Algorithmic Analysis of Channel machines Using Small Models

Jonathan Sharyari

25th August 2014

abstract

Contents

1 Introduction

Todays society grows more and more dependent on computer-applications. Often they are used directly, for example the task of paying one's bills online, in other cases applications are aiding us in a more abstract manner, e.g. controlling the elevator or planning a train schedule. The common factor between the online bank, the elevator relay and the train scheduler is that the correctness of these programs is of utter importance, where program failure could have devastating results on the economy, the infrastructure or even cause the loss of human life.

This has motivated research on various techniques to find and correct potential faults pre-deployment. The most common technique is that of *testing*, i.e., observing the system behaviour on a variety of likely or even unlikely scenarious. The widespread use of testing is well-motivated, but there are several scenarios, where testing by itself is not enough or maybe not even possible. It would for example be costly to verify the correctness of the aforementioned elevator only by testing, as a failure could result in damages on the system. Yet another weakness of this approach is that testing helps find faults at a late stage in the development process.

Another technique used for verification is the process of *simulation*. In contrast to testing, simulation can be done in an early stage of the development. Rather than first implementing an algorithm, one may simulate an abstracted model of the algorithm which may help ensure its correctness or find a fault in the algorithm before development has begun. An important note is that simulation techniques are not intended to be used *instead* of testing, but rather in combination with testing, as an abstract model of a system can never fully represent the system.

The goal of *formal verification* or *model checking* is to formally verify the correctness of a program with respect to a given specification. In general, given a model of a system and a set of properties, the task is to decide whether the properties are met by the model or not, by performing an exhaustive search in the system. Although different ways of representing the model and the properties have been proposed, most commonly the model is expressed as a *finite state machine* and the properties as propositional logic formulae. These properties are then checked by creating a *transition system* corresponding to the FSM at hand.

citation needed

FSM becomes TS, prop. logic becomes LTL/CTL?

A serious limitation of model checking is the problem of *state space explosion*, i.e. an exponential increase of states in the transition system, in relation to the size of the finite state machine. Related to this is the fact that a transition system need not even be finite, for example when the transition system is the composition of an infinite number of FSMs, but even such programming artifacts as using an unbounded integral value or relying on a buffer may introduce an infinit aspect into the model.

This essay focuses on this latter point; model checking systems relying on an unbounded buffer. This scenario is common in, but not bounded to, the verification of communication algorithms. Based on previous work by Parosh

et al. , I develop a system that in many cases is capable of reducing the inifinte state space of a buffer system to a finite set of states. Section 2 contains a formal definition of such a parameterized system. Section 5 contains a few extensions to this model, allowing for a wider variety of modelling scenarios. In section 4, some abstractions and techniques are examplained that allow for an efficient implementation of the verification algorithm. Further a protocol specification language is explained, that allows a user to define and use the verification tool without, without knowledge of the intricacies of the internal model. The results of the verifier when applied to a number of well-known communication algorithms is presented in section **??**, as well as some comparisons against the results of other verification tools.

Do I really wanna use this type of references? also, add the actual reference

2 Definitions and Terms

Before formally defining a channel system, we present a simple protocol, the *alternating bit protocol*. This protocol will continuously serve to examplify the theoretical concepts in this section in a practical setting. The formal definition below is associated to systems operating on FIFO buffers, other types of systems are covered in section 5.

Alternating Bit Protocol. The Alternating Bit Protocol (ABP) is a distributed protocol for transmitting data from a *sender* to a *receiver* in a network. The protocol uses two unbounded channels, ch_M used to transmit messages and ch_A to transmit *acknowledgements* of received messages. The sender sends a message with sequence number $x \in \{0,1\}$ to the receiver over channel ch_M, who upon reception sends an acknowledgement with the same sequence number over channel ch_A. Both the sender and the receiver may send the same message (with the same sequence number) repeatedly. When the sender receives an acknowledgement from the receiver, the next message can be sent using the sequence number *1-x* (thus the name Alternating Bit Protocol).

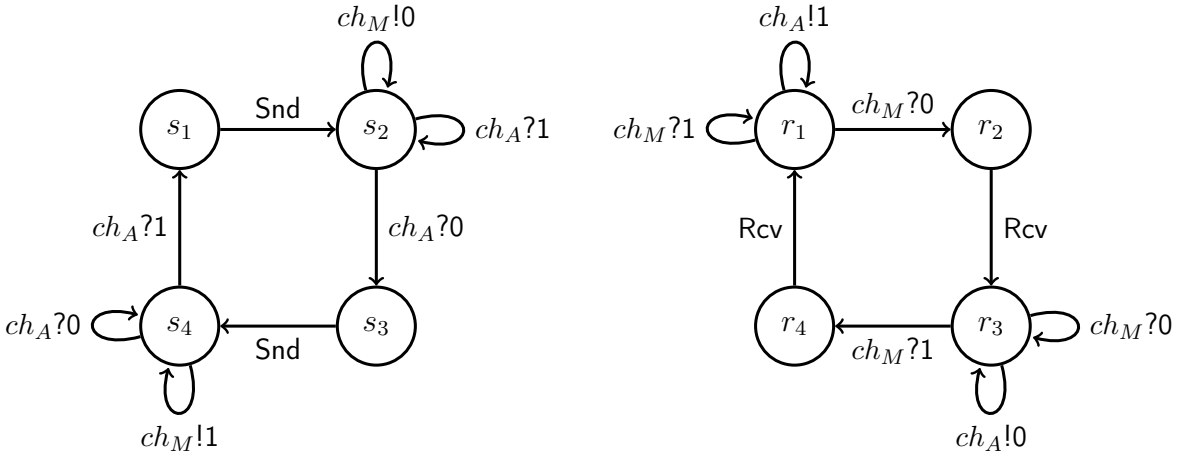
This is a simple protocol that operates on unbounded channels. The behaviour of the sender and receiver process is illustrated in figure 1.

2.1 Channel Systems

We present here the basic definition of a finite-state system with unbounded channels. Such a system can be seen as to have two parts, a *control part* and a *channel part*. The channel part is a set of channels, which may be empty or contain a sequence of messages, a *word*. The control part is a labeled finite-state transition system.

Definition Formally, a channel system CS over a set of processes P and channels Ch is a tuple $\langle \mathrm{S}, s_0, \mathrm{A}, \mathrm{C}, \mathrm{M}, \delta \rangle$, where

S is a finite set of control states,



(a) Sender (b) Receiver

Figure 1: Program graphs of sender and receiver in the ABP protocol.

s_0 is an initial control state,

A is a finite set of actions,

Ch is a finite set of channels,

M is a finite set of messages,

δ is a finite set of transitions, each of which is a triple of the form $\langle s_1, op, s_2 \rangle$, where s_1 and s_2 are control states, and op is a label of one of the forms

- $c!m$, where $c \in Ch$ and $m \in M$
- $c?m$, where $c \in Ch$ and $m \in M$
- $a \in A$.

The finite-state control part of CS is an ordinary labeled transition system with states S, initial state s_0 and transitions δ. The channel part is represented by the set Ch of channels, which may contain a string of messages in M. The set A denotes the set of observable interactions with the environment, whereas δ may either perform an action from A, or and unobservable action, where

$\langle s_1, c!m, s_2 \rangle$ represents a change of state from s_1 to s_2 while appending the message m to the tail of channel c

$\langle s_1, c?m, s_2 \rangle$ represents a change of state from s_1 to s_2 while removing the message m to the head of channel c

Example. The alternating bit protocol can be described by a channel system $CS = \langle S, i, A, Ch, M, \delta \rangle$ such that $S = \{(s,r)\}$ with $s \in \{s_1, s_2, s_3, s_4\}$, $r \in \{r_1, r_2, r_3, r_4\}$, i is the initial state (s_1, r_1), $A = \{Snd, Rcv\}$, $Ch = \{Ch_M, Ch_A\}$, $M = \{1,0\}$ and delta is the set of transitions

$\langle s_1, Snd, s_2 \rangle$	$\langle r_1, ch!1, r_1 \rangle$
$\langle s_2, ch!0, s_2 \rangle$	$\langle r_1, ch?1, r_1 \rangle$
$\langle s_2, ch?1, s_2 \rangle$	$\langle r_1, ch?0, r_2 \rangle$
$\langle s_2, ch?0, s_3 \rangle$	$\langle r_2, Rcv, r_3 \rangle$
$\langle s_3, Snd, s_4 \rangle$	$\langle r_3, ch!0, r_3 \rangle$
$\langle s_4, ch!1, s_4 \rangle$	$\langle r_3, ch?0, r_3 \rangle$
$\langle s_4, ch?0, s_4 \rangle$	$\langle r_3, ch?1, r_4 \rangle$
$\langle s_4, ch?1, s_1 \rangle$	$\langle r_4, Rcv, r_1 \rangle$

2.2 Channel Transition Systems

A *transition system* is an abstract machine, commonly used in model checking to describe the behaviour of a system. They are in ways similar to the notion of finite state automata, with the difference that the states and transitions in a transition system need not be finite. A state of a transition system describing a program may be determined, for example, the evaluation of all variables in the program. Transitions describe how a system in a certain state can move to another state.

Definition The operational behaviour of CS is defined by the inifinite-state transition system $TS = (C, \rightarrow)$ where

$C = (S \times \xi)$ is the set of its configurations, where ξ is an evaluation of the set of channels C in CS

$\rightarrow \subseteq (S \times S)$ contains the following transitions

- For each observable action a $\in A$ in CS

$$\frac{s \xrightarrow{a} s'}{(S, \xi) \rightarrow (S', \xi)}$$

- For each transmission action $\langle s_1, ch!m, s_2 \rangle$ in CS

$$\frac{s \xrightarrow{ch!m} s' \wedge ch \in \xi}{(S, \xi) \rightarrow (S', \xi')}$$

with

$$\xi' = \xi[ch := \xi(ch) \bullet m].$$

- For each reception action $\langle s_1, ch?m, s_2 \rangle$ in CS

$$\frac{s \xrightarrow{ch?m} s' \wedge \xi(ch) = m \bullet w_1..w_n}{(S, \xi) \rightarrow (S', \xi')}$$

with

$$\xi' = \xi[ch := w_1..w_n].$$

At times we may use a notation for a configuration c with explicit processes and channels, so that $c = \langle s_1, ..., s_n, ch_1, ..., ch_p \rangle$, or alternatively, with explicit processes and channel evaluations, $c = \langle s_1, ..., s_n, \xi(ch_1), ..., \xi(ch_p) \rangle$.

Example Let $c = \langle S, \xi \rangle$ be a configuration of the alternating bit protocol, with the channels containing the words 01 and 10 respectively. Then c may also be denoted as $c = \langle s, r, ch_M, ch_A \rangle$ or as $c = \langle s_1, s_2, 01, 10 \rangle$. There are finitely many control states in this system, but an infinite set of channel evaluations. A transmission transition in this system is for example $\langle \langle s_2, r_1, 01, 10 \rangle, ch_m!1, \langle s_2, r_1, 011, 10 \rangle \rangle$.

2.3 Reachability and Bad Configurations

An instance of the *reachability problem* is defined by a channel transition system $TS = \langle S, \xi \rangle$, a set of initial configurations $I \subseteq S^+$ and a set $BAD \subseteq S^+$ of *bad configurations*. We assume that Bad is the upward closure $\{c \mid c \in B : b \sqsubseteq c\}$ of a set of a given *finite* set of *minimal bad configurations*.

Let c denote a configuration, then c is said to be *reachable* in TS, if there are configurations $c_1 ... c_l$ such that c_0 is an initial configuration of TS and for each $0 \leq \mathrm{i} < \mathrm{l}$, $\langle c_i, c_{i+1} \rangle \in \rightarrow$.

We use $\mathcal{R}$ to denote the set of reachable states. We say that the system TS is *safe* if there are no reachable bad configurations, i.e. $\mathcal{R} \cap Bad = \emptyset$.

Is this true in my case?

2.4 Traces

Suppose s_n is a bad state of a channel system CS. Then a *bad trace* is a sequence $t = s_0 s_1 ... s_n$ such that for each $0 \leq i \leq (n-1)$, $\langle s_i, r, s_{i+1} \rangle \in \delta$. If t is the shortest bad trace in the system, we call it a *minimal bad trace*.

3 Problem Formulation

In (1), the authors propose a verification method for *parameterized systems* and show that how they can, under certain assumptions, be verified for an unbounded number of processes. This project is based on the results of this work, and aims to adapt this method for systems working on unbounded channels. This is done by formally defining a model for systems working on channels, and proving that such systems have a *small system property* that allows us to verify their correctness by only looking at finite-state representations of the system.

The goal is further to implement the proposed verification method. For such an implementation to be useful, several extensions are proposed in order to expand the context in which the system can be used. Also, a specification language is defined with which the user can easily model problems. This type of verification generally has a high demand of computational resources, thus the efficiency of the implementation is of great importance. The method is used

to model several well-known communication protocols, such as the alternating bit protocol and the sliding window protocol, in order to compare its efficiency against other verification methods and establish its correctness.

4 Model

4.1 Subwords and Views

Let $\sqsubseteq$ be the subword relation, then $u \sqsubseteq s_1...s_n{=}w$ iff u is an ordered subset of w. For example, if w=abc, then the set of subwords of w is abc, ab, bc, a, b, c.

We define the *views* of a configurations to be v' such that for $c = \langle s, \xi \rangle$, $v'{=}\langle s, \xi[ch \sqsubseteq \xi(ch)] | ch \in \xi \rangle$. We define size($ch$) to be equivalent to size($\xi(ch)$), i.e. the length of the word on the channel ch. We define the size of a configuration or a view to equal the size of its longest channel.

Example. Suppose c is a configuration $\langle s_1, s_2, ab, cd \rangle$. The configuration is of size 2 and its views are

$\langle s_1, s_2, ab, cd \rangle$	$\langle s_1, s_2, a, cd \rangle$	$\langle s_1, s_2, a, c \rangle$	$\langle s_1, s_2, a, d \rangle$	$\langle s_1, s_2, \epsilon, \epsilon \rangle$
	$\langle s_1, s_2, b, cd \rangle$	$\langle s_1, s_2, b, c \rangle$	$\langle s_1, s_2, b, d \rangle$	
	$\langle s_1, s_2, \epsilon, cd \rangle$	$\langle s_1, s_2, \epsilon, c \rangle$	$\langle s_1, s_2, \epsilon, d \rangle$	

4.2 Abstractions and Concretizations

The abstraction function $\alpha_k : C \to 2^{C_k}$ maps a configuration c into the set V of views of size up to k , such that for each $v \in V$, $\{v \sqsubseteq c\}$.

The concretization function $\gamma_k : 2^{C_k} \to 2^C$ returns, given a set of views V, the set of configurations that can be reconstructed from the views in V, in other words, $\gamma_c(V) = \{c \in C \mid \alpha_k(c) \subseteq V\}$

For a set V, we define the *post-image* of V, *post(V)* $= \{c' \mid c \to c' \wedge c \in V\}$. The *abstract post-image* of a set $V \in C_k$ is defined as $Apost_k(V) = \alpha_k(post(\gamma_k(V)))$ In general, γ_k is an infinite set of states. We show (4.3) that we only need to consider those configurations, whose sizes are up to k+1, i.e. a finite set of configurations. We define $\gamma_k^l(V) := \gamma_k(V) \cap C_l$ for some $l \geq 0$. The intuitive meaning of $\gamma_k^l(V)$ is the set of l-size configurations for which all views of length at most k are in V.

4.3 Small Model Property

The property that allows us to verify infinite-state problems, is that transitions have *small preconditions*, expressed formally by the following lemma.

Lemma 1. *For any* $k \in \mathbb{N}$, *and* $X \subseteq C_k$, $\alpha_k(post(\gamma_k(X))) \cup X = \alpha_k(post(\gamma_k^{k+1}(X))) \cup X$.

?

?

Is this really what we call the small model property?

Is

We will show that for any configuration $c \in \gamma_k(V)$ of size $m > k+1$ such that there is a c' induced by a transmission rule $r \in \rightarrow$ from c, then for each $v' \in \alpha_k(c')$, the following holds: There is a configuration $d \in \gamma_k(V)$ of size at most $k+1$ with a transition $d \xrightarrow{r} d'$ with $v' \in \alpha_k(d')$.

4.3.1 Proofs

Transmission rules First we note, that for any configuration $c \in V$, any view $v' \in \alpha_k(c)$ is also a valid configuration $c' \in \gamma_k(V)$, since $\alpha_k(v') \subseteq \alpha_k(c)$ and thus $v' \in \gamma_k^{k+1}(\alpha_k(c))$. Also note that if from c a transition r can be fired, then this transition can also be fired from any configuration $c' = v'$, as transmission rules are guarded only by the states of the channel system and not by channel evaluations.

A transmission rule changes the evaluation of at most one channel $ch \in c$, and (possibly) the state of the channel system, thus we need only reason about the evaluations of a single channel ch. Let $c = \langle S, w\rangle \xrightarrow{ch!w_{m+1}} \langle S', w \bullet m\rangle = c'$.

The views of c' of size up to k are either of the type 1) $\langle S', w' \sqsubset w\rangle$, with size$(w') \leq k$ (i.e. not including the newly transmitted message) or 2) of the form $\langle S', w' \bullet m | w' \sqsubseteq w\rangle$ with size$(w') < k$.

For any view of type 1, there exists a configuration of size k, $d = \langle S, w'\rangle \in \alpha_k c$ and the transition r can be taken, resulting in $d' = \langle S, w' \bullet m\rangle$ of size $k+1$. The view $v' \in \alpha_k w'$.

For any view of type 2, there exists a configuration of size $k-1$, $d = \langle S, w'\rangle$ and the transition r can be taken resulting in $d' = \langle S, w' \bullet m\rangle = v'$.

Example. Assume a system with two processes and a single channel. Let $c = \langle 1, 2, abc\rangle \rightarrow ch!d\ \langle 2, 2, abcd\rangle$. Assume that $c \in \gamma_2(V)$, then $\alpha_2(c) \in V$, i.e. $\langle 1, 2, a\rangle$, $\langle 1, 2, b\rangle$, $\langle 1, 2, c\rangle$, $\langle 1, 2, ab\rangle$, $\langle 1, 2, bc\rangle$, $\langle 1, 2, ac\rangle$ are in in V and also in $\gamma_k(V)$.

$\alpha_2(c') = \{\langle 2, 2, a\rangle, \langle 2, 2, b\rangle, \langle 2, 2, c\rangle, \langle 2, 2, d\rangle, \langle 2, 2, ab\rangle, \langle 2, 2, bc\rangle, \langle 2, 2, cd\rangle, \langle 2, 2, ac\rangle, \langle 2, 2, ad\rangle, \langle 2, 2, bd\rangle\}$. Consider a view with the newly transmitted message, $\langle 2, 2, cd\rangle$, it can by created by $\langle 1, 2, c\rangle \rightarrow ch!d\ \langle 2, 2, cd\rangle$. Considering instead a view without the transmitted message, $\langle 1, 2, ab\rangle$, it can be created by $\langle 1, 2, ab\rangle \rightarrow \langle 2, 2, abd\rangle$ for which $\langle 2, 2, ab\rangle$ is a view.

Under what pre-conditions does this hold?

Reception rules As opposed to the transmission rules, reception rules also rely on the state of the channel in order to be fired, but only the state of that channel need be considered. Consider $c = \langle S, m \bullet w\rangle \xrightarrow{ch?m} \langle S, w\rangle$. For any view $v' \in \alpha_k(c')$ with the word w' of size at most k on the channel, there exists a configuration of size at most $k+1$, $d = \langle S, m \bullet w'\rangle \in \alpha_k(c)$ such that $d \xrightarrow{ch?m} d' = v'$.

Why is that so?

Actions Actions can in this context be seen as equivalent to a reception rule, reading the empty symbol ϵ on some channel. The proof then follows directly

from 4.3.1.

5 Extensions

There are several ways the channel system and the channel transition system in 2.1 and 2.2 could be extended, in order to cope with a wider scope of application scenarios. For example, we may want to model a protocol working on LIFO buffers, rather than the FIFO buffers described above. Another context may be that of a protocol communicating over an unreliable channel, introducing the possibility of *message loss* on the channels. In this section, we shall create models for both of these scenarios. Doing this requires us to first adapt the channel system model and the corresponding channel transition system by adding appropriate transition rules and action, and second to prove that 1 holds for these models.

5.1 LIFO Channels

Stack Channel System In order to model a LIFO channel or a *stack*, we need only modify one of the transitions of the system described in 2.1 in such a way that transmissions and reception transitions append and delete messages on the same end of the channels. For simplicity, we only restate the parts of the channel system affected by these changes, all other parts of the system remain unchanged. The finite-state control part of *CS* is an ordinary labeled transition system with states S, initial state s_0 and transitions δ. The channel part is represented by the set Ch of channels, which may contain a string of messages in M. The set A denotes the set of observable interactions with the environment, whereas δ may either perform an action from A, or and unobservable action, where

> $\langle s_1, c!m, s_2\rangle$ represents a change of state from s_1 to s_2 while appending the message m to the **head** of channel c
>
> $\langle s_1, c?m, s_2\rangle$ represents a change of state from s_1 to s_2 while removing the message m to the **head** of channel c

Stack Channel Transmision System In order to describe the transition system induced by a Channel *CS*, we need only modify the transition rules of 2.2 so that they reflect the changes made in 5.1. Leaving the rest of the model unchanged, we restate the definitions of the transition relations (from which onlythe transmission transition is changed). This results in $\rightarrow \subseteq (S \times S)$ containing the additional transitions

- For each observable action a ∈ A in CS

$$\frac{s \xrightarrow{a} s'}{(S, \xi) \rightarrow (S', \xi)}$$

- For each transmission action $\langle s_1, ch!m, s_2 \rangle$ in CS

$$\frac{s \xrightarrow{ch!m} s' \wedge ch \in \xi}{(S, \xi) \rightarrow (S', \xi')}$$

with

$$\xi' = \xi[ch := m \bullet \xi(ch)].$$

- For each reception action $\langle s_1, ch?m, s_2 \rangle$ in CS

$$\frac{s \xrightarrow{ch?m} s' \wedge \xi(ch) = m \bullet w_1..w_n}{(S, \xi) \rightarrow (S', \xi')}$$

with

$$\xi' = \xi[ch := w_1..w_n].$$

Proof of Lemma 1 Only the part of the proof of lemma 1 regarding transmission rules is affected by the changes made to this system. Such a proof follows in a straightforward manner from the proof 4.3.1, by considering configurations of the form $\langle S, m \bullet w' \rangle$ rather than $\langle S, w' \bullet m \rangle$.

5.2 Lossy Channels

A lossy channel system is a system similar to 2.1, with the difference that the messages on channels may be lost. In practice, data loss may appear in several contexts, e.g. data corruption, inconsistencies on weak memory models or message loss during data transmission over a network.

Lossy Channel Systems A lossy channel system is described by 2.1 with an additional transition $\langle s, ch*, s \rangle$ such that $\xi(ch) = w_1 w_2..w_l \rightarrow w_1..w_{i-1} w_{i+1}..w_l$ for some $0 \geq i \geq$ l.

Lossy Channel Transition Systems A lossy channel transition system TS is described by 2.2 with an additional transition rule

$$\frac{s \xrightarrow{ch*} s \wedge \xi(ch) = w_1..w_{p-1} \bullet m \bullet w_{p+1}..w_n}{(S, \xi) \rightarrow (S, \xi')}$$

with

$$\xi' = \xi[ch := w_1..w_n].$$

Extended Proof of Lemma 1 Consider a configuration *c'* such that $c \xrightarrow{r} c'$, where r is a channel loss transition, as described above. Then one channel ch in c of length $n \leq k$, $\xi(ch) = w_1..w_{p-1} \bullet m \bullet w_{p+1}..w_n$ loses the symbol m, resulting in the channel evaluation $\xi(ch') = w_1..w_n$ in *c'*.

Since *c'* is a also a view of c, $c \in V$, it follows that the proofs as presented in 4.3 hold also for the lossy channel transition system.

5.3 Channel Systems with Synchronization

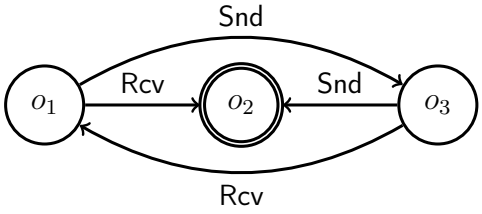
It is not uncommon that distributed programs rely on *synchronization* in their program behaviour. With synchronization, we mean that two or more programs take a joint step, i.e. a synchronization transition cannot be taken unless all programs affected by it take the transition simulateously. This is particularly common in parallel programs, which may perform independent calculations but occasionally rendezvous and synchronize. As we shall see, **??**, synchronization can also be used as a modelling technique even if the program being modelled does not synchronize.

A channel system is in a sense the interleaving of multiple *program graphs*. When operating such a level of detail, a synchronizaition action with a label l corresponds to an action that cannot be taken, unless each program where such an action is present take the action simultaneously.

As the working model in this paper, i.e. the channel system described in 2.1 and the corresponding transition system, have a higher level of abstraction, they have the mechanisms needed in order to model synchronizing programs. In essence, synchronizing and non-synchronizing actions are the same at this level of abstraction and cannot be differentiated.

5.4 Alternating Bit Protocol Revised

The alternating bit protocol is a protocol designed to be resistent to message loss, therefore it is reasonable to model it using a lossy model. The transition system induced by the program graphs 1 does not provide an intuitive way to describe a set *Bad* of bad states. This can easily be overcome by introducing an *observer* program, which synchronizes with the sender and receiver.



The observer synchronizes with the sender over transitions with the label *Snd* and with the receiver over *Rcv*. If either the sender performs two transmissions (with different sequence numbers) without the receiver having received in between, or if the receiver receives two messages without the sender having transmitted in between, the observer would reach its accepting state o_3. This state can therefore be considered to be a minimal bad state, and any configurations describing a system with the observer in its bad state is a bad configuration.

6 Verification method

As a result of lemma 1, if $\gamma_k^{k+1} \cap Bad =$ for any $k \geq 1$, then $\gamma_k(V) \cap Bad =$ for any buffer size larger than k. Given a set of bad states Bad, a set of initial states I and a set of transitions, Verifying the correctness of a system can be done with the following algorithm.

Algorithm 1 General Verification algorithm

```
1: for True do
2:     if R_k ∩ Bad ≠ ∅ then
3:         return Unsafe
4:     end if
5:     V := μX.α_k(I) ∪ Apost_k(X)
6:     if γ_k(V) ∩ Bad = ∅ then
7:         return Safe
8:     end if
9:     k := k+1
10: end for
```

The algorithm begins by performing a reachability analysis in order to compute $\mathcal{R}_k$ and check for bad states. For any buffer size k, if a bad state is found to be reachable in $\mathcal{R}_k$, the system is unsafe and the algorithm terminates. If no such bad state was found, the algorithm continues by computing an *overapproximation* of configurations of size k, reachable through configurations of size at most $k+1$ and checking for bad configurations. This is done by computing $Apost_k$ iteratively until a fixpoint V is reached. If at this point no bad configuration has been found, the system can be said to be safe (due to the result of 1) and the algorithm terminates. If on the other hand a bad state was found, the system is not necessarily unsafe, as V is an over-approximation of the reachable states in $\mathcal{R}_k$, the process is repeated with a buffer size of *k+1*.

In this rest of this section, we show how to implement this algorithm in an efficient way. We do this by beginning with a naive algorithm computing the configurations, and then step by step locating and addressing the performance issues.

6.1 Naive algorithm

A naive way to generate configurations would be to perform the described operations such that they correspond directly to the mathematical notations as described in section 4.2. This is possible, as most programming languages have built-in datastructures that support set operations.

A transition in the transition system corresponds to one or more *rules* in the implementation. A rule $r \in R$ is a description of a transition, seen as a set of state-predicates, channel-predicates, state-effects and channel-effects.If all predicates are true for a configuration c, the effects of the rule are applied to c resulting in a post-value *c'*.

Is the algorithm guaranteed to terminate? When does it not?

In this naive approach, all configurations are stored in a *set*, such that each configuration is stored uniquely. For each element c of the set V, the function *gamma*, *step* and *alpha* are performed in order, until a *fix-point* is reached, i.e. continuing the process does not result in any more configurations being found. Assuming that a set of channel symbols and a set of rules R describing the transitions are known, the algorithm does the following:

1. Generate a set $\Gamma(V)$: For each configurations $c \in V$, generate all potential concretizations *con* such that $/con/ = /c/+1$. For any such potential concretization *con*, remove those for which $alpha_k(c') \not\subset V$.

2. Generate the *post*(V): For every $con \in \Gamma(V)$, $r \in R$, compute *r(con)*.

3. Generate Apost(V): For each element $p \in$ *post(V)*, compute its views of size k. If $V' \cup V = V$, a fixpoint has been reached, otherwise the processes repeats with $V := V \cup V'$.

The bottle-neck in this algorithm lies in *gamma*. It first creates all potential concretizations of a configuration c, i.e. all combinations of channel evaluations where at least one of the channels is of a larger size than in c. Additionaly, for each of these, all their views must be computed in order to determine whether the concretization should be refuted or not. Although this method is correct, there is significant overlap of concretizations; it is often possible to create the same concretization *con* in several ways, and each time, α_k*con*, *post(con)* and *Apost(con)* need be performed. In the next section, we show abstractions of the functions *alpha* and *gamma* that are less computationally expensive.

6.2 Solution

We look closer at γ, in order to address the issues mentioned above. We show that all concretizations can be found, while considering only a subset of the potential concretizations. Furthermore, it is possible to determine whether a concretization should be accepted or not, examining only a subset of its views.

Abstracting the concretizations. Given a configuration c, with channel evaluations ξ such that all channels have size smaller or equal to k, a potential concretization of c is any configuration *con* such that *con* can be created from c by appending a symbol to at least one of the channels of c. As this is a large number of potential configurations, it is desireable to consider only a subset thereof, while still ensuring that all valid concretizations are eventually found.

We show that it is sufficient to in each iteration consider only the potential concretizations of c, for which only one channel has been modified, and only modifications where a symbol has been added at the **end** (or alternatively the beginning) of a channel.

Consider a potential concretization *con* of c, where n channels have been extended by a symbol. If such a potential concretization is valid, any evaluation where $n' < n$ channels have been modified in a similar way will also

be a valid concretization. Therefore, *con* will eventually be considered also when only one channel is extended in each iteration, but more iterations are required in order to find it.

As an example, consider a concretization with channel evaluation $e = w_1..w_l$, created by extending some configuration with a single symbol, not necessarily at the end of the channel. If *con* is a valid concretization, then a configuration *c'* must be in V such that *c'* has the channel evaluation $w_1..w_{l-1}$, since *c'* $\in \alpha_k(con)$. This channel evaluation can be extended to *con* by adding the symbol w_l at the end of the channel.

This shows that any valid concretization will eventually be created, even if only a subset of potential concretizations are created in each iteration. This result highly reduces the number of potential concretizations inspected. If $s = |symbols|$, t is the number of channels and $n = |V|$, the naive method creates $O(n * (s^k)^t)$ potential concretizations in each iteration. Using this abstraction, the number of potential concretizations is reduced to $O(n * s * t)$.

Reducing the views. Suppose we want to determine whether $c \in \gamma_k(V)$ given a configuration c and a set V. This would require that all views $v \in \alpha_k(c)$ are in V. Consider a view $v = \langle S, \xi(ch) = w\rangle$ with size$(w) = k$; if $v \in V$ then necessarily, any $v' = \langle S, \xi(ch) = w'\rangle$ with $w' \sqsubseteq w \in V$. Consequently, it is sufficient to assure that all configurations $c = \langle S, \xi(ch_i) = w_i\rangle$ are in V, with

- size(w_i)=k if size$(\xi(ch)) \geq k$
- size(w_i)= size$(\xi(ch))$ if size$(\xi(ch)) < k$.

Example. Suppose we want to determine whether the potential concretization $con = \langle S, abc, de\rangle$ is an element of γ_2. It is then sufficient to check that $c_1 = \langle S, ab, de\rangle$ and $c_2 = \langle S, bc, de\rangle$ are in V. The potential concretization must have been created from either $c1$ or c_2. Because of this, it is always sufficient to check for the existence of a single configuration, to determine wether *con* should be accepted.

Using these abstraction, the computational complexity of the verifier is greatly reduced, but there are yet several ways to optimize the procedure. The bottle-neck with the procedure at its current state is the choice of a *Set* as lone data structure to store configurations. We expect the number of configurations to grow rather large, due to state-space explotion which cannot be avoided automatically (in certain cases, it could be reduced by further abstraction of the model under testing or by removing unnecessary redundancy in the model). Assuming the Set is ordered, inserting an element to the set and finding an element in the set is done in *O(log n)* where n is the number of elements in the set. As a consequence, the function *alpha* which determines whether the views of a certain configuration are in the set has complexity $O(p \cdot logn)$ where

$p = |\alpha_k(c)$. We observe that by definition, a view (and a configuration) is of type *(S × ξ)*. For any configuration $c = \langle s \rangle \xi$, it's views will be of the type $\langle s \rangle \xi'$, i.e., the configuration and its views have identical control-parts. It is therefore reasonable to divide the single *Set* into a Set of *Set*s, where each *Set* corresponds to a specific control-state. With such a division, it is sufficient to check whether the views of a configuration reside in the *Set* that corresponds to its control-state.

In a best-case scenario, there would be an equal number of configurations for every state in the system. The complexity of checking whether $\alpha_k(c) \in V$ is reduced to $O(p \cdot log(n/s))$ where s denotes the number of states in the system. The gains of these modifications in terms of computational complexity is difficult to determine, as the number of configurations with a certain control-state are not necessarily well-distributed, but we anticipate that the gains in many cases will be close to the best case scenario.

6.3 Even less naive algorithm

6.3.1 Control-State Partitioning

Above it was suggested that the set V could be divided into several sets, distinguished by control-states. The number of control-states in a system is finite and unique; For a system with n processes, the number of states is $\prod_{i=1}^{n} |p_i|$. This makes it possible to partition the set of configurations into several smaller disjoint sets, such that all configurations in a set have equal control-state. These sets can be stored in a hashmap indexed by control-state.

6.3.2 Advantages of Partitioning

Using this partitioning, each node in the hashmap is a *Set* with equal-state configurations, addressed by the common control-state. Compared to the previous solution, this leads to a speed-up when inserting, retrieving and looking for the existence of elements in the sets, as the size of the sets will be smaller than before. Retrieving and inserting nodes into the hashmap is done in constant time in practice (the number of elements will be static, avoiding the need of resizing the hashmap).

Having partitioned the configurations, consider the act of applying a rule in order to create a new configuration. Recall that any rule has a state-predicate, a channel-predicate, a state-event and a channel-event. If the predicates are fulfilled, a new configuration can be created by applying the events to the current configuration.

By organizing the set of *rules* or transitions in a similar way in a hashmap of rules, we can ensure that the state-predicate is fulfilled without specifically testing for it. A state-predicate is a (possible empty) predicate on the states of the processes in the system, requiring that one or more processes are in a specific state. It is therefore possible to generate a static hashmap of rules, addressed again by control-states.

Considering a transition t_2 with no requirements on any processes, the state-predicate will be fulfilled by any configuration, thus a corresponding rule is created in every node of the tree. Consider then instead a transition t_2 with requirements on all the processes in the system, i.e. there is only one valid control-state. Such a transition corresponds to a single rule in the node addressed by that control-state. Last, consider a transition t_3 with requirements on a true subset of the processes in the system, then the transition can be taken from a number of control-states. These control states can be generated, and the transition will correspond to a set of rules in the nodes corresponding to those control-states.

Example. Consider yet again the alternating bit protocol, as described in **??**. The system has three processes *Sender*, *Receiver* and *Observer* with 4, 4 and 3 states respectively. Consider then the transition with the predicates that the sender is in state 3, the observer in state 3 and they may take the synchronized transition with the action *Snd* to states 4 and 2 respectively. This transition has no requirements on the receiver, thus it corresponds to four rules, (3,1,3)->(4,1,2), (3,2,3)->(4,2,2), (3,3,3)->(4,3,2), (3,4,3)->(4,4,2). These rules are inserted in the nodes (3,1,3), (3,2,3), (3,3,3) and (3,4,3).

There is a small one-time cost of creating the rule tree, but after creation, it is highly effective. The division by itself ensures that the state-predicate is fulfilled, if not, we never even attempt to apply the rule to a configuration. We illustrate this in figure 2

6.4 Final subsection

An efficient algorithm in this context is largely an algorithm that avoids performing unnecessary calculations. This can either be done by avoiding to create unnecessary configurations such as was done with the rule hashmap above or by avoiding re-calculating previously calculated results. The algorithm as described above reproduces its steps each iteration; if a configuration c can be extended to a concretization *con* at any point in the verification process, then *con* can and will be created in every following iteration. This includes checking whether $\alpha_k(con) \subset V$, applying a set of rules to the configuration and then adding all the views of the resulting configurations to the set. Each time the calculations are performed, the result will be duplicates and no new views are added to the system.

We solve this by maintaining another hashmap, *seen*, of configurations in parallell, containing exactly those concretization that have been accepted. If a configuration c can be extended to the concretization *con*, then we first check if *con* is an element of seen. If it is, we discard *con*, otherwise we add *con* to *seen* and also to the set of concretizations to be evaluated in this iteration.

Yet another source of repetition is the fact that there are multiple ways to create the same channel evaluations. Therefore, after a rule has been applied to a concretization, it may result in a configuration already in the set. Instead of performing the costly α-calculation, we first check if the newly created

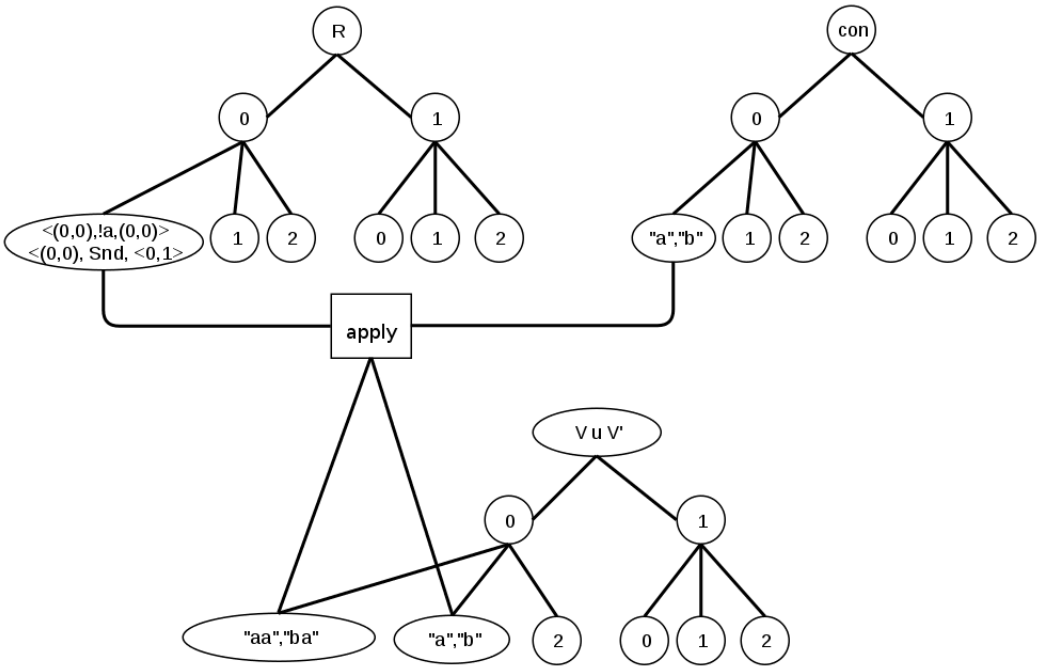


Figure 2: This picture shows the process of applying rules. For each control-state, each rules with that state are applied on each configuration with the same state. This will generate new configurations, not necessarily with the same control-state. This is visualized in a tree, rather than a hashmap, only since hashmap are less intuitive to draw.

configuration is not in fact a duplicate by checking if it is already in V. If so, the configuration can again be discarded.

The final algorithm amounts to the following pseudo-code representation:

6.5 Reachability Analysis

An important step of the verification process which has yet to be covered is that of performing a reachability analaysis, in order to find bad states, if any such states are reachable and find a minimal bad trace leading up to the bad state. See sections 2.3 and 2.4 for formal definitions of bad states and traces respectively.

Below, a simple technique of finding a minimal bad trace is presented, accompanied with a proof that the trace is in fact minimal.

Finding Minimal Traces When running the verification, if a bad state is found we want to produce a trace leading up to the bad state. Preferably, this would be a minimal trace that leads to the bad state.

The proposed verification method generates a finite set of reachable states (nodes in this context), but it does not record the available transitions between the nodes (i.e. the edges). It is possible to for each node n to save all nodes n from which an edge to n' exists, and thus build the complete reachability graph

Algorithm 2 The verification algorithm from section 6 in somewhat higher detail. This version includes

```
1: Gamma (V, Seen):
2:     con' := concretizations (V)
3:     con := c | c ∈ con ∧ c ∉ Seen
4:
5: Step (Con, Rules):
6: for state ∈ nodes(V) do
7:         S := r(c) | ∀ c ∈ con(state) ∧ ∀ r ∈ Rules(state)
8: end for
9:
10: Alpha (V, S):
11:     V:= V ∪ views(C)
12:
13: Verifier (V, Rules, Bad):
14: for True do
15:     if R_k ∩ Bad ≠ ∅ then
16:         return Unsafe
17:     end if
18:     V := μAlpha(Step(Gamma(V)))
19:     if γ_k(V) ∩ Bad = ∅ then
20:         return Safe
21:     end if
22:     k := k+1
23: end for
```

of the problem. There exists efficient algorithms to solve such a problem, e.g. *dijkstra shortest path*, or even the shortest path between any two nodes, e.g. flow-network techniques. Although these algorithms are efficient, building the complete reachability graph would be costly in terms of memory space, as the number of edges may be much larger than the number of states.

We show that due to the method of iteratively constructing the graph, nodes are created in such a way, that if a node n_i created in the i:th iteration is reached by a node n_{i-1} over an edge e_{i-1}, the shortest path from the initial node n_0 will necessarily be a path $e_1...e_{i-1}$.

Proof. This is proven using an induction proof. We hypothesize that, if at the point of creation of n_i, choosing the parent node n_{i-1} from which an edge e_{i-1} can be taken n_i, the path $e_1..e_i$ will be the shortest path to n_i and has length i. Note that the node n_{i-1} must have been created in the previous iteration; had it been created earlier, the edge e_i could have been taken in a previous iteration, and so n_{i+1} would already be a node in the tree.

The base case is that for any node reachable from n_0 over any edge e_0, e_0 will be the shortest path and has length 1. This is trivially true.

Now suppose a node n_{i+1} is reachable over an edge e_i from a node n_i, and the node n_{i+1} is not yet in the system. The induction hypothesis states that

the path $e_1...e_i$ is the shortest path leading up to n_i. If $e_0..e_{i-1}e_i$ would not be the shortest path to n_{i+1}, there would be a path $e'_0..e'_{k-1}$ to another node n_k with k < i from which n_{i+1} can be reached. But any such node would have been created in the k:th iteration of the algorithm, which would contradict the fact that the node n_{i+1} was not already in the system.

Having shown this, we need only record the information of a single parent of a node, in order to build up a tree from which the shortest path from n_0 to any node in the system can efficiently be found.

6.6 Specification Language

In order for the verification algorithm to be easily used, a specification language is needed in which algorithms can be formally defined. We expect such a specification language to

- be expressive enough to express all algorithms that are in the scope of the verification program
- be independent of the internal representations of the verifier and to demand as little knowledge of the actual verification process as possible from the user
- to be as clear as possible in order to ensure that the model at hand in fact corresponds to the actual algorithm, the way it was intended

The specification language used is an adaption of previous works by . The language simply uses XML to describe an algorithm. There are minor differences between the specification language used here, with respect to that used in by the MPass verification algorithm, and correspond to different expressivness of the models. More specifically, the MPass verifier allows for joint reception and transmission transitions, i.e. a single transition may read a message from a channel and write a message to a channel at the same time. On the other hand, it does not allow for synchronized transitions, this has therefore been added to the language.

Fixa referens

6.6.1 Example

We return yet again to the Alternating Bit Protocol to examplify the specification language. Bearing in mind the formal definition in section 2.1, a model needs a set of messages, channels, actions and transitions.

We begin by specifying a new protocol, it's name and that it is operating on FIFO buffers. We then specify a set of messages, a set of channels and a set of actions in a similar manner.

```
1 <protocol name="Alternating Bit Protocol" medium="FIFO>
2 <messages>
3 ______<message>ack0</message>
4 ______<message>ack1</message>
```

```
      <message>mesg0</message>
      <message>mesg1</message>
</messages>

<channels>
      <channel>c1</channel>
      <channel>c2</channel>
</channels>

<actions>
      <action>Rcv</action>
      <action>Snd</action>
</actions>
```

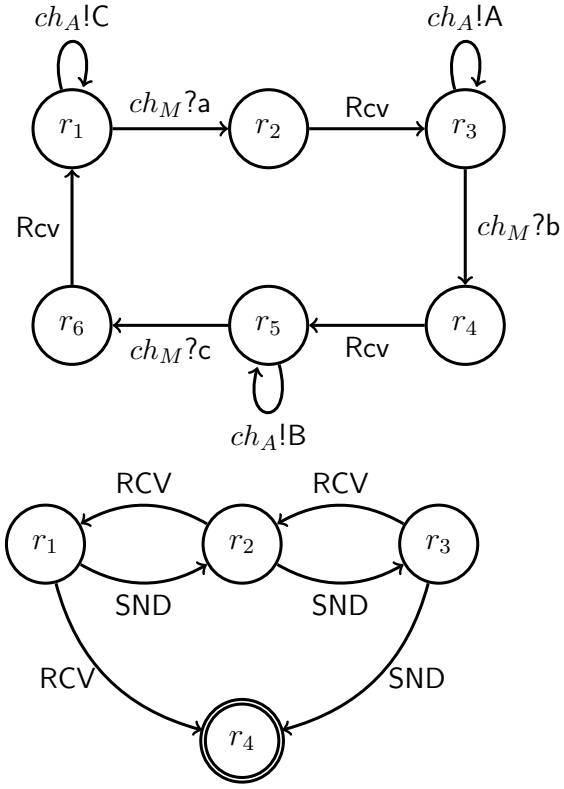
We then continue by defining the different processes in the system, i.e. the sender, receiver and observer. Such a process is defined by its states and its transitions. In the specification language, we differentiate between transitions over an action, and transitions that modify a channel. The latter type is called a *rule* in the specification language, in order to comply with the names used in the MPass specification language. This should not be confused with the word as used earlier in this section.

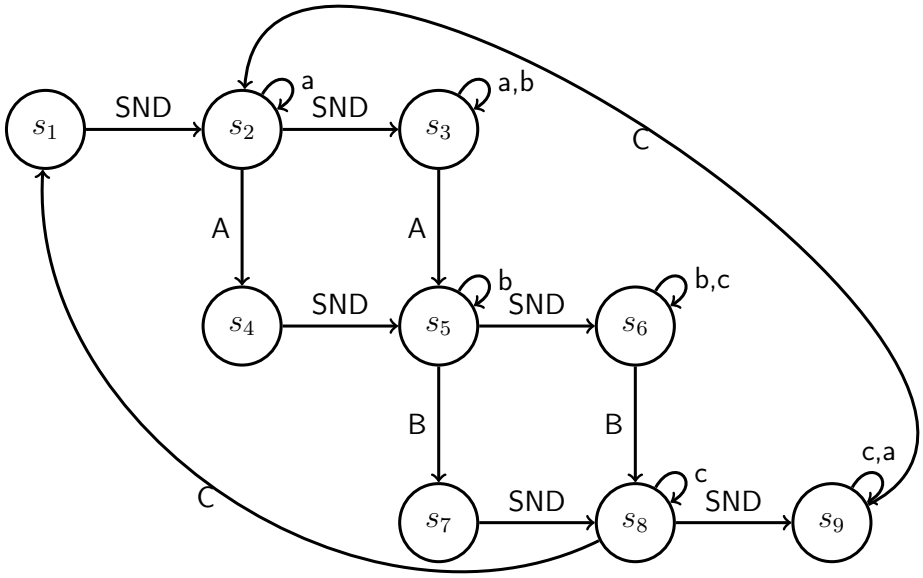
```
  <role name="SENDER">
    <states>
      <state type="initial">Q0</state>
      <state>Q1</state>
      <state>Q2</state>
      <state>Q3</state>
    </states>
    <action>
        <current_state>Q0</current_state>
        <type>Snd</type>
        <next_state>Q1</next_state>
    </action>
    <rule>
        <current_state>Q1</current_state>
        <send_message>mesg0</send_message>
        <next_state>Q1</next_state>
        <channel>c1</channel>
    </rule>
```

Last, we specify that the processes `SENDER` and `OBSERVER`, and `RECEIVER` and `OBSERVER` are to be synchronized over the actions `Snd` and `Rcv` respectively.

```
<synchronize>
        <first_role>SENDER</first_role>
        <second_role>OBSERVER</second_role>
        <action>Snd</action>
</synchronize>
<synchronize>
```

```
7           <first_role>RECEIVER</first_role>
8           <second_role>OBSERVER</second_role>
9           <action>Rcv</action>
10  </synchronize>
```





References

[1] P. Abdulla, F. Haziza, and L. Holík, "All for the price of few," in *Verification, Model Checking, and Abstract Interpretation* (R. Giacobazzi, J. Berdine, and I. Mastroeni, eds.), vol. 7737 of *Lecture Notes in Computer Science*, pp. 476–495, Springer Berlin Heidelberg, 2013.