s) := \mu(s) + \mu'(s)$ (as long as $|\mu''| \leq 1$). As long as $c \cdot |\mu| \leq 1$, we denote by $c\mu$ the subdistribution defined by $(c\mu)(s) := c \cdot \mu(s)$. For a subdistribution μ and a state $s \in Supp(\mu)$ we define $\mu - s$ by*

$$(\mu - s)(t) = \begin{cases} \mu(t) & \text{for } t \neq s \\ 0 & \text{for } t = s \end{cases}$$

Definition 6 (Lifting of relations on states to distributions)**.** *Whenever there is an equivalence relation $R \subseteq S \times S$, we may lift it to $Dist(S) \times Dist(S)$ in the following way. For μ, $\gamma \in Dist(S)$ we write $\mu L(R) \gamma$ (or simply, by abuse of notation, $\mu R \gamma$) if and only if for each $C \in S/R : \mu(C) = \gamma(C)$.*

Definition 7 (cf. [2])**.** *A probabilistic automaton (PA) P is a tuple (S, Act, T, s_0), where S is a countable set of* states*, $s_0 \in S$ is the* initial *state, Act is a countable set of* actions *($Act = H \dot\cup E$, H hidden actions, E external actions) and $T \subseteq S \times Act \times Dist(S)$ is a* transition relation *(can be uncountable). Whenever $(s,a,\mu) \in T$ we also write $s \xrightarrow{a} \mu$.*

In this paper we restrict ourselves to the case where $H = \{\tau\}$, i.e. $E = Act \setminus \{\tau\}$. Note that by the countability of S it is clear that every distribution over S has at most countable support.

2.2.1 Weak transitions

In the following we use the definitions and terminology of [9], but we leave out the definitions for labelled transition systems. The only major difference is that we do not assume *finite branching*, i.e. for each state s the set $\{(a,\mu) \in Act \times Dist(S) | s \xrightarrow{a} \mu\}$ does not have to be finite. Given a transition $tr = (s,a,\mu)$, we denote s by $source(tr)$ and μ by μ_{tr}. An execution fragment of a PA $P = (S, Act, T, s_0)$ is a finite or infinite sequence $\alpha = q_0 a_1 q_1 a_2 q_2 \cdots$ of alternating states and actions, starting with a state and, if the sequence is finite, ending in a state, where each $(q_i, a_{i+1}, \mu_{i+1}) \in T$ and $\mu_{i+1}(q_{i+1}) > 0$. State q_0, the first state of α, is denoted by $fstate(\alpha)$. If α is a finite sequence, then the last state of α is denoted by $lstate(\alpha)$. An *execution* of P is an execution fragment (of P) where $q_0 = s_0$. We let $frags(P)$ denote the set of execution fragments of P and $frags^*(P)$ the set of finite execution fragments of P. Similarly, we let $execs(P)$ denote the set of executions of P and $execs^*(P)$ the set of finite executions. Execution fragment α is a *prefix* of execution fragment α', denoted $\alpha \leq \alpha'$, if sequence α is a prefix of sequence α'.

The *trace* of an execution fragment α, written $trace(\alpha)$, is the sequence of actions obtained by restricting α to the set of external actions, i.e. $Act \setminus \{\tau\}$. For a set E of executions of a PA P, $traces(E)$ is the set of traces of the executions in E. We say that β is a trace of a PA P if there is an execution α of P with $trace(\alpha) = \beta$. Let $traces(P)$ denote the set of traces of P.

A *scheduler* for a PA P is a function $\sigma : frags^*(P) \to SubDisc(T)$ such that $tr \in supp(\sigma(\alpha))$ implies that $source(tr) = lstate(\alpha)$. This means that the image $\sigma(\alpha)$ is a *discrete* subdistribution over transitions. The defect of the subdistribution, i.e. $1 - |\sigma(\alpha)|$ is used for stopping in the current state. In other words, a scheduler is the entity that resolves nondeterminism in a probabilistic automaton by choosing randomly either to stop or to perform one of the transitions that are enabled from the current state. A scheduler σ is said to be *deterministic* if for each finite execution fragment α either $\sigma(\alpha)(T) = 0$ or $\sigma(\alpha) = \Delta(tr)$ (Dirac measure for tr) for some $tr \in T$. A scheduler is called *memoryless*, if it depends only on the last state of its argument, that is, for each pair α_1, α_2 of finite execution fragments, if $lstate(\alpha_1) = lstate(\alpha_2)$, then $\sigma(\alpha_1) = \sigma(\alpha_2)$.

A scheduler σ and a discrete initial probability measure $\mu_0 \in Dist(S)$ induce a measure ε on the sigma-field generated by cones of execution fragments as follows. If α is a finite execution fragment, then the *cone* of α is defined by $C_\alpha = \{\alpha' \in frags(P) | \alpha \leq \alpha'\}$. The measure ε of a cone C_α is defined recursively: If $\alpha = s$ for some $s \in S$ we define $\varepsilon(C_\alpha) = \mu_0(s)$. If α is of the form $\alpha' a' s'$ it is defined by the equation

$$\varepsilon(C_\alpha) = \varepsilon(C_{\alpha'}) \cdot \sum_{tr \in T(a')} \sigma(\alpha')(tr) \mu_{tr}(s'),$$

where $T(a')$ denotes the set of transitions of T that are labelled by a'. Standard measure theoretical arguments ensure that ε is well defined. We call the measure ε a probabilistic execution fragment of P, and we say that ε is generated by σ and μ_0.

Consider a probabilistic execution fragment ε of a PA P, with first state s, i.e. $\mu_0 = \Delta(s)$, that assigns probability 1 to the set of all finite execution fragments α with trace $trace(\alpha) = \beta$ for some $\beta \in (Act \setminus \{\tau\})^*$. Let μ be the discrete measure defined by $\mu(s') = \varepsilon(\{\alpha | lstate(\alpha) = s'\})$. Then $s \stackrel{\beta}{\Rightarrow}_C \mu$ is a *weak combined transition* of P. We call ε a *representation* of $s \stackrel{\beta}{\Rightarrow}_C \mu$. If $s \stackrel{\beta}{\Rightarrow}_C \mu$ is induced by a deterministic scheduler, we also write $s \stackrel{\beta}{\Rightarrow} \mu$. In case $trace(\alpha)$ is empty we write $s \stackrel{\tau}{\Rightarrow}_C \mu$.

Let $\{s \xrightarrow{a} \mu_i\}_{i \in I}$ be a collection of transitions of a PA P, and let $\{c_i\}_{i \in I}$ be a collection of probabilities such that $\sum_{i \in I} c_i = 1$. Then the triple $(s, a, \sum_{i \in I} c_i \mu_i)$ is called a *(strong) combined transition* of P and we write $s \xrightarrow{a}_C \sum_{i \in I} c_i \mu_i$.

2.3 Bisimulations

Definition 8 (Strong probabilistic bisimulation [12])**.** *An equivalence relation R on the set of states S of a PA $P = (S, Act, T, s_0)$ is called* strong probabilistic bisimulation *if and only if $x\,R\,y$ implies for all $a \in Act$: $(x \xrightarrow{a} \mu)$ implies $(y \xrightarrow{a}_C \mu')$ with $\mu(C) = \mu'(C)$ for all $C \in S/R$. Two PA are called* strongly bisimilar *if their initial states are related by a strong probabilistic bisimulation relation on the direct sum of their states.*

Definition 9 (Weak probabilistic bisimulation [12])**.** *An equivalence relation R on the set of states S of a PA $P = (S, Act, T, s_0)$ is called* weak probabilistic bisimulation *if and only if $x\,R\,y$ implies for all $a \in Act$: $(x \xrightarrow{a} \mu)$ implies $(y \stackrel{a}{\Rightarrow}_C \mu')$ with $\mu(C) = \mu'(C)$ for all $C \in S/R$. Two PA are called* weakly bisimilar *if their initial states are related by a weak probabilistic bisimulation relation on the direct sum of their states.*

Note that (as we always use combined transitions), in the following we generally omit the word "probabilistic" for our bisimulation relations, even if in the sense of [12] we speak about strong probabilistic and weak probabilistic bisimulations. In the sequel we denote, as usual, by $\sim$ a strong bisimulation relation and by $\approx$ a weak bisimulation relation.

It has been shown in [12] that decision of bisimilarities is closely related to convex sets of reachable distributions, i.e. $S_\sim(s,a) := \{\mu \in Dist(S) | s \xrightarrow{a}_C \mu\}/\sim$ and $S_\approx(s,a) := \{\mu \in Dist(S) | s \stackrel{a}{\Rightarrow}_C \mu\}/\approx$. Those sets are considered modulo the bisimilarities known and splitters are constructed out of them. For details we refer to [12]. The important point is that those sets can be defined for infinite state systems as well. The only difference is that they can no longer be regarded as subsets of $\mathbb{R}^n$ then.

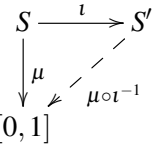
It is clear by definition that the unreachable parts of an automaton do not play any role for bisimilarity, which motivates the following definition.

Definition 10 (Reachable states)**.** *Let $P = (S, Act, T, s_0)$ be a PA, $S' \subseteq S$ its set of reachable states, i.e. those states that can be reached with non-zero probability by a scheduler starting from s_0. Let $T' := T|_{S' \times Act \times Dist(S')}$ be the restriction of the transition relation to S'. We define $r(P) := (S', Act, T', s_0)$ and call it the* reachable fraction *of P.*

2.4 Isomorphic & quotient automata

We want to be able to identify automata that are basically the same, only having different names of their states. The following definition formalises this.

Definition 11 (Isomorphic automata)**.** *Let $P = (S, Act, T, s_0)$ and S' a set with $|S'| = |S|$. We call a bijective mapping between sets $\iota : S \to S'$ a (set-)*isomorphism*. Via this isomorphism we may push forward distributions on S to distributions on S' by the mapping $\mu \circ \iota^{-1}$, as the following diagram shows.*



That in turn means that we may push forward transitions between states in the set S to transitions between states in the set S' using our isomorphism by the following mapping

$$\begin{array}{rccc} \iota: & S \times Act \times Dist(S) & \to & S' \times Act \times Dist(S') \\ & (s, a, \mu) & \mapsto & (\iota(s), a, \mu \circ \iota^{-1}) \end{array}$$

By this, we may define the automaton $\iota(P) := (\iota(S), Act, \iota(T), \iota(s_0))$ *where we, as usual, denote for mappings* $f : A \to B$ *by* $f(A) := \{f(a) | a \in A\} \subseteq B$ *the* image *of A under f. Two PA* $P = (S, Act, T, s_0)$, $P' = (S', Act, T', s_0')$ *are called* isomorphic*, if there exists an isomorphism* $\iota : S \to S'$ *such that* $P' = \iota(P)$. *For isomorphic automata we write* $P \equiv_{iso} P'$.

We would like to stress that we require the *same* set of actions for two automata to be isomorphic. The reason for this is that also for bisimulations the *same* set of actions is considered.

Lemma 2. *The relation* $\equiv_{iso}$ *is an equivalence relation.*

Proof. Follows by direct verification from the properties of isomorphisms □

Whenever an equivalence relation R is given (R will be chosen to be $\sim$ or $\approx$), it is common to look at a special automaton that is defined over equivalence classes of states. The following definition formalises this.

Definition 12 (Quotient automaton)**.** *Let* $P = (S, Act, T, s_0)$ *be a PA and R an equivalence relation over S. We write* P/R *to denote the quotient automaton of P wrt. R, that is*

$$P/R = (S/R, Act, T/R, [s_0]_R)$$

with $T/R \subseteq S/R \times Act \times Dist(S/R)$ *such that* $([s]_R, a, \mu) \in T/R$ *if and only if there exists a state* $s' \in [s]_R$ *such that* $(s', a, \mu') \in T$ *and* $\forall [t]_R \in S/R : \mu([t]_R) = \sum_{t' \in [t]_R} \mu'(t')$. *We call an automaton a* quotient wrt. *R if it holds that* $P \equiv_{iso} P/R$.

2.5 Compact automata

One key property when searching for lattice structures on PA is compactness. Compactness definitions already have been introduced in [3] for the alternating model.

Definition 13 (adapted from Def. 9 in [3])**.** *Let* $P = (S, Act, T, s_0)$ *be a PA. The function d on* $Dist(S) \times Dist(S)$ *is defined by*

$$d(\mu_1, \mu_2) := sup_{A \subseteq S} |\mu_1(A) - \mu_2(A)|$$

Even if in [3] it is mentioned without a proof, that function d defined above is really a metric, we give an explicit proof here.

Lemma 3. *The function d in Def. 13 is a metric on* $Dist(S)$.

Proof. It is clear by definition that d is non-negative and symmetric. Identity of indiscernibles is worth a thought: For general measures we only know when $d(\mu_1, \mu_2) = 0$ that μ_1 and μ_2 coincide up to a zero set. As $Dist(S)$ is a set of *discrete* probability measures, the values $\{(s, \mu(s)) | s \in S\}$ completely determine μ. Therefore $d(\mu_1, \mu_2) = 0 \Rightarrow \mu_1 = \mu_2$. Also the triangle inequality holds: $d(\mu_1, \mu_3) = sup_{A \subseteq S} |\mu_1(A) - \mu_3(A)| = sup_{A \subseteq S} |\mu_1(A) - \mu_2(A) + \mu_2(A) - \mu_3(A)| \leq sup_{A \subseteq S} (|\mu_1(A) - \mu_2(A)| + |\mu_2(A) - \mu_3(A)|) \leq sup_{A \subseteq S} |\mu_1(A) - \mu_2(A)| + sup_{A \subseteq S} |\mu_2(A) - \mu_3(A)| = d(\mu_1, \mu_2) + d(\mu_2, \mu_3)$ □

Corollary 1. *Let* $P = (S, Act, T, s_0)$ *be a PA and fix an equivalence relation* $R \in \{\sim, \approx\}$ *on S,* $s \in S$ *and* $a \in Act$. *Then for every pair* $(s, a) \in S \times Act$ *there is a metric space* $(S_R(s, a), d)$.

Definition 14 (adapted from Def. 10 in [3])**.** *Let* $P = (S, Act, T, s_0)$ *be a PA and consider some fixed equivalence relation* $R \in \{\sim, \approx\}$ *on S,* $s \in S$ *and* $a \in Act$. *We say that state s is a-compact wrt. R, if the set* $S_R(s, a)$ *is compact under the metric d. P is called* compact *wrt. R, if* $\forall s \in S \, \forall a \in Act$: *s is a-compact wrt. R. We omit "wrt. R" if the context is clear.*

The definition given by Desharnais et al. can be seen also in the view of products of metric spaces. We recall the definition of a product of metric spaces:

Definition 15. *If (M_i, d_i), $i \in \mathbb{N}$ are metric spaces, we define a metric on the countably infinite Cartesian product $\prod_{i\in\mathbb{N}} M_i$ by*

$$d(x,y) = \sum_{i=1}^{\infty} \frac{1}{2^i} \frac{d_i(x_i,y_i)}{1+d_i(x_i,y_i)}.$$

It is well-known that this construction leads to the metric space $\left(\prod_{i\in\mathbb{N}} M_i, d\right)$.

This metric metrizes the product topology - under this metric all projection functions are continuous.

We are now able to relate the metric space $\left(\prod_{(s,a)\in S\times Act} S_R(s,a), d\right)$ to the definition of compactness given in Def. 14.

Lemma 4 (Reformulation of Def. 14)**.** *Let $P = (S, Act, T, s_0)$ be a quotient of a PA wrt. some equivalence relation R. The compactness of P defined in Def. 14 is equivalent to the compactness of the metric space $\left(\prod_{(s,a)\in S\times Act} S_R(s,a), d\right)$.*

Proof. Assume that Def. 14 holds. Tychonoff's theorem (or equivalently the axiom of choice) ensures that the product of compact spaces is compact again, so the product space in Def. 15 is compact.

The projections $\prod_{(s,a)\in S\times Act} S_R(s,a) \to S_R(s,a)$ are continuous for every pair (s,a). Continuous mappings between metric spaces map compact sets to compact sets. So Def. 14 also holds. □

The question about compactness is only relevant in the case of infinite automata. As long as PA are constructed out of finite sets (both states and transitions), they are compact anyway. This is due to the following classical theorem:

Theorem 1 (Heine-Borel)**.** *For a subset of $M \subseteq \mathbb{R}^n$ ($\mathbb{R}^n$ the metric space of all n-Tuples with Euclidean metric) the following statements are equivalent:*

- *M is bounded and closed*
- *every open cover of M has a finite subcover, that is, M is compact*

An important theorem for compact sets is the Krein-Milman theorem (see below). It only works for locally convex vector spaces. We use as a basic fact, that the metric space $(l^\infty, d(x,y) = sup_{i\in\mathbb{N}}|x_i - y_i|)$ of bounded sequences in $\mathbb{R}$ is locally convex (using the seminorms $p_i(\{x_n\}_n) = |x_i|$, $i \in \mathbb{N}$). Sequences may be used to characterise distributions in the following way. Assume that the states are ordered by the natural numbers. Define the mapping $f : (S_R(s,a), d) \to (l^\infty, d)$, $\mu \mapsto (\mu(x_1), \mu(x_2), \ldots)$. We show that this mapping is continuous, which is straight-forward, as singletons are also subsets. $sup_{A\subseteq S}|\mu_1(A) - \mu_2(A)| < \varepsilon \Rightarrow sup_{a\in S}|\mu_1(a) - \mu_2(a)| < \varepsilon$. As continuous images of compact sets are compact, we know that $f(S_R(s,a))$ is compact. By definition of convex schedulers it is also clear that $f(S_R(s,a))$ is convex. The product of $\prod_{(s,a):S_R(s,a)\neq\emptyset} f(S_R(s,a))$ is also compact (Tychonoff's Theorem – accepting the axiom of choice) and convex as a product of convex sets. These sets will be used later to apply the Krein-Milman theorem.

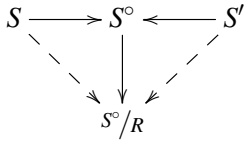
Definition 16 (Extreme points)**.** *An* extreme point *of a convex set A is a point $x \in A$ with the property that if $x = cy + (1-c)z$ with $y, z \in A$ and $c \in [0,1]$, then $y = x$ or $z = x$. $\mathbb{E}(A)$ will denote the set of extreme points of A.*

Now the Krein-Milman theorem says that a compact convex subset of a locally convex vector space is the convex hull of its extreme points:

Theorem 2 (Krein-Milman [7])**.** *Let A be a compact convex subset of a locally convex vector space X, then $A = CHull(\mathbb{E}(A))$.*

3 Operations on bisimilar quotient automata

Definition 17 (Set operations on bisimilar quotients)**.** *Let $P = (S, Act, T, s_0)$ and $P' = (S', Act, T', s'_0)$ be PA. Let $S^\circ := S \dot\cup S'$ be the disjoint union of state spaces and consider a fixed bisimulation relation $R \in \{\sim, \approx\}$, $R \subseteq S^\circ \times S^\circ$. Let P and P' be bisimilar, i.e. $s_0 R s'_0$. In the following, we assume that P and P' are quotients wrt. R. The intersection of S and S' (and the union) is performed in ${}^{S^\circ}/_R$ by means of the canonical mappings*

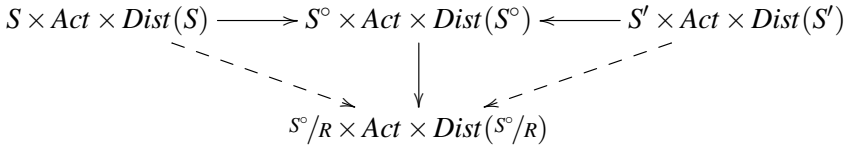


With this diagram in mind we may define

$$S' \cap_R S = \{[x] \in {}^{S^\circ}/_R | \exists s \in S : s \in [x] \wedge \exists s' \in S' : s' \in [x]\}$$

$$S' \cup_R S = \{[x] \in {}^{S^\circ}/_R | \exists s \in S : s \in [x] \vee \exists s' \in S' : s' \in [x]\}$$

Using these canonical mappings, similar operations can be defined on the sets of transitions. Clearly $T \subseteq S \times Act \times Dist(S)$ and $T' \subseteq S' \times Act \times Dist(S')$. The set operations are performed in the set ${}^{S^\circ}/_R \times Act \times Dist({}^{S^\circ}/_R)$ by means of the corresponding canonical mappings.



We thus may define (using the abbreviation $\mathfrak{T} = {}^{S^\circ}/_R \times Act \times Dist({}^{S^\circ}/_R)$):

$$T' \cap_R T = \{([x], a, [\mu]) \in \mathfrak{T} | (\exists (s, a, \gamma) \in T : s \in [x] \wedge \gamma \in [\mu]) \wedge (\exists (s', a, \gamma') \in T' : s' \in [x] \wedge \gamma' \in [\mu])\}$$

$$T' \cup_R T = \{([x], a, [\mu]) \in \mathfrak{T} | (\exists (s, a, \gamma) \in T : s \in [x] \wedge \gamma \in [\mu]) \vee (\exists (s', a, \gamma') \in T' : s' \in [x] \wedge \gamma' \in [\mu])\}$$

Finally we note that s_0 has to be mapped to ${}^{S^\circ}/_R$. Summing up, we define the intersection quotient wrt. R as

$$P \cap P' := (S' \cap_R S, Act, T \cap_R T', [s_0]_R)$$

and, similarly, the union quotient wrt. R as

$$P \cup P' := (S' \cup_R S, Act, T \cup_R T', [s_0]_R)$$

With the same mappings one may define $P \subseteq P'$ if and only if $S \subseteq S'$ (in ${}^{S^\circ}/_R$) and $T \subseteq T'$ in ${}^{S^\circ}/_R \times Act \times Dist({}^{S^\circ}/_R)$.

As we target on bisimilar automata in this paper, we would like to stress that only the reachable fraction of automata is relevant. Our quotient definition and the above set operations are defined for the general case, where unreachable states may occur, but of course they apply also to the case where only reachable states are present.

Lemma 5. *Let $R \in \{\sim, \approx\}$ and $P = (S, Act, T, s_0)$, $P' = (S', Act, T', s'_0)$ be bisimilar quotients of PA wrt. R. It holds that $P \cap P' \equiv_{iso} P' \cap P$ and $P \cup P' \equiv_{iso} P' \cup P$.*

Proof. Clear by identifying the different direct sums according to R □

As it will turn out that τ-loops leading back to the same state (as part of a distribution) can disturb the lattice property in the case of weak bisimulation, we give the following definition.

Definition 18 (Rescaled Automata)**.** *An automaton $P=(S,Act,T,s_0)$ is called* rescaled*, if for all $(s,\tau,\mu)\in T$ it holds either that $\mu(s)=0$ or $\mu(s)=1$.*

Note that the rescaling as defined in Def. 18 is only one possibility (incidently the one leading to the smallest transition fanout), but different rescalings, where each τ transition would loop back to its source state with fixed probability $0\leq p<1$ (or probability 1 for "loops") would also work.

Remark 2. *Let $P=(S,Act,T,s_0)$ be a PA. We split its set of transitions into the sets $T_{\neg\tau}:=\{(s,a,\mu)\in T|a\neq\tau\}$, $T_\Delta=\{(s,\tau,\mu)\in T|\mu=\Delta_s\}$ and $T_{\neg\Delta}=\{(s,\tau,\mu)\in T|\mu\neq\Delta_s\}$. Now we define the function $res:T_{\neg\Delta}\to S\times Act\times Dist(S)$ by $res(s,\tau,\mu):=(s,\tau,\frac{1}{1-\nu(s)}(\nu-s))$ which is well-defined, as we always have $\nu(s)\neq 1$. With*

$$T':=T_{\neg\tau}\cup T_\Delta\cup res(T_{\neg\Delta})$$

we may define the rescaled automaton $P^{res}:=(S,Act,T',s_0)$. When using randomised schedulers, it is a basic fact, that $P\approx P^{res}$.

3.1 Sets of quotients

Definition 19 (Sets of quotients)**.** *Let P be an automaton and $\mathscr{PA}$ be the set of all PA. Define the set*

$$\mathfrak{Q}_\sim(P):=\{A\in\mathscr{PA}|A \text{ quotient wrt. } \sim, A\sim P\}$$

and the set

$$\mathfrak{Q}_\approx(P):=\{A\in\mathscr{PA}|A \text{ quotient wrt. } \approx, A\approx P, A \text{ rescaled}\}$$

The reachable fractions can also be considered:

$$\mathfrak{Q}^*_\sim(P):=\{A\in\mathscr{PA}|A \text{ quotient wrt. } \sim, A\sim P, A=r(A)\}$$

and the set

$$\mathfrak{Q}^*_\approx(P):=\{A\in\mathscr{PA}|A \text{ quotient wrt. } \approx, A\approx P, A \text{ rescaled}, A=r(A)\}$$

Of course, the automata in the set $\mathfrak{Q}^*_\sim(P)$ (in the set $\mathfrak{Q}^*_\approx(P)$) all have the same number of states.

Lemma 6. *It holds that $\mathfrak{Q}^*_\sim(P)\subseteq\mathfrak{Q}_\sim(P)$ and $\mathfrak{Q}^*_\approx(P)\subseteq\mathfrak{Q}_\approx(P)$.*

Remark 3. *As usual for equivalence relations, we may consider quotients wrt. isomorphism, i.e. ${}^{\mathfrak{Q}_\sim(P)}/_{\equiv_{iso}}$ and ${}^{\mathfrak{Q}_\approx(P)}/_{\equiv_{iso}}$. Without loss of generality we may identify the states of every automaton in these quotients by a subset of the natural numbers $\mathbb{N}$ (using 1 for the initial states). For the sets $\mathfrak{Q}_\sim(P)$ and $\mathfrak{Q}_\approx(P)$ such an enumeration cannot be performed, as there are uncountably many unreachable parts (this is shown in detail by Lemma 13).*

For the rest of the paper we now assume that the states in ${}^{\mathfrak{Q}^*_\sim(P)}/_{\equiv_{iso}}$ and ${}^{\mathfrak{Q}^*_\approx(P)}/_{\equiv_{iso}}$ are consecutive natural numbers, starting with 1 for the initial state.

The rest of this section is devoted to show that for compact P there are well-defined operations

$$\cap:{}^{\mathfrak{Q}_\sim(P)}/_{\equiv_{iso}}\times{}^{\mathfrak{Q}_\sim(P)}/_{\equiv_{iso}}\to{}^{\mathfrak{Q}_\sim(P)}/_{\equiv_{iso}}$$

$$\cap:{}^{\mathfrak{Q}_\approx(P)}/_{\equiv_{iso}}\times{}^{\mathfrak{Q}_\approx(P)}/_{\equiv_{iso}}\to{}^{\mathfrak{Q}_\approx(P)}/_{\equiv_{iso}}$$

Similar operations exist for $\cup$, but their existence and well-definedness is clear. Note that it is a priori *not* clear that the result is again in ${}^{\mathfrak{Q}_\sim(P)}/_{\equiv_{iso}}$ (or ${}^{\mathfrak{Q}_\approx(P)}/_{\equiv_{iso}}$, respectively), with other words that the results are again bisimilar to P. This will be shown in the following.

Remark 4 (Counterexample)**.** *We show that there is in general no well-defined operation* $\cap : \mathfrak{Q}_{\sim(P)}/_{\equiv_{iso}} \times \mathfrak{Q}_{\sim(P)}/_{\equiv_{iso}} \to \mathfrak{Q}_{\sim(P)}/_{\equiv_{iso}}$*, that means the compactness of P is crucial. We use the (non-compact) automata*

$$P := (\{1,2,3\},\{\tau,a,b\},\{(1,\tau,\frac{1}{n}\Delta_2 \oplus (1-\frac{1}{n})\Delta_3)\}_{n\in\mathbb{N},n\geq 2} \cup \{(2,a,\Delta_2),(3,b,\Delta_3)\},1)$$

and

$$P' := (\{1,2,3\},\{\tau,a,b\},\{(1,\tau,\frac{\mathrm{e}}{n}\Delta_2 \oplus (1-\frac{\mathrm{e}}{n})\Delta_3)\}_{n\in\mathbb{N},n\geq 6}$$
$$\cup\{(1,\tau,\frac{1}{2}\Delta_2 \oplus \frac{1}{2}\Delta_3),(2,a,\Delta_2),(3,b,\Delta_3)\},1).$$

It is easy to see that $P \sim P'$ *and* $P,P' \in \mathfrak{Q}_{\sim(P)}/_{\equiv_{iso}}$*. It is clear that the intersection of both automata is only*

$$P^{\cap} := (\{1,2,3\},\{\tau,a,b\},\{(1,\tau,\frac{1}{2}\Delta_2 \oplus \frac{1}{2}\Delta_3),(2,a,\Delta_2),(3,b,\Delta_3)\},1),$$

as e *is* not *a rational number. So the result of the intersection is clearly not bisimilar, i.e. not in* $\mathfrak{Q}_{\sim(P)}/_{\equiv_{iso}}$*. The reason for this is, that both automata allow for a limiting distribution* $(1,\tau,\Delta_3)$*, but both don't reach this distribution. As the ways how this limit is reached (i.e. the sequences) are different, the intersection no longer leads to a bisimilar result. Note that there are many other examples (e.g. every irrational root may be taken instead of* e*).*

Thus we have shown:

Lemma 7. *In general* $(\mathfrak{Q}_{\sim(P)}/_{\equiv_{iso}},\subseteq)$ *and* $(\mathfrak{Q}^*_{\sim(P)}/_{\equiv_{iso}},\subseteq)$ *are* not *lattices.*

4 Lattice structures

For the rest of this paper we consider a subset of PA where $(\mathfrak{Q}_{R(P)}/_{\equiv_{iso}},\subseteq)$ and $(\mathfrak{Q}^*_{R(P)}/_{\equiv_{iso}},\subseteq)$, $R \in \{\sim,\approx\}$ are lattices.

Lemma 8 (compact sets of quotients)**.** *Let* $R \in \{\sim,\approx\}$*. The PAs contained in* $\mathfrak{Q}_{R(P)}/_{\equiv_{iso}}$ *are either all compact, or all non-compact. Thus the property of compactness is well-defined for* $\mathfrak{Q}_{R(P)}/_{\equiv_{iso}}$*.*

Proof. Let P be a PA, $R \in \{\sim,\approx\}$ and $P' \in \mathfrak{Q}_{R(P)}/_{\equiv_{iso}}$. P' is compact if and only if P is compact, as the sets $S_R(s,a)$ are unique up to isomorphism. □

In the following we will only consider compact PA and therefore also compact sets of quotients. In metric spaces, compactness is equivalent to sequence compactness.

Lemma 9. $(\mathfrak{Q}_{\sim(P)}/_{\equiv_{iso}},\subseteq)$ *is a poset. If P is compact, it is even a lattice with union (intersection) of automata as join (meet) operations. Intersections are also possible over arbitrary sets of bisimilar automata.*

Proof. By Krein-Milman Theorem (Theorem 2) we know that for countable state spaces the sets $S_\sim(s,a)$ (seen as subsets of l^∞) have a unique set of extreme points $\mathbb{E}(S_\sim(s,a))$. This set is included in every bisimilar automaton, therefore in every intersection. This is the reason why the intersection of all automata in a set of bisimilar automata still leads to a bisimilar automaton. Transitions leading to some distribution in $S_\sim(s,a)$ that is not extreme do not change the bisimilarity. This is the reason why the union of two bisimilar automata leads to a bisimilar automaton. Finally observe that the unreachable fraction of states and transitions does not play any role for bisimilarity. □

In the strong case, a minimal set of transitions leading to extreme points can be chosen from the strong transitions emanating from s. In the weak case this does not have to be the case, as an extreme point might be reached transitively via some intermediate distributions.

Astonishingly, there's a similar statement also for weak bisimilarity, when only rescaled automata are considered. The interesting point is that the arguments of the proof of Lemma 9 do not apply one-to-one, as the extreme points can now also be weak transitions. Therefore it is (with the arguments of the proof for strong bisimulation) not clear that the intersection leads again to a weakly bisimilar PA.

Lemma 10. *$(\mathfrak{Q}_{\approx(P)}/_{\equiv_{iso}}, \subseteq)$ is a poset. When $\mathfrak{Q}_{\approx(P)}/_{\equiv_{iso}}$ is compact, it is even a lattice with union (intersection) of automata as join (meet) operations. Intersections are also possible over arbitrary sets of bisimilar automata.*

Proof. The meet operation for *finite* PA is justified by Lemma 12 in [4] and the observation that the unreachable fraction of states and transitions does not play any role for bisimilarity. The join operation is clear by definition of weak bisimilarity and the same observation.

For the infinite case assume for the rest of the proof that there are two quotients P_1 and P_2, $P_1 \approx P_2$ and we identify the (countable) reachable state spaces S_1 and S_2 as in Def. 17 by ${}^{S^\circ}/_R$ (note that this induces a bijection).

Now fix one state $s \in {}^{S^\circ}/_\approx$. By assumption it must have two sets of emanating a-transitions – one in P_1, one in P_2. During the proof we use the notation $S^i_R(s,a)$, $i \in \{1,2\}$ to denote the sets of reachable distributions of automaton 1 and 2, respectively. We have to show that already the intersection of both sets is enough to generate $S_\approx(s,a)$. The main difference to the strong case is that we only know that $S^1_\approx(s,a) = S^2_\approx(s,a)$ for all $a \in Act$, but this does not necessarily imply that also $S^1_\sim(s,a) = S^2_\sim(s,a)$. According to Def. 19 both automata P_1 and P_2 are rescaled. We consider the one-step-transitions $T^i_a = \{\mu \in Dist({}^{S^\circ}/_R) | (s,a,\mu) \in T_i\}$. Assume further that the identical transitions (modulo bijection) already have been identified (in ${}^{S^\circ}/_R \times Act \times Dist({}^{S^\circ}/_R)$) and are denoted $T^\cap_a = T^1_a \cap T^2_a$ and let $T_a^{\neg\cap_i} = T^i_a \setminus T^\cap_a$. We will show that the identical transitions are enough to cover all the behaviour, i.e. all transitions in $T_a^{\neg\cap_i}$, $i \in \{1,2\}$, can be omitted.

We will first consider the case $a = \tau$. We fix an arbitrary $\mu \in T_\tau^{\neg\cap_1}$ for which we show that it can be generated by transitions from $T^\cap_\tau$. It is clear that both sets T^1_τ and T^2_τ generate the set $S_\approx(s,\tau)$ (in the sense that all weak rescaled transitions must start with a combination of these transitions as first step).

With the construction from the proof of Lemma 12 in [4] we get a weak transition[1] in $T_1 \setminus \{(s,\tau,\mu)\}$ such that $s \Rightarrow_C \mu$. Note that the construction in [4] has been given for *finite* state spaces, but all constructions there are also possible in the countable compact case due to sequential compactness. We split this transition in its first-step-probability $\mu^{(1)}$, such that $s \to_C \mu^{(1)} \Rightarrow_C \mu$. We see that without loss of generality we may reduce the set T_1 to the set $T_1 \setminus \{(s,\tau,\mu)\}$ without losing bisimilarity. With this construction we proceed as follows: Pick now some other transition (s,τ,μ') with $\mu' \in T_\tau^{\neg\cap_1} \setminus \{\mu\}$ that is used somewhere in the weak transition $s \to_C \mu^{(1)} \Rightarrow_C \mu$. Now, again by the construction in the proof of Lemma 12 in [4], substitute all strong τ transitions from s to μ' by weak transitions in $T_1 \setminus \{(s,\tau,\mu),(s,\tau,\mu')\}$ such that $s \to_C \mu^{(2)} \Rightarrow_C \mu^{(1)} \Rightarrow_C \mu$ (possibly $\mu^{(2)} = \mu^{(1)}$ when we substituted some transition not taken in the first step). Continuing in the same way yields a sequence of combined one-step-transitions leading to the series of distributions $(\mu^{(i)})_{i\in\mathbb{N}}$.

Now compactness comes into play: This sequence must have limit points *in* $S_\approx(s,\tau)$. If there is one limit point already in $T^\cap_\tau$, by construction there will be a scheduler that uses only transitions from $T^\cap_\tau$

[1] The proof in [4] shows that a transition $s \xrightarrow{\tau} \mu$ – there denoted $s \xrightarrow{\tau} \nu_s$ – is redundant, as long as it is not in the intersection $T^\cap_\tau$, by constructing the above mentioned transition. The only case where it is not redundant leads to a contradiction to the quotient property.

when leaving s. If there is a limit point μ^* in $T_\tau^{\neg\cap_1}$, there is a non-trivial loop $\mu^* \Rightarrow_c \gamma^* \Rightarrow_c \mu^*$ for some $\gamma^* \in T_\tau^{\neg\cap_2}$ which would contradict the quotient property as it would render at least two states bisimilar. Therefore we conclude that this case cannot happen and μ can be omitted from $T_\tau^{\neg\cap_1}$ without losing bisimilarity.

As μ was chosen arbitrarily, this construction can be done for all elements in $T_\tau^{\neg\cap_1}$ – even if they might be uncountably many – (and analogously for $T_\tau^{\neg\cap_2}$). Therefore all transitions in $T_\tau^{\neg\cap_1}$ (and analogously for $T_\tau^{\neg\cap_2}$) may be omitted without losing bisimilarity.

For the case $a \neq \tau$ we may omit those strong a-transitions that can be mimicked by a weak a transition (after leaving out this strong transition). Observe that there cannot be cyclic relations "s uses the a transition of t (with probability 1)" and vice versa, because then s and t would be bisimilar, which is a contradiction to the quotient property.

For the general case of an arbitrary intersection (i.e. possibly more than two automata) observe the following: Let A be an arbitrary set of weakly bisimilar quotients. We have to show that the automaton $P_0 = \cap_{P\in A} P$ is still bisimilar. Pick an arbitary automaton $P \in A$ we have to show that all transitions of P that are not in P_0 are redundant. We may find for every such transition tr a bisimilar automaton P_{tr} that does not have such a transition (for otherwise the transition would have to be in the intersection). By the above procedure for two automata we show that the transition tr is redundant. This can be done for all other transitions not in P_0. □

Remark 5. *The crucial point in the proof of Lemma 10 is compactness. For general quotients a sequence of redundant distributions may not have a limit in $S_\approx(s,\tau)$ (This would be e.g. the distribution $(1,\tau,\Delta_3)$ in Remark 4). Sequential compactness ensures that there exists such a limit in $S_\approx(s,\tau)$.*

As we have shown that the intersection of arbitrary sets of bisimilar automata is again bisimilar by Lemma 9 and 10, the following theorem is immediate. It extends Theorem 1 of [4] to the compact automata case.

Theorem 3. *Let P be compact wrt. R. Then $(\mathfrak{Q}_R(P)/\equiv_{iso}, \subseteq)$ has a unique minimal element.*

Definition 20. *Let P be compact wrt. R, $R \in \{\sim, \approx\}$. Then there is a well-defined mapping $\mathcal{N} : PA \to {}^{PA}/\equiv_{iso}$, given by $P \mapsto \bot(\mathfrak{Q}_R(P)/\equiv_{iso})$ that assign to every PA P the minimal element in $\mathfrak{Q}_R(P)/\equiv_{iso}$. The mapping to $\mathfrak{Q}_\sim(P)/\equiv_{iso}$ is just quotienting, while the mapping to $\mathfrak{Q}_\approx(P)/\equiv_{iso}$ is quotienting followed by rescaling. The mapping $\mathcal{N}$ (for R fixed) is called* normal form.

This definition corresponds to the normal form definition given in [4].

Corollary 2. *The notations for normal forms defined in Def. 20 are normal forms in the sense of [4].*

As the bisimilarity only considers the reachable fraction of the state space, the following corollary is immediate.

Corollary 3. *Lemma 9, Lemma 10 and Theorem 3 also hold for $\mathfrak{Q}^*_\sim(P)/\equiv_{iso}$ instead of $\mathfrak{Q}_\sim(P)/\equiv_{iso}$ and $\mathfrak{Q}^*_\approx(P)/\equiv_{iso}$ instead of $\mathfrak{Q}_\approx(P)/\equiv_{iso}$.*

Some of the lattices we have constructed above have the appealing property of being bounded.

Lemma 11. *The lattices $(\mathfrak{Q}^*_\sim(P)/\equiv_{iso}, \subseteq)$ and $(\mathfrak{Q}^*_\approx(P)/\equiv_{iso}, \subseteq)$ are bounded.*

Proof. The lower bound is clearly given by the normal forms defined above. So it remains to show that there is also an upper bound. Let $P = (S, Act, T, s_0)$, for $(\mathfrak{Q}^*_\sim(P)/\equiv_{iso}, \subseteq)$ and $(\mathfrak{Q}^*_\approx(P)/\equiv_{iso}, \subseteq)$ we know that it is sufficient to consider $(\mathfrak{Q}^*_\sim(r(P))/\equiv_{iso}, \subseteq)$ and $(\mathfrak{Q}^*_\approx(r(P))/\equiv_{iso}, \subseteq)$. As S is countable, also the set of reachable states will be countable. As there is no restriction on T wrt. finiteness, the union over all transition relations in $\mathfrak{Q}^*_R(P)/\equiv_{iso}$ (identified by R) will be a valid transition relation (i.e. $\cup_R$ of all transition relations of quotients in the set $\mathfrak{Q}^*_R(P)/\equiv_{iso}$). □

Lemma 12. *For a finite automaton P,*

1. *the maximal elements in* $(\mathfrak{Q}^*_{\sim}(P)/_{\equiv_{iso}}, \subseteq)$ *and* $(\mathfrak{Q}^*_{\approx}(P)/_{\equiv_{iso}}, \subseteq)$ *are not necessarily finite.*
2. *the minimal elements in* $(\mathfrak{Q}^*_{\sim}(P)/_{\equiv_{iso}}, \subseteq)$ *and* $(\mathfrak{Q}^*_{\approx}(P)/_{\equiv_{iso}}, \subseteq)$ *are finite.*

Proof. Assume that $P = (S, Act, T, s_0)$ is given where $S = \{A, B, C\}$. Assume further that transitions (A, τ, Δ_B) and (A, τ, Δ_C) exist. Then every $(A, \tau, c\Delta_B \oplus (1-c)\Delta_C)$ would also be a bisimilar transition for every $c \in (0,1)$. Thus, the number of such distributions is uncountably infinite. The union of all those transitions would therefore lead to a non-finite automaton. □

Corollary 4. *For a finite automaton P, the minimal elements in* $(\mathfrak{Q}_{\sim}(P)/_{\equiv_{iso}}, \subseteq)$ *and* $(\mathfrak{Q}_{\approx}(P)/_{\equiv_{iso}}, \subseteq)$ *are finite.*

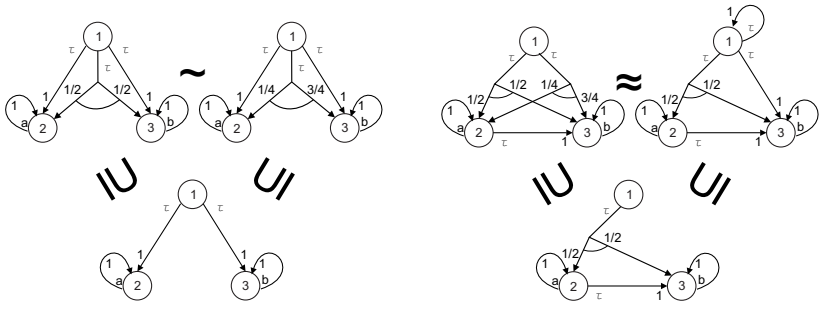
Lemma 13. *The lattices* $(\mathfrak{Q}_{\sim}(P)/_{\equiv_{iso}}, \subseteq)$ *and* $(\mathfrak{Q}_{\approx}(P)/_{\equiv_{iso}}, \subseteq)$ *are unbounded whenever there is at least one action* $a \in Act$, $a \neq \tau$.

Proof. Let $P = (S, Act, T, s_0)$ and $a \in Act$, $a \neq \tau$. We can construct two unreachable non-bisimilar states s_1, s_2 by using the transition $s_1 \xrightarrow{a} \Delta_{s_1}$ which is not possible by s_2 (possibly $s_1 \sim s$ or $s_2 \sim s$ for some $s \in S$, then we would use the corresponding state(s) from S, not adding them by a disjunct union in the following construction). So without loss of generality we may assume that $P = (S \dot\cup \{s_1, s_2\}, Act, T \cup \{s_1 \xrightarrow{a} \Delta_{s_1}\})$ The further construction goes in two steps. The first step is to construct a countable set of 'fresh' distinguished unreachable states $U = \mathbb{N}$ in the following way. use a single transition $i \xrightarrow{\tau} \frac{1}{i}\Delta_{s_0} \oplus (1 - \frac{1}{i})\Delta_{s_1}$ for every $i \in U$ (again, if $i \sim s$ for some $s \in S$, we may assume that $s \in U$, but will not add s again to the state space). So we reach the bisimilar automaton $P' = (S \dot\cup U, Act, T \cup \{i \xrightarrow{\tau} \frac{1}{i}\Delta_{s_0} \oplus (1 - \frac{1}{i})\Delta_{s_1}\}_{i \in U}, s_0)$. It is easy to see that all states in U are not bisimilar. For the second step notice that the powerset of a countable is uncountable. For every such subset $A \subseteq \mathbb{N}$ we may choose a distribution μ_A where $\mu_A(s) > 0$ if $s \in A$, 0 otherwise. Now, we can construct (uncountably many) PA P'_A bisimilar to P where we add one additional unreachable state s_A with the transition $s_A \xrightarrow{\tau} \mu_A$. Each of those unreachable states would need a separate state in the maximal element of our lattice (leaving alone the states that are bisimilar to one state of S by chance), meaning that there would be an uncountable number of states in the maximal element. This is a contradiction to the countability of the state space. The weak case follows similarly. □

5 Examples

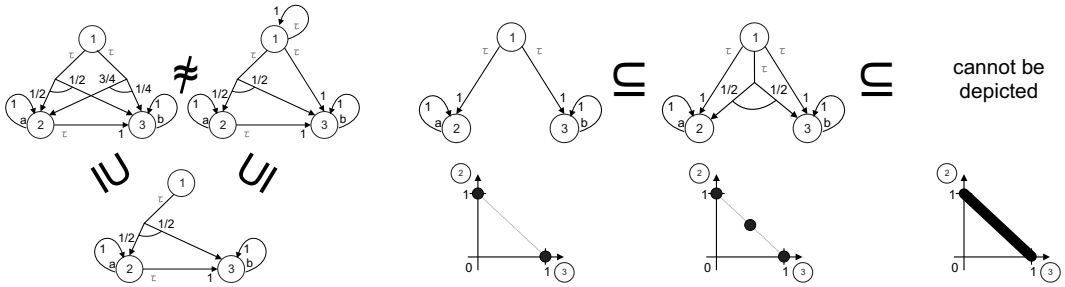
The first example is given in Fig. 1a. We start with the two strongly bisimilar automata at the top of the figure. The intersection of both is the automaton at the bottom of the figure.

The next example is given in Fig. 1b. We start with the two weakly bisimilar automata at the top of the figure Bisimilarity essentially stems from the following facts: Firstly, the transition $(1, \tau, \Delta_3)$ of the PA on the right can be mimicked by the left automaton by a Dirac determinate scheduler that in state 1 always chooses $(1, \tau, \frac{1}{2}\Delta_2 \oplus \frac{1}{2}\Delta_3)$ and in state 2 always $(2, \tau, \Delta_3)$ and stops in state 3. Secondly, the transition $(1, \tau, \frac{1}{4}\Delta_2 \oplus \frac{3}{4}\Delta_3)$ of the left automaton can be mimicked by the right automaton by choosing each of the τ-transitions emanating from state 1 with probability $\frac{1}{2}$. The intersection of both – which is the canonical form – is the automaton at the bottom of the figure.



(a) Strongly bisimilar case (b) Weakly bisimilar case

Figure 1: Intersection of bisimilar automata



(a) Non-bisimilar automata (b) The bounded case

Figure 2: Examples continued

The next example shows that it is essential to have elements of $\mathfrak{Q}_{\sim}(P)/\equiv_{iso}$, otherwise the intersection will not make sense. This example is given in Fig. 2a. It is clear that e.g. the distribution $\frac{3}{4}\Delta_2 \oplus \frac{1}{4}\Delta_3$ cannot be realised starting from state 1 by the right automaton, but it can be realised by the left automaton, so both cannot be bisimilar. Therefore, the intersection of both automata is not bisimilar to the left automaton.

5.1 A bounded example

Assume that we consider strong bisimulation and want to calculate $\mathfrak{Q}^*_{\sim}(P)/\equiv_{iso}$ for the automaton P given in Fig. 2b (middle). The least element (let's call it $\bot$) is shown on the left, the greatest element cannot be adequately depicted, as it has (uncountably) infinitely many transitions. The situation becomes clearer when (as suggested in [2]) considering distributions as points in $\mathbb{R}^n$. This is done in the lower line of the figure. We only show the distributions that are possible by Dirac determinate schedulers starting from state 1. As there are only two successor states, $\mathbb{R}^2$ suffices for our purpose. With this picture in mind it is clear that the greatest element is the PA given by $\top = (\{1,2,3\}, \{\tau, a, b\}, \{(2,a,\Delta_2),(3,b,\Delta_3)\} \cup \{(1,\tau,c\Delta_2 \oplus (1-c)\Delta_3) | c \in [0,1]\}, 1)$. Clearly this is not a finite PA. Generalising from this automaton, for $\mathfrak{A} \subseteq [0,1]$ we may define $P_{\mathfrak{A}} := (\{1,2,3\}, \{\tau, a, b\}, \{(2,a,\Delta_2),(3,b,\Delta_3)\} \cup \{(1,\tau,c\Delta_2 \oplus (1-c)\Delta_3) | c \in \mathfrak{A}\}, 1)$. Note that $\bot = P_{\{0,1\}}$, $\top = P_{[0,1]}$. But now it is clear how the set $\mathfrak{Q}^*_{\sim}(P)/\equiv_{iso}$ must look like: $\mathfrak{Q}^*_{\sim}(P)/\equiv_{iso} = \{P_{\mathfrak{A}} | \mathfrak{A} \subseteq [0,1], 0 \in \mathfrak{A}, 1 \in \mathfrak{A}\}$ Leaving out 0 or 1 from $\mathfrak{A}$ will break bisimilarity.

6 Conclusion

This paper extends the notion of normal forms introduced in [4] to the case of compact automata with countably infinite state space, countably infinite set of actions and possibly uncountably many transitions. We have justified the canonicity of normal forms by introducing them as the intersection of all bisimilar automata, not from an abstract point of view as in [4]. The structure presented is nice as a theoretical result, but (at least for the moment) there is no immediate practical applicability, as the hard part is to construct the sets $\mathfrak{Q}_{\sim}(P)$ and $\mathfrak{Q}_{\approx}(P)$ where an uncountably infinite number of automata would have to be constructed. The structure itself is particularly nice for teaching purposes and for better understanding the possible 'shapes' of infinite automata.

References

[1] T. Brázdil, A. Kučera & O. Stražovský (2004): *Deciding Probabilistic Bisimilarity Over Infinite-State Probabilistic Systems*. In P. Gardner & N. Yoshida, editors: *CONCUR 2004*, *Lecture Notes in Computer Science* 3170, Springer Berlin Heidelberg, pp. 193–208, doi:10.1007/978-3-540-28644-8_13.

[2] S. Cattani & R. Segala (2002): *Decision Algorithms for Probabilistic Bisimulation*. In: *CONCUR 2002*, *Lecture Notes in Computer Science* 2421, Springer, pp. 371–385, doi:10.1007/3-540-45694-5_25.

[3] J. Desharnais, V. Gupta, R. Jagadeesan & P. Panangaden (2010): *Weak bisimulation is sound and complete for PCTL**. *Inf. Comput.* 208(2), pp. 203–219, doi:10.1016/j.ic.2009.11.002.

[4] C. Eisentraut, H. Hermanns, J. Schuster, A. Turrini & L. Zhang (2013): *The Quest for Minimal Quotients for Probabilistic Automata*. In: *TACAS 2013*, *Lecture Notes in Computer Science* 7795, Springer, pp. 16–31, doi:10.1007/978-3-642-36742-7_2.

[5] V. Forejt, P. Jancar, S. Kiefer & J. Worrell (2012): *Bisimilarity of Probabilistic Pushdown Automata*. In: *FSTTCS 2012*, *Leibniz International Proceedings in Informatics (LIPIcs)* 18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, pp. 448–460, doi:10.4230/LIPIcs.FSTTCS.2012.448. Available at `http://drops.dagstuhl.de/opus/volltexte/2012/3880`.

[6] H. Hermanns & A. Turrini (2012): *Deciding Probabilistic Automata Weak Bisimulation in Polynomial Time*. In: *FSTTCS 2012*, *Leibniz International Proceedings in Informatics (LIPIcs)* 18, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 435–447, doi:10.4230/LIPIcs.FSTTCS.2012.435. Available at `http://drops.dagstuhl.de/opus/volltexte/2012/3879/`.

[7] M. Krein & D. Milman (1940): *On extreme points of regular convex sets*. *Studia Mathematica* 9(1), pp. 133–138. Available at `http://eudml.org/doc/219061`.

[8] N. Lynch, R. Segala & F. Vaandrager (2003): *Compositionality for probabilistic automata*. In: *CONCUR 2003*, Springer, pp. 208–221, doi:10.1007/978-3-540-45187-7_14.

[9] N. A. Lynch, R. Segala & F. W. Vaandrager (2007): *Observing Branching Structure through Probabilistic Contexts*. *SIAM J. Comput.* 37(4), pp. 977–1013, doi:10.1147/S0097539704446487.

[10] A. Parma & R. Segala (2007): *Logical characterizations of bisimulations for discrete probabilistic systems*. In: *Foundations of Software Science and Computational Structures*, Springer, pp. 287–301, doi:10.1007/978-3-540-71389-0_21.

[11] R. Segala (1995): *Modeling and Verification of Randomized Distributed Real-Time Systems*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Available at `http://profs.sci.univr.it/~segala/www/phd.html`.

[12] R. Segala & N. A. Lynch (1995): *Probabilistic Simulations for Probabilistic Processes*. *Nord. J. Comput.* 2(2), pp. 250–273, doi:10.1007/BFb0015027.

A Finite Exact Representation of Register Automata Configurations

Yu-Fang Chen
Academia Sinica, Taiwan

Bow-Yaw Wang
Academia Sinica, Taiwan

Di-De Yen
Academia Sinica, Taiwan

A register automaton is a finite automaton with finitely many registers ranging from an infinite alphabet. Since the valuations of registers are infinite, there are infinitely many configurations. We describe a technique to classify infinite register automata configurations into finitely many *exact* representative configurations. Using the finitary representation, we give an algorithm solving the reachability problem for register automata. We moreover define a computation tree logic for register automata and solve its model checking problem.

1 Introduction

Register automata are generalizations of finite automata to process strings over infinite alphabets [9]. In addition to a finite set of states, a register automaton has finitely many registers ranging from an infinite alphabet. When a register automaton reads a data symbol with parameters from the infinite alphabet, it compares values of registers and parameters and finite constants, updates registers, and moves to a new location. Since register automata allow infinitely many values in registers and parameters, they have been used to model systems with unbounded data. For instance, a formalization of user registration and account management in the XMPP protocol is given in [1, 2]. Since user identifiers are not fixed *a priori*, models in register automata are more realistic for the protocol.

Analyzing register automata nonetheless is not apparent. Since there are infinitely many valuations of registers, the number of configurations for a register automaton is inherently infinite. Moreover, register automata can update a register with values of registers or parameters in data symbols. The special feature makes register automata more similar to programs than to classical automata. Infinite configurations and register updates increase the expressive power of register automata. They also complicate analysis of the formalism as well.

In this paper, we develop a finitary representation for configurations of register automata. As observed in [6], register automata recognize strings modulo automorphisms on the infinite alphabet. That is, a string is accepted by a register automaton if and only if the image of the string under a one-to-one and onto mapping on the infinite alphabet is accepted by the same automaton. Subsequently, two valuations of registers are indistinguishable by register automata if one is the image of the other under an automorphism on the infinite alphabet. We therefore identify indistinguishable valuations and classify valuations into finitely many representative valuations. Naturally, our finitary representation enables effective analysis on register automata.

The first application of representative valuations is reachability analysis. Based on representative valuations, we define representative configurations. Instead of checking whether a given configuration is reachable in a register automaton, it suffices to check whether its representative configuration belongs to the finite set of reachable representative configurations. We give an algorithm to compute successors of an arbitrary representative configuration. The set of reachable representative configurations is obtained by fixed point computation.

L. Clemente and L. Holik (Eds.): 15th International Workshop on Verification of Infinite-State Systems (INFINITY'13)
EPTCS 140, 2014, pp. 16–34, doi:10.4204/EPTCS.140.2

Our second application is model checking on register automata. We define a computation tree logic (CTL) for register automata. Configurations in a representative configuration are shown to be indistinguishable in our variant of computation tree logic. The CTL model checking problem for register automata thus is solved by the standard algorithm with slight modifications.

As an illustration, we model an algorithm for the Byzantine generals problem under an interesting scenario. In the scenario, two loyal generals are trying to reach a consensus at the presence of a treacherous general. They would like to know how many soldiers should be sent to the front line. Since the total number of soldiers is unbounded,[1] we use natural numbers as the infinite alphabet and model the algorithm in a register automaton. By the CTL model checking algorithm, we compute the initial configurations leading to a consensus eventually.

Our formulation of register automata follows those in [1, 2]. It is easy to show that the expressive power of register automata with constant symbols is no difference from those versions without constants. A canonical representation theorem similar to Myhill-Nerode theorem for deterministic register automata is developed in [1]. In [2], a learning algorithm for register automata is proposed. Finite-memory automata is another generalization of finite automata to infinite alphabets [6]. Finite-memory automata and register automata have the same expressive power. In [6], we know that the emptiness problem for finite-memory automata is decidable. Therefore, the reachability problem for register automata is also decidable. In [4], it has been shown that the emptiness for register automata is in *PSPACE*. This is done by reducing an emptiness checking problem for register automata to an emptiness problem of a finite transition system over the so called "abstract states", which is very similar to the "equivalence classes" defined in this paper. However, in their reduction, they did not provide any algorithm to move from one abstract state to another abstract state, which is in fact non-trivial. In contrast, we provide an algorithm in Section 4. It has been shown in [5] that register automata together with a total order over the *alphabet* are equivalent to timed automata. In fact, the register automata model defined in this paper can be easily extended to support arbitrary order among alphabet symbols (the order can be partial) and hence is more general then the one defined in [5]. This because the finite representation of configurations defined in this paper can be extended to describe any finite relations between alphabet symbols by adding more possible values in the matrix. That is, instead of just 0 and 1 used in the current paper, we can add more possible values such as $\leq, <, >, \ldots$ to describe a richer relation between alphabet symbols. A survey on expressive power of various finite automata with infinite alphabets is given in [9]. We model the algorithm for the Byzantine generals problem [8] presented in [10].

The paper is organized as follows. We briefly review register automata in Section 2. Section 3 presents an exact finitary representation for configurations. It is followed by the reachability algorithm for register automata (Section 4). A computation tree logic for register automata and its model checking algorithm are given in Section 5. We discuss the Byzantine generals problem as an example (Section 6). Finally, we conclude the presentation in Section 7.

2 Preliminaries

Let S, S', and S'' be sets. An *automorphism* on S is a one-to-one and onto mapping from S to S. Given a subset T of S, an automorphism σ on S is invariant on T if $\sigma(x) = x$ for every $x \in T$. If f is an onto mapping from S to S' and h is a mappings from S' to S'', $(h \circ f)$ is a mapping from S to S'' that $(h \circ f)(a) = h(f(a))$ for $a \in S$. We write $S_{n\times n}$ for the set of square matrices of size n with entries in S.

[1] This is certainly an ideal simplification. The number of soldiers of course is bounded by the population of the empire.

Let Σ be an infinite *alphabet*. A set of constants, denoted by C, is a finite subset of Σ. Let A be a finite set of *actions*. Each action has a finite *arity*. A *data symbol* $\alpha(\bar{d_n})$ consists of an action $\alpha \in A$ and $\bar{d_n} = d_1 d_2 \cdots d_n \in \Sigma^n$ when α is of arity n. A *string* is a sequence of data symbols.

Fix a finite set X of *registers*. Define $X' = \{x' | x \in X\}$. A *valuation* v is a mapping from X to Σ. Since X is finite, we represent a valuation by a string of $\Sigma^{|X|}$. We write $V_{(X,\Sigma)}$ for the set of valuations from X to Σ.

Let $P = \{p_1, p_2, \ldots\}$ be an infinite set of *formal parameters* and $P_n = \{p_1, p_2, \ldots, p_n\} \subseteq P$. A *parameter valuation* $v_{\bar{d_n}}$ is a mapping from P_n to Σ such that $v_{\bar{d_n}}(p_i) = d_i$ for every $1 \leq i \leq n$. We write $V_{(P,\Sigma)}$ for the set of parameter valuations. Obviously, each finite sequence $\bar{d_n} \in \Sigma^n$ corresponds to a parameter valuation $v_{\bar{d_n}} \in V_{(P,\Sigma)}$.

Given a valuation v, a parameter valuation $v_{\bar{d_n}}$, and $e \in X \cup P_n \cup C$, define

$$[\![e]\!]_{v,v_{\bar{d_n}}} = \begin{cases} v(e) & \text{if } e \in X \\ v_{\bar{d_n}}(e) & \text{if } e \in P_n \\ e & \text{if } e \in C \end{cases}$$

Thus $[\![e]\!]_{v,v_{\bar{d_n}}}$ is the value of e on the valuation v, parameter valuation $v_{\bar{d_n}}$, or constant e.

An *assignment* π is of the form

$$(x_{k_1} x_{k_2} \ldots x_{k_n}) \mapsto (e_{l_1} e_{l_2} \ldots e_{l_n})$$

where $x_{k_i} \in X$, $e_{l_i} \in X \cup P_n \cup C$, and $x_{k_i} \neq x_{k_j}$ whenever $i \neq j$. Let Π denote the set of assignments. For valuation v and parameter valuation $v_{\bar{d_n}}$, define

$$[\![\pi]\!]_{v,v_{\bar{d_n}}} \triangleq \{v' | v'(x_{k_i}) = [\![e_{l_i}]\!]_{v,v_{\bar{d_n}}} \text{ for every } 1 \leq i \leq n\}.$$

That is, $[\![\pi]\!]_{v,v_{\bar{d_n}}}$ contains the valuations obtained by executing the assignment under the valuation v and parameter valuation $v_{\bar{d_n}}$.

An *atomic guard* is of the form $e = f$ or its negation $\neg(e = f)$ (written $e \neq f$) where $e, f \in X \cup P_n \cup C$. A *guard* is a conjunction of atomic guards. We write Γ for the set of guards. For any valuation v and parameter valuation $v_{\bar{d_n}}$, define

$$\begin{array}{ll} v, v_{\bar{d_n}} \models e = f & \text{if } [\![e]\!]_{v,v_{\bar{d_n}}} = [\![f]\!]_{v,v_{\bar{d_n}}} \\ v, v_{\bar{d_n}} \models e \neq f & \text{if } [\![e]\!]_{v,v_{\bar{d_n}}} \neq [\![f]\!]_{v,v_{\bar{d_n}}} \\ v, v_{\bar{d_n}} \models g_1 \wedge g_2 \wedge \cdots \wedge g_k & \text{if } v, v_{\bar{d_n}} \models g_i \text{ for every } 1 \leq i \leq k \end{array}$$

Definition 1. *A* register automaton *is a tuple* $(\Sigma, A, X, L, l_0, \Delta)$ *where*

- *A is a finite set of actions;*
- *L is a finite set of* locations*;*
- *$l_0 \in L$ is the* initial location*;*
- *X is a finite set of registers.*
- *$\Delta \subseteq L \times A \times \Gamma \times \Pi \times L$ is a finite set of* transitions*.*

A *configuration* $\langle l, v \rangle$ of a register automaton $(\Sigma, A, X, L, l_0, \Delta)$ consists of a location $l \in L$ and a valuation $v \in V_{(X,\Sigma)}$. For configurations $\langle l, v \rangle$ and $\langle l', v' \rangle$, we say $\langle l, v \rangle$ *transits* to $\langle l', v' \rangle$ on $\alpha(\bar{d_n})$ (written $\langle l, v \rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v' \rangle$) if there is a transition $(l, \alpha, g, \pi, l') \in \Delta$ such that $v, v_{\bar{d_n}} \models g$ and $v' \in [\![\pi]\!]_{v,v_{\bar{d_n}}}$.

A *run* of a register automaton $(\Sigma, A, X, L, l_0, \Delta)$ on a string $\alpha_0(\bar{d^0_{n_0}})\ \alpha_1(\bar{d^1_{n_1}}) \cdots\ \alpha_{k-1}(\bar{d^{k-1}_{n_{k-1}}})$ is a sequence of configurations $\langle l_0, v_0 \rangle\ \langle l_1, v_1 \rangle \cdots\ \langle l_k, v_k \rangle$ such that $\langle l_i, v_i \rangle \xrightarrow{\alpha(\bar{d^i_{n_i}})} \langle l_{i+1}, v_{i+1} \rangle$ for every $0 \leq i < k$.

Example 1. *Let $\mathbb{N}$ denote the set of natural numbers, $\Sigma = \mathbb{N}$, $A = \{\alpha, \beta\}$, $L = \{l_0, l_1\}$, $C = \{2\}$, and $X = \{x_1, x_2\}$ where α and β have arities 2 and 1 respectively. Consider the register automaton in Figure 1. In the figure, $\frac{\alpha|g}{\pi}$ denotes a transition with action α, guard g, and assignment π. Here is a run of the automaton:*

$$\langle l_0, 77\rangle \xrightarrow{\alpha(1,3)} \langle l_1, 13\rangle \xrightarrow{\beta(1)} \langle l_1, 13\rangle \xrightarrow{\beta(2)} \langle l_1, 23\rangle \xrightarrow{\beta(1)} \langle l_0, 69\rangle$$

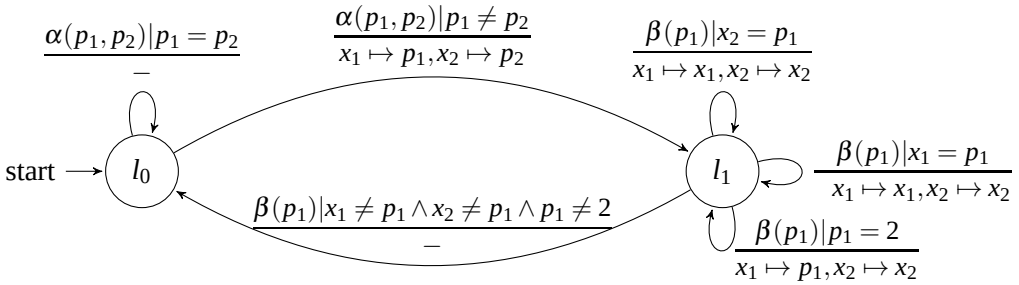


Figure 1: A Register Automaton

Let $(\Sigma, A, X, L, l_0, \Delta)$ be a register automaton. A configuration $\langle l, v\rangle$ is *reachable* if there is a run $\langle l_0, v_0\rangle\langle l_1, v_1\rangle \cdots \langle l_k, v_k\rangle$ of $(\Sigma, A, X, L, l_0, \Delta)$ with $\langle l_k, v_k\rangle = \langle l, v\rangle$. The *reachability* problem for register automata is to decide whether a given configuration is reachable in a given register automaton.

Definition 2 ([7])**.** *An* equality logic formula *is defined as follows.*

$$\begin{array}{rcl} \phi & : & \phi \wedge \phi \mid \neg\phi \mid \phi \implies \phi \mid var = var \\ var & : & x \mid x' \mid p \mid c \end{array}$$

where $x \in X$, $x' \in X'$, $p \in P$, and $c \in C$.

Note that a guard is also an equality logic formula. An equality logic formula ϕ is *valid* if ϕ always evaluates to true by assigning each member of $X \cup X' \cup P$ with an arbitrary element in Σ. We write $\vdash \phi$ when ϕ is valid. The formula ϕ is *consistent* if it is not the case that $\vdash \neg\phi$. Given an equality logic formula ϕ, the *validity* problem for equality logic is to decide whether $\vdash \phi$.

Theorem 1 ([7])**.** *The validity problem for equality logic is coNP-complete.*

3 Representative Configurations

Consider a register automaton $(\Sigma, A, X, L, l_0, \Delta)$. Since Σ is infinite, there are an infinite number of valuations in $V_{(X,\Sigma)}$. A register automaton subsequently has infinitely many configurations. In this section, we show that configurations can be partitioned into finitely many classes. Any two configurations in the same class are indistinguishable by register automata.

Definition 3. *Let $u, v \in V_{(X,\Sigma)}$. u is* equivalent *to v with respect to C (written $u \sim_C v$) if there is an automorphism σ on Σ such that σ is invariant on C and $(\sigma \circ u)(x) = v(x)$ for every $x \in X$.*

For example, let $\Sigma = \mathbb{N}$, $X = \{x_1, x_2, x_3\}$, $C = \{1\}$, $v_1 = 123$, $v_2 = 134$, and $v_3 = 523$. We have $v_1 \sim_C v_2$ but $v_1 \not\sim_C v_3$.

It is easy to see that $\sim_C$ is an equivalence relation on $V_{(X,\Sigma)}$. For any valuation $v \in V_{(X,\Sigma)}$, we write $[v]$ for the equivalence class of v. That is,

$$[v] \triangleq \{u \in V_{(X,\Sigma)} | u \sim_C v\}.$$

The equivalence class $[v]$ is called a *representative valuation*. Note that there are only finitely many representative valuations for X is finite.

Definition 4. *A* representative configuration $\langle l, [v] \rangle$ *is a pair where* $l \in L$ *and* $[v]$ *is a representative valuation.*

Since X and L are finite sets, the number of representative configurations is finite. Our next task is to show that every configurations in a representative configuration behave similarly. Let $\langle l, [v] \rangle$ and $\langle l', [v'] \rangle$ be two representative configurations. Define $\langle l, [v] \rangle \rightsquigarrow \langle l', [v'] \rangle$ if

- for each $u \in [v]$, there is a valuation $u' \in [v']$ and a data symbol $\alpha(\bar{d_n})$ such that $\langle l, u \rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', u' \rangle$; and
- for each $u' \in [v']$, there is a valuation $u \in [v]$ and a data symbol $\alpha(\bar{d_n})$ such that $\langle l, u \rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', u' \rangle$.

Let $\langle \Sigma, A, X, L, l_0, \Delta \rangle$ be a register automaton and $\langle l_k, [v_k] \rangle$ a representative configuration. We say $\langle l_k, [v_k] \rangle$ is *reachable* if there is a sequence of representative configurations $\langle l_0, [v_0] \rangle \langle l_1, [v_1] \rangle \cdots \langle l_k, [v_k] \rangle$ such that $\langle l_i, [v_i] \rangle \rightsquigarrow \langle l_{i+1}, [v_{i+1}] \rangle$ for every $0 \leq i < k$. The following three propositions are useful to our key lemma.

Proposition 1. *Let* $v \in V_{(X,\Sigma)}$ *be a valuation,* $v_{\bar{d_n}} \in V_{(P,\Sigma)}$ *a parameter valuation, and* $g \in \Gamma$ *a guard.* $v, v_{\bar{d_n}} \models g$ *if and only if* $\sigma \circ v, \sigma \circ v_{\bar{d_n}} \models g$ *for every automorphism* σ *on* Σ *which is invariant on* C.

Proposition 2. *Let* $v, w \in V_{(X,\Sigma)}$ *be valuations,* $v_{\bar{d_n}} \in V_{(P,\Sigma)}$ *a parameter valuation, and* $\pi \in \Pi$ *an assignment.* $w \in [\![\pi]\!]_{v, v_{\bar{d_n}}}$ *if and only if* $\sigma \circ w \in [\![\pi]\!]_{\sigma \circ v, \sigma \circ v_{\bar{d_n}}}$ *for every automorphism* σ *on* Σ *which is invariant on* C.

Proposition 3. *Let* $(\Sigma, A, X, L, l_0, \Delta)$ *be a register automaton,* $l, l' \in L$ *locations,* $v, v' \in V_{(X,\Sigma)}$ *valuations, and* $\alpha(\bar{d_n})$ *a data symbol with* $\bar{d_n} = d_1 d_2 \cdots d_n$. *If* $\langle l, v \rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v' \rangle$, *then* $\langle l, \sigma \circ v \rangle \xrightarrow{\alpha(\sigma(\bar{d_n}))} \langle l', \sigma \circ v' \rangle$ *for every automorphism* σ *on* Σ *which is invariant on* C, *where* $\sigma(\bar{d_n}) \triangleq \sigma(d_1)\sigma(d_2)\cdots\sigma(d_n)$.

By Proposition 3, we get the following key lemma.

Lemma 1. *Let* $(\Sigma, A, X, L, l_0, \Delta)$ *be a register automaton,* $l, l' \in L$, *and* $v, v' \in V_{(X,\Sigma)}$. $\langle l, v \rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v' \rangle$ *for some* $\alpha(\bar{d_n})$ *if and only if* $\langle l, [v] \rangle \rightsquigarrow \langle l', [v'] \rangle$.

Lemma 1 shows that representative configurations are exact representations for configurations with respect to transitions. The configuration $\langle l, v \rangle$ transits to another configuration $\langle l', v' \rangle$ in one step precisely when their representative configurations have a transition. There are however infinitely many valuations. In order to enumerate $[v]$ effectively, we use a matrix-based representation.

Let $[\underline{v}]$ be a representative valuation with $v \in V_{(X,\Sigma)}$. Assume $\{\overline{0}, \overline{1}\} \cap \Sigma = \emptyset$. A *representative matrix* $R_{[v]} \in (\{\overline{0}, \overline{1}\} \cup C)_{|X| \times |X|}$ of $[v]$ is defined as follows.

$$(R_{[v]})_{ij} \triangleq \begin{cases} \underline{v}(x_i) & \text{if } v(x_i) = v_(x_j) \in C \\ \overline{1} & \text{if } v(x_i) = v(x_j) \notin C \\ \overline{0} & \text{otherwise} \end{cases}$$

Let $v \in V_{(X,\Sigma)}$ be a valuation. The entry $(R_{[v]})_{ij}$ denotes the equality relation among registers x_i, x_j, and constant c for every $c \in C$. If $v(x_i) = v(x_j)$, $(R_{[v]})_{ij} \in \{\overline{1}\} \cup C$; otherwise, $(R_{[v]})_{ij} = \overline{0}$; moreover, if $v(x_i) = c \in C$, $(R_{[v]})_{ii} = c$. The following proposition shows that $R_{[v]}$ is well-defined.

Proposition 4. *For any $u, v \in V_{(X,\Sigma)}$, $[u] = [v]$ if and only if $R_{[u]} = R_{[v]}$.*

By Proposition 4, we will also call $R_{[v]}$ a representative valuation and write $R_{[v]}$ for $[v]$. Subsequently, $\langle l, R_{[v]} \rangle \rightsquigarrow \langle l', R_{[v']} \rangle$ if and only if $\langle l, [v] \rangle \rightsquigarrow \langle l', [v'] \rangle$.

Example 2. *By example 1, we have $v_0 = 77$, $v_1 = v_2 = 13$, $v_3 = 23$, $v_4 = 69$ and $R_{[v_0]} = \begin{pmatrix} \overline{1} & \overline{1} \\ \overline{1} & \overline{1} \end{pmatrix}, R_{[v_1]} = R_{[v_2]} = R_{[v_4]} = \begin{pmatrix} \overline{1} & \overline{0} \\ \overline{0} & \overline{1} \end{pmatrix}, R_{[v_3]} = \begin{pmatrix} 2 & \overline{0} \\ \overline{0} & \overline{1} \end{pmatrix}$. Hence, $\langle l, R_{[v_0]} \rangle \rightsquigarrow \langle l', R_{[v_1]} \rangle \rightsquigarrow \langle l', R_{[v_2]} \rangle \rightsquigarrow \langle l', R_{[v_3]} \rangle \rightsquigarrow \langle l', R_{[v_4]} \rangle$.*

Every representative valuation corresponds to a matrix. However, not every matrix has a corresponding representative valuation. For instance, the zero matrix $(\overline{0}) \in \{\overline{0}, \overline{1}\}_{1\times 1}$ does not correspond to any representative valuation. If $(\overline{0}) = R_{[v]}$ for some valuation v, one would have the absurdity $v(x_1) \neq v(x_1)$. Such matrices are certainly not of our interests and should be excluded.

For any $R \in (\{\overline{0}, \overline{1}\} \cup C)_{|X|\times|X|}$, define the equality logic formula $E(R)$ as follows.

$$E(R) \stackrel{\triangle}{=} \bigwedge_{R_{ij} \in C} (x_i = x_j \wedge x_i = R_{ij}) \wedge \bigwedge_{R_{ij} = \overline{1}} (x_i = x_j \wedge \bigwedge_{c \in C} x_i \neq c) \wedge \bigwedge_{R_{ij} = \overline{0}} x_i \neq x_j$$

Idea: If we do not add the equalities of form $x_i = c \in C$ for some i or the inequalities $x_i \neq c \in C$ for some i to the conjunction $E(R)$, we can not distinguish the following four kinds of matrices:
(1) $\begin{pmatrix} c & \overline{1} \\ \overline{1} & c \end{pmatrix}$ (2) $\begin{pmatrix} c & \overline{1} \\ c & c \end{pmatrix}$ (3) $\begin{pmatrix} c & c \\ \overline{1} & c \end{pmatrix}$ (4) $\begin{pmatrix} c & c \\ c & c \end{pmatrix}$
The fourth kind of matrix is the matrix we hope for.

We say the matrix R is *consistent* if $E(R)$ is consistent. It can be shown that a consistent matrix is also a representative matrix. Indeed, Algorithm 1 computes a valuation v such that $R_{[v]} = R$ for any consistent matrix R.

Algorithm 1 starts from a valuation where the register x_i is assigned to R_{ii} for every $R_{ii} \in C$, the rest of registers are assigned to distinct elements in $\Sigma \setminus C$. It goes through entries of the given consistent matrix R by rows. At row i, the algorithm assigns $w(x_i)$ to the register x_j if $R_{ij} \in \{\overline{1}\} \cup C$. Hence the first i rows of R are equal to the first i rows of $R_{[w]}$ after iteration i. When Algorithm 1 returns, we obtain a valuation whose representative matrix is R.

Lemma 2. *Let $R \in (\{\overline{0}, \overline{1}\} \cup C)_{|X|\times|X|}$ be a consistent matrix and $w = \mathsf{CanonicalVal}(R)$. $R = R_{[w]}$.*

For a consistent matrix R, the valuation computed by $\mathsf{CanoncalVal}(R)$ is called the *canonical valuation* of R. The following lemma follows from Lemma 2.

Lemma 3. *Let $R \in (\{\overline{0}, \overline{1}\} \cup C)_{|X|\times|X|}$. R is consistent if and only if $R = R_{[v]}$ for some $v \in V_{(X,\Sigma)}$.*

By Lemma 3, it is now straightforward to enumerate all representative matrices. Algorithm 2 computes the set of all representative matrices.

4 Reachability

Let $(\Sigma, A, X, L, l_0, \Delta)$ be a register automaton and $\langle l, v \rangle$ a configuration with $l \in L$ and $v \in V_{(X,\Sigma)}$. In order to solve the reachability problem for register automata, we show how to compute all $\langle l', R_{[v']} \rangle$ such that $\langle l, R_{[v]} \rangle \rightsquigarrow \langle l', R_{[v']} \rangle$.

// $c_1, c_2, \ldots, c_{|X|}$ are distinct elements in $\Sigma \setminus C$
Input: R : a consistent matrix
Output: $w \in V_{(X,\Sigma)} : R = R_{[w]}$
foreach $1 \leq i \leq |X|$ **do**
  if $R_{ii} \in C$ **then**
    $w(x_i) \leftarrow R_{ii}$;
  else
    $w(x_i) \leftarrow c_i$;
  end
end
foreach $i = 1$ **to** $|X| - 1$ **do**
  foreach $j = i + 1$ **to** $|X|$ **do**
    if $R_{ij} \in \{\overline{1}\} \cup C$ **then** $w(x_j) \leftarrow w(x_i)$;
  end
end
return w;

Algorithm 1: CanonicalVal(R)

Output: $\mathscr{R} : \mathscr{R} = \{R_{[v]} : v \in V_{(X,\Sigma)}\}$
$\mathscr{R} \leftarrow \emptyset$;
foreach *matrix* $R \in (\{\overline{0}, \overline{1}\} \cup C)_{|X| \times |X|}$ **do**
  if R *is consistent* **then** $\mathscr{R} \leftarrow \mathscr{R} \cup \{R\}$;
end
return $\mathscr{R}$;

Algorithm 2: UniverseR(X)

By Lemma 1, $\langle l, R_{[v]}\rangle \rightsquigarrow \langle l', R_{[v']}\rangle$ if $\langle l, v\rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v'\rangle$ for some $\alpha(\bar{d_n})$. A first attempt to find $\langle l', R_{[v']}\rangle$ with $\langle l, R_{[v]}\rangle \rightsquigarrow \langle l', R_{[v']}\rangle$ is to compute all $\langle l', v'\rangle$ with $\langle l, v\rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v'\rangle$ for some $\alpha(\bar{d_n})$. The intuition however would not work. Since Σ is infinite, there can be infinitely many data symbols $\alpha(\bar{d_n})$ and valuations v' with $\langle l, v\rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v'\rangle$. It is impossible to enumerate them.

Instead, we compute $\langle l', R'\rangle$ with $\langle l, R_{[v]}\rangle \rightsquigarrow \langle l', R'\rangle$ directly. Based on equality relations among registers in the given configuration $\langle l, v\rangle$, we infer equality relations among registers in a configuration $\langle l', v'\rangle$ with $\langle l, v\rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v'\rangle$. Since there are finitely many representative matrices, we enumerate those representative matrices conforming to the inferred equality relations among registers. The conforming representative matrices give desired representative configurations.

We start with extracting equality relations among registers in the given configuration $\langle l, v\rangle$. For any valuation $v \in V_{(X,\Sigma)}$, define

$$E(v) \stackrel{\triangle}{=} \bigwedge_{v(x)=c\in C} x = c \wedge \bigwedge_{v(x)=v(y)} x = y \wedge \bigwedge_{v(x)\neq v(y)} x \neq y, \text{ and}$$

$$E'(v) \stackrel{\triangle}{=} \bigwedge_{v(x)=c\in C} x' = c \wedge \bigwedge_{v(x)=v(y)} x' = y' \wedge \bigwedge_{v(x)\neq v(y)} x' \neq y'.$$

Let (l, α, g, π, l') be a transition and $\langle l, v\rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v'\rangle$. Equality relations among registers in $\langle l', v'\rangle$ are determined by the assignment π. Let $\pi = (x_{k_1} x_{k_2} \ldots x_{k_n}) \mapsto (e_{l_1} e_{l_2} \ldots e_{l_n})$. Define

$$E(\pi) \stackrel{\triangle}{=} \bigwedge_{i=1}^{n} x'_{k_i} = e_{l_i}.$$

Observe that $E(v)$ and $E(\pi)$ are equality logic formulae for any valuation v and assignment π. By Lemma 3, $\langle l, R\rangle$ is a representative configuration when R is a consistent matrix. For any representative configuration $\langle l, R\rangle$, we characterize a representative configuration $\langle l', R'\rangle$ with $\langle l, R\rangle \rightsquigarrow \langle l', R'\rangle$ as follows.

Definition 5. *Let $RA = (\Sigma, A, X, L, l_0, \Delta)$ be a register automaton, $(l, \alpha, g, \pi, l') \in \Delta$ a transition, and R a consistent matrix. Define the set $Post_{RA}(\langle l, R\rangle)$ of representative matrices as follows. $\langle l', R'\rangle \in Post_{RA}(\langle l, R\rangle)$ if $g \wedge E(w) \wedge E(\pi) \wedge E'(w')$ is consistent, where w and w' are the canonical valuations of R and R' respectively.*

Example 3. *Let $\Sigma = \mathbb{N}$, $X = \{x_1, x_2, x_3\}$, and $R = \begin{pmatrix} \overline{1} & \overline{0} & \overline{0} \\ \overline{0} & \overline{1} & \overline{1} \\ \overline{0} & \overline{1} & \overline{1} \end{pmatrix}$. By Algorithm 1, $w = 122$ is the canonical valuation of R. Consider a transition (l, α, g, π, l') where g is $(x_1 \neq x_2) \wedge (p_1 \neq p_2)$ and π is $(x_1 x_2 x_3) \mapsto (x_2 p_1 p_2)$. Then $E(v)$ is $(x_1 \neq x_2) \wedge (x_1 \neq x_3) \wedge (x_2 = x_3)$ and $E(\pi)$ is $(x'_1 = x_2) \wedge (x'_2 = p_1) \wedge (x'_3 = p_2)$. Let F denote the equality logic formula $g \wedge E(v) \wedge E(\pi)$. F is consistent. Observe that $\vdash F \implies x'_2 \neq x'_3$. Consider the following three cases:*

1. *R'_0 is $\begin{pmatrix} \overline{1} & \overline{1} & \overline{0} \\ \overline{1} & \overline{1} & \overline{0} \\ \overline{0} & \overline{0} & \overline{1} \end{pmatrix}$. Since $\vdash F \implies x'_1 = x'_2 \wedge x'_1 \neq x'_3$, $\langle l', R'_0\rangle \in Post_{RA}(\langle l, R\rangle)$;*
2. *R'_1 is $\begin{pmatrix} \overline{1} & \overline{0} & \overline{1} \\ \overline{0} & \overline{1} & \overline{0} \\ \overline{1} & \overline{0} & \overline{1} \end{pmatrix}$. Since $\vdash F \implies x'_1 = x'_3 \wedge x'_1 \neq x'_2$, $\langle l', R'_1\rangle \in Post_{RA}(\langle l, R\rangle)$;*

3. R'_2 *is* $\begin{pmatrix} \overline{1} & \overline{0} & \overline{0} \\ \overline{0} & \overline{1} & \overline{0} \\ \overline{0} & \overline{0} & \overline{1} \end{pmatrix}$. *Since* $\vdash F \implies x'_1 \neq x'_2 \wedge x'_1 \neq x'_3$, $\langle l', R'_2 \rangle \in Post_{RA}(\langle l, R \rangle)$.

Lemma 4. *$E(v)$ is consistent for every $v \in V_{(X,\Sigma)}$. Moreover, $E(v) = E(w)$ if $[v] = [w]$.*

Lemma 5. *Let $RA = (\Sigma, A, X, L, l_0, \Delta)$ be a register automaton, R and R' be consistent. $\langle l, R \rangle \rightsquigarrow \langle l', R' \rangle$ iff there is $(l, \alpha, g, \pi, l') \in \Delta$ and $g \wedge E(w) \wedge E(\pi) \wedge E'(w')$ is consistent, where w and w' are the canonical valuations of R and R' respectively.*

The following lemma is directly from Lemma 5. It shows that Definition 5 correctly characterizes successors of any given representative configuration.

Lemma 6. $Post_{RA}(\langle l, R \rangle) = \{\langle l', R' \rangle | \langle l, R \rangle \rightsquigarrow \langle l', R' \rangle\}$.

Using Algorithm 2 to enumerate representative matrices, it is straightforward to compute the set $Post_{RA}(\langle l, R \rangle)$ for any representative configuration $\langle l, R \rangle$ (Algorithm 3). We first obtain the canonical valuation w for R. The algorithm iterates through transitions of the given register automaton. For a transition (l, α, g, π, l'), define the equality logic formula F to be $g \wedge E(w) \wedge E(\pi)$. The algorithm then checks if F is consistent. If so, it goes through every representative matrices and adds them to the successor set U by Lemma 6.

```
Input: RA: (Σ, A, X, L, l_0, Δ); ⟨l, R⟩ : a representative configuration
ℛ ← UniverseR(X);
U, w ← ∅, CanonicalVal(R);
foreach (l, α, g, π, l') ∈ Δ do
    F ← g ∧ E(w) ∧ E(π);
    if F is consistent then
        foreach R' ∈ ℛ do
            w' ← CanonicalVal(R');
            F' ← g ∧ E(w) ∧ E(π) ∧ E'(w');
            if F' is consistent then U ← U ∪ {R'};
        end
end
return U;
```

Algorithm 3: $\mathsf{Post}(RA, \langle l, R \rangle)$

Theorem 2. *Let $RA = (\Sigma, A, X, L, l_0, \Delta)$ be a register automaton and $\langle l, R \rangle$ a representative configuration. $R' \in Post_{RA}(\langle l, R \rangle)$ iff $R' \in \mathsf{Post}(RA, \langle l, R \rangle)$.*

With the algorithm $\mathsf{Post}(RA, \langle l, R \rangle)$ at hand, we are ready to present our solution to the reachability problem for register automata. By Lemma 1, $\langle l_0, v_0 \rangle \langle l_1, v_1 \rangle \cdots \langle l_k, v_k \rangle$ is a run precisely when $\langle l_0, R_{[v_0]} \rangle \rightsquigarrow \langle l_1, R_{[v_1]} \rangle \rightsquigarrow \cdots \rightsquigarrow \langle l_k, R_{[v_k]} \rangle$. In order to check if the configuration $\langle l, v \rangle$ is reachable, we compute reachable representative configurations and check if the $\langle l, R_{[v]} \rangle$ belongs to the reachable representative configurations (Algorithm 4).

Our first technical result is summarized in the following theorem.

Theorem 3. *Let $(\Sigma, A, X, L, l_0, \Delta)$ be a register automaton and $\langle l, v \rangle$ a configuration. $\langle l, v \rangle$ is reachable iff* $\mathsf{Reach}((\Sigma, A, X, L, l_0, \Delta), (l, R_{[v]}))$ *returns true.*

Input: $(\Sigma, A, X, L, l_0, \Delta)$: a register automaton; $\langle l, R\rangle$: a representative configuration
Output: *true* if $\langle l, R\rangle$ is reachable; *false* otherwise
$\mathscr{R} \leftarrow \mathsf{UniverseR}(X)$;
$U, V \leftarrow \{\langle l_0, R_0\rangle | R_0 \in \mathscr{R}\}, \emptyset$;
while $U \neq V$ **do**
  $U' \leftarrow \bigcup_{\langle l,R\rangle \in U} \mathsf{Post}(RA, \langle l, R\rangle)$;
  $V, U \leftarrow U, U \cup U'$;
end
result $\leftarrow$ **if** $\langle l, R\rangle \in U$ **then** *true* **else** *false*;
return *result*;

Algorithm 4: $\mathsf{Reach}((\Sigma, A, X, L, l_0, \Delta), \langle l, R\rangle)$

5 $CTL(X, L)$ Model Checking

In addition to checking whether a configuration is reachable, it is often desirable to check patterns of configurations in runs of a register automaton. We define a computation tree logic to specify patterns of configurations in register automata. Representative configurations are then used to design an algorithm that solves the model checking problem for register automata.

Let X be the set of registers and L the set of locations. An *atomic formula* is an equality over X, an equality one side over X another side over C, or a location $l \in L$. We write AP for the set of atomic formulae. Consider the computation tree logic $CTL(X, L)$ defined as follows [3].

- If $f \in AP$, f is a $CTL(X, L)$ formula;
- If f_0 and f_1 are $CTL(X, L)$ formulae, $\neg f_0$ and $f_0 \wedge f_1$ are $CTL(X, L)$ formulae;
- If f_0 and f_1 are $CTL(X, L)$ formulae, $EX\, f_0$, $E(f_0\, U\, f_1)$, and $EG\, f_0$ are $CTL(X, L)$ formulae.

We use the standard abbreviations: $false(\equiv \neg(x = x))$, $true(\equiv \neg false)$, $f_0 \vee f_1(\equiv \neg(\neg f_0 \wedge \neg f_1))$, $f_0 \implies f_1(\equiv \neg f_0 \vee f_1)$, $AX\, f_0(\equiv \neg EX \neg f_0)$, $EF\, f_0(\equiv E(true\, U\, f_0))$, $AG\, f_0(\equiv \neg EF \neg f_0)$, and $AF\, f_0(\equiv \neg EG \neg f_0)$. Examples of $CTL(X, L)$ are $AF(l_{end} \wedge x_1 = x_2)$, $AG((l_{start} \wedge \neg(x_1 = x_2)) \implies EF(l_{end} \wedge (x_1 = x_2)))$.

Let $\langle l, v\rangle$ be a configuration of a register automaton $RA = (\Sigma, A, X, L, l_0, \Delta)$ and f a $CTL(X, L)$ formula. Define $\langle l, v\rangle$ *satisfies* f *in* RA ($\langle l, v\rangle \models_{RA} f$) by

- $\langle l, v\rangle \models_{RA} l$;
- $\langle l, v\rangle \models_{RA} x = y$ if $v(x) = v(y)$;
- $\langle l, v\rangle \models_{RA} \neg f$ if not $\langle l, v\rangle \models_{RA} f$;
- $\langle l, v\rangle \models_{RA} f_0 \wedge f_1$ if $\langle l, v\rangle \models_{RA} f_0$ and $\langle l, v\rangle \models_{RA} f_1$;
- $\langle l, v\rangle \models_{RA} EX\, f$ if $\langle l', v'\rangle \models_{RA} f$ for some $\alpha(\bar{d_n})$ such that $\langle l, v\rangle \xrightarrow{\alpha(\bar{d_n})} \langle l', v'\rangle$;
- $\langle l, v\rangle \models_{RA} E(f_0\, U\, f_1)$ if there are $k \geq 0$, $\alpha_i(\bar{d^i_{n_i}})$, $\langle l_i, v_i\rangle$ with $\langle l_0, v_0\rangle = \langle l, v\rangle$, and $\langle l_i, v_i\rangle \xrightarrow{\alpha_i(\bar{d^i_{n_i}})} \langle l_{i+1}, v_{i+1}\rangle$ for every $0 \leq i < k$ such that (1) $\langle l_k, v_k\rangle \models_{RA} f_1$; and (2) $\langle l_i, v_i\rangle \models_{RA} f_0$ for every $0 \leq i < k$.
- $\langle l, v\rangle \models_{RA} EG f$ if there are $\alpha_i(\bar{d^i_{n_i}})$, $\langle l_i, v_i\rangle$ with $\langle l_0, v_0\rangle = \langle l, v\rangle$, and $\langle l_i, v_i\rangle \xrightarrow{\alpha_i(\bar{d^i_{n_i}})} \langle l_{i+1}, v_{i+1}\rangle$ for every $i \geq 0$ such that $\langle l_i, v_i\rangle \models_{RA} f$.

Let $RA = (\Sigma, A, X, L, l_0, \Delta)$ be a register automaton and f a $CTL(X,L)$ formula. We say *RA satisfies* f (written $\models_{RA} f$) if $\langle l_0, v\rangle \models_{RA} f$ for every $v \in V_{(X,\Sigma)}$. The $CTL(X,L)$ *model checking* problem for register automata is to decide whether $\models_{RA} f$. The following lemma shows that any two configurations in a representative configuration satisfy the same $CTL(X,L)$ formulae.

Lemma 7. *Let $RA = (\Sigma, A, X, L, l_0, \Delta)$ be a register automaton, $l \in L$, $u, v \in V_{(X,\Sigma)}$, and f a $CTL(X,L)$ formula. If $u \sim_C v$, then*

$$\langle l, u\rangle \models_{RA} f \text{ if and only if } \langle l, v\rangle \models_{RA} f.$$

By Lemma 7, it suffices to compute representative configurations for any $CTL(X,L)$ formula. For any $CTL(X,L)$ formula f, we compute the set of representative configurations $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f\}$. Our model checking algorithm essentially follows the classical algorithm for finite-state models.

Input: RA: $(\Sigma, A, X, L, l_0, \Delta)$; ap : a $CTL(X,L)$ atomic formula
Output: $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} ap\}$
$\mathscr{R} \leftarrow \mathsf{UniverseU}(X)$;
switch ap **do**
  case l: **return** $\{\langle l, R\rangle | R \in \mathscr{R}\}$;
  case $x_i = x_j$: **return** $L \times \{R \in \mathscr{R} | R_{ij} = \overline{1} \text{ or } R_{ij} = c \in C\}$;
  case $x_i = c$: **return** $L \times \{R \in \mathscr{R} | R_{ii} = c \in C\}$;
endsw

Algorithm 5: $\mathsf{ComputeAP}(RA, ap)$

Algorithm 5 computes the set of representative configurations for atomic propositions. Clearly, $\mathsf{ComputeAP}(RA, ap) = \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} ap\}$.

Input: RA: $(\Sigma, A, X, L, l_0, \Delta)$; $S : \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f\}$
Output: $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} \neg f\}$
$\mathscr{R} \leftarrow \mathsf{UniverseR}(X)$;
return $(L \times \mathscr{R}) \setminus S$;

Algorithm 6: $\mathsf{ComputeNot}(RA, S)$

Input: $RA : (\Sigma, A, X, L, l_0, \Delta)$; $S_0 : \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f_0\}$; $S_1 : \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f_1\}$
Output: $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f_0 \wedge f_1\}$
return $S_0 \cap S_1$;

Algorithm 7: $\mathsf{ComputeAnd}(RA, S_0, S_1)$

For Boolean operations, we assume that representative configurations for operands have been computed. Algorithm 6 and 7 give details for the negation and conjunction of $CTL(X,L)$ formulae respectively.

Given the set S of representative configurations for a $CTL(X,L)$ formula f, Algorithm 8 shows how to compute representative configurations for EXf. For every possible representative configuration $\langle l, R\rangle$, it checks if $\langle l', R'\rangle \in S$ for some $\langle l', R'\rangle$ with $\langle l, R\rangle \rightsquigarrow \langle l', R'\rangle$. If so, $\langle l, R\rangle$ is added to the result.

To compute representative configurations for $f_0 U f_1$, recall that $f_0 U f_1$ is the least fixed point of the function $\Psi(Z) = f_1 \vee (f_0 \wedge EXZ)$. Algorithm 9 thus follows the standard fixed point computation for the $CTL(X,L)$ formula $f_0 U f_1$.

Input: RA: $(\Sigma, A, X, L, l_0, \Delta)$; $S : \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f\}$
Output: $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} EX\, f\}$
$\mathscr{R}, U \leftarrow \mathsf{UniverseR}(X), \emptyset$;
foreach $\langle l, R\rangle \in L \times \mathscr{R}$ **do**
  if $\mathsf{Post}(RA, \langle l, R\rangle) \cap S \neq \emptyset$ **then** $U \leftarrow U \cup \{\langle l, R\rangle\}$;
end
return U;

Algorithm 8: $\mathsf{ComputeEX}(RA, S)$

Input: RA: $(\Sigma, A, X, L, l_0, \Delta)$; $S_0 : \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f_0\}$; $S_1 : \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f_1\}$
Output: $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f_0\, U\, f_1\}$
$U, V \leftarrow S_1, \emptyset$;
while $U \neq V$ **do**
  $W \leftarrow \mathsf{ComputeEX}(RA, U)$;
  $V, U \leftarrow U, U \cup (W \cap S_0)$;
end
return U;

Algorithm 9: $\mathsf{ComputeEU}(RA, S_0, S_1)$

Input: $RA : (\Sigma, A, X, L, l_0, \Delta)$; $S : \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f\}$
Output: $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} EGf\}$
$U, V \leftarrow S, \mathsf{UniverseR}(X)$;
while $U \neq V$ **do**
  $W \leftarrow \mathsf{ComputeEX}(RA, U)$;
  $V, U \leftarrow U, U \cap W$;
end
return U;

Algorithm 10: $\mathsf{ComputeEG}(RA, S)$

For the $CTL(X,L)$ formula EGf, recall that EGf is the greatest fixed point of the function $\Phi(Z) = f \wedge EXZ$. Algorithm 10 performs the greatest fixed point computation to obtain representative configurations for EGf.

Input: $RA : (\Sigma, A, X, L, l_0, \Delta)$; f : a $CTL(X,L)$ formula
Output: $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f\}$
switch f **do**
  case *l, $x_i = x_j$, or $x_i = c$:*
    $U \leftarrow \mathsf{ComputeAP}(RA, f)$;
  case *$\neg f_0$:*
    $V \leftarrow \mathsf{ComputeCTL}(RA, f_0)$;
    $U \leftarrow \mathsf{ComputeNot}(RA, V)$;
  case *$f_0 \wedge f_1$:*
    $V_0, V_1 \leftarrow \mathsf{ComputeCTL}(RA, f_0), \mathsf{ComputeCTL}(RA, f_1)$;
    $U \leftarrow \mathsf{ComputeAnd}(RA, V_0, V_1)$;
  case *$EX\, f_0$:*
    $V \leftarrow \mathsf{ComputeCTL}(RA, f_0)$;
    $U \leftarrow \mathsf{ComputeEX}(RA, V)$;
  case *$E(f_0\, U\, f_1)$:*
    $V_0, V_1 \leftarrow \mathsf{ComputeCTL}(RA, f_0), \mathsf{ComputeCTL}(RA, f_1)$;
    $U \leftarrow \mathsf{ComputeEU}(RA, V_0, V_1)$;
  case *$EG\, f_0$:*
    $V \leftarrow \mathsf{ComputeCTL}(RA, f_0)$;
    $U \leftarrow \mathsf{ComputeEG}(RA, V)$;
endsw
return U;

Algorithm 11: $\mathsf{ComputeCTL}(RA, f)$

The representative configurations for a $CTL(X,L)$ formula are computed by induction on the formula (Algorithm 11). Theorem 4 summaries the algorithm.

Theorem 4. *Let $RA = (\Sigma, A, X, L, l_0, \Delta)$ be a register automaton, f a $CTL(X,L)$ formula, $l \in L$, and $v \in V_{(X,\Sigma)}$. $\langle l, v\rangle \models_{RA} f$ if and only if $\langle l, R_{[v]}\rangle \in \mathsf{ComputeCTL}(RA, f)$.*

It is easy to check whether $\models_{RA} f$ for any register automaton RA and $CTL(X,L)$ formula f by Theorem 4 (Algorithm 12). We compute the set $\{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} f\}$ of representative configurations and check if $\langle l, R\rangle$ belongs to the set for every representative matrix R.

6 An Example

In the Byzantine generals problem, one commanding and $n-1$ lieutenant generals would like to share information through one-to-one communication. However, not all generals are loyal. Some of them (the commanding general included) may be traitors. Traitors need not follow rules. The problem is to devise a mechanism so that all loyal generals share the same information at the end.

Consider the scenario with a commanding general, two loyal lieutenant, and one treacherous general. The emperor decides to send m soldiers to the front line, and asks the commanding general to inform

Input: $RA : (\Sigma, A, X, L, l_0, \Delta)$; f : a $CTL(X, L)$ formula
Output: *true* if $\models_{RA} f$; *false* otherwise
$U \leftarrow \mathsf{ComputeCTL}(RA, f)$;
$\mathscr{R} \leftarrow \mathsf{UniverseR}(X)$;
$W \leftarrow \{\langle l_0, R\rangle | R \in \mathscr{R}\}$;
result $\leftarrow$ **if** $W \subseteq U$ **then** *true* **else** *false*;
return *result*;

Algorithm 12: $\mathsf{ModelCheck}(RA, f)$

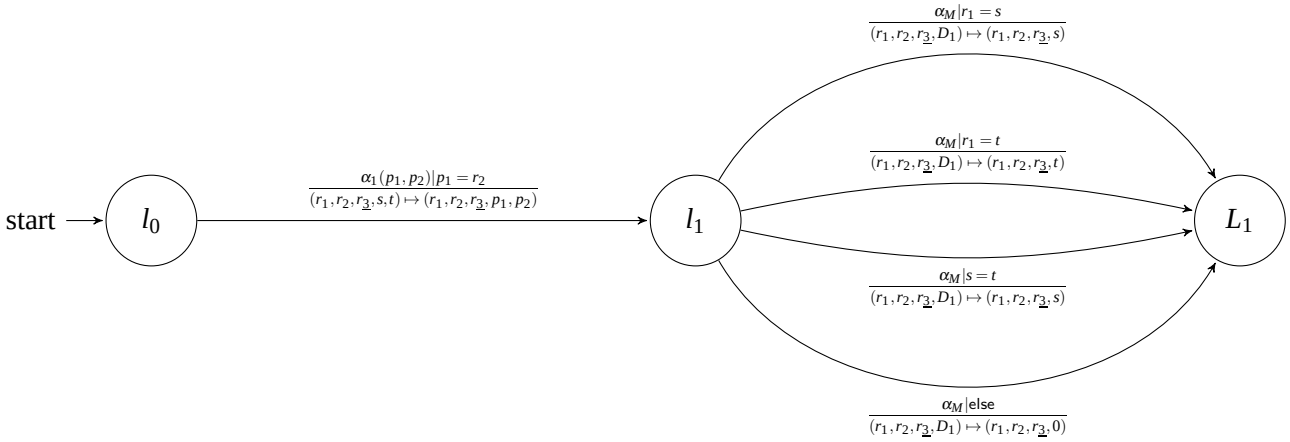


Figure 2: The Lieutenant General 1

the lieutenant generals. Based on the algorithm in [10], we give a model where a loyal, the treacherous, and the other loyal lieutenant generals act in turn. We want to know the initial configurations where both loyal generals agree upon the same information in this setting.

Since the number of soldiers is unbounded, we choose $\mathbb{N}$ as the infinite alphabet. The set of constants C is $\{0\}$, it is for default decision. When a lieutenant general cannot decide, he will take the default decision. We identify lieutenant generals by numbers: 1 and 2 are loyal, $\underline{3}$ is treacherous. The action set A has four actions: $\alpha_1, \alpha_2, \alpha_{\underline{3}}$, and α_M. The action α_i means that the lieutenant general i receives messages from the other lieutenant generals. Each lieutenant general computes the majority of messages in action α_M. Eight registers will be used. The registers $r_1, r_2, r_{\underline{3}}$ contain the commanding general's messages sent to each lieutenant general respectively. The final decisions of each lieutenant generals are stored in the registers D_1, D_2, and $D_{\underline{3}}$ respectively. Finally, s and t are temporary registers.

Assume the lieutenant generals have received a decision from the commanding general initially. Since the commanding general may be treacherous, the registers r_1, r_2, r_3 have arbitrary values at location l_0 (Figure 2).

In our scenario, the lieutenant general 1 acts first. He receives two messages from the other lieutenant generals in the action $\alpha_1(p_1, p_2)$. Since the lieutenant general 2 is loyal, he sends the message received from the commanding general. Thus we have the guard $p_1 = r_2$. The message from the lieutenant general $\underline{3}$ is arbitrary because the general is treacherous. We record the messages from the lieutenant generals 2 and $\underline{3}$ in the registers s and t respectively (location l_1). The lieutenant general 1 makes his decision by the majority of the message from the commanding general (r_1), the message from the lieutenant general

2 (s), and the message from the treacherous lieutenant general $\underline{3}$ (t). For instance, if the messages from the other lieutenant generals are equal ($s = t$), the lieutenant general 1 will have his decision equal to s through the transition $(l_1, \alpha_M, s = t, (r_1, r_2, r_{\underline{3}}, D_1) \mapsto (r_1, r_2, r_{\underline{3}}, s), L_1)$.

The other lieutenant generals are modeled similarly. Appendix A gives the model in register automata for the scenario where the location L_2 denotes the end of communication. Since the commanding general is not necessary loyal, we are interested in finding initial configurations that satisfy the $CTL(X,L)$ property $AF(D_1 = D_2) \equiv \neg EG\neg(D_1 = D_2)$.

Let $\mathscr{R} = \mathsf{UniverseR}(X)$ be the set of representative matrices. We begin with $U_0 = \{\langle l, R_{[v]}\rangle | \langle l, v\rangle \models_{RA} \neg(D_1 = D_2)\} = L \times \{R_{[v]} | v(D_1) \neq v(D_2)\}$. Then $W_0 = \mathsf{ComputeEX}(RA, U_0) = (\{l_0, l_1, L_1, L_3\} \times \mathscr{R}) \cup \{\langle l_2, R_{[v]}\rangle | (v(r_2) = v(s) \wedge v(D_1) \neq v(s)) \vee (v(r_2) = v(t) \wedge v(D_1) \neq v(t)) \vee (v(s) = v(t) \wedge v(D_1) \neq v(s)) \vee (v(r_2) \neq v(s) \wedge v(r_2) \neq v(t) \wedge v(s) \neq v(t) \wedge v(D_1) \neq v(0))\} \cup \{\langle L_2, R_{[v]}\rangle | v(D_1) \neq v(D_2)\}$. Consider a configuration $\langle l_1, v_1\rangle \in \langle l_1, R_{[v_1]}\rangle \in W_0$. Since the outgoing transitions at location l_1 do not assign values to the register D_2, D_2 can have an arbitrary value at the location L_1. Particularly, $\langle l_1, v_1\rangle \xrightarrow{\alpha_M} \langle l_1, v_1'\rangle$ for some $v_1'(D_2) \neq v_1'(D_1)$. We have $\langle l_1, v_1\rangle \models_{RA} EX\neg(D_1 = D_2)$. More interestingly, let us consider another configuration $\langle l_2, v_2\rangle \in \langle l_2, R_{[v_2]}\rangle \in W_0$ with $v_2(s) = v_2(t) \wedge v_2(D_1) \neq v_2(s)$. Since $v_2(s) = v_2(t)$, the register D_2 will be assigned to the value of the register s by the transition $(l_2, \alpha_M, s = t, (r_1, r_2, r_{\underline{3}}, D_1, D_2, D_3) \mapsto (r_1, r_2, r_{\underline{3}}, D_1, s, D_3), L_2)$ (Figure 3). Particularly, define $v_2'(D_2) = v_2(s)$ and $v_2'(x) = v_2(x)$ for $x \neq s$. We have $\langle l_2, v_2\rangle \xrightarrow{\alpha_M} \langle L_2, v_2'\rangle$, $v_2'(D_2) = v_2(s) \neq v_2(D_1) = v_2'(D_1)$, and $\langle L_2, v_2'\rangle \models_{RA} \neg(D_1 = D_2)$. $\langle l_2, v_2\rangle \models_{RA} EX\neg(D_1 = D_2)$.

We manually compute the representative configurations obtained by $\mathsf{ComputeCTL}(RA, EG\neg(D_1 = D_2))$ (Appendix B). Particularly, we have $\{\langle l_0, R_{[v]}\rangle | D_1 = D_2 \vee r_1 = r_2\} \subseteq \mathsf{ComputeCTL}(RA, AF(D_1 = D_2))$. The loyal lieutenant generals will agree on the same information provided they have the same decision, or the commanding general sends them the same message initially.

7 Conclusion

We develop an exact finitary representation for valuations in register automata. Based on representative valuations, we show that the reachability problem for register automata is decidable. We also define $CTL(X,L)$ for register automata and propose a model checking algorithm for the logic. As an illustration, we model a scenario in the Byzantine generals problem. We discuss the initial condition for correctness by the $CTL(X,L)$ model checking algorithm in the example.

$CTL(X,L)$ has very primitive modal operators. We believe that our technique applies to more expressive modal μ-calculus. It will also be interesting to investigate structured infinite alphabets. For instance, a totally ordered infinite alphabet is useful in the bakery algorithm. Representative valuations for such infinite alphabets will be essential to verification as well.

References

[1] Sofia Cassel, Falk Hower, Bengt Jonsson, Maik Merten & Bernhard Steffen (2011): *A Succinct Canonical Register Automaton Model*. *ATVA 2011* LNCS 6996, pp. 366–380, doi:10.1007/978-3-642-24372-1_26.

[2] Sofia Cassel, Falk Hower, Bengt Jonsson & Bernhard Steffen (2012): *Inferring Canonical Register Automata*. *VMCAI 2012* LNCS 7148, pp. 251–266, doi:10.1007/978-3-642-27940-9_17.

[3] Edmund M. Clarke, Jr., Orna Grumberg & Doron A. Peled (1999): *Model Checking*. MIT Press.

[4] Stphane Demri & Ranko Lazic (2009): *LTL with the Freeze Quantifier and Register Automata*. *ACM Transactions on Computational Logic* 10(16), doi:10.1145/1507244.1507246.

[5] Diego Figueira, Piotr Hofman & Slawomir Lasota (2010): *Relating timed and register automata*. *EPTCS 41*, pp. 61–75, doi:10.4204/EPTCS.41.5.

[6] Michael Kaminiski & Nissim Francez (1994): *Finite-memory automata*. *Theoretical Computer Science* 134, pp. 329–363, doi:10.1016/0304-3975(94)90242-9.

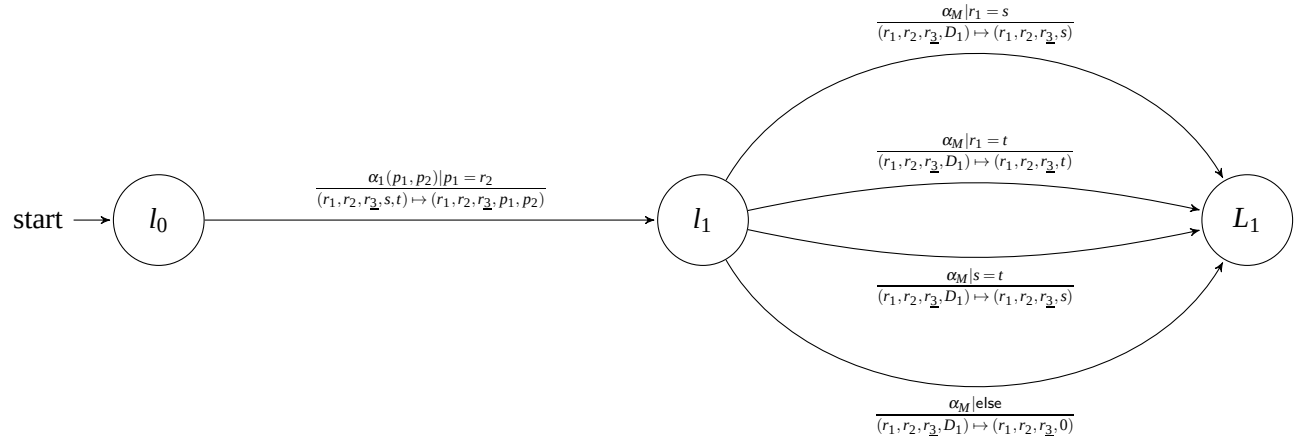
[7] D. Kroening & O. Strichman (2008): *Decision Procedures - an algorithmic point of view*. EATCS, Springer, doi:10.1007/s10817-013-9295-4.

[8] Leslie Lamport, Robert Shostak & Marshall Pease (1982): *The Byzantine Generals Problem*. *ACM Transactions on Programming Languages and Systems* 4(3), pp. 382–401, doi:10.1145/357172.357176.

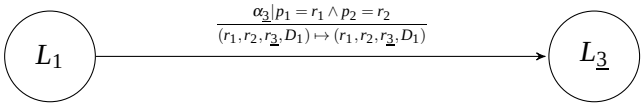
[9] Frank Neven, Thomas Schwentick & Victor Vianu (2004): *Finite State Machines for Strings Over Infinite Alphabets*. *ACM Transsactions on Computational Logic* 5(3), pp. 403–435, doi:10.1145/1013560.1013562.

[10] Junxing Wang (2012): *A Simple Byzantine Generals Protocol*. *Journal of Combinatorial Optimization*, doi:10.1007/s10878-012-9534-3.

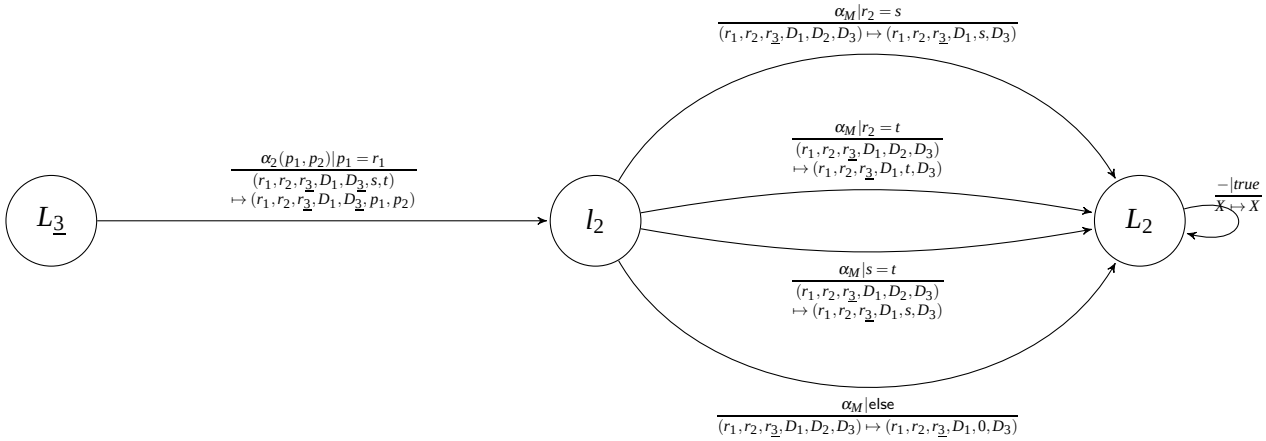
A A Scenario of the Byzantine Generals Problem



(a) Lieutenant General 1



(b) Lieutenant General $\underline{3}$



(c) Lieutenant General 2

Figure 3: The Byzantine Generals Problem

Figure 3 shows the register automaton for the scenario described in Section 6. The transition $\frac{-|true}{X\mapsto X}$ at location L_2 denotes that the automaton keeps the same valuation upon reading any data symbol at location L_2.

B ComputeCTL($RA, EG\neg(D_1 = D_2)$)

Let $\mathscr{R} = \mathsf{UniverseR}(X)$. In the following, the set comprehension represents the requirements of valuations. For instance, the notation $\{R_{[v]} | D_1 \neq D_1\}$ denotes the set $\{R_{[v]} | v(D_1) \neq v(D_2)\}$. The following table shows the details of computation.

U_0	$L \times \{R_{[v]} \mid D_1 \neq D_2\}$
W_0	$\{l_0, l_1, L_1, L_3\} \times \mathscr{R}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle \mid D_1 \neq D_2\}$
U_1	$\{l_0, l_1, L_1, L_3\} \times \{R_{[v]} \mid D_1 \neq D_2\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle \mid D_1 \neq D_2\}$
W_1	$\{l_0, l_1, L_1\} \times \mathscr{R}$ $\cup$ $\{\langle L_3, R_{[v]} \rangle \mid (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2)\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle \mid D_1 \neq D_2\}$
U_2	$\{l_0, l_1, L_1\} \times \{R_{[v]} \mid D_1 \neq D_2\}$ $\cup$ $\{\langle L_3, R_{[v]} \rangle \mid (D_1 \neq D_2) \wedge (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle \mid D_1 \neq D_2\}$
W_2	$\{l_0, l_1\} \times \mathscr{R}$ $\cup$ $\{\langle L_1, R_{[v]} \rangle \mid (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2)\}$ $\cup$ $\{\langle L_3, R_{[v]} \rangle \mid (D_1 \neq D_2) \wedge (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle \mid D_1 \neq D_2\}$
U_3	$\{l_0, l_1\} \times \{R_{[v]} \mid D_1 \neq D_2\}$ $\cup$ $\{\langle L_1, R_{[v]} \rangle \mid (D_1 \neq D_2) \wedge ((D_1 \neq r_2) \vee (D_1 \neq r_1) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\{\langle L_3, R_{[v]} \rangle \mid (D_1 \neq D_2) \wedge (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle \mid D_1 \neq D_2\}$

W_3	$\{l_0\} \times \mathscr{R}$ $\cup$ $\left\{ \langle l_1, R_{[v]} \rangle \;\middle|\; \begin{array}{l} [(r_1 = s \wedge s \neq r_2) \vee (r_1 = t \wedge t \neq r_2) \vee (s = t \wedge s \neq r_2) \vee \\ \quad (r_1 \neq s \wedge r_1 \neq t \wedge s \neq t \wedge r_2 \neq 0)] \vee \\ [(r_1 = s \wedge s \neq 0 \wedge r_1 \neq r_2) \vee (r_1 = t \wedge t \neq 0 \wedge r_1 \neq r_2) \vee \\ \quad (s = t \wedge s \neq 0 \wedge r_1 \neq r_2)] \end{array} \right\}$ $\cup$ $\{\langle L_1, R_{[v]} \rangle | (D_1 \neq D_2) \wedge ((D_1 \neq r_2) \vee (D_1 \neq r_1) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\{\langle L_3, R_{[v]} \rangle | (D_1 \neq D_2) \wedge (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle | D_1 \neq D_2\}$
U_4	$\{\langle l_0, R_{[v]} \rangle | D_1 \neq D_2\}$ $\cup$ $\left\{ \langle l_1, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ([(r_1 = s \wedge s \neq r_2) \vee (r_1 = t \wedge t \neq r_2) \vee (s = t \wedge s \neq r_2) \vee \\ \quad (r_1 \neq s \wedge r_1 \neq t \wedge s \neq t \wedge r_2 \neq 0)] \vee \\ [(r_1 = s \wedge s \neq 0 \wedge r_1 \neq r_2) \vee (r_1 = t \wedge t \neq 0 \wedge r_1 \neq r_2) \vee \\ \quad (s = t \wedge s \neq 0 \wedge r_1 \neq r_2)]) \end{array} \right\}$ $\cup$ $\{\langle L_1, R_{[v]} \rangle | (D_1 \neq D_2) \wedge ((D_1 \neq r_2) \vee (D_1 \neq r_1) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\{\langle L_3, R_{[v]} \rangle | (D_1 \neq D_2) \wedge (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle | D_1 \neq D_2\}$
W_4	$\{\langle l_0, R_{[v]} \rangle | r_1 \neq r_2\}$ $\cup$ $\left\{ \langle l_1, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ([(r_1 = s \wedge s \neq r_2) \vee (r_1 = t \wedge t \neq r_2) \vee (s = t \wedge s \neq r_2) \vee \\ \quad (r_1 \neq s \wedge r_1 \neq t \wedge s \neq t \wedge r_2 \neq 0)] \vee \\ [(r_1 = s \wedge s \neq 0 \wedge r_1 \neq r_2) \vee (r_1 = t \wedge t \neq 0 \wedge r_1 \neq r_2) \vee \\ \quad (s = t \wedge s \neq 0 \wedge r_1 \neq r_2)]) \end{array} \right\}$ $\cup$ $\{\langle L_1, R_{[v]} \rangle | (D_1 \neq D_2) \wedge ((D_1 \neq r_2) \vee (D_1 \neq r_1) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\{\langle L_3, R_{[v]} \rangle | (D_1 \neq D_2) \wedge (D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\}$ $\cup$ $\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ((r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)) \end{array} \right\}$ $\cup$ $\{\langle L_2, R_{[v]} \rangle | D_1 \neq D_2\}$

Finally, the following representative configurations satisfy $EG\neg(D_1 = D_2)$.

$$\begin{array}{l}
\{\langle l_0, R_{[v]} \rangle | D_1 \neq D_2 \wedge r_1 \neq r_2\} \quad \cup \\
\left\{ \langle l_1, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ ([(r_1 = s \wedge s \neq r_2) \vee (r_1 = t \wedge t \neq r_2) \vee (s = t \wedge s \neq r_2) \vee \\ \quad (r_1 \neq s \wedge r_1 \neq t \wedge s \neq t \wedge r_2 \neq 0)] \vee \\ [(r_1 = s \wedge s \neq 0 \wedge r_1 \neq r_2) \vee (r_1 = t \wedge t \neq 0 \wedge r_1 \neq r_2) \vee \\ \quad (s = t \wedge s \neq 0 \wedge r_1 \neq r_2)]) \end{array} \right\} \quad \cup \\
\{\langle L_1, R_{[v]} \rangle | (D_1 \neq D_2) \wedge ((D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\} \quad \cup \\
\{\langle L_3, R_{[v]} \rangle | (D_1 \neq D_2) \wedge ((D_1 \neq r_2) \vee (D_1 \neq 0 \wedge r_1 \neq r_2))\} \quad \cup \\
\left\{ \langle l_2, R_{[v]} \rangle \;\middle|\; \begin{array}{l} (D_1 \neq D_2) \wedge \\ [(r_2 = s \wedge D_1 \neq s) \vee (r_2 = t \wedge D_1 \neq t) \vee (s = t \wedge D_1 \neq s) \vee \\ \quad (r_2 \neq s \wedge r_2 \neq t \wedge s \neq t \wedge D_1 \neq 0)] \end{array} \right\} \quad \cup \\
\{\langle L_2, R_{[v]} \rangle | D_1 \neq D_2\}
\end{array}$$

Zenoness for Timed Pushdown Automata

Parosh Aziz Abdulla Mohamed Faouzi Atig Jari Stenman

Timed pushdown automata are pushdown automata extended with a finite set of real-valued clocks. Additionaly, each symbol in the stack is equipped with a value representing its age. The enabledness of a transition may depend on the values of the clocks and the age of the topmost symbol. Therefore, dense-timed pushdown automata subsume both pushdown automata and timed automata. We have previously shown that the reachability problem for this model is decidable. In this paper, we study the zenoness problem and show that it is EXPTIME-complete.

1 Introduction

Pushdown automata [9, 20, 16, 17] and timed automata [6, 11, 10] are two of the most widely used models in verification. Pushdown automata are used as models for (discrete) recursive systems, whereas timed automata model timed (nonrecursive) systems. Several models have been proposed that extend pushdown automata with timed behaviors [8, 14, 12, 13, 15].

We consider the model of (Dense-)Timed Pushdown Automata (TPDA), introduced in [1], that subsumes both pushdown automata and timed automata. As in the case of a pushdown automaton, a TPDA has a stack which can be modified by pushing and popping. A TPDA extends pushdown automata with time in the sense that the automaton (1) has a finite set of real-valued clocks, and (2) stores with each stack symbol its (real-valued) age. Pushing a symbol adds it on top of the stack with an initial age chosen nondeterministically from a given interval. A pop transition removes the topmost symbol from the stack provided that it matches the symbol specified by the transition, and that its age lies within a given interval. A TPDA can also perform timed transitions, which simulate the passing of time. A timed transition synchronously increases the values of all clocks and the ages of all stack symbols with some non-negative real number. The values of the clocks can be tested for inclusion in a given interval or nondeterministically reset to a value in a given interval. The model yields a transition system that is infinite in two dimensions; the stack contains an unbounded number of symbols, and each symbol is associated with a unique real-valued clock.

In [1], we showed that the reachability problem, i.e. the problem of deciding whether there exists a computation from the initial state to some target state, is decidable (specifically, EXPTIME-complete). In this paper, we address the zenoness problem for TPDA. The zenoness problem is the problem of deciding whether there is a computation that contains infinitely many discrete transitions (i.e. transitions that are not timed transitions) in *finite time* [5, 4, 21]. Zeno computations may represent specification errors, since these kinds of runs are not possible in real-world systems. We show that the zenoness problem for TPDA can be reduced to the problem of deciding whether a pushdown automaton has an infinite run with the labelling a^ω. The latter problem is polynomial in the size of the pushdown automaton, which is itself exponential in the size of the TPDA.

Related Work

The works in [8, 14, 12, 13, 15] consider pushdown automata extended with clocks. However, these models separate the timed part and the pushdown part of the automaton, which means that the stack

L. Clemente and L. Holik (Eds.): 15th International Workshop on
Verification of Infinite-State Systems (INFINITY'13)
EPTCS 140, 2014, pp. 35–47, doi:10.4204/EPTCS.140.3

symbols are not equipped with clocks.

In [7], the authors define the class of *extended pushdown timed automata*. An extended pushdown timed automaton is a pushdown automaton enriched with a set of clocks, with an additional stack used to store/restore clock valuations. In our model, clocks are associated with stack symbols and store/restore operations are disallowed. The two models are quite different. This is illustrated, for instance, by the fact that the reachability problem is undecidable in their case.

In [22], the authors introduce *recursive timed automata*, a model where clocks are considered as variables. A recursive timed automaton allows passing the values of clocks using either *pass-by-value* or *pass-by-reference* mechanism. This feature is not supported in our model since we do not allow pass-by-value communication between procedures. Moreover, in the recursive timed automaton model, the local clocks of the caller procedure are stopped until the called procedure returns. The authors show decidability of the reachability problem when either all clocks are passed by reference or none is passed by reference. This is the model that is most similar to ours, since in both cases, the reachability problem reduces to the same problem for a pushdown automaton that is abstract-time bisimilar to the timed system.

In a recent work [2] we have shown decidability of the reachability problem for *discrete-timed* pushdown automata, where time is interpreted as being incremented in discrete steps and thus the ages of clocks and stack symbols are in the natural numbers. This makes the reachability problem much simpler to solve, and the method of [2] cannot be extended to the dense-time case.

Finally, the zenoness problem for different kinds of timed systems is well studied in the literature (see, e.g., [4, 18] for timed automata and [3] for dense-timed Petri nets).

2 Preliminaries

We use $\mathbb{N}$ and $\mathbb{R}^{\geq 0}$ to denote the set of natural numbers and non-negative reals, respectively. For values $n, m \in \mathbb{N}$, we denote by the intervals $[n:m]$, $(n:m)$ $[n:m)$, $(n:m]$, $[n:\infty)$ and $(n:\infty)$ the sets of values $r \in \mathbb{R}^{\geq 0}$ satisfying the constraints $n \leq r \leq m$, $n < r < m$, $n \leq r < m$, $n < r \leq m$, $n \leq r$, and $n < r$, respectively. We let $\mathcal{I}$ denote the set of all such intervals.

For a non-negative real number $r \in \mathbb{R}^{\geq 0}$, with $r = n + r'$ $n \in \mathbb{N}$, and $r' \in [0:1)$, we let $\lfloor r \rfloor = n$ denote the *integral part*, and $frac(r) = r'$ denote the *fractional part* of r. Given a set S, we use 2^S for the powerset of S. For sets A and B, $f : A \to B$ denotes a (possibly partial) function from A to B. We write $f(a) = \perp$ when f is undefined at $a \in A$. We use $dom(f)$ and $range(f)$ to denote the domain and range of f. We write $f[a \leftarrow b]$ to denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ for $x \neq a$. The set of partial functions from A to B is written as $[A \to B]$.

Let A be an alphabet. We denote by A^*, (resp. A^+) the set of all *words* (resp. non-empty words) over A. The empty word is denoted by ϵ. For a word w, $|w|$ denotes the length of w (we have $|\epsilon| = 0$). For words w_1, w_2, we use $w_1 \cdot w_2$ for the concatenation of w_1 and w_2. We extend the operation $\cdot$ to sets W_1, W_2 of words by defining $W_1 \cdot W_2 = \{w_1 \cdot w_2 \,|\, w_1 \in W_1, w_2 \in W_2\}$. We denote by $w[i]$ the ith element a_i of $w = a_1 \ldots a_n$.

We use A^ω to denote the set of all infinite words over the alphabet A. We let a^ω denote the infinite word $aaa\ldots$ and write $|w| = \infty$ for any infinite word w over A.

We define a binary *shuffle operation* $\otimes$ inductively: For $w \in (2^A)^*$, define $w \otimes \epsilon = \epsilon \otimes w = \{w\}$. For sets $r_1, r_2 \in 2^A$ and words $w_1, w_2 \in (2^A)^*$, define $(r_1 \cdot w_1) \otimes (r_2 \cdot w_2) = (r_1 \cdot (w_1 \otimes (r_2 \cdot w_2))) \cup (r_2 \cdot ((r_1 \cdot w_1) \otimes w_2))) \cup ((r_1 \cup r_2) \cdot (w_1 \otimes w_2))$.

Let $w = a_1 \dots a_m$ and $w' = b_1 \dots b_n$ be words in A^*. An *injection* from w to w' is a partial function $h : \{1,\dots,m\} \to \{1,\dots,n\}$ that is *strictly monotonic*, i.e. for all $i,j \in \{1,\dots,m\}$, if $i < j$ and $h(i),h(j) \neq \perp$, then $h(i) < h(j)$. The *fragmentation* w/h of w w.r.t. h is the sequence $\langle w_0\rangle a_{i_1} \langle w_1\rangle a_{i_2} \dots \langle w_{k-1}\rangle a_{i_k} \langle w_k\rangle$, where $dom(h) = \{i_1,\dots,i_k\}$ and $w = w_0 \cdot a_{i_1} \cdot w_1 \cdot \dots \cdot a_{i_k} \cdot w_k$. The fragmentation w'/h is the sequence $\langle w'_0\rangle b_{j_1} \langle w'_1\rangle \dots \langle w'_{l-1}\rangle b_{j_l} \langle w'_l\rangle$, where $range(h) = \{j_1,\dots,j_l\}$ and $w' = w'_0 \cdot b_{j_1} \cdot \dots \cdot b_{i_l} \cdot w'_l$.

Pushdown Automata

A pushdown automaton is a tuple $(Q, q_{init}, \Sigma, \Gamma, \Delta)$, where Q is a finite set of states, q_{init} is an initial state, Σ is a finite input alphabet, Γ is a finite stack alphabet and Δ is a set of transition rules of the form $\langle q, \sigma, \mathbf{nop}, q'\rangle$, $\langle q, \sigma, \mathbf{pop}(a), q'\rangle$ or $\langle q, \sigma, \mathbf{push}(a), q'\rangle$, where $q, q' \in Q$, $a \in \Gamma$ and $\sigma \in \Sigma \cup \{\epsilon\}$.

A configuration is a pair (q, w), where $q \in Q$ and $w \in \Gamma^*$. We define $\gamma_{init} = (q_{init}, \epsilon)$ to be the *initial configuration*, meaning that the automaton starts in the initial state and with an empty stack. We define a transition relation $\to$ on the set of configurations in the following way: Given two configurations $\gamma_1 = (q_1, w_1)$, $\gamma_2 = (q_2, w_2)$ and a transition rule $t = \langle q_1, \sigma, \mathbf{op}, q_2\rangle \in \Delta$, we write $\gamma_1 \xrightarrow{t} \gamma_2$ if one of the following conditions is satisfied:

- $\mathbf{op} = \mathbf{nop}$ and $w_2 = w_1$,
- $\mathbf{op} = \mathbf{push}(a)$ and $w_2 = a \cdot w_1$,
- $\mathbf{op} = \mathbf{pop}(a)$ and $w_1 = a \cdot w_2$.

For any transition rule $t = \langle q_1, \sigma, \mathbf{op}, q_2\rangle \in \Delta$, define $\Sigma(t) = \sigma$. We define $\longrightarrow = \cup_{t\in\Delta} \xrightarrow{t}$ and let $\longrightarrow^*$ be the reflexive transitive closure of $\longrightarrow$. We say that an infinite word $\sigma_1\sigma_2\sigma_3 \dots \in \Sigma^\omega$ is a *trace* of $\mathcal{P}$ if there exists configurations $\gamma_1, \gamma_2, \gamma_3, \dots$ such that $\gamma_1 = \gamma_{init}$, $\gamma_1 \xrightarrow{t_1} \gamma_2 \xrightarrow{t_2} \gamma_3 \xrightarrow{t_3} \dots$, and $\Sigma(t_i) = \sigma_i$ for all $i \in \mathbb{N}$. We denote by $Traces(\mathcal{P})$ the set of all traces of $\mathcal{P}$.

3 Timed Pushdown Automata

Syntax

A *Timed Pushdown Automaton* (TPDA) is a tuple $\mathcal{T} = (Q^\mathcal{T}, q^\mathcal{T}_{init}, X^\mathcal{T}, \Gamma^\mathcal{T}, \Delta^\mathcal{T})$. Here, $Q^\mathcal{T}$ is a finite set of *states*, $q^\mathcal{T}_{init} \in Q^\mathcal{T}$ is an initial state, $X^\mathcal{T}$ is a finite set of *clocks*, $\Gamma^\mathcal{T}$ is a finite *stack alphabet* and $\Delta^\mathcal{T}$ is finite set of *transition rules* of the form $(q, \mathbf{op}, q')$, where $q, q' \in Q^\mathcal{T}$ and $\mathbf{op}$ is one of the following:

$\mathbf{nop}$ An "empty" operation that does not modify the clocks or the stack,

$\mathbf{push}(a, I)$ Pushes $a \in \Gamma^\mathcal{T}$ to the stack with a (nondeterministic) initial age in $I \in \mathcal{I}$,

$\mathbf{pop}(a, I)$ Pops the topmost symbol if it is a and its age is in $I \in \mathcal{I}$,

$\mathbf{test}(x, I)$ Tests if the value of $x \in X^\mathcal{T}$ is within $I \in \mathcal{I}$,

$\mathbf{reset}(x, I)$ Sets the value of $x \in X^\mathcal{T}$ (nondeterministically) to some value in $I \in \mathcal{I}$.

Intuitively, a transition rule $\langle q, \mathbf{op}, q'\rangle$ means that the automaton is allowed to move from state q to state q' while performing the operation $\mathbf{op}$. The $\mathbf{nop}$ operation can be used to switch states without changing the stack or the values of clocks.

Semantics

The semantics of TPDA is defined by a transition relation over the set of *configurations*. A configuration is a tuple $(q, \mathtt{X}, w)$, where $q \in Q^{\mathcal{T}}$ is a state, $\mathtt{X} : X^{\mathcal{T}} \to \mathbb{R}^{\geq 0}$ is a *clock valuation* which assigns concrete values to clocks, and $w = (a_1, y_1) \dots (a_n, y_n) \in (\Gamma^{\mathcal{T}} \times \mathbb{R}^{\geq 0})^*$ is a *stack content*. In other words, the stack content is a sequence of pairs, each pair consisting of a symbol and its age. Here, (a_1, y_1) is on the top and (a_n, y_n) is on the bottom of the stack. Given a TPDA $\mathcal{T}$, we denote by $Conf(\mathcal{T})$ the set of all configurations of $\mathcal{T}$.

The transition relation consists of two types of transitions; *discrete* transitions, which correspond to applications of the transition rules, and *timed* transitions, which simulate the passing of time.

Discrete Transitions. Let $t = (q, \mathbf{op}, q') \in \Delta^{\mathcal{T}}$ be a transition rule and let $\gamma = (q, \mathtt{X}, w)$ and $\gamma' = (q', \mathtt{X}', w')$ be configurations. We have $\gamma \xrightarrow{t} \gamma'$ if one of the following conditions is satisfied:

- $\mathbf{op} = \mathbf{nop}$, $w' = w$ and $\mathtt{X}' = \mathtt{X}$,
- $\mathbf{op} = \mathbf{push}(a, I)$, $w' = (a, v)w$ for some $v \in I$, and $\mathtt{X}' = \mathtt{X}$,
- $\mathbf{op} = \mathbf{pop}(a, I)$, $w = (a, v)w'$ for some $v \in I$, and $\mathtt{X}' = \mathtt{X}$,
- $\mathbf{op} = \mathbf{test}(x, I)$, $w' = w$, $\mathtt{X}' = \mathtt{X}$ and $\mathtt{X}(x) \in I$,
- $\mathbf{op} = \mathbf{reset}(x, I)$, $w' = w$, and $\mathtt{X}' = \mathtt{X}[x \leftarrow v]$ for some $v \in I$.

Timed Transitions. Let $r \in \mathbb{R}^{\geq 0}$ be a real number. Given a clock valuation $\mathtt{X}$, let $\mathtt{X}^{+r}$ be the function defined by $\mathtt{X}^{+r}(x) = \mathtt{X}(x) + r$ for all $x \in X$. For any stack content $w = (a_1, y_1) \dots (a_n, y_n)$, let w^{+r} be the stack content $(a_1, y_1 + r) \dots (a_n, y_n + r)$. Let $\gamma = (q, \mathtt{X}, w)$ and $\gamma' = (q', \mathtt{X}', w')$ be configurations. Then $\gamma \xrightarrow{r} \gamma'$ if and only if $q' = q$, $\mathtt{X}' = \mathtt{X}^{+r}$ and $w' = w^{+r}$.

Computations. A *computation* (or *run*) π is a (finite or infinite) sequence of the form $(\gamma_1, \tau_1, \gamma_2)(\gamma_2, \tau_2, \gamma_3) \cdots$ (written as $\gamma_1 \xrightarrow{\tau_1} \gamma_2 \xrightarrow{\tau_2} \gamma_3 \cdots$) such that $\gamma_i \xrightarrow{\tau_i} \gamma_{i+1}$ for all $1 \leq i \leq |\pi|$. For $\tau \in (\Delta^{\mathcal{T}} \cup \mathbb{R}^{\geq 0})$, we define $Disc(\tau) = 1$ if $\tau \in \Delta^{\mathcal{T}}$ and $Disc(\tau) = 0$ if $\tau \in \mathbb{R}^{\geq 0}$. Then, the number of discrete transitions in π is defined as $|\pi|_{disc} = \sum_{i=1}^{|\pi|} Disc(\tau_i)$. Note that if $|\pi| = \infty$, then it may be the case that $|\pi|_{disc} = \infty$.

In this paper, we will consider the *duration* of transitions. Given a $\tau \in (\Delta^{\mathcal{T}} \cup \mathbb{R}^{\geq 0})$, the duration $\delta(\tau)$ is defined in the following way:

- $\delta(\tau) = 0$ if $\tau \in \Delta^{\mathcal{T}}$. Discrete transitions have no duration.
- $\delta(\tau) = \tau$ if $\tau \in \mathbb{R}^{\geq 0}$.

For a computation π, we define the duration $\delta(\pi)$ to be $\sum_{i=1}^{|\pi|} \delta(\tau_i)$. If the automaton can perform infinitely many discrete transitions in finite time, it exhibits a behavior called *zenoness*.

Definition 1 (Zenoness)**.** *A computation π is* zeno *if it contains infinitely many discrete transitions and has a finite duration, i.e. if $|\pi|_{disc} = \infty$ and $\delta(\pi) \leq c$ for some $c \in \mathbb{N}$. π is* non-zeno *if it is not zeno.*

The *zenoness problem* is the question whether a given TPDA contains a zeno run starting from the initial configuration:

Definition 2 (The Zenoness Problem)**.** *Given a TPDA $\mathcal{T}$, decide if there exists a computation $\pi = \gamma_{init} \longrightarrow \gamma_1 \longrightarrow \gamma_2 \longrightarrow \dots$ from the initial configuration of $\mathcal{T}$ such that π is zeno.*

Given two computations $\pi = \gamma_1 \xrightarrow{\tau_1} \gamma_2 \xrightarrow{\tau_2} \gamma_3 \xrightarrow{\tau_3} \cdots$ and π', we say that π' is a *prefix* of π if $\pi = \pi'$ or $\pi' = \gamma_1 \xrightarrow{\tau_1} \gamma_2 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_{n-1}} \gamma_n$ for some $1 \leq n$. We say that π' is a *suffix* of π if either $\pi' = \pi$ or $\pi' = \gamma_n \xrightarrow{\tau_n} \gamma_{n+1} \xrightarrow{\tau_{n+1}} \cdots$ for some $n \in \mathbb{N}$. We define the *concatenation* of a finite computation $\pi = \gamma_1 \xrightarrow{\tau_1} \gamma_2 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_{n-1}} \gamma_n$ with a (finite or infinite) computation $\pi' = \gamma'_1 \xrightarrow{\tau'_1} \gamma'_2 \xrightarrow{\tau'_2} \cdots$, where $\gamma_n = \gamma'_1$, as $\pi \cdot \pi' = \gamma_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{n-1}} \gamma_n \xrightarrow{\tau'_1} \gamma'_2 \xrightarrow{\tau'_2} \cdots$.

Let $\pi = \pi_1 \cdot \pi_2$ be a computation. We call the suffix π_2 a *unit suffix* if $\delta(\pi) < 1$. The question whether a TPDA $\mathcal{T}$ has a zeno run starting from the initial configuration can be reduced to the question whether there exists a run from the initial configuration which contains a zeno unit suffix:

Lemma 1. *A TPDA $\mathcal{T}$ contains a zeno run iff $\mathcal{T}$ contains a run $\pi = \pi_1 \cdot \pi_2$ such that π_2 is zeno and $\delta(\pi_2) < 1$.*

Proof. We prove both directions:

If: By the definition of zenoness.

Only if: Assume π is a zeno run of $\mathcal{T}$. Then there exists a smallest $n \in \mathbb{N}$ such that $\delta(\pi) \leq n$. Call it c. This means that the longest prefix π' of π for which $\delta(\pi') \leq c-1$ contains finitely many discrete transitions. We have that after π', the next transition in π will be a timed transition $\gamma \xrightarrow{r} \gamma'$ for some $r \in \mathbb{R}^{\geq 0}$, and $\delta(\pi') + r > c-1$. Now, let $\pi_1 = \pi' \cdot \gamma \xrightarrow{r} \gamma'$, and let π_2 be the remaining suffix in π. We can conclude that $\delta(\pi_2) = c - \delta(\pi_1) < c - (c-1) = 1$.

□

In the rest of the paper, we will show how to decide whether $\mathcal{T}$ contains a run that has a zeno unit suffix. Intuitively, given a TPDA $\mathcal{T}$, we will construct a pushdown automaton $\mathcal{P}$ which simulates the behavior of $\mathcal{T}$. The pushdown automaton $\mathcal{P}$ operates in two modes.

Initially, $\mathcal{P}$ runs in the first mode, in which it simulates the behavior of $\mathcal{T}$ exactly as described in [1]. While $\mathcal{P}$ runs in the first mode, all transitions are labelled with ϵ. At any time, $\mathcal{P}$ may guess that it can simulate a unit suffix. In this case, $\mathcal{P}$ switches to the second mode, in which it reads symbols from a unary alphabet (say $\{a\}$) while simulating discrete transitions of $\mathcal{T}$. The question whether $\mathcal{T}$ contains a unit suffix then reduces to the question whether $Traces(\mathcal{P})$ includes a^ω.

4 Symbolic Encoding

In this section, we show how to construct a symbolic PDA $\mathcal{P}$ that simulates the behavior of a TPDA $\mathcal{T}$. The PDA uses a symbolic *region* encoding to represent the infinitely many clock valuations of $\mathcal{T}$ in a finite way. The notion of regions was introduced in the classical paper on timed automata [6], in which a timed automaton is simulated by a region automaton (a finite-state automaton that encodes the regions in its states). This abstraction relies on the set of clocks being fixed and finite. Since a TPDA may in general operate on unboundedly many clocks (the stack is unbounded, and each symbol has an age), we cannot rely on this abstraction. Instead, we use regions of a special form as stack symbols in $\mathcal{P}$. For each symbol in the stack of $\mathcal{T}$, the stack of $\mathcal{P}$ contains, at the same position, a region that relates the stack symbol with all clocks. A problem with this approach is that we might need to record relations between clocks and stack symbols that lie arbitrarily far apart in the stack. However, in [1], we show that it is enough to enrich the regions in finite way (by recording the relationship between clocks and adjacent stack symbols), thus keeping the stack alphabet of $\mathcal{P}$ finite.

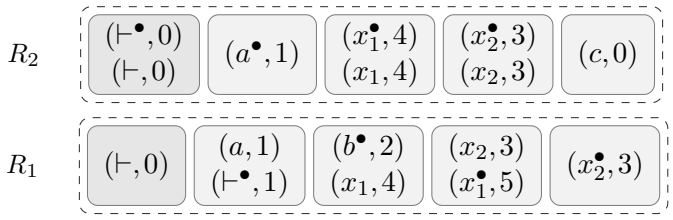


Figure 1: Two examples of regions

Regions

A region is a word over sets, where each set consists of a number of *items*. There are *plain items*, which represent the values of clocks and the topmost stack symbols. In addition, this set includes a reference clock $\vdash$, which is always 0 except when simulating a pop transition. Furthermore, we have *shadow items* which record the values of the corresponding plain items in the region below. Shadow items are used to remember the time that elapses while the plain symbols they represent are not on the top of the stack.

To illustrate this, assume that the region R_1 in Figure 1 is the topmost region in the stack. R_1 records the integral values and the relationships between the clocks x_1, x_2, the topmost stack symbol a and the reference clock $\vdash$. It also relates these symbols to the values of x_1, x_2, b and $\vdash$ in the previous topmost region. Now, if we simulate the pushing of c with inital age in $[0:1]$, one of the possible resulting regions is R_2. The region R_2 uses $x_1^\bullet$, $x_2^\bullet$ and $\vdash^\bullet$ to record the previous values of the clocks (initially, their values are identical to those of their plain counterparts). The value of the previous topmost symbol a is recorded in $a^\bullet$. Finally, the region relates the new topmost stack symbol c with all the previously mentioned symbols.

We define the set $Y = X \cup \Gamma \cup \{\vdash\}$ of plain items and a corresponding set $Y^\bullet = X^\bullet \cup \Gamma^\bullet \cup \{\vdash^\bullet\}$ of shadow items. We then define the set of *items* $Z = Y \cup Y^\bullet$.

Let c_{max} be the largest constant in the definition of $\mathcal{T}$. We denote by Max the set $\{0, 1, \ldots, c_{max}, \infty\}$. A *region* R is a word $r_1 \ldots r_n \in (2^{Z \times Max})^+$ such that the following holds:

- $\sum_{i=1}^n |(\Gamma \times Max) \cap r_i| = 1$ and $\sum_{i=1}^n |(\Gamma^\bullet \times Max) \cap r_i| = 1$. There is exactly one occurrence of a stack symbol and one occurrence of a shadow stack symbol.
- $\sum_{i=1}^n |(\{\vdash\} \times Max) \cap r_i| = 1$ and $\sum_{i=1}^n |(\{\vdash^\bullet\} \times Max) \cap r_i| = 1$. There is exactly one occurrence of $\vdash$ and one occurrence of $\vdash^\bullet$.
- For all clocks $x \in X$, $\sum_{i=1}^n |(\{x\} \times Max) \cap r_i| = 1$ and $\sum_{i=1}^n |(\{x^\bullet\} \times Max) \cap r_i| = 1$. Each plain clock symbol and shadow clock symbol occurs exactly once.
- $r_i \neq \varnothing$ for all $2 \leq i \leq n$. Only the first set may be empty.

For items $z \in Z$, if we have $(z, k) \in r_i$ for some $i \in \{1, \ldots, n\}$ and some (unique) $k \in Max$, then define $Val(R, z) = k$ and $Index(R, z) = i$. Otherwise, define $Val(R, z) = \perp$ and $Index(R, z) = \perp$ (this may be the case for stack symbols). We define $R^\top = \{z \in Z \mid Index(R, z) \neq \perp\}$.

Operations on Regions

In order to define the transition rules of the symbolic PDA, we need a number of operations on regions:

Testing Satisfiability

When we construct new regions, we need to limit the values of the items to certain intervals. To do this, we define what it means for a region to *satisfy* a membership predicate. Given an item $z \in Z$, an interval $I \in \mathcal{I}$ and a region R such that $z \in R^{\top}$, we write $R \vDash (z \in I)$ if and only if one of the following conditions is satisfied:

- $Index(R,z) = 1$, $Val(R,z) \neq \infty$ and $Val(R,z) \in I$,
- $Index(R,z) > 1$, $Val(R,z) \neq \infty$ and $Val(R,z) + v \in I$ for all $v \in \mathbb{R}^{\geq 0}$ such that $0 < v < 1$,
- $Val(R,z) = \infty$ and I is of the form $(m : \infty)$ or the form $[m : \infty)$ for some $m \in \mathbb{N}$.

Adding and Removing Items

In the following, we define operations that describe how items are added and deleted from regions. We also define, in terms of these operations, an operation that assigns a new value to an item.

For a region $R = r_1 \dots r_n$, an item $z \in Z$ and an $k \in Max$, we define $R \oplus (z,k)$ to be the set of regions R' satisfying the following conditions:

- $R = r_1 \dots r_{i-1}(r_i \cup \{(z,k)\})r_{i+1} \dots r_n$, where $1 \leq i \leq n$
- $R = r_1 \dots r_i\{(z,k)\}r_{i+1} \dots r_n$, where $1 \leq i \leq n$

We extend the definition of $\oplus$ by letting $R \oplus a$ denote the set $\bigcup_{m \in Max} R \oplus (a,m)$, i.e. the set of regions where we have added all possible values of a.

We define $R \ominus z$ to be the region $R' = r'_1 \dots r'_n$, where, for $1 \leq i \leq n$, we have $r'_i = r_i \setminus \{\{z\} \times Max\}$ if $r_i \setminus \{\{z\} \times Max\} \neq \varnothing$, and $r'_i = \epsilon$ otherwise. We extend the definition of $\ominus$ to sets of items in the following way: $R \ominus \varnothing = R$ and $R \ominus \{z_1, \dots z_n\} = (R \ominus z_1) \ominus \{z_2, \dots z_n\}$.

Given a region R, an item $z \in Z$ and an interval $I \in \mathcal{I}$, we define an *assignment* operation. We write $R[z \leftarrow I]$ to mean the set of regions R' such that $R' \in (R \ominus z) \oplus z$ and $R' \vDash (z \in I)$. For any number $n \in \mathbb{N}$, we write $R[z \leftarrow n]$ to mean $R[z \leftarrow [n : n]]$.

Creating New Regions

When we push a new stack symbol, we need to record the values of clocks and the value of the current top-most stack symbol. The operation $Make$ takes as arguments a region, a stack symbol, and an interval, It constructs the set of regions in which the shadow items record the values of the plain items in the old topmost region, and the value of the stack symbol is in the given interval.

Given a region R, a stack symbol $a \in \Gamma$ and an interval $I \in \mathcal{I}$, we define $Make(R, a \in I)$ to be the set of regions R' such that there are R_1, R_2, R_3 satisfying the following:

- $R_1 = R \ominus (R^{\top} \cap Y^{\bullet})$,
- If $R_1 = r_1 \dots r_n$, then $R_2 = r'_1 \dots r'_n$, where $r'_i = r_i \cup \{(y^{\bullet}, k) \,|\, (y,k) \in r_i\}$ for $i \in \{1, \dots, n\}$,
- $R_3 = R_2 \ominus (R^{\top} \cap \Gamma)$,
- $R' \in R_3 \oplus a$ and $R' \vDash (a \in I)$.

Passage of Time

We implement the passage of time by *rotating* the region. A rotation describes the effect of the smallest timed transition that changes the region. If the leftmost set (i.e. the set which represents items with fractional part 0) is nonempty, a timed transition, no matter how small, will "push" those items out. If the leftmost set is empty, the smallest timed transition that changes the regions is one that makes the fractional parts of those items 0.

Given a pair $(z,k) \in Z \times Max$, define $(z,k)^+ = (z,k')$, where $k' = k+1$ if $k < c_{max}$ and $k' = \infty$ otherwise. For a set $r \in 2^{Z \times Max}$, define $r^+ = \{(z,k)^+ \mid (z,k) \in r\}$. For a region $R = r_1 \dots r_n$, we define $R^+ = R'$ such that one of the following conditions is satisfied:

- $r_1 \neq \varnothing$ and $R' = \varnothing r_1 \dots r_n$,
- $r_1 = \varnothing$ and $R' = r_n^+ r_1 \dots r_{n-1}$.

We denote by R^{++} the set $\{R, R^+, (R^+)^+, ((R^+)^+)^+, \dots\}$. Note that this set is finite.

Product

When we simulate a pop transition, the region that we pop contains the most recent values of all clocks. On the other hand, the region below it contains shadow items that record relationships between items further down the stack. We need to keep all of this information. To do this, we define a product operation $\odot$ that merges the information contained in two regions. For regions $P = p_1 \dots p_{|P|}$ and $Q = q_1 \dots q_{|Q|}$, and an injection h from $\{1,\dots,|P|\}$ to $\{1,\dots,|Q|\}$, we write $P \preceq_h Q$ iff the following conditions are satisfied:

- $Val(P, y^\bullet) = Val(Q, y)$ for all $y \in P^\top \cap Y$,
- For every $i > 1$, $h(i) \neq \perp$ iff there exists a $y \in Y$ such that $Index(P,y) = i$,
- $h(1) = 1$,
- For all $y \in Y$, $i \in \{1,\dots,|P|\}$ and $j \in \{1,\dots,|Q|\}$, if $Index(P,y) = i$ and $Index(Q, y^\bullet) = j$, then $h(i) = j$.

We say that P *supports* Q, written $P \preceq Q$, if $P \preceq_h Q$ for some h. Let $P/h = p_{i_1}\langle P_1\rangle p_{i_2} \dots p_{i_m}\langle P_m\rangle$ and $Q/h = q_{j_1}\langle Q_1\rangle q_{j_2} \dots q_{j_m}\langle Q_m\rangle$. We define $p'_k = p_{i_k} \cap (Y^\bullet \cup \Gamma)$ and $q'_k = q_{j_k} \cap (X \cup \{\vdash\})$. Finally, define $r_1 = p'_1 \cup q'_1$ and, for $k \in \{2,\dots,m\}$, define $r_k = p'_k \cup q'_k$ if $p_k \cup q'_k \neq \varnothing$ and $r_k = \epsilon$ if $p_k \cup q'_k = \varnothing$. Then, $R \in P \odot Q$ if $R = r_1 \cdot R_1 \cdot r_2 \dots r_m \cdot R_m$ and $R_k \in P_k \otimes Q_k$ for $k \in \{1,\dots,m\}$.

5 An EXPTIME Upper Bound for the Zenoness Problem

In this section, we prove our main result:

Theorem 2. *The Zenoness problem for TPDA is in* EXPTIME.

The rest of this section will be devoted to the proof of Theorem 2. Given a TPDA $\mathcal{T} = (Q^\mathcal{T}, q^\mathcal{T}_{init}, \Gamma^\mathcal{T}, X^\mathcal{T}, \Delta^\mathcal{T})$, we construct an (untimed) PDA $\mathcal{P} = (Q^\mathcal{P}, q^\mathcal{P}_{init}, \Sigma^\mathcal{P}, \Gamma^\mathcal{P}, \Delta^\mathcal{P})$ such that $\mathcal{P}$ simulates zeno runs of $\mathcal{T}$. More specifically, $\mathcal{P}$ simulates a zeno run of $\mathcal{T}$ by first simulating the prefix, and then simulating the unit suffix. In order to do this, $\mathcal{P}$ runs in two modes. In the first mode, it simulates the prefix. In the second mode, it simulates the suffix while keeping track of the fact that the value of a special *control clock* $x_{control}$ is smaller than 1. We now describe the components of $\mathcal{P}$.

The states of $\mathcal{P}$ are composed of two disjoint sets; the *genuine* states $\{\mathbf{0},\mathbf{1}\} \times Q^{\mathcal{T}}$ and some *temporary* states $\mathtt{Tmp}$. Each genuine state (m,q) contains a state q from $Q^{\mathcal{T}}$ and a symbol m indicating the current simulation mode. If $m = \mathbf{0}$, $\mathcal{P}$ is currently simulating the prefix of a run. Conversely, if $m = \mathbf{1}$, $\mathcal{P}$ is simulating the suffix. The temporary states are used for intermediate transitions between configurations containing genuine states. We assume that we have functions $\mathtt{tmp}$, $\mathtt{tmp}_1$ and $\mathtt{tmp}_2$ that input arguments and map them to a unique element in $\mathtt{Tmp}$. The initial state $q^{\mathcal{P}}_{init}$ of $\mathcal{P}$ is the state $(\mathbf{0}, q^{\mathcal{T}}_{init})$. The input alphabet $\Sigma^{\mathcal{P}}$ is the unary alphabet $\{a\}$. The automaton reads an a when (and only when) it simulates a discrete transition in the suffix. When it simulates any other transition, it reads ϵ. Let $x_{control} \notin X^{\mathcal{T}}$ be a special control clock. The stack alphabet $\Gamma^{\mathcal{P}}$ contains all possible regions over the items $Z \cup \{x_{control}, x^{\bullet}_{control}\}$. The purpose of the control clock is to limit the duration of the suffix. We will now describe the set $\Delta^{\mathcal{T}}$ of transition rules:

nop For each transition rule $\langle q_1, \mathbf{nop}, q_2\rangle \in \Delta^{\mathcal{T}}$, the set $\Delta^{\mathcal{P}}$ contains the transition rules $\langle (\mathbf{0},q_1), \epsilon, \mathbf{nop}, (\mathbf{0},q_2)\rangle$ and $\langle (\mathbf{1},q_1), a, \mathbf{nop}, (\mathbf{1},q_2)\rangle$. Nop transitions are used for switching states without modifying the clocks or the stack.

test$(x \in I)$ We simulate a test transition in $\mathcal{T}$ with two transition in $\mathcal{P}$. If the topmost region satisfies the constraint, we pop it and move to a temporary state. Since a test transition is not supposed to modify the stack, we push back the same region we popped, while moving to the second genuine state. Formally, for each transition rule $\tau = \langle q_1, \mathbf{test}(x \in I), q_2\rangle \in \Delta^{\mathcal{T}}$, and region R such that $R \vDash (x \in I)$, the set $\Delta^{\mathcal{T}}$ contains the transition rules:

- $\langle (\mathbf{0},q_1), \epsilon, \mathbf{pop}(R), \mathtt{tmp}(\tau, R, \mathbf{0})\rangle$,
- $\langle \mathtt{tmp}(\tau, R, \mathbf{0}), \epsilon, \mathbf{push}(R), (\mathbf{0},q_2)\rangle$ (for simulating the prefix),
- $\langle (\mathbf{1},q_1), a, \mathbf{pop}(R), \mathtt{tmp}(\tau, R, \mathbf{1})\rangle$,
- $\langle \mathtt{tmp}(\tau, R, \mathbf{1}), \epsilon, \mathbf{push}(R), (\mathbf{1},q_2)\rangle$ (for simulating the suffix).

reset$(x \leftarrow I)$ We simulate reset transitions by popping the topmost region and pushing it back, in a similar way to test transitions, except that the given clock is nondeterministically set to some value in the given interval. Formally, for each transition rule $\tau = \langle q_1, \mathbf{reset}(x \leftarrow I), q_2\rangle \in \Delta^{\mathcal{T}}$, and each pair of regions R, R' such that $R' \in R[x \leftarrow I]$, the set $\Delta^{\mathcal{T}}$ contains the transition rules:

- $\langle (\mathbf{0},q_1), \epsilon, \mathbf{pop}(R), \mathtt{tmp}(\tau, R, \mathbf{0})\rangle$,
- $\langle \mathtt{tmp}(\tau, R, \mathbf{0}), \epsilon, \mathbf{push}(R'), (\mathbf{0},q_2)\rangle$ (for simulating the prefix),
- $\langle (\mathbf{1},q_1), a, \mathbf{pop}(R), \mathtt{tmp}(\tau, R, \mathbf{1})\rangle$,
- $\langle \mathtt{tmp}(\tau, R, \mathbf{1}), \epsilon, \mathbf{push}(R'), (\mathbf{1},q_2)\rangle$ (for simulating the suffix).

push(a, I) We will need two temporary states to simulate a push. First, we move to a temporary state while popping the topmost region. This is done in order to remember its content. Then, we push back that region unmodified. Finally, we push a region containing the given symbol, constructed from the previous topmost region such that the initial age of the symbol is in the given interval. Formally, for each transition rule $\tau = \langle q_1, \mathbf{push}(a, I), q_2\rangle \in \Delta^{\mathcal{T}}$, and each pair of regions R, R' such that $R' \in Make(R, a \in I)$, the set $\Delta^{\mathcal{T}}$ contains the transition rules:

- $\langle(\mathbf{0}, q_1), \epsilon, \mathbf{pop}(R), \mathtt{tmp}_1(\tau, R, \mathbf{0})\rangle$,
- $\langle\mathtt{tmp}_1(\tau, R, \mathbf{0}), \epsilon, \mathbf{push}(R), \mathtt{tmp}_2(\tau, R, \mathbf{0})\rangle$,
- $\langle\mathtt{tmp}_2(\tau, R, \mathbf{0}), \epsilon, \mathbf{push}(R'), (\mathbf{0}, q_2)\rangle$ (for simulating the prefix),
- $\langle(\mathbf{1}, q_1), a, \mathbf{pop}(R), \mathtt{tmp}(\tau, R, \mathbf{1})\rangle$,
- $\langle\mathtt{tmp}_1(\tau, R, \mathbf{1}), \epsilon, \mathbf{push}(R), \mathtt{tmp}_2(\tau, R, \mathbf{1})\rangle$,
- $\langle\mathtt{tmp}(\tau, R, \mathbf{1}), \epsilon, \mathbf{push}(R'), (\mathbf{1}, q_2)\rangle$ (for simulating the suffix).

$\mathbf{pop}(a, I)$ The simulation of pop transitions also requires two temporary states. First, we pop the topmost region and move to a temporary state. Then, in order to update the new topmost region, we need to first pop it, then rotate and merge it with the first region we popped, and finally push back the result. Formally, for each transition rule $\tau = \langle q_1, \mathbf{pop}(a, I), q_2\rangle \in \Delta^{\mathcal{T}}$, and all regions R_1, R_1', R_2, such that $R_2 \vDash (a \in I)$ and $R_1' \in \bigcup\{R_2 \odot R' \mid R' \in R_1^{++} \text{ and } R' \preceq R_2\}$, the set $\Delta^{\mathcal{T}}$ contains the transition rules:

- $\langle(\mathbf{0}, q_1), \epsilon, \mathbf{pop}(R_2), \mathtt{tmp}_1(\tau, R_2, \mathbf{0})\rangle$,
- $\langle\mathtt{tmp}_1(\tau, R_2, \mathbf{0}), \epsilon, \mathbf{pop}(R_1), \mathtt{tmp}_2(\tau, R_2, \mathbf{0})\rangle$,
- $\langle\mathtt{tmp}_2(\tau, R_2, \mathbf{0}), \epsilon, \mathbf{push}(R'), (\mathbf{0}, q_2))\rangle$ (for simulating the prefix),
- $\langle(\mathbf{1}, q_1), a, \mathbf{pop}(R_2), \mathtt{tmp}_1(\tau, R_2, \mathbf{1})\rangle$,
- $\langle\mathtt{tmp}_1(\tau, R_2, \mathbf{1}), \epsilon, \mathbf{pop}(R_1), \mathtt{tmp}_2(\tau, R_2, \mathbf{1})\rangle$,
- $\langle\mathtt{tmp}_2(\tau, R_2, \mathbf{1}), \epsilon, \mathbf{push}(R'), (\mathbf{1}, q_2))\rangle$ (for simulating the suffix).

Timed Transitions

For every state $q \in Q^{\mathcal{T}}$ and every pair of regions R, R' such that $R' \in R^+[\vdash \leftarrow [0:0]]$ (this is a singleton set), the set $\Delta^{\mathcal{P}}$ contains the transition rules:

- $\langle(\mathbf{0}, q), \epsilon, \mathbf{pop}(R), \mathtt{tmp}(timed, q, R, \mathbf{0})\rangle$,
- $\langle\mathtt{tmp}(timed, q, R, \mathbf{0}), \epsilon, \mathbf{push}(R'), (\mathbf{0}, q)\rangle$.

Additionally, if $R' \vDash (x_{control} \in [0:1))$, then $\Delta^{\mathcal{P}}$ also contains the transitions

- $\langle(\mathbf{1}, q), \epsilon, \mathbf{pop}(R), \mathtt{tmp}(timed, R, \mathbf{1})\rangle$,
- $\langle\mathtt{tmp}(timed, R, \mathbf{1}), \epsilon, \mathbf{push}(R'), (\mathbf{1}, q)\rangle$.

Switching Modes

In addition to the transitions described so far, $\mathcal{P}$ must also be able to switch from mode $\mathbf{0}$ to mode $\mathbf{1}$. This is done nondeterministically at any point in the simulation of the prefix. When the automaton changes mode, it resets the control clock $x_{control}$. For each state $q \in Q^{\mathcal{T}}$ and region R, the set $\Delta^{\mathcal{P}}$ contains the transition rules $\langle(\mathbf{0}, q), \epsilon, \mathbf{pop}(R), \mathtt{tmp}(switch, q, R)\rangle$ and $\langle\mathtt{tmp}(switch, q, R), \epsilon, \mathbf{push}(R'), (\mathbf{1}, q)\rangle$, where R' is the region in the singleton set $R[x_{control} \leftarrow 0]$.

Correctness. The simulation of the prefix (mode $\mathbf{0}$) works exactly like the simulation in [1]. The simulation of the suffix (mode $\mathbf{1}$) only imposes a restriction on the duration of the remaining run, namely that the value of the control clock $x_{control}$ may not reach 1. In other words, the automaton may simulate any unit suffix. Additionally, it reads an a each time it simulates a discrete transition. This, together with Lemma 1 implies the following result:

Lemma 3. *There exists a zeno run in $\mathcal{T}$ if and only if for the corresponding symbolic automaton $\mathcal{P}$, we have $a^\omega \in Traces(\mathcal{P})$.*

Using our construction, the size of $\mathcal{P}$ is exponential in the size of $\mathcal{T}$. The problem of checking $a^\omega \in Traces(\mathcal{P})$ is polynomial in the size of $\mathcal{P}$ [9]. This gives membership in EXPTIME for Theorem 2.

6 An EXPTIME Lower Bound for the Zenoness Problem

The following theorem gives EXPTIME-hardness for the zenoness problem for TPDA (matching its upper bound).

Theorem 4. *The zenoness problem for TPDA is* EXPTIME*-hard.*

Proof. The following problem is EXPTIME-complete [19]: Given a labelled pushdown automaton $\mathcal{P}$ recognizing the language L and n finite automata $A_1, \ldots, A_n$ recognizing languages $L_1, \ldots, L_n$, is the intersection $L \cap \bigcap_{i=1}^{n} L_i$ empty? This problem can be reduced, in polynomial time, to the zenoness problem for a TPDA $\mathcal{T}$. The pushdown part of $\mathcal{T}$ simulates $\mathcal{P}$, while a clock x_i encodes the state of the finite automaton A_i. We can use an additional control clock to ensure that no time passes during the simulation. We may assume w.l.o.g. that the finite automata are free of ϵ-transitions. An ϵ-transition of $\mathcal{P}$ is simulated by the pushdown part of $\mathcal{T}$. A labelled transition of $\mathcal{P}$ is first simulated by the pushdown part of $\mathcal{T}$ and then followed by a sequence of transitions that checks and updates the clocks in order to ensure that each finite automaton A_i is able to match the transition.

From a final state of $\mathcal{P}$, we introduce a series of transitions that checks if all finite-state automata A_i are also in their final states. If they are, we move to a special state of $\mathcal{T}$ from which there exists a zeno run. In this special state, we remove the restriction that time cannot pass and we add a self-loop performing a $\mathbf{nop}$ operation. Thus, the intersection $L \cap \bigcap_{i=1}^{n} L_i$ is empty if and only if $\mathcal{T}$ does not contain a zeno run. □

7 Conclusion and Future Work

In this paper, we have considered the problem of detecting zeno runs in TPDA. We showed that the zenoness problem for TPDA is EXPTIME-complete. The proof uses a reduction from the zenoness problem for TPDA to the problem of deciding whether a^ω is contained in the set of traces of a PDA. More specifically, given a TPDA $\mathcal{T}$, we construct a PDA $\mathcal{P}$ which simulates zeno runs of $\mathcal{T}$ and whose size is exponential in the size of $\mathcal{T}$.

We are currently considering the problem of computing the minimal (or infimal, if it does not exist) *reachability cost* in the model of *priced* TPDA, in which discrete transitions have *firing costs* and stack contents have *storage costs*, meaning that the cost of taking a timed transition depends on the stack content.

Another interesting question is whether there are fragments of some suitable metric logic for which model checking TPDA is decidable.

References

[1] P. A. Abdulla, M. F. Atig & J. Stenman (2012): *Dense-timed pushdown automata*. In: *Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on*, IEEE, doi:10.1109/LICS.2012.15.

[2] P. A. Abdulla, M. F. Atig & J. Stenman (2012): *The Minimal Cost Reachability Problem in Priced Timed Pushdown Systems*. In: *LATA*, doi:10.1007/978-3-642-28332-1_6.

[3] P. A. Abdulla, P. Mahata & R. Mayr (2005): *Decidability of Zenoness, syntactic boundedness and token-liveness for dense-timed petri nets*. In: *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, Springer, pp. 58–70, doi:10.1007/978-3-540-30538-5_6.

[4] R. Alur (1991): *Techniques for Automatic Verification of Real-Time Systems*. Ph.D. thesis, Dept. of Computer Sciences, Stanford University.

[5] R. Alur & D. L. Dill (1990): *Automata For Modeling Real-Time Systems*. In: *ICALP*, *LNCS* 443, Springer, pp. 322–335, doi:10.1007/BFb0032042.

[6] R. Alur & D. L. Dill (1994): *A Theory of Timed Automata*. *Theor. Comput. Sci.* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[7] M. Benerecetti, S. Minopoli & A. Peron (2010): *Analysis of Timed Recursive State Machines*. In: *TIME*, IEEE Computer Society, pp. 61–68, doi:10.1109/TIME.2010.10.

[8] A. Bouajjani, R. Echahed & R. Robbana (1994): *On the Automatic Verification of Systems with Continuous Variables and Unbounded Discrete Data Structures*. In: *Hybrid Systems*, LNCS 999, Springer, pp. 64–85, doi:10.1007/3-540-60472-3_4.

[9] A. Bouajjani, J. Esparza & O. Maler (1997): *Reachability Analysis of Pushdown Automata: Application to Model-Checking*. In: *CONCUR*, LNCS 1243, Springer, pp. 135–150, doi:10.1007/3-540-63141-0_10.

[10] P. Bouyer, F. Cassez, E. Fleury & K. G. Larsen (2004): *Optimal Strategies in Priced Timed Game Automata*. In: *FSTTCS*, LNCS 3328, Springer, pp. 148–160, doi:10.1007/978-3-540-30538-5_13.

[11] P. Bouyer & F. Laroussinie (2008): *Model Checking Timed Automata*. In Stephan Merz & Nicolas Navet, editors: *Modeling and Verification of Real-Time Systems*, ISTE Ltd. – John Wiley & Sons, Ltd., pp. 111–140, doi:10.1002/9780470611012.ch4.

[12] Z. Dang (2003): *Pushdown timed automata: a binary reachability characterization and safety verification*. *Theor. Comput. Sci.* 302(1-3), pp. 93–121, doi:10.1016/S0304-3975(02)00743-0.

[13] Z. Dang, T. Bultan, O. H. Ibarra & R. A. Kemmerer (2004): *Past pushdown timed automata and safety verification*. *Theor. Comput. Sci.* 313(1), pp. 57–71, doi:10.1016/j.tcs.2003.10.004.

[14] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer & J. Su (2000): *Binary Reachability Analysis of Discrete Pushdown Timed Automata*. In: *CAV*, LNCS 1855, Springer, pp. 69–84, doi:10.1007/10722167_9.

[15] M. Emmi & R. Majumdar (2006): *Decision Problems for the Verification of Real-Time Software*. In: *HSCC*, LNCS 3927, Springer, pp. 200–211, doi:10.1007/11730637_17.

[16] J. Esparza, D. Hansel, P. Rossmanith & S. Schwoon (2000): *Efficient Algorithms for Model Checking Pushdown Systems*. In: *CAV*, *LNCS* 1855, Springer, doi:10.1007/10722167_20.

[17] J. Esparza & S. Schwoon (2001): *A BDD-Based Model Checker for Recursive Programs*. In: *CAV*, *LNCS* 2102, Springer, pp. 324–336, doi:10.1007/3-540-44585-4_30.

[18] F. Herbreteau, B. Srivathsan & I. Walukiewicz (2012): *Efficient emptiness check for timed büchi automata*. *Formal Methods in System Design* 40(2), pp. 122–146, doi:10.1007/s10703-011-0133-1.

[19] A. Heußner, J. Leroux, A. Muscholl & G. Sutre (2012): *Reachability Analysis of Communicating Pushdown Systems*. *Logical Methods in Computer Science* 8(3:23), pp. 1–20, doi:10.2168/LMCS-8(3:23)2012.

[20] S. Schwoon (2002): *Model-Checking Pushdown Systems*. Ph.D. thesis, Technische Universität München.

[21] S. Tripakis (1999): *Verifying Progress in Timed Systems*. In: *ARTS*, *LNCS* 1601, Springer, pp. 299–314, doi:10.1007/3-540-48778-6_18.

[22] A. Trivedi & D. Wojtczak (2010): *Recursive timed automata*. In: *Proceedings of the 8th international conference on Automated technology for verification and analysis*, ATVA, pp. 306–324, doi:10.1007/978-3-642-15643-4_23.

Algorithmic Verification of Continuous and Hybrid Systems

Oded Maler
CNRS-VERIMAG,
University of Grenoble
France

We provide a tutorial introduction to *reachability computation*, a class of computational techniques that exports verification technology toward continuous and hybrid systems. For open under-determined systems, this technique can sometimes replace an infinite number of simulations.

1 Introduction

The goal of this article is to introduce the fundamentals of algorithmic verification of *continuous dynamical systems* defined by *differential equations* and of hybrid dynamical systems defined by *hybrid automata* which are dynamical systems that can switch between several modes of continuous dynamics. There are two types of audience that I have in mind. The first is verification-aware computer scientists who know finite-state automata and their algorithmic analysis. For this audience, the conceptual scheme underlying the presented algorithms will be easy to grasp as it is mainly inspired by symbolic model-checking of non-deterministic automata. Putting aside dense time and differential equations, dynamical systems can be viewed by this audience as a kind of infinite-state reactive program defined over the so-called real numbers. For this reason, I will switch from continuous to discrete-time discourse quite early in the presentation.

The other type of audience is people coming from control or other types of engineering and applied mathematics. These are relatively well versed in the concrete mathematics of continuous systems and should first be persuaded that the verification question is interesting, despite the fact that airplanes can fly (and theoretical papers can be written) without it. In my attempts to accommodate these two types of audience, I have written some explanation that will look trivial to some but this cannot be avoided while trying to do genuine inter-disciplinary research[1] (see also [65] for an attempt to unify discrete and continuous systems and [69] for some historical reflections). My intention is to provide a synthetic introduction to the topic rather than an exhaustive survey, hence the paper is strongly biased toward techniques closer to my own research. In particular, I will not deal with decidability results for hybrid systems with simplified dynamics and not with deductive verification methods that use invariants, barriers and Lyapunov functions. I sincerely apologize to those who will not find citations of their relevant work.

The rest of the paper is organized as follows. Section 2 describes the problem and situates it in context. Section 3 gives the basic definitions of reachability notions used throughout the paper. Section 4 presents the principles of set-based computation as well as basic issues related to the computational treatment of sets in general and convex polytopes in particular. Section 5 is devoted to reachability techniques for *linear* and affine dynamical systems in both discrete and continuous time, the domain

[1] The need to impress the members of one's own community is perhaps the main reason for the sterility of many attempts to do inter-disciplinary research.

L. Clemente and L. Holik (Eds.): 15th International Workshop on
Verification of Infinite-State Systems (INFINITY'13)
EPTCS 140, 2014, pp. 48–69, doi:10.4204/EPTCS.140.4

where a lot of progress has been made in recent years. The extension of these techniques to hybrid and non-linear systems, an active area of research, is discussed in Section 6. I conclude with some discussion of related work and new research directions.

2 The Problem

This paper is concerned with the following problem that we define first in a quasi-formal manner:

Consider a continuous dynamical system with input defined over some bounded state space X and governed by a differential equation of the form[2]

$$\dot{x} = f(x, v)$$

where $v[t]$ ranges, for every t, over some pre-specified bounded set V of admissible input values. Given a set $X_0 \subset X$, compute all the states visited by trajectories of the system starting from any $x_0 \in X_0$.

The significance of this question to control is the following: consider a controller that has been designed and connected to its plant and which is subject to external disturbances modeled by v. Computing the reachable set allows one to verify that all the behaviors of the closed-loop system stay within a desired range of operation and do not reach a forbidden region of the state space. Proving such properties for systems subject to uncontrolled interaction with the external environment is the main issue in verification of programs and digital hardware from which this question originates. In verification you have a large automaton with inputs that represent non-deterministic (non-controllable) effects such as behaviors of users and interactions with other systems and you would like to know whether there is an input sequence that drives the automaton into a forbidden state.

Before going further, let me try to situate this problem in the larger control context. After all, control theory and practice have already existed for many years without asking this question nor trying to answer it. This question distinguishes itself from traditional control questions in the following respects:

1. It is essentially a *verification* rather than a *synthesis* question, that is, the controller is assumed to exist already. However, it has been demonstrated that variants of reachability computation can be used for synthesizing switching controllers for timed [14, 8, 21] and hybrid [9] systems.
2. External disturbances are modeled *explicitly* as a *set* of admissible inputs, which is not the case for certain control formulations.[3] These disturbances are modeled in a *set-theoretic* rather than *stochastic* manner, that is, only the set of *possible* disturbances is specified without any probability induced over it. This makes the system in question look like a system defined by *differential inclusions* [16] which are the continuous analog of non-deterministic automata: if you project away the input you move from $\dot{x} = f(x, v)$ to $\dot{x} \in F(x)$. Adding probabilities the the inputs will yield a kind of a stochastic differential equation [15].
3. The information obtained from reachability computation covers also the *transient behavior* of the system in question, and not only its *steady-state* behavior. This property makes the approach particularly attractive for the analysis of *hybrid* (discrete-continuous, numerical-logical) systems where the applicability of analytic methods is rather limited. Such hybrid models can express, for example, deviation from idealized linear models due to constraints and saturation as well as other switching phenomena such as thermostat-controlled heating or gear shifting, see [66] for

[2] We use the physicists' notation where $\dot{x}$ indicates dx/dt. It is amusing to note that the idea of having a dedicated notation for the special variable called Time, is not unique to Temporal Logic.

[3] See [68] for a short discussion of this intriguing fact.

a lightweight introduction to hybrid systems and more elaborate accounts in books, surveys and lecture notes such as [74, 17, 64, 48, 61, 80, 76, 30, 20, 4].

4. The notion of *to compute* has a more effective flavor, that is, to develop algorithms that produce a representation of the set of reachable states (or an approximation of it) which is computationally usable, for example, it can be checked for intersection with a bad set of states.

Perhaps the most intuitive explanation of what is going on in reachability computation (and verification in general) can be given in terms of *numerical simulation*, which is by far the most commonly-used approach for validating complex systems. Each individual simulation consists of picking *one* initial condition and *one* input stimulus (random, periodic, step, etc.), producing the corresponding trajectory using numerical integration and observing whether this trajectory behaves properly. Ideally, to be on the safe side, one would like to repeat this procedure with *all* possible disturbances which are uncountably many. Reachability computation achieves the same effect as *exhaustive* simulation by exploring the state space in a "breadth-first" manner: rather than running each individual simulation to completion and then starting a new one, we compute at each time step all the states reachable by *all* possible one-step inputs from states reachable in the previous step (see [67] for a more elaborate development of this observation and [70] for a more general discussion of *under-determined* systems and their simulation). This set-based simulation is, of course, much more costly than the simulation of an individual trajectory but it provides more confidence in the correctness of the system than a small number of individual simulations would. The paper is focused on one popular approach to reachability computation based on discretizing time and performing a kind of set-based numerical integration. Alternative approaches are mentioned briefly at the end.

3 Preliminaries

We assume a time domain $T = \mathbb{R}_+$ and a state space $X \subseteq \mathbb{R}^n$. A trajectory is a measurable partial function $\xi : T \to X$ defined over all T (infinite trajectory) or over an interval $[0,t] \subset T$ (a finite trajectory). We use the notation $\mathscr{T}(X)$ for all such trajectories and $|\xi| = t$ to denote the length (duration) of finite signals. We consider an input space $V \subseteq \mathbb{R}^m$ and likewise use $\mathscr{T}(V)$ to denote input signals $\zeta : T \to V$. A continuous dynamical system $S = (X, V, f)$ is a system defined by the differential equation

$$\dot{x} = f(x, v). \tag{1}$$

We say that ξ is the *response* of f to ζ from x if ξ is the solution of (1) for initial condition x and $v(.) = \zeta$. We denote this fact by $\xi = f_x(\zeta)$ and also as

$$x \xrightarrow{\zeta/\xi} x'$$

when $|\zeta| = t$ and $\xi[t] = x'$. In this case we say that x' is reachable from x by ζ within t time and write this as

$$R(x, \zeta, t) = \{x'\}.$$

This notion speaks of *one* initial state, *one* input signal and *one* time instant and its generalization for a *set* X_0 of initial states, for *all* time instants in an interval $I = [0,t]$ and for *all* admissible input signals in $\mathscr{T}(V)$ yields the definition of the reachable set:

$$R_I(X_0) = \bigcup_{x \in X_0} \bigcup_{t \in I} \bigcup_{\zeta \in \mathscr{T}(V)} R(x, \zeta, t).$$

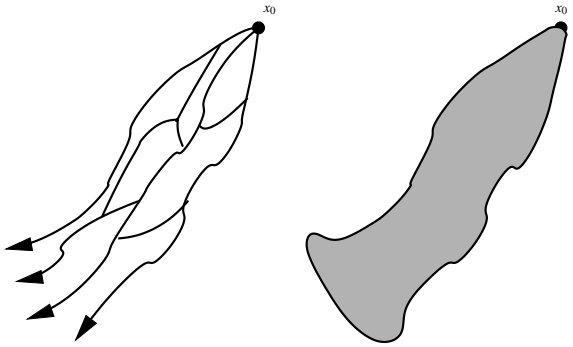


Figure 1: Trajectories induced by input signals from x_0 and the set of reachable states.

Figure 1 illustrates the induced trajectories and the reachable states for the case where $X_0 = \{x_0\}$. We will use the same R_I notation also when I is not an interval but an arbitrary time set. For example $R_{[1..r]}(X_0)$ can denote either the states reachable from X_0 by a continuous-time systems at discrete time instants, or states reachable by a discrete-time system during the first r steps.

Note that our introductory remark equating the relation between simulation and reachability computation to the relation between breadth-first and depth-first exploration of the space of trajectories corresponds to the commutativity of union:

$$\bigcup_{t\in I}\bigcup_{\zeta\in\mathscr{T}(V)} R(x,\zeta,t) = \bigcup_{\zeta\in\mathscr{T}(V)}\bigcup_{t\in I} R(x,\zeta,t).$$

4 Principles

In what follows we lay down the principles of one of the most popular approaches for computing reachable sets which is essentially a set-based extension of numerical integration.

4.1 The Abstract Algorithm

The *semigroup property* of dynamical systems, discrete and continuous alike, allows one to compute trajectories incrementally. The reachability operator also admits this property which is expressed as:

$$R_{[0,t_1+t_2]}(X_0) = R_{[0,t_2]}(R_{[0,t_1]}(X_0)).$$

Hence, the computation of $R_I(X_0)$ for an interval $I = [0,L]$ can be carried out by picking a time step r and executing the following algorithm:

Algorithm 1 (Abstract Incremental Reachability)

Input: *A set* $X_0 \subset X$
Output: $Q = R_{[0,L]}(X_0)$

$P := Q := X_0$
repeat $i = 1, 2 \ldots$
  $P := R_{[0,r]}(P)$
  $Q := Q \cup P$
until $i = L/r$

Remark: When interested in reachability for *unbounded* horizon, the termination condition $i = L/r$ should be replaced by $P \subseteq Q$, that is, the newly-computed reachable states are included in the set of states already computed. With this condition the algorithm is not guaranteed to terminate. Throughout most of this article we focus on reachability problems for a bounded time horizon.

4.2 Representation of Sets

The most urgent thing needed in order to convert the above scheme into a working algorithm is to choose a class of subsets of X that can be *represented* in the computer and be subject to the *operations* appearing in the algorithm. This is a very important issue, studied extensively (but often in low dimension) in computer graphics and computational geometry, but less so in the context of dynamical systems and control, hence we elaborate on it a bit bringing in, at least informally, some notions related to *effective computation*.

Mathematically speaking, subsets of $\mathbb{R}^n$ are defined as those points that satisfy some predicate. Such predicates are *syntactic* descriptions of the set and the points that satisfy them are the *semantic* objects we are interested in. The syntax of mathematics allows one to define weird types of sets which are not subject to any useful computation, for example, the set of irrational numbers. In order to compute we need to restrict ourselves to (syntactically characterized) classes of sets that satisfy the following properties:

1. Every set P in the class $\mathscr{C}$ admits a finite representation.
2. Given a representation of a set $P \in \mathscr{C}$ and a point x, it is possible to check in a finite number of steps whether $x \in P$.
3. For every operation $\circ$ on sets that we would like to perform and every $P_1, P_2 \in \mathscr{C}$ we have $P_1 \circ P_2 \in \mathscr{C}$. Moreover, given representations of P_1 and P_2 it should be possible to compute a representation of $P_1 \circ P_2$.

The latter requirement is often referred to as $\mathscr{C}$ being effectively *closed* under $\circ$. This requirement will later be relaxed into requiring that $\mathscr{C}$ contains a reasonable *approximation* of $P_1 \circ P_2$. To illustrate these notions, let us consider first a negative example of a class of sets admitting a finite representation but not satisfying requirements 2 and 3 above. The reachable set of a linear system $\dot{x} = Ax$ can be "computed" and represented by a finite formula of the form

$$R_I(X_0) = \{x : \exists x_0 \in X_0 \; \exists t \in I \; x = x_0 e^{At}\},$$

however this representation is not very useful because, in the general case, checking the membership of a point x in this set amounts to solving the reachability problem itself! The same holds for checking

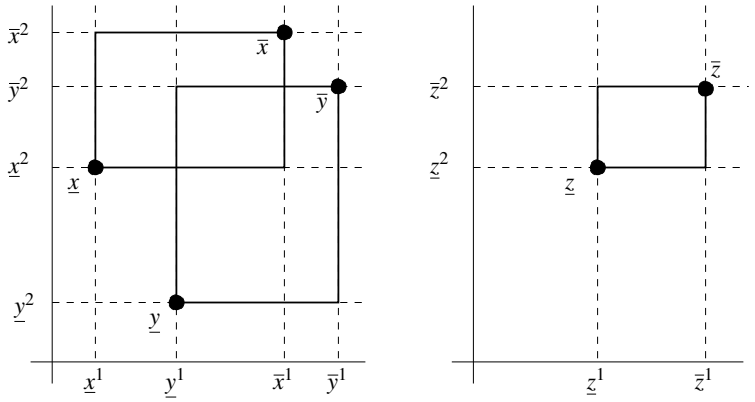


Figure 2: Intersecting two rectangles represented as $\langle \underline{x}, \overline{x} \rangle$ and $\langle \underline{y}, \overline{y} \rangle$ to obtain a rectangle represented as $\langle \underline{z}, \overline{z} \rangle$. The computation is done by letting $\underline{z}^1 = \max(\underline{x}^1, \underline{y}^1)$, $\underline{z}^2 = \max(\underline{x}^2, \underline{y}^2)$, $\overline{z}^1 = \min(\overline{x}^1, \overline{y}^1)$ and $\overline{z}^2 = \min(\overline{x}^2, \overline{y}^2)$.

whether this set intersects another set. On the other hand, a set defined by a quantifier-free formula of the form

$$\{x : g(x) \geq 0\},$$

where g is some computable function, admits in principle a membership check for every x: just evaluate $g(x)$ and compare with 0.

As a further illustration consider one of the simplest classes of sets, hyper-rectangles with axes-parallel edges and rational endpoints. Such a hyper-rectangle can be represented by its leftmost and rightmost corners $\underline{x} = (\underline{x}^1, \ldots, \underline{x}^n)$ and $\overline{x} = (\overline{x}^1, \ldots, \overline{x}^n)$. The set is defined as all points $x = (x^1, \ldots, x^n)$ satisfying

$$\bigwedge_{i=1}^{n} \underline{x}^i \leq x^i \leq \overline{x}^i,$$

a condition which is easy to check. As for operations, this class is effectively closed under translation (just add the displacement vector to the endpoints), dilation (multiply the endpoints by a constant) but not under rotation. As for Boolean set-theoretic operations, it is not hard to see that rectangles are effectively closed under intersection by component-wise max of their leftmost corners and component-wise min of their rightmost corners, see Figure 2. However they are not closed under union and complementation. This is, in fact, a general phenomenon that we encounter in reachability computations, where the basic sets that we work with are convex, but their union is not and hence the reachable sets computed by concrete realizations of Algorithm 1 will be stored as unions (lists) of convex sets (the recent algorithm of [38] is an exception).

As mentioned earlier, sets can be defined using combinations of inequalities and, not surprisingly, linear inequalities play a prominent role in the representation of some of the most popular classes of sets. We will mostly use *convex polytopes*, bounded polyhedra definable as conjunctions of linear inequalities. Let us mention, though, that Boolean combinations of *polynomial* inequalities define the *semi-algebraic* sets, which admit some interesting mathematical and computational results. Their algorithmics is, however, much more complex than that of polyhedral sets. The only class of sets definable by nonlinear inequalities for which relatively-efficient algorithms have been developed is the class of *ellipsoids*, convex sets defined as deformations of a unit circle by a (symmetric and positive definite)

linear transformation [54]. Ellipsoids can be finitely represented by their center and the transformation matrix and like polytopes, they are closed under linear transformations, a fact that facilitates their use in reachability computation for linear systems. Ellipsoids differ from polytopes by not being closed under intersection but such intersections can be approximated to some extent.

4.3 Convex Polytopes

In the following we list some facts concerning convex polytopes. These objects, which underlie other domains such as linear programming, admit a very rich theory of which we only scratch the surface. Readers interested in more details and precision may consult textbooks such as [75, 82].

A *linear inequality* is an inequality of the form $a \cdot x \leq b$ with a being an n-dimensional vector. The set of all points satisfying a linear inequality is called a *halfspace*. Note that the relationship between halfspaces and linear inequalities is not one-to-one because any inequality of the form $ca \cdot x \leq cb$, with c positive, will represent the same set. However using some conventions one can establish a unique representation for each halfspace. A convex polyhedron is an intersection of finitely many halfspaces. A convex polytope is a bounded convex polyhedron. A *convex combination* of a set $\{x_1, \ldots, x_l\}$ of points is any $x = \lambda_1 x_1 + \cdots + \lambda_l x_l$ such that

$$\bigwedge_{i=1}^{l} \lambda_i \geq 0 \wedge \sum_{i=1}^{l} \lambda_i = 1.$$

The *convex hull* of a set $\tilde{P}$ of points, denoted by $P = conv(\tilde{P})$, is the set of all convex combinations of its elements. Convex polytopes admit two types of canonical representations:

1. Vertices: each convex polytope P admits a finite minimal set $\tilde{P}$ such that $P = conv(\tilde{P})$. The elements of $\tilde{P}$ are called the *vertices* of P.
2. Inequalities: a convex polytope P admits a minimal set $H = \{H^1, \ldots, H^k\}$ of halfspaces such that $P = \bigcap_{i=1}^{k} H^i$. This set is represented syntactically as a conjunction of inequalities

$$\bigwedge_{i=1}^{k} a^i \cdot x \leq b^i.$$

Some operations are easier to perform on one representation and some on the other. Testing membership $x \in P$ is easier using inequalities (just evaluation) while using vertices representation, one needs to solve a system of linear equations to find the λ's. To check whether $P_1 \cap P_2 \neq \emptyset$ one can first combine syntactically the inequalities of P_1 and P_2 but in order to check emptiness, these inequalities should be brought into a canonical form. On the other hand, $conv(\tilde{P})$ is always non-empty for any non-empty $\tilde{P}$. Various (worst-case exponential) algorithms convert polytopes from one representation to the other.

The most interesting property of convex polytopes, which is also shared by ellipsoids, is the fact that they are closed under linear operators, that is, for a matrix A, if P is a convex polytope (resp. ellipsoid) so is the set

$$AP = \{Ax : x \in P\}$$

and this property is evidently useful for set-based simulation. The operation can be carried out using both representations of polytopes: if $P = conv(\{x_1, \ldots, x_l\})$ then $AP = conv(\{Ax_1, \ldots, Ax_l\})$ and we leave the computation based on inequalities as an exercise to the reader.

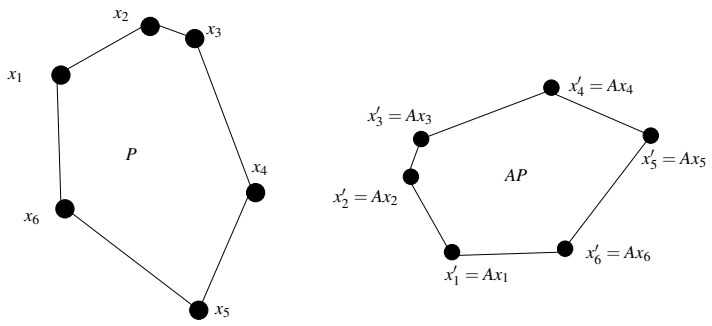


Figure 3: Computing AP from P by applying A to the vertices.

5 Linear Systems

Naturally, the most successful results on reachability computation have been obtained for systems with linear and affine dynamics, that is, systems defined by *linear differential equations*, not to be confused with the simpler "linear" hybrid automata (LHA) where the derivative of x in any state is independent of x. We will start by explaining the treatment of autonomous systems in discrete time, then move to continuous time and then to systems with bounded inputs. In particular we describe some relatively-recent complexity improvements based on a "lazy" representation of the reachable sets. This algorithm underlies the major verification procedure in the **SpaceEx** tool [39] and can handle quite large systems.

5.1 Discrete-Time Autonomous Systems

Consider a system defined by the recurrence equation

$$x_{i+1} = Ax_i.$$

In this case

$$R_{[0..L]}(X_0) = \bigcup_{i=0}^{L} A^i X_0$$

and the abstract algorithm can be realized as follows:

Algorithm 2 (Discrete-Time Linear Reachability)

Input: *A set* $X_0 \subset X$ *represented as* $conv(\tilde{P}_0)$
Output: $Q = R_{[0..L]}(X_0)$ *represented as a list* $\{conv(\tilde{P}_0), \ldots, conv(\tilde{P}_L)\}$

$P := Q := \tilde{P}_0$
repeat $i = 1, 2 \ldots$
  $P := AP$
  $Q := Q \cup P$
until $i = L$

The complexity of the algorithm, assuming $|\tilde{P}_0| = m_0$ is $O(m_0 LM(n))$ where $M(n)$ is the complexity of matrix-vector multiplication in n dimensions which is $O(n^3)$ for simple algorithms and slightly less for

fancier ones. As noted, this algorithm can be applied to other representations of polytopes, to ellipsoids and any other class of sets closed under linear transformations. If the purpose of reachability is to detect intersection with a set B of bad states we can weaken the loop termination condition into $(i = L) \vee (P \cap B \neq \emptyset)$ where the intersection test is done by transforming P into an inequalities representation. If we consider unbounded horizon and want to detect termination we need to check whether the newly-computed P is included in Q which can be done by "sifting" P through all the polytopes in Q and checking whether it goes out empty. This is not a simple operation but can be done. In any case, there is no guarantee that this condition will ever become true.

5.2 Continuous-Time Autonomous Systems

The approach just described can be adapted to *continuous-time* systems of the form

$$\dot{x} = Ax$$

as follows. First it is well known that by choosing a time step r and computing the corresponding matrix exponential $A' = e^{Ar}$ we obtain a discrete-time system

$$x_{i+1} = A'x_i$$

which approximates the original system in the sense that for every i, x_i of the discrete-time system is close to $x[ir]$ of the continuous-time system. The quality of the approximation can be indefinitely improved by taking smaller r. We then use the discrete time reachability operator to compute $P' = R_{\{r\}}(P) = A'P$, that is, the successors of P at time r and can use one out of several techniques to compute an approximation of $R_{[0,r]}(P)$ from P and P':

5.2.1 Make r Small

This approach, used implicitly by [53], just makes r small enough so that subsequent sets overlap each other and the difference between their unions and the continuous-time reachable set vanishes.

5.2.2 Bloating

Let P and P' be represented by the sets of vertices $\tilde{P}$ and $\tilde{P}'$ respectively. The set $\overline{P} = conv(\tilde{P} \cup \tilde{P}')$ is a good approximation of $R_{[0,r]}(P)$ but since in general, we would like to obtain an *over-approximation* (so that if the computed reachable sets does not intersect with the bad set, we are sure that the actual set does not either) we can bloat this set to ensure that it is an *outer* approximation of $R_{[0,r]}(P)$.

This can be done, for example, by pushing the facets of $\overline{P}$ outward by a constant derived from the Taylor approximation of the curve [12]. To this end we need first to transform $\overline{P}$ into an inequalities representation. An alternative approach [24] is to find this over-approximation via an optimization problem. Note that for autonomous systems we can modify Algorithm 1 by replacing the initialization by $P := Q := R_{[0,r]}(X_0)$ and the iteration by $P := R_{[r,r](P)}$, that is, the successors after exactly r time units. This way the over-approximation is done only *once* for $conv(\tilde{P}_0 \cup \tilde{P}_1)$ and then A is applied successively to this set [58].

5.2.3 Adding an Error Term

The last approach that we mention is particularly interesting because it can be used, as we shall see later, also for non-autonomous systems as well as nonlinear ones. Let Y and Y' be two subsets of $\mathbb{R}^n$. Their *Minkowski sum* is defined as

$$Y \oplus Y' = \{y + y' : y \in Y \wedge y' \in Y'\}.$$

The maximal distance between the sets $R_r(P)$ and $R_{[0,r]}(P)$ can be estimated globally. Then, one can fix an "error ball" E (could be a polytope for that matter) of that radius and over-approximate $R_{[0,r]}(P)$ as $AP \oplus E$. Since this computation is equivalent to computing the reachable set of the discrete time system $x_{i+1} = A'x_i + e$ with $e \in E$, we can use the techniques for systems with input described in the next section.

5.3 Discrete-Time Systems with Input

We can now move, at last, to open systems of the form

$$x_{i+1} = Ax_i + Bv_i$$

where v ranges over a bounded convex set V. The one-step successor of a set P is defined as

$$P' = \{Ax + Bv : x \in P, v \in V\} = AP \oplus BV.$$

Unlike linear operations that preserve the number of vertices of a convex polytope, the Minkowski sum increases their number and its successive application may prohibitively increase the representation size. Consequently, methods for reachability under disturbances need some compromise between exact computation that leads to explosion, and approximations which keep the representation size small but may accumulate errors to the point of becoming useless, a phenomenon also known in numerical analysis as the "wrapping effect" [50, 51]. We illustrate this tradeoff using three approaches.

5.3.1 Using Vertices

Assume both P and V are convex polytopes represented by their vertices, $P = conv(\tilde{P})$ and $V = conv(\tilde{V})$. Then it is not hard to see that

$$AP \oplus BV = conv(\{Ax + Bv : x \in \tilde{P}, v \in \tilde{V}\}).$$

Hence, applying the affine transformation to all combinations of vertices in $\tilde{P} \times \tilde{V}$ we obtain all the candidates for vertices of P' (see Figure 4). Of course, not all of these are actual vertices of P' but there is no known efficient procedure to detect which are and which are not. Moreover, it may turn out that the number of actual vertices indeed grows in a super-linear way. Neglecting the elimination of fictitious vertices and keeping all these points as a representation of P' will lead to $|\tilde{P}| \cdot |\tilde{V}|^k$ vertices after k steps, a completely unacceptable situation.

5.3.2 Pushing Facets

This approach over-approximates the reachable set while keeping its complexity more or less fixed. Assume P to be represented in (or converted into) inequality representation. For each supporting halfspace H^i defined by $a^i \cdot x \leq b^i$, let $v^i \in V$ be the disturbance vector which pushes H^i in the "outermost" way, that is, the one which maximizes the product $v \cdot a^i$ with the normal to H^i. In the discrete time setting described

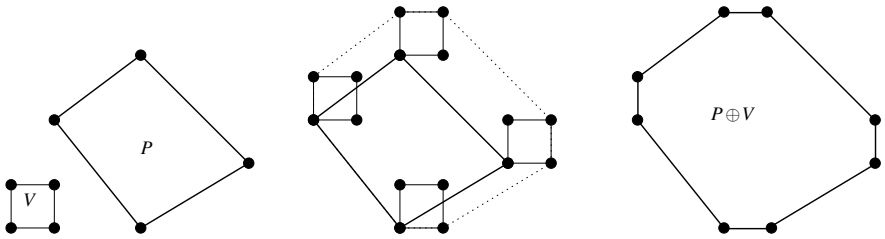


Figure 4: Adding a disturbance polytope V to a polytope P leads to a polytope $P \oplus V$ with more vertices. The phenomenon is more severe in higher dimensions.

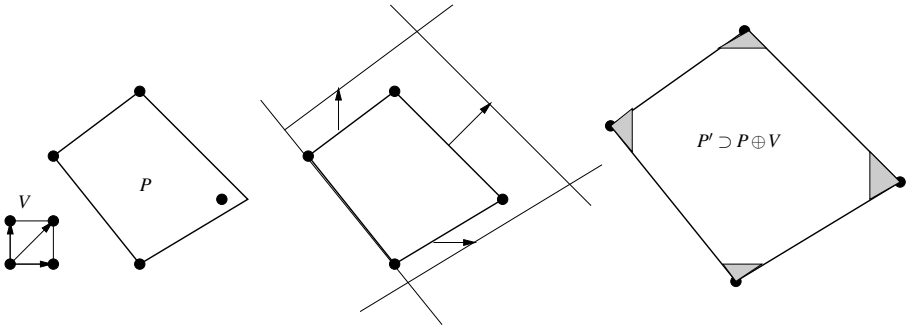


Figure 5: Pushing each face of P by the element of V which pushes it to the maximum. The result will typically not have more facets or vertices but may be a proper superset of $P \oplus V$ (shaded triangles represent the over-approximation error).

here, $v^i \in \tilde{V}$ for every i. We then apply to each H^i the transformation $Ax + Bv^i$ and the intersection of the hyperplanes thus obtained is an over-approximation of the successors (see Figure 5).

This approach has been developed first in the context of continuous time, starting with [81] who applied it to supporting planes of ellipsoids and then adapted in [28] for polytopes. It is also similar in spirit to the face lifting technique of [32]. In continuous time, the procedure of finding each v^i is a linear program derived from the maximum principle of *optimal control*. Recently is has been demonstrated that by using redundant constraints [13] the error can be reduced dramatically for quite large systems.

5.3.3 Lazy Representation

The previous sections showed that until recently, the treatment of linear systems with input provided two alternatives: either apply the linear transformation to an ever-increasing number of objects (vertices, inequalities, zonotope generators) and thus slowing down the computation in each step, or accumulate large over-approximation errors. The following observation due to Colas Le Guernic [57, 41] allows us to benefit from both worlds: it can restrict the application of the linear transformation A to a *fixed* number of points at each step while at the same time *not accumulating* approximation errors. Let us look at two consecutive sets P_k and P_{k+1} in the computation:

$$P_k = A^k P_0 \oplus A^{k-1} BV \oplus A^{k-2} BV \oplus \ldots \oplus BV$$

and

$$P_{k+1} = A^{k+1} P_0 \oplus A^k BV \oplus A^{k-1} BV \oplus \ldots \oplus BV.$$

As one can see, these two sets "share" a lot of common terms that need not be recomputed (and this holds also for continuous-time and variable time-steps). And indeed, the algorithm described in [41] computes the sequence $P_0,\ldots,P_k$ in $O(k(m_0+m)M(n))$ time. From this symbolic/lazy representation of P_i, one can produce an approximating polytope with any desired precision, but this object is *not* used to compute P_{i+1} and hence the wrapping effect is avoided.

A first prototype implementation of that algorithm could compute the reachable set after 1000 steps for linear systems with 200 state variables within 2 minutes. This algorithm was first discovered for zonotopes (a class of centrally symmetric polytopes closed under Minkowski sum proposed for reachability in [40]) but was later adapted to arbitrary convex sets represented by *support functions* [60, 58]. A re-engineered version of the algorithm has been implemented into the **SpaceEx** tool [39]. Although one might get the impression that the reachability problem for linear systems can be considered as solved, the development of **SpaceEx** under the direction of Goran Frehse has confirmed once more that a lot of work is needed in order to transform bright ideas into a working tool that can robustly handle large non-trivial systems occurring in practice.[4]

6 Hybrid and Nonlinear Systems

Being able to handle quite large linear systems, a major challenge is to extend reachability to richer classes of systems admitting hybrid or nonlinear dynamics.

6.1 Hybrid systems

The analysis of hybrid systems was the major motivation for developing reachability algorithms because unlike analytical methods, these algorithms can be easily adapted to handle discrete transitions and mode switching. Figure 6 shows a very simple hybrid automaton with two states, each with its own linear dynamics (using another terminology we have here a piecewise-linear or piecewise-affine dynamical system). An (extended) state of a hybrid system is a pair $(s,x)\in S\times X$ where s is the discrete state (mode). A transitions from state s_i to state s_j may occur when the condition $G_{ij}(x)$ (the transition guard) is satisfied by the current value of x. Such conditions are typically comparisons of state variables with thresholds or more generally linear inequalities. Moreover, while staying at discrete state s, the value of x should satisfy additional constraints, known as *state invariants*.[5] Like timed automata, hybrid automata can exhibit dense non-determinism in parts of the state-space where both a transition guard and a state invariant hold. The runs/trajectories of such an automaton are of the form

$$(s_1,x[0])\xrightarrow{t_1}(s_1,x[t_1])\longrightarrow(s_2,x[t_1])\xrightarrow{t_2}(s_2,x[t_1+t_2])\longrightarrow\cdots,$$

that is, an interleaving of *continuous trajectories* and *discrete transitions* taken at extended states where guards are satisfied. The adaptation of linear reachability computation so as to compute the reachable subset of $S\times X$ follows the procedure proposed already in [5] for simpler dynamics and implemented in the HyTech tool [45]. It goes like this: first, continuous reachability is applied using the dynamics A_1 of s_1, while respecting the state invariant I_1. Then the set of reachable states is intersected with the (semantics of the) transition guard G_{12}. The outcome serves as an initial set of states in s_2 where continuous linear reachability with A_2 and I_2 is applied and so on.

[4] Readers are encouraged to download **SpaceEx** at `http://spaceex.imag.fr/` to obtain a first hand experience in reachability computation.

[5] The full hybrid automaton model may also associate transitions with *reset maps* which are transformations (jumps) applied to x upon a transition.

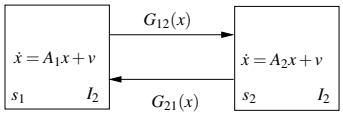


Figure 6: A simple hybrid system with two modes.

The real story is, of course, not that simple for the following reasons.

1. The intersection of the reachable states with the state invariant breaks the symbolic lazy representation as soon as some part of the reachable set leaves the invariant. Likewise, the change of dynamics after the transition invalidates the update scheme of the lazy representation and the set has to be over-approximated before doing intersection and reachability in the next state.
2. The dynamics might be "grazing" the transition guard, intersecting different parts of it at different time steps, thus spawning many subsequent computations. In fact, even a dynamics which proceeds orthogonally toward a single transition guard may spawn many such computations because when small time steps are used, many consecutive sets may have a non-empty intersection with the guard. Consequently, techniques for hybrid reachability such as [59] tend to cluster these sets together before conducting reachability in the next state thus increasing the over-approximation error. This error can now be controlled using the techniques of [38].
3. Even in the absence of these phenomena, when there are several transitions outgoing from a state we may end up with an exponentially growing number of runs to explore.

All these are problems, some tedious and some glorious, that need to be resolved if we want to provide a robustly working tool.

6.2 Nonlinear Systems

Many challenging problems in numerous engineering and scientific domains boil down to exploring the behaviors of nonlinear systems. The techniques described so far take advantage of the intimate relationships between linearity and convexity, in particular the identity $A \cdot conv(\tilde{P}) = conv(A\tilde{P})$. For nonlinear functions such properties do not hold and new ideas are needed. I sketch briefly two approches for adapting reachability for such systems: one which is general and is based on linearizing the system at various parts of the state-space thus obtaining a piecewise-linear system (a hybrid automaton) to which linear reachability techniques are applied. Other techniques look for more sophisticated data-structures and syntactical objects that can represent sets reached by specific classes of nonlinear systems such as those defined by polynomial dynamics. Unlike linear systems, linear reachability is still in an exploratory phase and it is too early to predict which of the techniques described below will survive.

6.2.1 Linearization/Hybridization

Consider a nonlinear system $\dot{x} = f(x)$ and a partition of its state space, for example into cubes (see Figure 7). For each cube s one can compute a linear function A_s and an error polytope V_s such that for every $x \in s$, $f(x) - A_s x \in V_s$ and hence we have a conservative approximation $f(x) \in A_s x \oplus V_s$. We can now build a hybrid automaton (a piecewise-affine dynamical system) whose states correspond to the cubes, and which makes transitions (mode switching) from s to s' whenever x crosses the boundary between them (see Figure 7). The automaton provides an over approximation of the nonlinear system in the sense

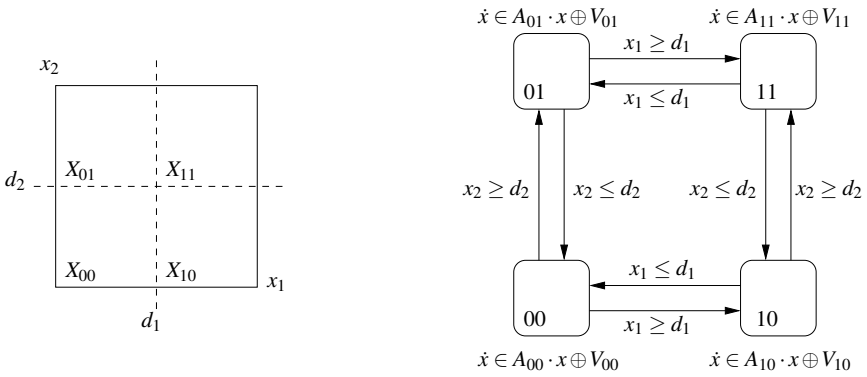


Figure 7: Hybridization: a nonlinear system is over-approximated by a hybrid automaton with an affine dynamics in each state. The transition guards indicate the conditions for switching between neighboring linearizations.

that any run of the latter corresponds to (a projection on X of) a run of the hybrid automaton. This technique, initially developed for doing simulations, has been coined *hybridization* in [11]. An earlier work [46] partitions the state-space similarly but approximates f in each cube by a simpler dynamics of the form $\dot{x} \in C$ for a constant polytope C.

To perform reachability computation on the automaton one can apply the linear techniques described in the preceding section using A_s and V_s, as long as the reachable states remain within cube s. Whenever some P_i reaches the boundary between s and s' we need to intersect it with the switching surface (the transition guard) and use the obtained result as an initial set for reachability computation in s' using $A_{s'}$ and $V_{s'}$. However, the difficulties previously mentioned concerning reachability for hybrid systems, and in particular the fact that the reachable set may leave a cube and penetrate into exponentially many neighboring cubes, renders hybridization impractical beyond 3 dimensions. A dynamic version of hybridization, not based on a fixed grid, has been introduced in [31] and is illustrated in Figure 8. The idea is quite simple: first a linearization domain around the initial set is constructed with the corresponding affine dynamics. Linear reachability computation is performed until the first time the computed polytope leaves the domain. Then the last step is undone, and a new linearization domain is constructed around the current set and linear reachability is resumed. Unlike static hybridization, linearization domains overlap and do not form a partition, but the inclusion of trajectories still hold by construction (see more details in [31]). The intersection operation and the artificial splitting of sets due to the fixed grid are altogether avoided.

The computation of the linear approximation of f and its error bound are costly procedures and should not be done too often. Hence it is important to choose the size and shape of the linearization domains such that the error is small and the computation stays in each domain as long as possible. This topic has been studied in [34] where the curvature of f has been used to construct and orient simplical linearization domains.

6.2.2 Specialized Methods

We mention a few other approaches for reachability computation for nonlinear systems. For polynomial systems, the Bernstein form of polynomials is used to either enclose the image of a set by the polynomial, or to approximate the polynomial by affine bound functions. The different representations using the

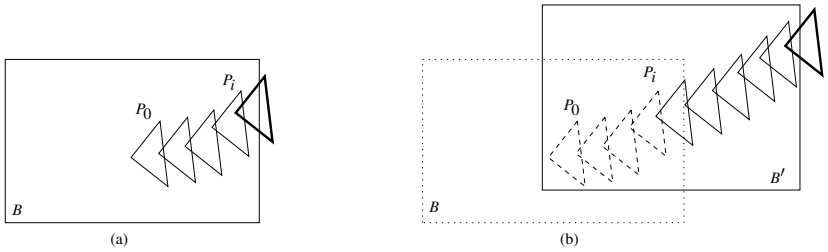


Figure 8: Dynamic hybridization: (a) Computing in some box until intersection with the boundary; (b) Backtracking one step and computing in a new box.

Bernstein form include the Bézier simplices [29] and the Bernstein expansion [35, 77]. A nonlinear function can be over-approximated, based on its Taylor expansion, by a polynomial and an interval which includes all the remainder values. Then, the integration of an ODE system can be done using the Picard operator with reachable sets are represented by boxes [22]. Similarly, in [2] non-linear functions are approximated by linear differential inclusions with dynamical error estimated as the remainder of the Taylor expansion around the current set. As a set representation, "polynomial zonotopes" based on a polynomial rather than linear combination of generators are used.

We may conclude that the extension of reachability computation to nonlinear and hybrid systems is a challenging problem which is still waiting for several conceptual and algorithmic breakthroughs. We believe that the ability to perform reachability computation for nonlinear systems of non-trivial size can be very useful not only for control systems but also for other application domains such as *analog circuits* and *systems biology*. In biological models, uncertainty in parameters and environmental condition is a rule, not an exception, and set-based simulation, combined with other approaches for exploring the uncertainty space of under-determined dynamical systems can be very beneficial.

7 Related and Future Work

The idea of set-based numerical integration has several origins. In some sense it can be seen as related to those parts of numerical analysis, for example *interval analysis* [72, 47], that provide "robust" set-based results for numerical computation to compensate for numerical errors. This motivation is slightly different from verification and control where the uncertainty is attributed to an external environment not to the internal computation. Set-based computations also underlie the *abstract interpretation* framework [27] for static analysis of software.

The idea of applying set-based computation to hybrid systems was among the first contributions of the verification community to hybrid systems research [5] and it has been implemented in the pioneering HyTech tool [45]. However this idea was restricted to hybrid automata with very simple continuous dynamics (a constant derivative in each state) where trajectories can be computed without numerical integration. To the best of our knowledge, the first explicit mention of combining numerical integration with approximate representation by polyhedra in the context of verification appeared in [43]. It was recently brought to my attention[6] that an independent thread of reachability computation, quite similar in motivation and techniques, has developed in the USSR starting from the early 70s [62]. More citations from this school can be found in [63].

[6] G. Frehse, personal communication.

The polytope-based techniques described here were developed independently in [24, 23, 25] and in [12, 28]. Among other similar techniques that we have not described in detail, let us mention again the extensive work on ellipsoids [54, 53, 19, 55] and another family of methods [71, 79] which uses techniques such as *level sets*, inspired by numerical solution of partial differential equations, to compute and represent the reachable states. Among the symbolic (non numerical) approaches to the problem let us mention [56] which computes an effective representation of the reachable states for a restricted class of linear systems. Attempts to scale-up reachability techniques to higher dimensions using compositional methods that analyze an abstract approximate systems obtained by projections on subsets of the state variables are described in [44] and [10].

The interpretation of V as the controller's output rather than disturbance transforms the reachability problem into some open-loop variant of controller synthesis [67, 68]. Hence it is natural that the optimization-based approach developed in [17] for synthesis, has also been applied to reachability computation [18]. On the other hand, reachability computation can be used to synthesize controllers in the spirit of dynamic programming, as has been demonstrated in [9] where a backward reachability operator has been used as part of an algorithm for synthesizing switching controllers.

Another class of methods, called *simulation-based*, for example [49, 42, 33, 37, 36, 1], attempts to obtain the same effect as reachability computation by finitely many simulations, not necessarily of extremal points as in the methods described in this paper. Such techniques may turn out to be superior for nonlinear systems whose dynamics does not preserve convexity and systems whose dynamics is expressed by programs that do not always admit a clean mathematical model.

Alternative approaches to verify continuous and hybrid systems algorithmically attempt to approximate the system by a simpler one, typically a finite-state automaton[7] [52, 7, 78]. This can be done by simply partitioning the state space into cubes and defining transitions between adjacent cubes which are connected by trajectories, or by more modern methods, inspired by program verification, such as *predicate abstraction* and *counter-example based refinement* [6, 26, 73]. It should be noted, however, that finite-state models based on space partitions suffer from the problem of *false transitivity*: the abstract system may have a run of the form $s_1 \to s_2 \to s_3$ while the concrete one has no trajectory $x_1 \to x_2 \to x_3$ passing through these regions. As a result, the finite-state model will often have too many spurious behaviors to be useful for verification.

Finally let me mention some other issues not discussed so far:

- Disturbance models: implicit in reachability computation is the assumption that the only restriction on the input signal is that it always remains in V. This means that it may oscillate in any frequency between the extremal values of V. Such a non realistic assumption may increase the reachable set and render the analysis too pessimistic. This effect can be reduced by composing the system with a bounded-variability non-deterministic model of the input generator but this will increase the size of the system. In fact, the technique of abstraction by projection [10] does the opposite: it projects away state-variables and converts them into bounded input variables. A more systematic study of precision/complexity tradeoffs could be useful here.
- Temporal properties: in our presentation we assumed implicitly that systems specifications are simply *invariance* properties, a subclass of safety properties which are violated once a trajectory reaches a forbidden state. It is, of course, possible to follow the usual procedure of taking a more complex temporal property, constructing its automaton and composing with the system model. This procedure will extend the discrete state-space of the system and will make the analysis harder by virtue of having more discrete transitions with the usual additional complications associated

[7] In fact, hybridization is another instance of this approach.

with detection of cycles in the reachability graph and extraction of concrete trajectories that realize them.

- Adaptive algorithms: although we attempt to be as general as possible, it should be admitted that different systems lead to different behaviors of the reachability algorithm, even for linear systems. If we opt for general techniques that do not require a dedicated super-intelligent user, we need to make the algorithms more adaptive to their own behavior and automatically explore different values of their parameters such as time steps, size and shapes of approximating polytopes, quick and approximate inclusion tests, different search strategies (for hybrid models) and more.
- Numerical aspects: the discourse in this paper treated real-valued computations as well-functioning black boxes. In practice, certain systems can be more problematic in terms of numerical stability or operate in several time scales and this should be taken into account.
- Differential-algebraic equations: many dynamical systems that model physical phenomena obeying conservation laws are modeled using differential-algebraic equations with relational constraints over variables and derivatives. In addition to the existing simulation technology for such systems, a specialized reachability theory should be developed, see for example [3].
- Discrete state explosion: the methods described here focus on scalability with respect to the dimensionality of the continuous state-space and tacitly assume that the number of discrete modes is not too large. This assumption can be wrong in at least two contexts: when discrete states are used to encode different parts of a high dimensional state-space as in hybridization or in the verification of complex control software when there is a relatively small number of continuous variables used to model the environment of the software which has many states. The problem of combining the symbolic representations for discrete and continuous spaces, for example BDDs and polytopes, is an unsolved problem already for the simplest case of timed automata.
- Finally, in terms of usability, the whole set-based point of view is currently not the natural one for practitioners (this is true also for verification versus testing in the discrete case) and the extraction of individual trajectories that violate the requirements as well as input signals that induce them will make reachability more acceptable as a powerful model debugging method.

Acknowledgments: This work benefitted from discussions with George Pappas, Bruce Krogh, Charles Rockland, Eugene Asarin, Thao Dang, Goran Frehse, Anoine Girard, Alexandre Donzé and Colas Le Guernic.

References

[1] Houssam Abbas, Georgios Fainekos, Sriram Sankaranarayanan, Franjo Ivančić & Aarti Gupta (2013): *Probabilistic temporal logic falsification of cyber-physical systems*. *ACM Transactions on Embedded Computing Systems (TECS)* 12(2s), p. 95, doi:10.1145/2465787.2465797.

[2] Matthias Althoff (2013): *Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets*. In: *HSCC*, ACM, pp. 173–182, doi:10.1145/2461328.2461358.

[3] Matthias Althoff & Bruce Krogh (2013): *Reachability Analysis of Nonlinear Differential-Algebraic Systems*. *Automatic Control, IEEE Transactions on* PP(99), doi:10.1109/TAC.2013.2285751.

[4] Rajeev Alur (2011): *Formal verification of hybrid systems*. In: *EMSOFT*, pp. 273–278, doi:10.1145/2038642.2038685.

[5] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis & Sergio Yovine (1995): *The Algorithmic Analysis of Hybrid Systems*. *Theoretical Computer Science* 138(1), pp. 3–34. Available at http://dx.doi.org/10.1016/0304-3975(94)00202-T.

[6] Rajeev Alur, Thao Dang & Franjo Ivancic (2006): *Counterexample-guided predicate abstraction of hybrid systems*. *Theoretical Computer Science* 354(2), pp. 250–271. Available at http://dx.doi.org/10.1016/j.tcs.2005.11.026.

[7] Rajeev Alur, Thomas A Henzinger, Gerardo Lafferriere & George J Pappas (2000): *Discrete abstractions of hybrid systems*. *Proceedings of the IEEE* 88(7), pp. 971–984, doi:10.1109/5.871304.

[8] E Asarin, O Maler, A Pnueli & J Sifakis (1998): *Controller synthesis for timed automata*. In: *Proc. System Structure and Control*, Elsevier. Available at http://www-verimag.imag.fr/PEOPLE/Oded.Maler/Papers/newsynth.pdf.

[9] Eugene Asarin, Olivier Bournez, Thao Dang, Oded Maler & Amir Pnueli (2000): *Effective synthesis of switching controllers for linear systems*. *Proceedings of the IEEE* 88(7), pp. 1011–1025, doi:10.1109/5.871306. Available at http://www-verimag.imag.fr/~maler/Papers/procieee.pdf.

[10] Eugene Asarin & Thao Dang (2004): *Abstraction by Projection and Application to Multi-affine Systems*. In: *HSCC*, *LNCS* 2993, Springer, pp. 32–47, doi:10.1007/978-3-540-24743-2_3. Available at http://springerlink.metapress.com/openurl.asp?genre=article{&}issn=0302-9743{&}volume=2993{&}spage=32.

[11] Eugene Asarin, Thao Dang & Antoine Girard (2007): *Hybridization methods for the analysis of nonlinear systems*. *Acta Informatica* 43(7), pp. 451–476. Available at http://dx.doi.org/10.1007/s00236-006-0035-7.

[12] Eugene Asarin, Thao Dang, Oded Maler & Olivier Bournez (2000): *Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems*. In: *HSCC*, *LNCS* 1790, Springer, pp. 20–31, doi:10.1007/3-540-46430-1_6. Available at http://link.springer.de/link/service/series/0558/bibs/1790/17900020.htm.

[13] Eugene Asarin, Thao Dang, Oded Maler & Romain Testylier (2010): *Using Redundant Constraints for Refinement*. In: *ATVA*, pp. 37–51, doi:10.1007/978-3-642-15643-4_5. Available at http://www-verimag.imag.fr/~maler/Papers/redundant.pdf.

[14] Eugene Asarin, Oded Maler & Amir Pnueli (1995): *Symbolic Controller Synthesis for Discrete and Timed Systems*. In: *Hybrid Systems II*, 999, Springer, pp. 1–20, doi:10.1007/3-540-60472-3_1. Available at http://www-verimag.imag.fr/~maler/Papers/symbolic.pdf.

[15] Karl J Astrom (1970): *Introduction to stochastic control theory*. Academic Press.

[16] Jean-Pierre Aubin & Arrigo Cellina (1984): *Differential inclusions : set-valued maps and viability theory*. *Grundlehren der mathematischen Wissenschaften* 264, Springer-Verlag, doi:10.1007/978-3-642-69512-4.

[17] Alberto Bemporad & Manfred Morari (1999): *Control of systems integrating logic, dynamics, and constraints*. *Automatica* 35(3), pp. 407–427, doi:10.1016/S0005-1098(98)00178-2.

[18] Alberto Bemporad, Fabio Danilo Torrisi & Manfred Morari (2000): *Optimization-based verification and stability characterization of piecewise affine and hybrid systems*. In: *Hybrid Systems: Computation and Control*, Springer, pp. 45–58, doi:10.1007/3-540-46430-1_8.

[19] Oleg Botchkarev & Stavros Tripakis (2000): *Verification of Hybrid Systems with Linear Differential Inclusions Using Ellipsoidal Approximations*. In: *HSCC*, *LNCS* 1790, Springer, pp. 73–88, doi:10.1007/3-540-46430-1_10. Available at http://link.springer.de/link/service/series/0558/bibs/1790/17900073.htm.

[20] Christos G Cassandras & John Lygeros (2010): *Stochastic hybrid systems*. CRC Press.

[21] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G Larsen & Didier Lime (2005): *Efficient on-the-fly algorithms for the analysis of timed games*. In: *CONCUR*, Springer, pp. 66–80, doi:10.1007/978-3-540-75454-1_3.

[22] Xin Chen, Erika Ábrahám & Sriram Sankaranarayanan (2012): *Taylor model flowpipe construction for non-linear hybrid systems*. In: *Proc. RTSS12*, IEEE, pp. 183–192.

[23] Alongkrit Chutinan (1999): *Hybrid System Verification using Discrete Model Approximations*. Ph.D. thesis, Carnegie Mellon University.

[24] Alongkrit Chutinan & Bruce H Krogh (1999): *Verification of Polyhedral-Invariant Hybrid Automata Using Polygonal Flow Pipe Approximations*. In: *HSCC*, *LNCS* 1569, Springer, pp. 76–90, doi:10.1007/3-540-48983-5_10. Available at `http://link.springer.de/link/service/series/0558/bibs/1569/15690076.htm`.

[25] Alongkrit Chutinan & Bruce H Krogh (2003): *Computational techniques for hybrid system verification*. *IEEE Transactions on Automatic Control* 48(1), pp. 64 – 75. Available at `http://dx.doi.org/10.1109/TAC.2002.806655`.

[26] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Joël Ouaknine, Olaf Stursberg & Michael Theobald (2003): *Abstraction and counterexample-guided refinement in model checking of hybrid systems*. *International Journal of Foundations of Computer Science* 14(04), pp. 583–604, doi:10.1142/S012905410300190X.

[27] Patrick Cousot & Radhia Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *POPL*, ACM, pp. 238–252, doi:10.1145/512950.512973.

[28] Thao Dang (2000): *Verification and Synthesis of Hybrid Systems*. Ph.D. thesis, Institut National Polytecnique de Grenoble.

[29] Thao Dang (2006): *Approximate Reachability Computation for Polynomial Systems*. In: *HSCC*, Springer, pp. 138–152, doi:10.1007/11730637_13.

[30] Thao Dang, Goran Frehse, Antoine Girard & Colas Le Guernic (2009): *Tools for the Analysis of Hybrid Models*. In: *Communicating Embedded Systems: Software and Design: Formal Methods*, John Wiley & Sons, Inc., pp. 227–251, doi:10.1002/9781118558188.ch7.

[31] Thao Dang, Colas Le Guernic & Oded Maler (2011): *Computing reachable states for nonlinear biological models*. *Theoretical Computer Science* 412(21), pp. 2095–2107, doi:10.1016/j.tcs.2011.01.014. Available at `http://www-verimag.imag.fr/~maler/Papers/nonlinear-bio-tcs.pdf`.

[32] Thao Dang & Oded Maler (1998): *Reachability Analysis via Face Lifting*. In: *HSCC*, *LNCS* 1386, Springer, pp. 96–109, doi:10.1007/3-540-64358-3_34. Available at `http://www-verimag.imag.fr/PEOPLE/Oded.Maler/Papers/facelift.pdf`.

[33] Thao Dang & Tarik Nahhal (2009): *Coverage-guided test generation for continuous and hybrid systems*. *Formal Methods in System Design* 34(2), pp. 183–213, doi:10.1007/s10703-009-0066-0.

[34] Thao Dang & Romain Testylier (2011): *Hybridization domain construction using curvature estimation*. In: *HSCC*, pp. 123–132, doi:10.1145/1967701.1967721.

[35] Thao Dang & Romain Testylier (2012): *Reachability analysis for polynomial dynamical systems using the Bernstein expansion*. *Reliable Computing* 17(2), pp. 128–152.

[36] Alexandre Donzé (2010): *Breach, a toolbox for verification and parameter synthesis of hybrid systems*. In: *CAV*, Springer, pp. 167–170, doi:10.1007/978-3-642-14295-6_17.

[37] Alexandre Donzé & Oded Maler (2007): *Systematic Simulation Using Sensitivity Analysis*. In: *HSCC*, *LNCS* 4416, Springer, pp. 174–189. Available at `http://dx.doi.org/10.1007/978-3-540-71493-4_16`.

[38] Goran Frehse, Rajat Kateja & Colas Le Guernic (2013): *Flowpipe approximation and clustering in space-time*. In: *HSCC*, pp. 203–212, doi:10.1145/2461328.2461361.

[39] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang & Oded Maler (2011): *SpaceEx: Scalable verification of hybrid systems*. In: *Computer Aided Verification*, pp. 379–395, doi:10.1007/978-3-642-22110-1_30. Available at http://www-verimag.imag.fr/~maler/Papers/spaceex-cav.pdf.

[40] Antoine Girard (2005): *Reachability of Uncertain Linear Systems Using Zonotopes*. In: *HSCC*, *LNCS* 3414, Springer, pp. 291–305, doi:10.1007/978-3-540-31954-2_19. Available at http://springerlink.metapress.com/openurl.asp?genre=article{&}issn=0302-9743{&}volume=3414{&}spage=291.

[41] Antoine Girard, Colas Le Guernic & Oded Maler (2006): *Efficient Computation of Reachable Sets of Linear Time-Invariant Systems with Inputs*. In: *HSCC*, *LNCS* 3927, Springer, pp. 257–271. Available at http://dx.doi.org/10.1007/11730637_21.

[42] Antoine Girard & George Pappas (2006): *Verification using simulation*. In: *Hybrid Systems: Computation and Control*, Springer, pp. 272–286, doi:10.1007/11730637_22.

[43] Mark R. Greenstreet (1996): *Verifying Safety Properties of Differential Equations*. In: *CAV*, *LNCS* 1102, Springer, pp. 277–287. Available at http://dx.doi.org/10.1007/3-540-61474-5_76.

[44] Mark R. Greenstreet & Ian Mitchell (1999): *Reachability Analysis Using Polygonal Projections*. In: *Hybrid Systems: Computation and Control*, *LNCS* 1569, Springer, pp. 103–116, doi:10.1007/3-540-48983-5_12. Available at http://link.springer.de/link/service/series/0558/bibs/1569/15690103.htm.

[45] Thomas A Henzinger, Pei-Hsin Ho & Howard Wong-Toi (1997): *HyTech: A model checker for hybrid systems*. In: *Computer aided verification*, Springer, pp. 460–463, doi:10.1007/3-540-63166-6_48.

[46] Thomas A Henzinger, Pei-Hsin Ho & Howard Wong-Toi (1998): *Algorithmic analysis of nonlinear hybrid systems*. *Automatic Control, IEEE Transactions on* 43(4), pp. 540–554, doi:10.1109/9.664156.

[47] Luc Jaulin, Michel Kieffer, Oliver Didrit & Éric Walter (2001): *Applied Interval Analysis*. Springer-Verlag, doi:10.1007/978-1-4471-0249-6.

[48] Mikael Johansson (2002): *Piecewise linear control systems*. Springer Verlag.

[49] Jim Kapinski, Bruce H Krogh, Oded Maler & Olaf Stursberg (2003): *On systematic simulation of open continuous systems*. In: *HSCC*, Springer, pp. 283–297, doi:10.1007/3-540-36580-X_22. Available at http://www-verimag.imag.fr/~maler/Papers/simulation.pdf.

[50] Wolfgang Kühn (1998): *Rigorously computed orbits of dynamical systems without the wrapping effect*. *Computing* 61(1), pp. 47–67, doi:10.1007/BF02684450.

[51] Wolfgang Kühn (1999): *Towards an optimal control of the wrapping effect*. In: *Developments in Reliable Computing*, Springer, pp. 43–51, doi:10.1007/978-94-017-1247-7_4.

[52] Robert P Kurshan & Kenneth L McMillan (1991): *Analysis of digital circuits through symbolic reduction*. *IEEE Trans. on CAD of Integrated Circuits and Systems* 10(11), pp. 1356–1371. Available at http://doi.ieeecomputersociety.org/10.1109/43.97615.

[53] Alexander B. Kurzhanski & Pravin Varaiya (2000): *Ellipsoidal Techniques for Reachability Analysis*. In: *HSCC*, *LNCS* 1790, Springer, pp. 202–214, doi:10.1007/3-540-46430-1_19. Available at http://link.springer.de/link/service/series/0558/bibs/1790/17900202.htm.

[54] Alexandr B Kurzhanski & István Vályi (1997): *Ellipsoidal Calculus for Estimation and Control*. Birkhauser, doi:10.1007/978-1-4612-0277-6.

[55] Alex A Kurzhanskiy & Pravin Varaiya (2007): *Ellipsoidal Techniques for Reachability Analysis of Discrete-Time Linear Systems*. *IEEE Transactions on Automatic Control* 52(1), pp. 26 –38. Available at http://dx.doi.org/10.1109/TAC.2006.887900.

[56] Gerardo Lafferriere, George J Pappas & Sergio Yovine (1999): *A new class of decidable hybrid systems*. In: *Hybrid Systems: Computation and Control*, Springer Berlin Heidelberg, pp. 137–151, doi:10.1007/3-540-48983-5_15.

[57] Colas Le Guernic (2005): *Calcul Efficace de l'Ensemble Atteignable des Systèmes Linéaires avec Incertitudes*. Master's thesis, Université Paris 7. Available at http://www.mpri.master.univ-paris7.fr/attached-documents/Stages-2005-rapports/rapport-2005-LeGuernic.pdf.

[58] Colas Le Guernic (2009): *Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics*. Ph.D. thesis, Université Grenoble 1 – Joseph Fourier. Available at http://tel.archives-ouvertes.fr/docs/00/43/07/40/PDF/CLeGuernic_thesis.pdf.

[59] Colas Le Guernic & Antoine Girard (2009): *Reachability Analysis of Hybrid Systems Using Support Functions*. In: *CAV*, pp. 540–554. Available at http://dx.doi.org/10.1007/978-3-642-02658-4_40.

[60] Colas Le Guernic & Antoine Girard (2010): *Reachability analysis of linear systems using support functions*. *Nonlinear Analysis: Hybrid Systems* 4(2), pp. 250–262, doi:10.1016/j.nahs.2009.03.002.

[61] Daniel Liberzon (2003): *Switching in systems and control*. Springer, doi:10.1007/978-1-4612-0017-8.

[62] A V Lotov (1971): *Construction of domains of attainability for a linear discrete system with bottle-neck constraints*. *Aerophysics and Applied Mathematics*, pp. 113–119. In Russian.

[63] Alexander V Lotov, Vladimir A Bushenkov & Georgy K Kamenev (2004): *Interactive decision maps: Approximation and visualization of Pareto frontier*. 89, Springer, doi:10.1007/978-1-4419-8851-5.

[64] John Lygeros, Shankar Sastry & Claire Tomlin (2001): *The art of hybrid systems*. Available at robotics.eecs.berkeley.edu/~sastry/ee291e/book.pdf. Unpublished manuscript.

[65] Oded Maler (1998): *A unified approach for studying discrete and continuous dynamical systems*. In: *CDC*, 2, pp. 2083–2088, doi:10.1109/CDC.1998.758641. Available at http://www-verimag.imag.fr/~maler/Papers/unified.pdf.

[66] Oded Maler (2001): *Guest Editorial: Verification of Hybrid Systems*. *European Journal of Control* 7(1), pp. 357–365, doi:10.3166/ejc.7.357-365. Available at http://www-verimag.imag.fr/~maler/Papers/guest.pdf.

[67] Oded Maler (2002): *Control from Computer Science*. *Annual Reviews in Control* 26(2), pp. 175–187, doi:10.1016/S1367-5788(02)00030-5. Available at http://www.sciencedirect.com/science/article/B6V0H-485P0W5-3/2/b5160ff386c03f13f06db257df3547e0.

[68] Oded Maler (2007): *On optimal and reasonable control in the presence of adversaries*. *Annual Reviews in Control* 31(1), pp. 1–15, doi:10.1016/j.arcontrol.2007.02.001. Available at http://www-verimag.imag.fr/~maler/Papers/annual.pdf.

[69] Oded Maler (2010): *Amir Pnueli and the dawn of hybrid systems*. In: *HSCC*, pp. 293–295, doi:10.1145/1755952.1755953. Available at http://www-verimag.imag.fr/~maler/Papers/amir-cpsweek.pdf.

[70] Oded Maler (2011): *On under-determined dynamical systems*. In: *EMSOFT*, pp. 89–96. Available at http://www-verimag.imag.fr/~maler/Papers/under-det.pdf.

[71] Ian Mitchell & Claire Tomlin (2000): *Level Set Methods for Computation in Hybrid Systems*. In: *HSCC*, *LNCS* 1790, Springer, pp. 310–323, doi:10.1007/3-540-46430-1_27. Available at http://link.springer.de/link/service/series/0558/bibs/1790/17900310.htm.

[72] Ramon E Moore (1979): *Methods and applications of interval analysis*. SIAM, doi:10.1137/1.9781611970906.

[73] Stefan Ratschan & Zhikun She (2005): *Safety verification of hybrid systems by constraint propagation based abstraction refinement*. In: *HSCC*, Springer, pp. 573–589, doi:10.1145/1210268.1210276.

[74] Abraham J van der Schaft & Johannes M Schumacher (2000): *An introduction to hybrid dynamical systems*. 251, Springer London.

[75] Alexander Schrijver (1986): *Theory of Linear and Integer Programming*. Wiley.

[76] Paulo Tabuada (2009): *Verification and control of hybrid systems: a symbolic approach*. Springer, doi:10.1007/978-1-4419-0224-5.

[77] Romain Testylier & Thao Dang (2013): *NLTOOLBOX: A Library for Reachability Computation of Nonlinear Dynamical Systems*. In: *ATVA*, pp. 469–473, doi:10.1007/978-3-319-02444-8_37.

[78] Ashish Tiwari (2008): *Abstractions for hybrid systems*. *Formal Methods in System Design* 32(1), pp. 57–83, doi:10.1007/s10703-007-0044-3.

[79] Claire J Tomlin, Ian Mitchell, Alexandre M Bayen & Meeko Oishi (2003): *Computational techniques for the verification of hybrid systems*. *Proceedings of the IEEE* 91(7), pp. 986–1001, doi:10.1109/JPROC.2003.814621.

[80] Stavros Tripakis & Thao Dang (2009): *Modeling, verification and testing using timed and hybrid automata*. In: *Model-Based Design for Embedded Systems*, CRC Press, pp. 383–436, doi:10.1201/9781420067859-c13.

[81] Pravin Varaiya (1998): *Reach set computation using optimal control*. In: *Proc. KIT Workshop on Verification of Hybrid Systems*, Verimag, Grenoble, pp. 377–383, doi:10.1007/978-3-642-59615-5_15.

[82] Günter M. Ziegler (1995): *Lectures on Polytopes*. *Graduate Texts in Mathematics* 152, Springer, doi:10.1007/978-1-4613-8431-1.

Synthesis of Parametric Programs using Genetic Programming and Model Checking

Gal Katz Doron Peled
Department of Computer Science, Bar Ilan University
Ramat Gan 52900, Israel

Formal methods apply algorithms based on mathematical principles to enhance the reliability of systems. It would only be natural to try to progress from verification, model checking or testing a system against its formal specification into constructing it automatically. Classical algorithmic synthesis theory provides interesting algorithms but also alarming high complexity and undecidability results. The use of genetic programming, in combination of model checking and testing, provides a powerful heuristic to synthesize programs. The method is not completely automatic, as it is fine tuned by a user that sets up the specification and parameters. It also does not guarantee to always succeed and converge towards a solution that satisfies all the required properties. However, we applied it successfully on quite nontrivial examples and managed to find solutions to hard programming challenges, as well as to improve and to correct code. We describe here several versions of our method for synthesizing sequential and concurrent systems.

1 Introduction

Formal methods [16] assist software and hardware developers in enhancing the reliability of systems. They provide methods and tools to search for design and programming errors. While these methods are effective in the software development process, they also suffer from severe limitations: testing is not exhaustive, formal verification is extremely tedious and model checking is limited to particular domains (usually, finite state systems) and suffers from high complexity, where memory and time requirements are sometimes prohibitively high.

A natural progress from formal methods are algorithms for automatically converting the formal specification into code or a description of hardware. Such algorithms would create correct-by-design code or piece of hardware. However, high complexity [19] and even undecidability [20] appear in some main classical automatic synthesis problems.

The approach presented here is quite different from algorithmic synthesis. We perform a generate-and-check kind of synthesis and use model checking or SAT solving to evaluate the generated candidates. An extreme approach would be to enumerate the possible programs (say, up to a certain size) and use model checking to find the correct solution(s). This was applied in Taubenfeld [3] to find mutual exclusion algorithms. Our synthesis method is based on *genetic programming*. It allows us to generate multiple candidate solutions at random and to mutate them, as a stochastic process. We employ enhanced model checking (model checking that does not only produce an affirmation to the checked properties or a counterexample, but distinguishes also some finer level of correctness) to provide *fitness* levels that are used to direct the search towards solutions that satisfy the given specification. Our synthesis method can be seen as a heuristic search in the space of syntactically fitting programs. It is not completely automatic, in the sense that the user can refine the specification and change the way the fitness is evaluated when the formal properties are satisfied. Our method is not guaranteed to terminate with a correct solution; we might give up after some time and can restart the search from a new random seed or with a refinement of the way the method assigns fitness.

L. Clemente and L. Holik (Eds.): 15th International Workshop on
Verification of Infinite-State Systems (INFINITY'13)
EPTCS 140, 2014, pp. 70–84, doi:10.4204/EPTCS.140.5

Although this marriage between genetic programming and model checking is quite promising, it suffers from some limitations of model checking. First, model checking is primarily designed for finite state systems. Although some extensions of it exist (e.g., to programs with a single stack), model checking does not work in general for parametric systems. Unfortunately, most systems that we would like to synthesize are parametric in nature: almost every abstract algorithm on data structures, be it queue, tree, graph, is parametric, where the size of the structure, is not fixed. It is easy to demonstrate model checking on a sorting program with a fixed vector of numbers and some fixed initial assignment of values. However, when the length of the vector is parametric, and we need to prove correctness with respect to arbitrary set of values, existing model checking techniques often fail.

For this reason, we use model checking in our approach for synthesizing parametric systems not as a comprehensive method for finding correctness, but as a generalized testing tool, which can make exhaustive checks for fixed parameters. Under this setting, we accept candidate programs when there is ample evidence that they are correct, specifically, when they passed enough checks, rather than when we establish comprehensive correctness.

Our genetic programming synthesis approach allows us not only to generate code that satisfies a given temporal specification but also to improve and correct code. We can start with an existing solution for a specification, and use the genetic process to improve it. We can also start with some flawed version of the code and use our method to correct it.

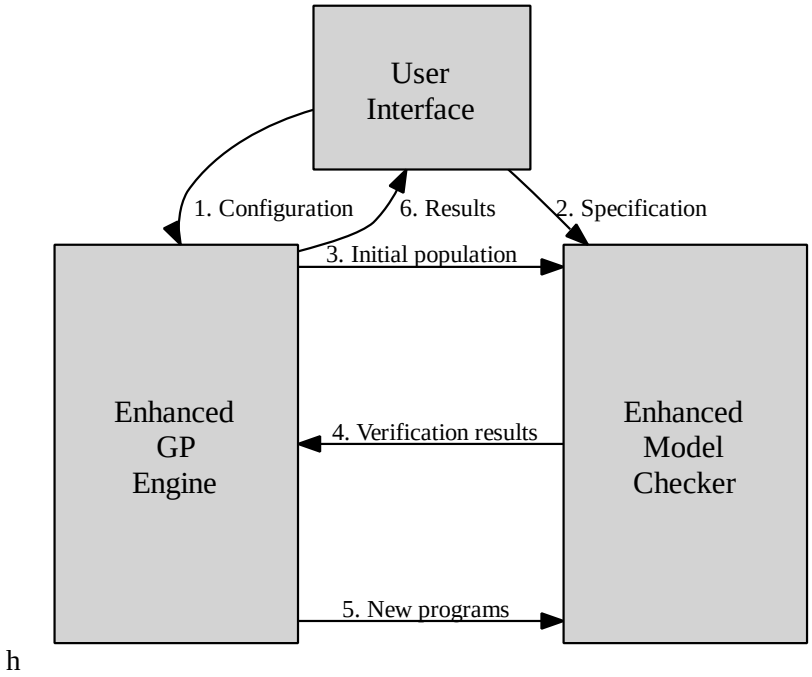
2 Genetic Programming Based on Model Checking

We present in [8, 9, 10, 11, 12] a framework combining genetic programming and model checking, which allows to automatically synthesize code for given problems. The framework we suggest is depicted in Figure 1.

- The *formal specification* of the problem, as well as the required architecture and constraints on the structure of the desired solutions is provided by the user. This may also include some initial versions of the desired code that either need correction or improvement.
- An *enhanced GP engine* that generates random programs and then evolves them using mutation operations that allow to change the code randomly.
- A *verifier* that analyzes the generated programs, and provides useful information about their correctness. This can be a model checker, often enhanced to provide more information than yes/no (and counterexample), or a SAT solver.

The synthesis process goes through the following steps:

1. The user feeds the GP engine with the desired architecture and a set of constraints regarding the programs that are allowed to be generated. This includes:
   (a) a set of functions literals and instructions used as building blocks for the generated programs,
   (b) the number of concurrent processes, the methods of communication between processes (in case of concurrent programs),
   (c) limitations on the size and structure of the generated programs, and the maximal number of permitted iterations.
   (d) The user may also provide some initial versions of the code that may be either incorrect or suboptimal. The genetic process can exploit these versions to evolve into better (correct or optimized) code.



h

Figure 1: The Suggested Framework

2. The user provides a formal specification for the problem. This consists, in our case, of a set of linear temporal logic properties, as well as additional quantitative requirements on the program behavior.
3. The GP engine randomly generates an *initial population* of candidate programs based on the provided building blocks and constraints.
4. The verifier analyzes the behavior of the generated candidates against the specification properties, and provides *fitness measures* based on the amount of satisfaction.
5. The GP engine creates new programs by applying the genetic operations of *mutation*, which performs small random changes to the code, and *crossover*, which glues together parts of different candidate solutions. Steps 4 and 5 are repeated until either a perfect program is found (fully satisfying the specification), or until the maximal number of iterations is reached.
6. The results are sent back to the user. This includes programs that satisfy all the specification properties, if one exists, or the best partially correct programs that was found, along with its verification results.

For steps 4 and 5 above, we use the following selection method:

- Randomly select μ candidate programs.
- Create λ new candidates by applying mutation (and optionally crossover) operations (as explained below) to the above μ candidates. We now have $\mu + \lambda$ candidates.
- Calculate the fitness function for each of the new candidates based on "enhanced model checking".
- Based on the calculated fitness, choose new μ candidates from the set of $\mu + \lambda$ candidates. Candidates with higher fitness values are selected with a higher probability than others. Replace the originally selected μ with the ones selected at this step.

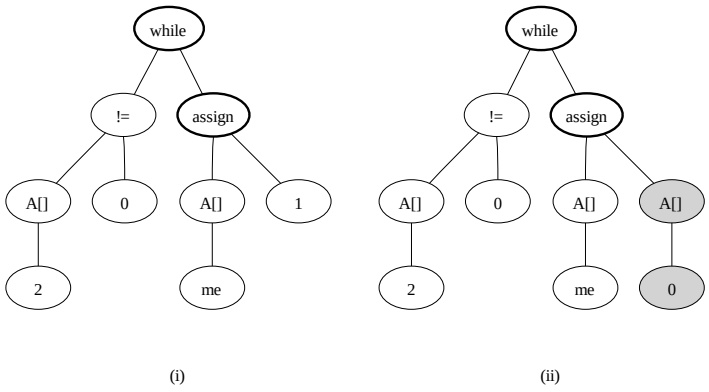


Figure 2: (i) Randomly created program tree, (ii) the result of a replacement mutation

We represent programs as trees, where an instruction or an expression is represented by a single node, having its parameters as its offspring. Terminal nodes represent constants. Examples of the instructions we use are *assignment, while , if* and *block.* The latter construct is a sequential composition of a pair of instructions.

At the first step, an initial population of candidate programs is generated. Each program is generated recursively, starting from the root, adding nodes until the tree is completed. The root node is chosen randomly from the set of instruction nodes, and each child node is chosen randomly from the set of nodes allowed by its parent type, and its place in the parameter list. Figure 2(i) shows an example of a randomly created tree that represents the following program:

```
while (A[2] != 0)
    A[me] = 1
```

The main operation we use is *mutation.* It allows making small changes on existing trees. The mutation includes the following steps:

1. Randomly choose a node *s* from the program tree.
2. Apply one of the following operations to the tree with respect to ihe chosen node:
   (a) Replace the subtree with root *s* with a new randomly generated subtree.
   (b) Add an immediate parent to *s*. Randomly create other offspring to the new parent, if needed.
   (c) Replace the node *s* by one of its offspring. Delete the remaining offspring of that node.
   (d) Delete the subtree with root *s*. The node ancestors should be updated recursively.

Mutation of type (a) can replace either a single terminal or an entire subtree. For example, the terminal "1" in the tree of Figure 2(i), is replaced by the grayed subtree in 2(ii), changing the assignment instruction into `A[me] = A[0]`. Mutations of type (b) can extend programs in several ways, depending on the new parent node type. In case a "block" type is chosen, a new instruction(s) will be inserted before or after the mutation node. For instance, the grayed part of Figure 3 represents a second assignment instruction inserted into the original program. Similarly, choosing a parent node of type "while" will have the effect of wrapping the mutation node with a while loop. The type of mutation applied to candidate programs is randomly selected. All mutations must of course produce legal code. This affects the possible mutation type for the chosen node, and the type of new generated nodes.

Another operation that is frequently used in genetic programming is *crossover*. The crossover operation creates new candidates by merging building blocks of two existing programs. The crossover steps are:

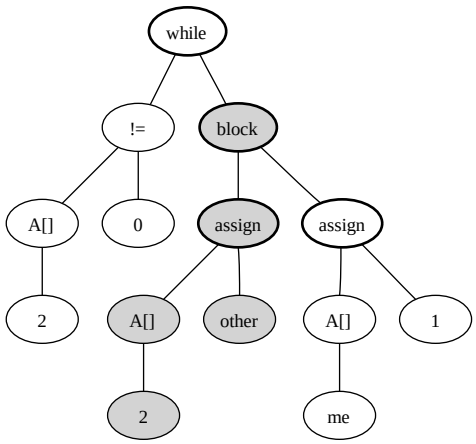


Figure 3: Tree after insertion mutation

1. Randomly choose a node from the first program.
2. Randomly choose a node from the second program that has the same type as the first node.
3. Exchange between the subtrees rooted by the two nodes, and use the two new programs created by this method.

While traditional GP is heavily based on crossover, it is quite a controversial operation (see [2], for example). Crossover is not used in our work.

Fitness is used by GP in order to choose which programs have a higher probability to survive and participate in the genetic operations. In addition, the success termination criterion of the GP algorithm is based on the fitness value of the most fitted candidate. Traditionally, the fitness function is calculated by running the program on some set of inputs (a training set), which represent the possible inputs. In contrast, our fitness function is not based on running the programs on sample data, but on an enhanced model checking procedure. While the classical model checking provides a yes/no answer to the satisfiablity of the specification (thus yielding a two-valued fitness function), our model checking algorithm generates a smoother function by providing several levels of correctness. Often, we have the following four levels of correctness, per each linear temporal logic property:

1. None of the executions of the program satisfy the property.
2. Some, but not all the executions of the program satisfy the property.
3. The only executions that do not satisfy the property must have infinitely many decisions that avoid a path that does satisfy the property.
4. All the executions satisfy the property.

We provided several methods for generating the various fitness levels:

- Using Streett Automata, and a strongly-connected component analysis of the program graph [9].
- Enhanced model checking logic and algorithm [8, 15].
- Probabilistic model checking.

There are several other considerations in setting up the calculation of the fitness. First, priority between the properties is used to suppress assigning fitness value due to the satisfaction of a liveness property (e.g., “when a process wants to enter its critical section, it would eventually be able to do

so") when the safety property does not hold (e.g., "the two processes cannot enter their critical sections simultaneously"). Another consideration is to prevent needless growth of the program by useless code. To alleviate this, we use some negative fitness value related to the program's length. This entails that a solution that satisfies all the specification is accepted even if it does not have perfect fitness value (due to length).

3 Example: Mutual Exclusion Algorithms

As an example, we used our method in order to automatically generate solutions to several variants of the Mutual Exclusion Problem. In this problem, first described and solved by Dijkstra [5], two or more processes are repeatedly running critical and non-critical sections of a program. The goal is to avoid the simultaneous execution of the critical section by more than one process. We limit our search for solutions to the case of only two processes. The problem is modeled using the following program parts that are executed in an infinite loop:

```
Non Critical Section
Pre Protocol
Critical Section
Post Protocol
```

These parts are fixed, and, together with the number of processes involved (two) and the number of variables allowed, consist of the architecture provided to our genetic programming tool, together with the temporal specification.

The `Non Critical Section` part represents the process part on which it does not require an access to the shared resource. A process can make a nondeterministic choice whether to stay in that part, or to move into the `Pre Protocol` part. From the `Critical Section` part, a process always has to move into the `Post Protocol` part. The `Non Critical Section` and `Critical Section` parts are fixed, while our goal is to automatically generate code for the `Pre Protocol` and `Post Protocol` parts, such that the entire program will fully satisfy the problem's specification.

We use a restricted high level language based on the C language. Each process has access to its id (0 or 1) by the `me` literal, and to the other process' id by the `other` literal. The processes can use an array of shared bits with a size depended on the exact variant of the problem we wish to solve. The two processes run the same code. The available node types are: *assignment, if, while, empty-while, block, and, or* and *array*. Terminals include the constants: *0, 1, 2, me* and *other*.

Table 1 describes the properties that define the problem specification. The four program parts are denoted by `NonCS`, `Pre`, `CS` and `Post` respectively. Property 1 is the basic safety property requiring the

Table 1: Mutual Exclusion Specification

No.	Type	Definition	Description	Level
1	Safety	$\Box\neg(p_0$ in `CS` $\wedge\ p_1$ in `CS`$)$	Mutual Exclusion	1
2,3	Liveness	$\Box(p_{me}$ in `Post` $\rightarrow \Diamond(p_{me}$ in `NonCS`$))$	Progress	2
4,5		$\Box(p_{me}$ in `Pre` $\wedge\ \Box(p_{other}$ in `NonCS`$)) \rightarrow \Diamond(p_{me}$ in `CS`$))$	No Contest	3
6		$\Box((p_0$ in `Pre` $\wedge\ p_1$ in `Pre`$) \rightarrow \Diamond(p_0$ in `CS` $\vee\ p_1$ in `CS`$))$	Deadlock Freedom	4
7,8		$\Box(p_{me}$ in `Pre` $\rightarrow \Diamond(p_{me}$ in `CS`$))$	Starvation	4

mutual exclusion. Properties displayed in pairs are symmetrically defined for the two processes. Prop-

erties 2 and 3 guarantee that the processes are not hung in the `Post Protocol` part. Similar properties for the `Critical Section` are not needed, since it is a fixed part without an evolved code. Properties 4 and 5 require that a process can enter the critical section, if it is the only process trying to enter it. Property 6 requires that if both processes are trying to enter the critical section, at least one of them will eventually succeed. This property can be replaced by the stronger requirements 7 and 8 that guarantee that no process will starve.

There are several known solutions to the Mutual Exclusion problem, depending on the number of shared bits in use, the type of conditions allowed (simple / complex) and whether starvation-freedom is required. The variants of the problem we wish to solve are showed in Table 2.

Table 2: Mutual Exclusion Variants

Variant No.	Number of bits	Conditions	Requirement	Relevant properties	Known algorithm
1	2	Simple	Deadlock Freedom	1,2,3,4,5,6	One bit protocol [4]
2	3	Simple	Starvation Freedom	1,2,3,4,5,7,8	Dekker [5]
3	3	Complex	Starvation Freedom	1,2,3,4,5,7,8	Peterson [18]

Three different configurations where used, in order to search for solutions to the variants described in Table 2. Each run included the creation of 150 initial programs by the GP engine, and the iterative creation of new programs until a perfect solution was found, or until a maximum of 2000 iterations. At each iteration, 5 programs were randomly selected, bred, and replaced using mutation. The values $\mu = 5, \lambda = 150$ where chosen.

In addition to the temporal specification of mutual exclusion, our configuration allows three shared bits. The famous Dekker's algorithm [5] uses two bits to announce that they want to enter the critical section, and the third bit is used to set turns between the two processes. Many runs initially converged into deadlock-free algorithms using only two bits. Those algorithms have executions in which one of the processes starve, hence only partially satisfying properties 7 or 8. Program (a) shows one of those algorithms, which later evolved into program (b). The evolution first included the addition of the second line to the *post protocol* section (which only slightly decreased its fitness level due to the parsimony measure). A replacement mutation then changed the inner while loop condition, leading to a perfect solution similar to Dekker's algorithm.

```
Non Critical Section
A[me] = 1
While (A[other] == 1)
   While (A[0] != other)
      A[me] = 0
   A[me] = 1
Critical Section
A[me] = 0
```

(a) [94.34]

```
Non Critical Section
A[me] = 1
While (A[other] == 1)
   While (A[2] == me)
      A[me] = 0
A[me] = 1
Critical Section
A[2] = me
A[me] = 0
```

(b) [96.70]

Inspired by algorithms developed by Tsay [21] and by Kessels [13], our next goal was to start from an existing algorithm, and by adding more constraints and building blocks, try to evolve into more advanced algorithms.

First, we allowed a minor asymmetry between the two processes. This is done by the operators *not0* and *not1*, which act only on one of the processes. Thus, for process 0, $not0(x) = \neg x$ while for process 1, $not0(x) = x$. This is reversed for $not1(x)$, which negates its bit operand x only in process 1, and do nothing on process 0.

As a result, the tool found two algorithms that may be considered simpler than Peterson's. The first one has only one condition in the *wait* statement (written here using the syntax of a *while* loop), although with a more complicated atomic comparison, between two bits. Note that the variable turn is in fact A[2] and is renamed here turn to accord with classical presentation of the extra global bit that does not belong to a specific process.

```
Pre CS
A[me] = 1
turn = me
While (A[other] != not1(turn));
Critical Section
A[me] = 0
```

The second algorithm uses the idea of setting the turn bit one more time after leaving the critical section. This allows the while condition to be even simpler. Tsay [21] used a similar refinement, but his algorithm needs an additional if statement, which is not used in our algorithm.

```
Pre CS
A[me] = 1
turn = not0(A[other])
While (A[2] != me);
Critical Section
A[me] = 0
turn = other
```

Next, we aimed at finding more advanced algorithms satisfying additional properties. The configuration was extended into four shared bits and two private bits (one for each process). The first requirement was that each process can change only its 2 local bits, but can read all of the 4 shared bits This yielded the following algorithm.

```
Pre CS
A[me] = 1
B[me] = not1(B[other])
While (A[other] == 1 and B[0] == not1(B[1]));
Critical Section
A[me] = 0
```

The algorithm uses the idea of using two bits as the "turn", were each process changes only its bit to set its turn, but compares both of them in the while loop. Finally, we added the requirement for busy waiting only on local bits (i.e. using local spins). The following algorithm (similar to Kessels') was generated, satisfying all properties from the table above.

```
Non Critical Section
A[other] = 1
B[other] = not1(B[0])
T[me] = not1(B[other])
While (A[me] == 1 and B[me] == T[me]);
Critical Section
A[other] = 0
```

4 Synthesizing Parametric Programs

Our experience with genetic program synthesis quickly hits a difficulty that stems from the limited power of model checking: there are few interesting fixed finite state programs that can also be completely specified using pure temporal logic. Most programming problems are, in fact, parametric. Model checking is undecidable for parametric families of programs (say, with n processes, each with the same code, initialized with different parameters) even for a fixed property [1]. One may look at mutual exclusion for a parametric number of processes. Examples are, sorting, where the number of processes and the values to be sorted are the parameters, network algorithms, such as finding the leader in a set of processes, etc. In order to synthesize parametric concurrent programs, in particular those that have a parametric number of processes, and even a parametric architecture, we use a different genetic programming strategy.

First, we assume that a solution that is checked for a large number of instances/parameters is acceptable. This is not a guarantee of correctness, but under the prohibitive undecidability of model checking for parametric programs, at least we have a strong evidence that the solution may generalize to an arbitrary configuration. In fact, there are several works on particular cases where one can calculate the parameter size that guarantees that if all the smaller instances are correct, then any instance is correct [6]. Unfortunately, this is not a rule that can be applied to any arbitrary parametric problem. We apply a *co-evolution* based synthesis algorithm: we collect parameters from failed checked cases and keep them as counterexamples. When suggesting a new solution, we check it against the collected counterexamples. We can view this process as a genetic search for both correct programs and counterexamples. The fitness is different, of course, for both tasks: a program gets higher fitness by being close to satisfying the full set of properties, while a counterexample is obtaining a high fitness if it fails the program.

In this sense, the model checking of a particular set of instances can be considered as a generalized *testing* for these values: each set of instances of the parameters provides a single finite state system that is itself comprehensively tested using model checking. This idea can be also used, independently, for model checking parametric systems. For example, consider a concurrent sorting program consisting of a parametric array of processes, each containing some initial value. Adjacent processes may exchange values during the algorithm. For any particular size and set of values, the model checking provides automatic and exhaustive test for a particular set of values, but the check is not exhaustive for all the array sizes or array values, but rather samples them.

In the classical *leader election in a ring* problem, the processes initially have their own values that they can transfer around, with the goal of finding a process that has the highest value. Then, the parameters include the size of the ring, and the initial assignment of values to processes. While we can check solutions up to a certain size, and in addition, check all possible initial values, the time and state explosion is huge. Instead, we can then store each set of instances of the parameters that failed for some candidate solution, and, when checking a new candidate solution, check it against the failed instances. A solution for the leader election, albeit not the most optimal one, was obtained using our genetic programming methods [10].

5 Correcting Erroneous Program

Our method is not limited to finding new program that satisfy the given specification. In fact, we can start with the code of an existing program instead of a completely random population and try to improve or correct it. In order to *improve code*, our fitness measure may include some quantitative evaluation; then the initial program may be found inferior to some later generated candidates. If the program we start with is *erroneous*, then it would not get a very high fitness value by failing to satisfy some of the properties.

In [11] we approached the ambitious problem of correcting a known protocol for obtaining inter-process interaction called α-core [17]. The algorithm allows multiparty synchronization of several processes. It needs to function in a system that allows nondeterministic choices, which makes it challenging, as processes that may consider one possible interaction may also decide to be engaged in another interaction. The algorithm uses asynchronous message passing in order to enforce selection of the interactions by the involved processes. This nontrivial algorithm, which is used in practice for distributed systems, contains an error.

The protocol is quite big, involving sending different messages between the controlled processes, and the controlling processes, one per each possible multiparty interaction. These messages include announcing the willingness to be engaged in an interaction, committing an interaction, canceling an interaction, request for commit from the interaction manager processes, as well as announcement that the interaction can start, or is canceled due to the departure of at least one participant. The state space of such a protocol is obviously high. In addition, the protocol can run on any number of processes, each process with arbitrary number of choices to be involves in interactions, and each interaction includes any number of processes.

Recall that model checking of parametric programs is undecidable in general [1]. In fact, we use our genetic programming approach first to find the error, and then to correct it. We use two important ideas:

1. Use the genetic engine not only to generate programs, but also to evolve different architectures on which programs can run.
2. Apply a co-evolution process, where candidate programs, and test cases (architectures) that may fail these programs, are evolved in parallel.

Specifically, the architecture for the candidate programs is also represented as code (or, equivalently, a syntactic tree) for spanning processes and their interactions, which can be subjected to genetic mutations. The fitness function directs the search for a program that may falsify the specification for the given erroneous program. After finding a "bad" architecture for a program, one that causes the program to fail its specification, our next goal is to reverse the genetic programming direction, and try to automatically correct the program, where a "correct" program at this step, is one that has passed model checking against the architecture. Yet, correcting the program for the first found wrong architecture only, does not guarantee its correctness under different architectures, hence more architectures that fail candidate solutions are collected. Note that we use for the co-evolution two separate fitness functions: one for searching for "bad" architectures, and one for searching for a correct solution.

In Figure 4 we show the architecture that was found to produce the error in the original α-core algorithm. A message sequence chart in Figure 5 demonstrate the found bad scenario. The correction consisted of changing the line of code

$$\mathsf{if\ n{>}0\ then\ n{:=}n\text{-}1}$$

into

$$\mathsf{if\ sender{\in}shared\ then\ n{:=}n\text{-}1}$$

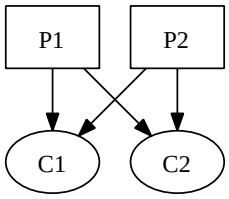


Figure 4: An architecture violating the assertion

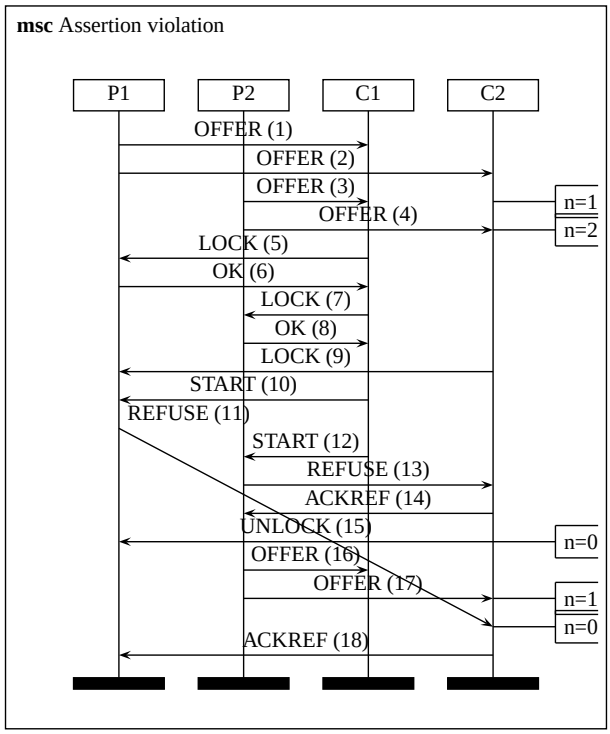


Figure 5: A Message Sequence Chart showing the counterexample for the α-core protocol

6 A Tool for Genetic Programming Based on Model Checking

We constructed a tool, MCGP [12], that implements our ideas about model checking based genetic programming. Depending on these settings, the tool can be used for several purposes:

- Setting all parts as *static* will cause the tool to just run the enhanced model checking algorithm on the user-defined program, and provide its detailed results.
- Setting the *init* process as *static* and all or some of the other processes as *dynamic* will order the tool to synthesize code according to the specified architecture. This can be used for synthesizing programs from scratch, synthesizing only some missing parts of a given partial program, or trying to correct or improve a complete given program.
- Setting the *init* process as *dynamic* and all other processes as static, is used when trying to falsify a given parametric program by searching for a configuration that violates its specification (see [11]).

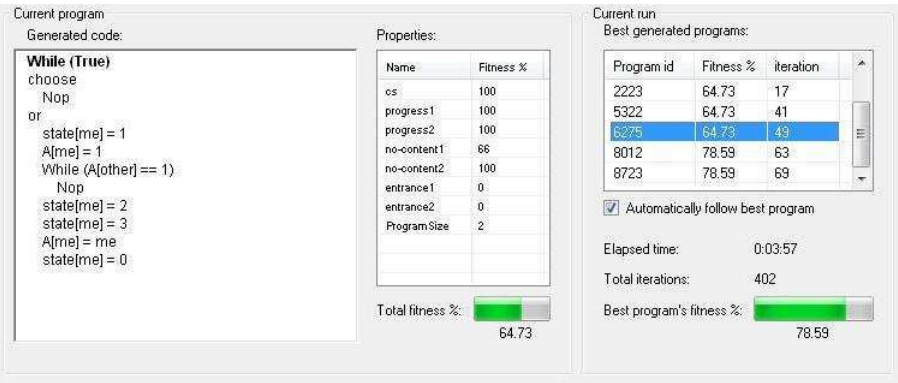


Figure 6: MCGP screen shot during synthesis of a mutual exclusion algorithm

- Setting both the *init* and the program processes as *dynamic* is used for synthesizing parametric programs, where the tool alternatively evolves various programs and configurations under which the programs have to be satisfied.

7 Replacing Model Checking by SAT Solving

Our approach can use automated deductive techniques instead of model checking in order to prove the correctness of the synthesized algorithms. However, it requires the verification procedures to be both fully automatic, and quite fast, so it can be repeated a large number of times. Obviously, most theorem provers that require some user interaction during the proof process cannot be used along with our framework. Furthermore, verification in this case is in general undecidable, so fast and complete procedure is not achievable.

Recently, there is a growing use of *SAT* and *SMT solvers* for verification purposes. These tools can function as high performance, and light-weight theorem provers for a broad range of decidable theories over first order logic, such as those of equalities with uninterpreted functions, bit-vectors and arrays. If we restrict our domain and structure of the synthesized programs as shown later, we can successfully (and quite quickly) verify their correctness for all inputs in the related domains. For some theories, variables are theoretically unbounded, while for other theories, we must limit their width.

Our work is inspired by [7], in which a set of short but ingenious and nontrivial programs, selected from the book *Hacker's Delight* [22], were successfully synthesized. These programs are loop-free, and use expressions over the decidable theory of bit-vectors. Thus, they can be easily converted into first order formulas which can then be verified by an SMT solver. The theory of bit-vectors is decidable only when limiting the width of its related variables. From a practical point of view, this does not impose a real constraint, since we can easily check the correctness of programs even with 128-bit variables. Unlike [7], we do not use the SMT solver for the direct synthesis of programs. Instead, we generate and evolve programs using our GP engine, and integrate the SMT solver into our verification component.

We modify our original framework in order to adopt it to the synthesis of sequential programs. In this new framework, the configuration provided by the user to the GP engine includes a set of building blocks, such as variables and functions that are related to the theory in use. Only loop-free programs are generated. The specification provided by the user consists of first order logic formulas describing pre and post-conditions over the above variables. A new verification component is built for dealing with

sequential programs, including two modules. A *Prover* module is able to get programs from the GP engine, and transfer them into logical formulas that are then checked for correctness by the SMT solver against the specification. The results received from the SMT solver are then used for calculating the fitness function, and for generating counterexamples. The core of this module is based on the *Microsoft Z3* SMT Solver [14]. A *Runner* module is able to run programs directly, and check their correctness for specific given test cases.

For sequential programs, we can use the Hoare notation $\{\varphi\}P\{\psi\}$ to denote the requirement that if the execution of the program starts with a state satisfying the (first order formula) φ, upon termination, it satisfies ψ. The formula φ is over the input variables. Assume they do not change. Otherwise, we can use an additional copy of them; a fixed part of the code copies them to the changeable copy. The formula ψ represents the connection between the input and output variables upon termination. Termination is not an issue here, as our generated loop-free programs must always terminate by construction (they contain no loops). Let φ be the common precondition and $\psi_1..\psi_n$ be a set of post-conditions. We want to check for each ψ_i whether $\{\varphi\}P\{\psi_i\}$ holds. Using standard construction, we obtain a formula η_P that represents the relationship between the input and output variables.

For each postcondition ψ_i we define

$$F_i := \varphi \wedge \eta_P \wedge \neg\psi_i$$

and

$$F_i' := \varphi \wedge \eta_P \wedge \psi_i$$

We can define the following three fitness levels in order of increasing value:

1. F_i' *is not satisfiable.* Then, the program P is incorrect (w.r.t. ψ_i) for all possible inputs.
2. *both* F_i *and* F_i' *are satisfiable.* There exists an input for which P satisfies ψ_i.
3. F_i *is unsatisfiable.* P is correct (for all inputs).

As an example for using our basic method, we tried first to synthesize one of the simplest programs from [22], which is required to output 0 in the variable R if and only if its input X equals $2^n - 1$ for some non-negative n. The GP engine was allowed to generate straight line programs, using only *assignment* instructions, bit-vectors related operators (such as *and*, *or* and *xor*) constants (*0* and *1*), and variables. Within a few seconds, the following correct program was generated.

```
T = X + 1
R = T and X
```

Solutions found by the method described above are guaranteed to be correct for every possible input (in the domain of the variables, such as bit-vectors with a specific width). This is a major advantage over solutions generated by traditional GP, which can usually guarantee correctness only for the set of test cases. However, using test cases can help in building a smoother fitness function that can direct the generated programs into gradual improvements. Hence, we used for calculating the fitness function, in addition to the above satisfiability based levels, a collection of test cases that failed on previous selected candidates. Each test case is obtained using the SAT solver when checking satisfiability of F_i on a previous candidate program. It consists of initial values from which we can run new checked candidates. Note that running the code on test cases, using the runner module, is faster than applying SAT solving using prover module.

After adding the ability to use test cases, we tried to synthesize a more advanced program that is required to compute the floor of the average of its inputs X and Y without overflowing (which may be

```
R = X + Y
R = R >> 1
```

(a)

```
T = X >> 1
R = Y >> 1
R = T + R
```

(b)

```
R = X and Y
T = X xor Y
T = T >> 1
R = R + T
```

(c)

Figure 7: Synthesized Programs for Computing *avg(X,Y)*

caused by simply summing the inputs before dividing by two). Figure 7 shows some of the programs generated during the synthesis process (the logical shift right operator is denoted by ">>").

Program *(a)* is the naive way for computing the average. However, the addition may cause an overflow, and indeed the program was refuted by the SMT solver, yielding a counterexample with big inputs. At the next iteration, program *(b)* was generated. While not overflowing, the program is still incorrect if both of its inputs are odd, which was reflected by a second counterexample. Finally, the more ingenious program *(c)* was generated, and verified to be a correct solution (identical to the one presented in [22]).

8 Conclusions

We suggested the use of a methodology and a tool that perform synthesis of programs based on genetic programming guided by model checking. Code mutation is at the kernel of genetic programming (crossover is also extensively used, but we did not implement it). Our method can be used for

- synthesizing correct-by-design programs,
- finding errors in protocols with complicated architectures,
- automatically correcting erroneous code with respect to a given specification, and
- improving code, e.g., to perform more efficiently.

We demonstrated our method on the classical mutual exclusion problem, and were able to find existing solutions, as well as new solutions.

In general, the verification of parametric systems is undecidable, and in the few methods that promise termination, quite severe restrictions are required. The same apply to code synthesis. Nevertheless, we provide a co-evolution method for synthesizing parametric systems based on accumulating cases to be checked. Parameters or architectures on which the synthesis failed before, or test cases based on previous counterexamples are accumulated to be checked later with new candidate solutions. As the model checking itself is undecidable, we finish if we obtain a strong enough evidence that the solution is correct on the accumulated cases.

We allowed constructing the architecture (processes and the channels between them) as part of the code that can be mutated. Then the genetic mutation operation can be used in finding architectures in which given algorithms fail. This can be used to model check code with varying architecture, and furthermore, to correct it.

We started recently to look at replacing model checking by SAT and SMT tools. This provides an efficient alternative for some synthesis problem. In particular, SMT solvers may succeed in some parametric cases where model checking fails.

Although our method does not guarantee termination, neither for finding the error, nor for finding a correct version of the algorithm, it is quite general and can be fine tuned through provided heuristics in a convenient human-assisted process of code correction.

References

[1] Krzysztof R. Apt & Dexter Kozen (1986): *Limits for Automatic Verification of Finite-State Concurrent Systems*. *Inf. Process. Lett.* 22(6), pp. 307–309, doi:10.1016/0020-0190(86)90071-2.

[2] W. Banzhaf, P. Nordin, R. E. Keller & F. D. Francone (2001): *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications (3rd edition)*. Morgan Kaufmann, dpunkt.verlag.

[3] Yoah Bar-David & Gadi Taubenfeld (2003): *Automatic discovery of mutual exclusion algorithms*. In: *PODC*, p. 305, doi:10.1145/872035.872080.

[4] James E. Burns & Nancy A. Lynch (1993): *Bounds on Shared Memory for Mutual Exclusion*. *Information and Computation* 107(2), pp. 171–184, doi:10.1006/inco.1993.1065.

[5] Edsger W. Dijkstra (1965): *Solution of a problem in concurrent programming control*. *Commun. ACM* 8(9), p. 569, doi:10.1145/365559.365617.

[6] E. Allen Emerson & Kedar S. Namjoshi (1995): *Reasoning about Rings*. In: *POPL*, pp. 85–94, doi:10.1145/199448.199468.

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari & Ramarathnam Venkatesan (2011): *Synthesis of loop-free programs*. In: *PLDI*, pp. 62–73, doi:10.1145/1993498.1993506.

[8] Gal Katz & Doron Peled (2008): *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*. In: *ATVA*, *LNCS* 5311, pp. 33–47, doi:10.1007/978-3-540-88387-6_5.

[9] Gal Katz & Doron Peled (2008): *Model Checking-Based Genetic Programming with an Application to Mutual Exclusion*. In: *TACAS*, *LNCS* 4963, pp. 141–156, doi:10.1007/978-3-540-78800-3_11.

[10] Gal Katz & Doron Peled (2009): *Synthesizing Solutions to the Leader Election Problem using Model Checking and Genetic Programming*. In: *HVC*, *LNCS* 6405, pp. 117–132, doi:10.1007/978-3-642-19237-1_13.

[11] Gal Katz & Doron Peled (2010): *Code Mutation in Verification and Automatic Code Correction*. In: *TACAS*, LNCS, pp. 435–450, doi:10.1007/978-3-642-12002-2_36.

[12] Gal Katz & Doron Peled (2010): *MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming*. In: *ATVA*, pp. 359–364, doi:10.1007/978-3-642-15643-4_28.

[13] Joep L. W. Kessels (1982): *Arbitration Without Common Modifiable Variables*. *Acta Inf.* 17, pp. 135–141, doi:10.1007/BF00288966.

[14] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS*, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[15] Peter Niebert, Doron Peled & Amir Pnueli (2008): *Discriminative Model Checking*. In: *CAV*, *LNCS* 5123, Springer, pp. 504–516, doi:10.1007/978-3-540-70545-1_48.

[16] Doron Peled (2001): *Software Reliability Methods*. Springer, doi:10.1007/978-1-4757-3540-6.

[17] Jose Antonio Perez, Rafael Corchuelo & Miguel Toro (2004): *An order-based algorithm for implementing multiparty synchronization*. *Concurrency - Practice and Experience* 16(12), pp. 1173–1206, doi:10.1002/cpe.903.

[18] Peterson & Fischer (1977): *Economical Solutions to the Critical Section Problem in a Distributed System*. In: *STOC: ACM Symposium on Theory of Computing (STOC)*, pp. 91–97, doi:10.1145/800105.803398.

[19] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, pp. 179–190, doi:10.1145/75277.75293.

[20] Amir Pnueli & Roni Rosner (1990): *Distributed Reactive Systems Are Hard to Synthesize*. In: *FOCS*, pp. 746–757, doi:10.1109/FSCS.1990.89597.

[21] Yih-Kuen Tsay (1998): *Deriving a Scalable Algorithm for Mutual Exclusion*. In: *DISC*, pp. 393–407, doi:10.1007/BFb0056497.

[22] Henry S. Warren (2002): *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.