Algorithmic Analysis of Channel Machines Using Small Models

Jonathan Sharyari

29th December 2014

Abstract

Verification of infinite-state systems is in general an undecidable problem, but nevertheless, solid correctness results are important in many real-life applications. It is commonly the case that such algorithms rely on unbounded buffers for their operation, communication protocols being a typical example. Building upon abstract interpretation techniques, this project presents a verification algorithm capable of verifying the correctness of algorithms relying on unbounded *lossy* buffers.

We have implemented our approach and used it to verify a number of well-known communication protocols. The experimental results are mixed, but in most cases comparable to that of other exisiting tools, justifying further research on the topic.

Contents

1 Introduction **4**

2 Formal Model for Lossy Channel Systems **7**
2.1 Preliminaries 7
2.1.1 Words and Alphabet 7
2.1.2 Fixpoints 7
2.2 Lossy Channel Systems 8
2.2.1 Syntax 8
2.2.2 Configurations 8
2.3 Semantics 9
2.4 Alternating Bit Protocol 9
2.4.1 The Verification Problem 11

3 Verification of Lossy Channel Systems **13**
3.1 Views 13
3.2 Abstraction function 13
3.3 Concretisation function 13
3.4 Post-Image 14
3.5 Abstract Post-Image 14
3.6 Reachability Analysis 14
3.7 Small Models 14
3.7.1 Proofs 15
3.8 Verification Algorithm 15

4 Extensions **17**
4.1 LIFO Channels 17
4.2 Lossy Channels 18
4.3 Channel Systems with Synchronization 19
4.4 Alternating Bit Protocol Revised 19

5 Implementation of the Verification Method **21**
5.1 Naive implementation 21
5.2 Improving the Implementation 22
5.3 Better Implementation 23
5.3.1 Control-State Partitioning 23
5.3.2 Advantages of Partitioning 24
5.4 Final Implementation 25
5.5 Reachability Analysis 26
5.6 Specification Language 27
5.6.1 Example 28

6 Result **30**
6.1 Experimental Results 31
6.2 Analysis of Results 31

7 Summary and Further Work **32**

8 Related Work **33**

A Automata for the Sliding Window Protocol **36**

1 Introduction

Todays society grows more and more dependent on computer applications. Often they are used directly, for example the task of paying 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 harm.

This has motivated research on various techniques to find and correct potential faults, particularly in safety-critical systems. The most common technique is that of *testing*, i.e. observing the system behaviour on a variety of likely or even unlikely scenarios. The widespread use of testing is well-motivated, but there are several scenarios where testing by itself is insufficient, as testing does not provide any guarantee of correctness.

Another technique used for verification is the process of *simulation*. In comparison to testing, simulation can be done in an earlier stage of the development. Rather than first implementing an algorithm, one may simulate an abstracted model of the algorithm which may help in ensuring 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 as a whole.

The goal of *model checking* is to formally verify the correctness of concurrent programs with respect to a given specification. In general, given a model of a system and a specification of its intended properties, the task is to decide whether the model meets its specification or not.

Different representations of models and properties have been proposed, as well as techniques to perform the verification. One of the most successful and common methods is that of *temporal logic model checking*. Traditionally, the model is expressed as a finite state machine and the properties as propositional logic formulae. Several verification methods for such models have been proposed[21] and although these methods have been applied to infinite models[13], the focus has in large been on the verification of finite models.

A trend in todays computing is that applications become concurrent or distributed. Such programs are inherently difficult to verify, as although processes themselves have finite models, the results of the asynchronous behaviour of processes may be infinite. This is the case for example when the composition of a possibly infinite number of processes is observed, a common scenario when verifying concurrent and distributed programs. Also, we are often not interested in performing verification of a *specific* composition of concurrent programs, but rather in verifying that it works for *all* compositions.

Related to this is the necessity for distributed and concurrent programs to communicate, and to do so accurately. In general, the underlying systems that we use for our communication are unreliable, in that message loss or message corruption may occur. This happens due to a variety of reasons, may that

be errors in the communication links, relays or the communicating parties themselves. To overcome this problem, communication protocols are designed to be resistent to such faults, and ensuring the correctness of these protocols is paramount when ensuring reliable communication. Even when the number of communicating processes is finite, communication protocols commonly result in infinite models, as the number of sent messages may be unbounded, so there could be an infinite number of messages in the buffer. The verification of systems relying asynchronous unbounded buffers, is the focus of this thesis.

Such *channel systems* are common in communication protocols using channels and in distributed computing[16]. Channels systems may also represent other types of programs, such as cache coherence protocols and sensor systems[23]. An important note is that channel systems using *perfect channels*, i.e. channels without message loss, are Turing powerful, and all non-trivial problems are therefore undecidable. However it has been proven that verification of *safety properties* of channel systems over *lossy channels*, i.e. channels that may nondeterministicly lose messages, is decidable[2][8]. Nevertheless, the problem is difficult to verify, having high computational complexity.

The main problem when verifying such systems is to find a way to construct a finite approximation of the infinite system. In order to solve this problem, we find inspiration in [1], where the verification of *parameterized systems* is attempted. *Parameterized systems* are systems composed of a set of processes, where the number of processes is itself a parameter of the system – this is often the case when discussing concurrent and distributed programs. Although an undecidable problem, there are methods that provide good approximation techniques which consist in reducing the verification task to the case of finite state systems.

In [1], the authors take advantage of the fact that parameterized systems often enjoy a *small model property*, meaning that the verification of small instances of the system is enough, to capture the reachability of bad configurations. An important contribution of [1] is that they show that this property holds for any *well-structured transition systems*[3] (i.e. transitions systems monotonic with respect to a well-quasi ordering).

Building upon the work in [1], we explore the possibility of adapting the verification algorithm proposed in [1], in order to verify systems with unbounded lossy channels.

The verification algorithm is based on an enumerative exploration of all the finite approximations by bounding the size of the channels. To verify each bounded instance of the system (let us assume that the size of the buffers is bounded by $k \in \mathbb{N}$) we use a standard forward reachability algorithm. If the reachability analysis returns a counter-example, then this a real counter-example of our system and the analysis procedure terminates. Otherwise, if the analysis procedure declares the finite approximation as safe, then we proceed to a second verification step. This extra verification step uses abstract interpretation techniques[10][11], with the goal to check if k is a *cut-off* point which means that there is no need to verify any larger instance and we can claim that the program is safe. If the second step determines that the algorithm is

safe, then the algorithm is truly safe for that instance and all larger instances. If, on the otherhand, a counter-example is found, this means that the reachability analysis must be repeated, with the buffers bounded to size $k + 1$.

As lossy channel systems are well-structured, the small model property holds for the classes of systems being considered in this thesis, thus a cut-off point will eventually be found, leaving a termination guarantee for the algorithm.

The verification technique was implemented and tested for some common communication protocols. The results were compared to that of two other verification techniques for the same classes of systems; the MPass[4] verifier translates verification problems to satisfiability problems, solvable using third party SAT-solvers, and a *backward reachability analysis* as described in [2]. The backward analysis algorithm was was implemented within the scope of this project for the purpose of comparison. The experimental results confirm the correctness of the results, and show that the verification times are comparable or faster than the alternative methods for most of the verified protocols.

Reading this paper. Section 2 contains formal definitions to some of the key concepts – these are previously known concepts, asides from their adaption to the specific context of this thesis. Section 3 introduces the concepts of small models and abstract interpretation, and relates these to buffer channels to create the theoretical model for this paper. Section 4 introduces some extensions to this model, allowing for a wider variety of modeling scenarios. In section 3.8, some abstractions and techniques are explained 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 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 6, as well as some comparisons against the results of other verification tools.

2 Formal Model for Lossy Channel Systems

In this chapter we will give the formal definition of a channel system. First we define some necessary terminology, such as alphabets and words, before giving the syntax and semantics of a channel system. Then we illustrate these ideas by applying them to a well-known communication protocol, the *alternating bit protocol*. Finally, we state the formal definition of the verification problem we are investigating.

2.1 Preliminaries

2.1.1 Words and Alphabet

A word w is a finite sequence of letters from the alphabet; that is $w = a_1 a_1 \cdots a_n$. We use ϵ to denote the empty word. We define the length of a word w to be the number of letters appearing in it (i.e., the length of the word $w = a_1 a_2 \cdots a_n$, with $a_1 a_2 \cdots a_n \in \Sigma$, is n). We use $|w|$ to denote the length of the word w. Observe that $|\epsilon| = 0$. We use Σ^* (resp. Σ^+) to denote the set of all words (resp. non-empty words) over the alphabet Σ. Let k be a natural number. We use Σ_k^* (resp. Σ_k^+) to denote the set of all words (resp. non-empty words) of length at most k. At times, we use the symbol $\bullet$ to emphasize the concatenation of two words, i.e. $w_1 w_2 w_3$ can be written as $w_1 \bullet w_2 \bullet w_3$.

Let the relation $\leq$ be defined such that for $w_i, w_j \in w$, $w_i \leq w_j$ if $i \leq j$. We define the relation $\sqsubseteq$ to be the *subword* relation, s.t. $u \sqsubseteq w_1 ... w_n = w$ iff u is an ordered subset of w. Mathematically the subword relation can the be defined such that $u = u_1 ... u_n \sqsubseteq w_1 ... w_n = w$ if $\forall u_i, u_j \in u$ there exists $w_k, w_l \in w$ such that $u_i = w_k$, $u_j = w_l$ and $u_i \leq u_j \iff w_k \leq w_l$.

As an example, if w=abc, then the set of subwords of w is abc, ab, ac, bc, a, b, c, ε.

2.1.2 Fixpoints

Let A be a set. We use 2^A to denote the set of all the subsets of A. Let F be a function with 2^A as its domain and co-domain. We say that F is a *monotonic function* if and only if for every subset $X, Y \subseteq A$, if $X \subseteq Y$ then $F(X) \subseteq F(Y)$. We say that X is a *fix-point* of F iff $F(X) = X$. We use $\mu X.F(X)$ to denote the least fixpoint of F with respect to the subset relation (i.e., $\mu X.F(X)$ denotes the smallest set $D \subset A$, such that D is a fix-point of F). From the Knaster-Tarski theorem [**?**], we know that the least-fix point for a monotonic function exists. Furthermore, if A is finite, then the sequence $X_0 = I$, $X_{i+1} = f(A_{n-1})$ for all $i \geq 0$, converges in finitely many iterations to the least-fixpoint of F.

2.2 Lossy Channel Systems

2.2.1 Syntax

We present here the syntax of a finite-state system with unbounded channels. Such a system can be seen as two parts, a *control part* and a *channel part*. The channel part is a set of *channels* containing a word. The control part is a labeled finite-state transition system.

Formally, a lossy channel system LCS over a set of processes P, channels Ch and messages M is a tuple $\langle S, s^0, A, Ch, M, delta \rangle$, where

- S is a finite set of control states
- s^0 is the initial control state
- A is a set of labeled actions
- Ch is a finite set of channels,
- $M \subset \Sigma^+$ is a finite set of messages (words) over a finite alphabet Σ,
- δ is a finite set of transitions, each of which is a triple of the form $\langle s, op, s' \rangle$, where $s, s' \in S$ are control states, and op is a label of one of the forms
  - $ch!m$, where $ch \in Ch$ and $m \in M$
  - $ch?m$, where $ch \in Ch$ and $m \in M$
  - $a \in A$.

The finite-state control part of LCS is set of states S, with the initial state s^0, labeled actions a and transitions δ. The channel part is represented by the set Ch of channels, which contain a word over M. The set A denotes the set of internal transitions, which only alter the control state of the system, wheras δ_i may either be an action from A, or an action where

> $\langle s, ch!m, s' \rangle$ represents a change of state from $s \in S$ to $s' \in S$ while appending the message $m \in M$ to the tail of channel $ch \in Ch$.
>
> $\langle s, ch?m, s' \rangle$ represents a change of state from $s \in S$ to $s' \in S$ while removing the message $m \in M$ from the head of channel $ch \in Ch$

2.2.2 Configurations

Let $\xi : ch \rightarrow \Sigma^*$ be a function that maps the content of a channel $ch \in Ch$ to the word on the channel. We will call this the *evaluation* of the channel. We define a *configuration* to be a tuple $c = (s, W)$, such that s is the global control state and W is a sequence of channel evaluations: $W = (W_1, W_2 ... W_m)$ s.t. $W_i = \xi(ch_i)$ for a channel $ch_i \in Ch$, $1 \leq i \leq m$. For a sequence of evaluations $W = (W_1, W_2 \cdots W_n)$ we use W_i to denote the $i : th$ evaluation in the sequence. Let C denote the set of all possible configurations. We define $state : C \rightarrow S$ to be a function mapping a configuration to its state, and $eval : C- > W$ to be

a function mapping a configuration to its evaluation, and use the shorthand $c_S = state(c)$ and $c_E = eval(c)$ for a configuration c.

Let ϕ denote a sequence of empty words, that is $\phi = \{\epsilon, \epsilon \cdots \epsilon\}$. We define c^0 to be the initial configuration $\langle s^0, \epsilon \rangle$, i.e. the configuration with all processes in their initial state and all channel evaluations being the empty word.

We define the size of a configuration $c = (s, W)$ with $W = (w_1, w_2, \ldots, w_m)$ to be the length of the longest word w_i, i.e., size(c)$=max\{|w_i| \ |i \in \{1, ..., m\}\}$

2.3 Semantics

In the following section, we explain the semantics of a lossy channel system, which describes the behaviour of the system. The operational behaviour of LCS induces the inifinite-state transition system $LTS = (C, \rightarrow)$ where C is a possibly infinite set of configurations and $\rightarrow \subseteq (C \times C)$ is the smallest transition relation defined as follows:

- For each action $\langle s, a, s' \rangle \in \delta$, $\langle s, W \rangle \xrightarrow{a} \langle s', W \rangle$. This means that the control state changes with the action a.
- For each transmission transition $\langle s, ch_j!m, s' \rangle$ in δ, $\langle s, W \rangle \xrightarrow{ch_j!m} \langle s', W' \rangle$ where $W'_j = W_j \bullet m, W'_i = W_i, \forall i \neq j$. This means that the control state and the evaluation of the configuratoin change, so that the message m appended to the end of the $j : th$ evaluation.
- For each reception transition $\langle s, ch_j?m, s \rangle$ in δ_i, $\langle s, W \rangle \xrightarrow{ch_j!m} \langle s', W' \rangle$ such that if $W_j = m \bullet x$, $W'_j = x, W_i = W'_i, \forall i \neq j$. This means that the control state and the evaluation of the configuratoin change, so that the message m is removed from the head of the $j : th$ evaluation.
- Additionally, $\langle s, W \rangle \xrightarrow{ch_j*} \langle s, W' \rangle$ such that if $W_j = x_1 \bullet m \bullet x_2$, $W'_j = x_1 \bullet x_2, W_i = W'_i, \forall i \neq j$. This signifies that a message loss has occurred on the $j : th$ channel, leaving the control state unchanged.

Notations for configurations Depending on the context, different notations to describe a configuration $c \in C$ may be used. For example, $c = \langle s, W \rangle$ can be denoted as $c = \langle s, W_1, W_2, ..., W_3 \rangle$ when talking about the specific channel evaluations, or with the words on a channel explicitely written, e.g. $c = \langle s, abc, def \rangle$.

2.4 Alternating Bit Protocol

Here we present a simple protocol, the *alternating bit protocol* [7]. This protocol will serve as a running example of the theoretical concepts in this section and following sections.

The Alternating Bit Protocol (ABP)is a distributed protocol for transmitting reference 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 som data with with a 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*, hence the name Alternating Bit Protocol. The behaviour of the sender and receiver process are illustrated in figures 1a and 1b, and the pseudo-code for this algorithm is seen below.

Algorithm 1 ABP Sender

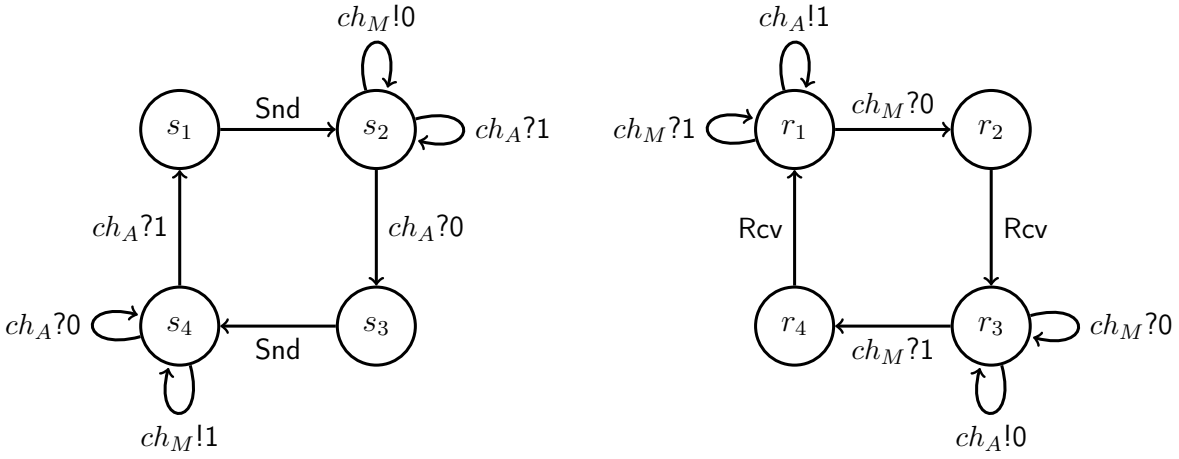
```
1: b := False                                          ▷ Initially False
2: for True do
3:     msg.send(data, b);                              ▷ Send the message
4:     while ack.receive().b != b do      ▷ Wait until ack has the correct ID
5:         msg.send(data, b);                          ▷ Re-send the message
6:     end while
7:     b := !b                                         ▷ Alternate the bit
8: end for
```

Algorithm 2 ABP Receiver

```
1: b := False                                          ▷ Initially False
2: for True do
3:     while msg.receive().b != b do      ▷ Wait until msg has the correct ID
4:         ack.send(b);                ▷ (Re-)transmit the acknowledgement
5:     end while
6:     b := !b                                         ▷ Alternate the bit
7: end for
```

In the pseudo-code above, both the sender and receiver have an internal boolean variable b, which at any time has the value of the ID they expect the next message to have, which may either be False (0) or True (1). The sender will repeatedly send a messages with the current value of b to as ID, until it receives an acknowledgement from the receiver with the same ID, at which point it alternates the value of b. The receiver behaves the same as the sender, with the exception that it does not initiate the communication.

Syntax. The alternating bit protocol can be described by a channel system $CS = \langle S, s^0, A, Ch, M, \delta \rangle$ such that $S = \{(s_i, r_i)\}$ with $i \in \{1, 2, 3, 4\}$, s^0 is the initial state (s_1,r_1), $A = \{Snd, Rcv\}$, $Ch = \{Ch_M, Ch_A\}$, $M = \{1,0\}$ and δ is the set of transitions



(a) Sender (b) Receiver

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

$$
\begin{array}{ll}
(s_1, r_1) \xrightarrow{Snd} (s_2, r_1) & (s_1, r_1) \xrightarrow{ch_A!1} (s_1, r_1) \\
(s_2, r_1) \xrightarrow{ch_M!0} (s_2, r_1) & (s_1, r_1) \xrightarrow{ch_M?1} (s_1, r_1) \\
(s_2, r_1) \xrightarrow{ch_A?1} (s_2, r_1) & (s_1, r_1) \xrightarrow{ch_M?0} (s_2, r_1) \\
(s_2, r_1) \xrightarrow{ch_A?0} (s_3, r_1) & (s_2, r_1) \xrightarrow{Rcv} (s_3, r_1) \\
(s_3, r_1) \xrightarrow{Snd} (s_4, r_1) & (s_3, r_1) \xrightarrow{ch_A!1} (s_3, r_1) \\
(s_4, r_1) \xrightarrow{ch_M!1} (s_4, r_1) & (s_3, r_1) \xrightarrow{ch_M?0} (s_3, r_1) \\
(s_4, r_1) \xrightarrow{ch_A?1} (s_1, r_1) & (s_3, r_1) \xrightarrow{ch_M?1} (s_4, r_1) \\
(s_4, r_1) \xrightarrow{ch_A?0} (s_4, r_1) & (s_4, r_1) \xrightarrow{Rcv} (s_1, r_1)
\end{array}
$$

Notations Let $c = \langle s, W \rangle$ 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 $c = \langle s, ch_M, ch_A \rangle$ or as $c = \langle s, 01, 10 \rangle$. There are finitely many control states in this system (precisely 16), but an infinite set of channel evaluations. The initial configuration $c^0 = \langle (s_1, r_1), \epsilon \rangle$.

Although the channel system has a finite number of transitions (listed above) between states, the transition system has an infinite number of transitions between configurations, as these depend also on the evaluations. An example of such a transition $\langle s_2, r_1, 01, 10 \rangle$, $ch_M!1$, $\langle s_2, r_1, 011, 10 \rangle \rangle$.

2.4.1 The Verification Problem

A channel system can potentially reach a *bad* state, i.e. a state representing unintended behaviour. Suppose Bad is a set of bad configurations of a lossy transition system LTS, as defined in **??**. We call a sequence $t = c^0 c^1 \cdots c^n$ of configurations such that for each $0 \leq i \leq (n-1)$, there exists a transition $c^i \rightarrow c^{i+1}$ a *trace* from c^0 to c^n. We call a trace a *bad trace* if there exists

$c^{bad} \in Bad$ such that $c_S^{bad} = c_S^n$ and $c_E^n \subseteq c_E^{bad}$. Note that a configuration with a bad control state is a bad configuration, regardless of the content of the channel evaluation. If t is the shortest bad trace in the system, we call it a *minimal bad trace*.

A configuration c is said to be *reachable* in *LTS*, if there is a trace from the initial configuration c^0 to c. Let $\mathcal{R}$ denote the set of all reachable configurations. The verification problem is to determine, for a channel transition system *LTS* with an initial configuration c^0 and a set $Bad \subseteq C$ of bad configurations, whether a system is safe, that is, does not have any reachable bad configurations. More precisely, this means determining if $\mathcal{R} \cap Bad = \emptyset$.

3 Verification of Lossy Channel Systems

The goal of this chapter is to formalize the use of *small models* for the verification of lossy channel systems, as is done in [1]. The technique is based on the use of *abstract interpretation* techniques, first formalized by Cousot and Cousot[10]. Abstract interpretation techniques are techniques of approximating programs. Using information about the control and data flow, i.e. the semantics of a system, an overapproximation of the possible configurations of a system can be created, which is typically an infinite set of configurations. In [1] the authors show that such an infinite set of configurations can be safely bounded to a finite set of configurations, by finding a *cut-off* point for the maximum size of the evaluations.

3.1 Views

We define the *views* of a configurations $c = \langle s, W \rangle$ to be the set $V = \{\langle s, W' \rangle | W' \sqsubseteq W$. We say that $views(c) = V$. By definition, the views of a configuration are themselves configurations.

Example will be updated.

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

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

3.2 Abstraction function

For a given parameter $k \in \mathbb{N}$, we use C and V to denote sets of configurations and views respectively, and C_k and V_k to denote configurations and views of size up to k.

The abstraction function $\alpha_k : C \to 2^{C_k}$ maps a configuration c into the powerset V of views of size up to k, such that for each $v \in V$, $\{v_eval \subseteq c_eval\}$, $v_state = c_state$ and $size(v_eval) \leq k$.

3.3 Concretisation function

The concretisation 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, $\gamma_k(V) = \{c \in C \mid \alpha_k(c) \subseteq V\}$

In general $\gamma_k(V)$ is an infinite set of configurations. We define $\gamma_k^l(V) := \gamma_k(V) \cap C_l$ for some $l \geq 0$. The intuitive meaning is that $\gamma_k^l(V)$ is the set of configurations of size at most l for which all views of length at most k are in V.

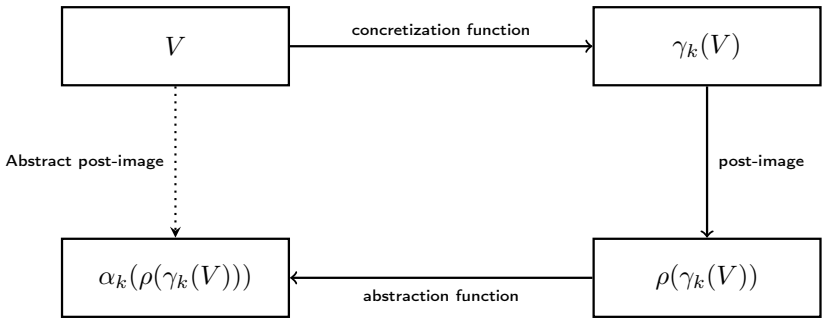


Figure 2: The application order of the α, ρ and γ

3.4 Post-Image

For a configuration $c \in C$ of a lossy transition system $LTS = (C, ->)$, we define the *post-image* of c denoted $\rho(c) = \{c' \mid c \rightarrow c' \in \rightarrow\}$. Intuitively, the post-image of a configuration is the configuration obtained by firing the transitions of the transition system.

For a *set* of configurations V we define the post-image of the set $\rho(V) = \{c' | c \rightarrow c' \in \rightarrow, c \in C\}$. This means that the post-image of a set V of configurations is the set $\rho(V)$ containing the post-images of each configurations $c \in V$.

3.5 Abstract Post-Image

The *abstract post-image* of a set $V \subseteq C_k$ is defined as $Apost_k(V) = \alpha_k(\rho(\gamma_k(V)))$. This means that the abstract post-image of a set V is the set obtained by applying γ_k, ρ and α_k in order to the set V. The procedure is illustrated in figure 2.

Note to self: When do we say views, when do we say configurations?

Note that since $V \subset Apost_k(V)$, $Apost_k$ is a monotonic function, and that the set V is a powerset because the function α returns a powerset. This means that by the Tarski fixed point theorem, the function $Apost_k$ has a fixed point.

Even the infinte powerset must reach a fixpoint, right?

3.6 Reachability Analysis

$\mathbb{R}_k$ was defined in the last chapter, here I would add a section on how to compute it using forward reachability

.

3.7 Small Models

Calculating the abstract post-image of a set of views $V \subseteq C_k$ is essentiatial for the verification procedure. As $\gamma_k(V)$ typically is infinite, this cannot be done straightforwardly. It is the main result of **??** that it suffices to consider

configurations of $\gamma_k(V)$ of sizes up to $k+1$ (γ_k^{k+1}), which is a finite set of configurations for which the abstract post-image can be computed. Formally, they show that

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

Below, we show that this lemma holds in the context of lossy channel systems. We show that for any configuration $c \in \gamma_k(X)$ of size $m > k+1$ such that there is a configuration c' where $c' \xrightarrow{r} c$, 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 $v' \in \alpha_k(d')$.

3.7.1 Proofs

The proofs need to be updated, but I await your comments on the definitions in the last chapter. Also, they need a restructuring anyway. Maybe move this to an appendix?

Transmission rules

Reception rules

Actions

3.8 Verification Algorithm

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 larger k. Given a set of bad states Bad, a set of initial states I and a set of transitions, Verification of a system can be done with algorithm 3, which was first presented in [1].

Algorithm 3 General Verification algorithm

for $k := 1$ **to** ∞
  if $\mathcal{R}_k \cap Bad \neq \emptyset$ **then return** Unsafe
  $V := \mu X.\alpha_k(I) \cup Apost_k(X)$
  if $\gamma_k(V) \cap Bad = \emptyset$ **then return** Safe

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

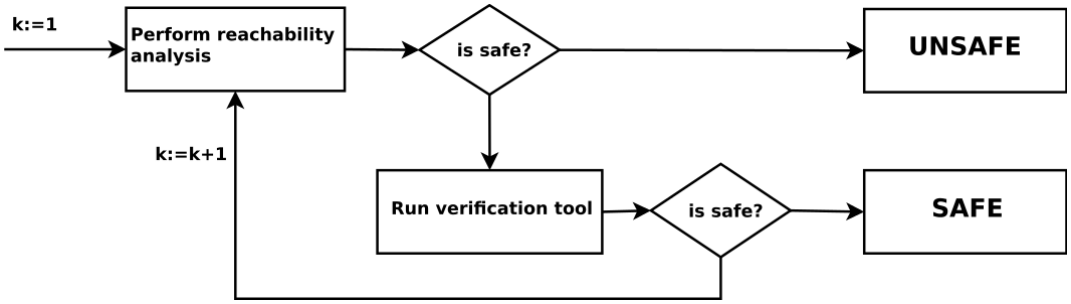


Figure 3: The general flow of the verifier.

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 $k+1$. This process is abstractly described in the flowchart in figure 3.

4 Extensions

There are several ways the channel system and the channel transition system in 2.2.1 and **??** 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.

4.1 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 2.2.1 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 **??** so that they reflect the changes made in 4.1. 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 3.7.1, by considering configurations of the form $\langle S, m \bullet w' \rangle$ rather than $\langle S, w' \bullet m \rangle$.

4.2 Lossy Channels

A lossy channel system is a system similar to 2.2.1, 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 2.2.1 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 **??** 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 3.7 hold also for the lossy channel transition system.

4.3 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, 4, synchronization can also be used as a modelling technique even if the program being modelled does not synchronize.

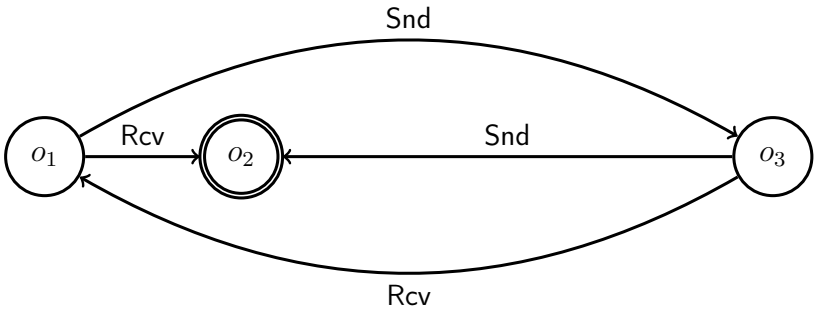
A channel system is in a sense the interleaving of multiple *program graphs*. When operating such a level of detail, a synchronizaition action with a label l corresponds to an action that cannot be taken, unless each program where such an action is present take the action simultaneously.

As the working model in this paper, i.e. the channel system described in 2.2.1 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.

4.4 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.

Figure 4: An observer machine which takes synchronized transitions with the sender and the 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.

5 Implementation of the Verification Method

In this section, we show how to implement the 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.

Ref

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

5.1 Naive implementation

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 3.2. 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'.

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, $\alpha_k con$, *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.

5.2 Improving the Implementation

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 $(S \times \xi)$. 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.

5.3 Better Implementation

5.3.1 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.

5.3.2 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 4. 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 5

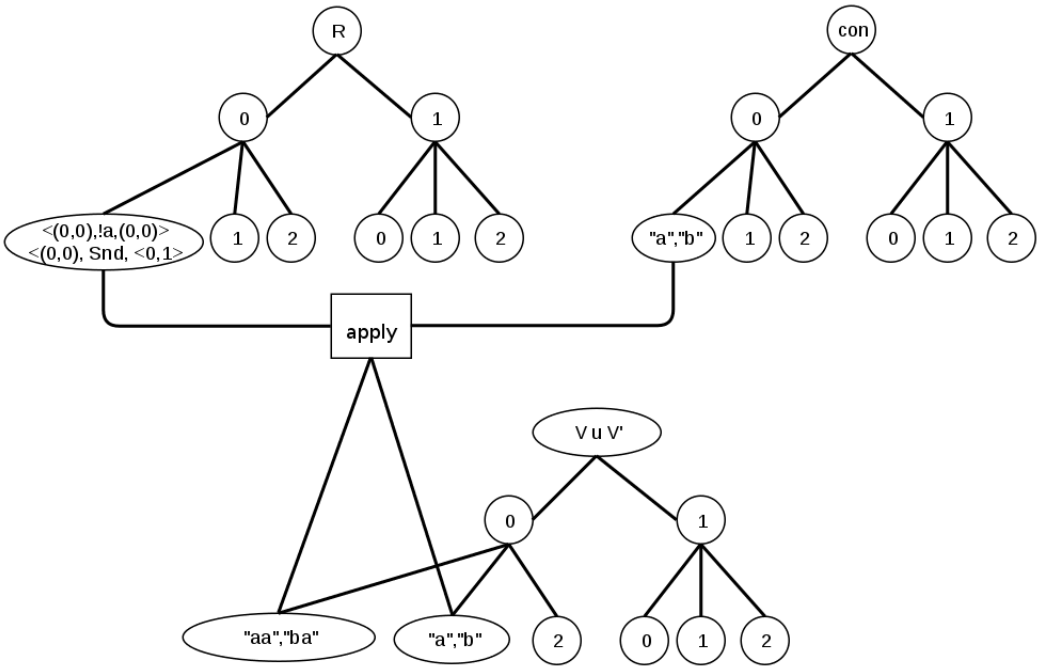


Figure 5: This picture shows the process of applying rules. For each control-state, each rules with that state are applied on each configuration with the same state. This will generate new configurations, not necessarily with the same control-state. This is visualized in a tree, rather than a hashmap, only since hashmap are less intuitive to draw.

5.4 Final Implementation

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

Algorithm 4 The verification algorithm from section 3 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
```

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:

5.5 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 **??** and **??** 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 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.

5.6 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 in [4]. 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.

5.6.1 Example

We return yet again to the Alternating Bit Protocol to examplify the specification language. Bearing in mind the formal definition in section 2.2.1, 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>
5  ␣␣␣␣␣␣<message>mesg0</message>
6  ␣␣␣␣␣␣<message>mesg1</message>
7  </messages>
8
9  <channels>
10 ␣␣␣␣␣␣<channel>c1</channel>
11 ␣␣␣␣␣␣<channel>c2</channel>
12 </channels>
13
14 <actions>
15 ␣␣␣␣␣␣<action>Rcv</action>
16 ␣␣␣␣␣␣<action>Snd</action>
17 </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.

```
1    <role name="SENDER">
2      <states>
3        <state type="initial">Q0</state>
4        <state>Q1</state>
5        <state>Q2</state>
6        <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>
        <first_role>RECEIVER</first_role>
        <second_role>OBSERVER</second_role>
        <action>Rcv</action>
</synchronize>
```

6 Result

An implementation of this method has been written in the *Haskell* programming language (`https://www.haskell.org/haskellwiki/Haskell`). It was used to verify a number of parameterized systems, all of which are communication protocols;

ABP	The alternating bit protocol, as described in 1 with the extension suggested in 4. The Alternating Bit Protocol corresponds to the *Sliding Window Protocol* with k=2. Automatas for the the Sliding Window Protocol can be found in A.
ABP_F	A purpously faulty version of the ABP.
SW3	The sliding window protocol, with k=3.
ABP_F	A purpously faulty version of the SW3.
SW4	The sliding window protocol, with k=3.
SW5	The sliding window protocol, with k=3.
BRP	The *Bounded Retransmission Protocol*, a variant of the alternating bit protocol.
BRP_F	A purpously faulty version of the BRP.

The `Backward` verifier uses *backward reachability* for verification, and follows the algorithm described in [2]. The algorithm was implemented as part of this project with the purpose of comparison in mind. It was therefore implemented in such a way that it accepts exactly the same specification language, as that of the verifier of this project.

The *MPass* verifier addresses the verification bounded-phase reachability problem, i.e. every process may perform only a bounded number of *phases* during a run in the system. This problem is then translated into a satisfiability formula (a quantifier-free Presburger formula to be exact) which in turn is solved by a third party SMT-solver (Z3 SMT-solver, `http://z3.codeplex.com`).

Note on the MPass verification results. The input language used in this project (see 5.6) was based on that of the *MPass* verifier, but does not fully coincide. The tool ships with some example protocols (ABP, ABP_F, SW3_F, BRP), which with minor changes could be adapted such that they would comply to the specification language used here. The remaining protocols were written from scratch, and use specification artifcats not comprehensible by *MPass*. Reformulating these would result in very different models, making it difficult to draw any conclusions from a comparison. For this reason, *MPass* was only used to verify a subset of the protocols mentioned above.

It should be mentioned that the time estimation for the *MPass* solver is part the time to generate an *SMT* formula, and part to verify that formula with a third party *SMT*-solver. The times generated in the latter part were approximated to full seconds, meaning the values have a measurement error up to ± 0.5 seconds.

						Backward			MPass		
	k	**size(V)**	**Result**	**Time**	**Mem**	**Size V**	**Result**	**Time**	**Bound**	**Result**	**Time**
ABP	2	108	Safe	0.00s	1MB	56	Safe	0.01s	3	Safe	1.04s
ABP_F	1	–	Fail	0.00s	1MB	–	Fail	0.01s	3	Fail	6.04s
SW3	3	4247	Safe	0.10s	3MB	270	Safe	0.17s	–	–	–
SW3_F	1	–	Fail	0.00s	1MB	–	Fail	0.10s	3	Fail	26.08s
SW4	4	98629	Safe	3.64s	36MB	840	Safe	2.03s	–	–	–
SW5	5	1834345	Safe	120.20s	924MB	2028	Safe	24.31s	–	–	–
BRP	2	45	Safe	0.02s	3MB	–	??	Timeout	3	Safe	1.23s
BRP_F	1	–	Fail	0.00	2	–	Fail	0.15	–	–	–

Table 1: Runtime and verification results for the verification method in this paper, in comparison to backward reachability and the MPass verifier.

6.1 Experimental Results

The results can be seen in 1. The table shows the size of the variable k, whether the result was safe or unsafe, the runetime and the amount of working memory used by the verifier for each of the communication protocols above. Further, the table shows the runtime and result of two other verifiers, verifying the same protocols. For these verifiers, only the runtime and the safetyness result is specified.

The results were generated on a 3.2GHz Intel Core i3 550 with 4GB of memory, running Debian Linux, using a single core.

6.2 Analysis of Results

The results in 1 show that the verification tool is in large efficient, yielding comparable verification times to backward reachability and faster than the *MPass* verifier for all of the protocols tested. Further, all verifiers yield the same safetyness results, which is to be expected as all three solvers are free from both false positives and negatives.

Examining the sliding window protocols of increasing size, i.e. ABP, SW3, SW4 and SW5, it is evident that the verifier suffers from a higher degree of exponential growth with respect to the problem size, than does the backward reachability verifier (which also suffers from exponential growth, as this is a inherent property of the model). First and foremost, it is the amount of working memory that is the bottleneck.

Increasing the size of the sliding window protocol means increasing the window size of the protocol – this not only means that the value of k for which the protocol can be found safe increases, but also that there is additional symbols to consider, and that the number of states and transitions increase (i.e. 1 compared to A). The number of potential configurations depend on exactly these factors and due to the overapproximation configurations, we expect that a large set of these will be reachable. As the verification tool stores *all* reachable configurations, without approximation or abstraction, this leads to memory becoming a bottle-neck for this particular protocol.

7 Summary and Further Work

In this paper I have investigated the possibility of verifying systems depending on lossy channels. The approach was to model such systems as parameterized systems and to use abstract interpretation techniques in order to bound systems with otherwise inifinte reachable states, to a carefully chosen finite subset of those states. This method is influenced by and builds upond previous work[1], adapting the existing verification method to work for unbounded lossy channels.

The verification method was implemented, and a protocol specification language was designed, influenced by the XML-layout used by the MPASS verification language[4]. Experimental results with the verifier, in comparison to that of the MPASS verifier and a verification algorithm based on backward reachability analysis[2], show that the method achieved comparable or better results on most of the protocols tested.

A notable weakness of the verification method is that it struggles with the space and time complexity caused by an increased number of messages, but copes relatively well with an increase of local states in the processes, because all reachable channel evaluations are explicitly stored. This is a point that could be improved, by finding a method to abstractly represent a larger number of channel evaluations more compactly, possibly by introducing further overapproximation.

In addition to this, an important theoretical issue has been largely overlooked in this paper, i.e. the fact that the verifier is not guaranteed to ever converge towards a solution. Considering that the verifier terminates for all tested protocols, it is likely that the verification algorithm could leave such a guarantee for atleast some classes of problems, as is done in [1]. Identifying and formally proving such guarantee of termination would be a valuable theoretical addition worth investigation.

8 Related Work

Parameterized model checking focuses verification scenarious where the size of the problem under observation is a parameter of the system, and the size has no trivial upper bound. The goal is to verify the correctness of the system regardless of the value of the parameter. As this generally corresponds to a transition system of infinite size, research in this field of verification focuses on techniques to limit the size of the transition system.

On such method is the *invisible invariants* method[5], which similar to [1] and [22] use *cut-off* points to check invariants. Such cut-off points can be found dynamicly[19] or be constant[14]. The works of Namjoshi[22] show that methods based on process invariants or cut-offs are complete for safety properties.

Small model systems have been investigated in [19], which combines it with ininite state backward coverability analysis. Such methods prevent its use on undecidable reachability problems, such as those considered in [1]. The work of [1] also show that the problem is decidable for a large class of well quasi-ordered systems, including Petri Nets[1]. Practical results indicate that these methods may be decidable for yet larger classes of systems. More on well-structured infinite transition systems can be found in [15].

Abstract interpretation techniques were first described by Cousot and Cousot[10][11]. Similar work to [1] can be found in [18, 17].

Edsger Dijkstra was the first to consider the interaction between processes communicating over channels[12][6]. In [9], the authors show that verification of perfect channels systems is undecidable, by relating it to the simulation of a Turing Machine[9]. The decidability of lossy channels has been researched in[2, 8].

Similar to this thesis, [20] make use of abstract interpretation to channel systems, applying the technique to regular model checking. MPass[4] is a verification tool for channel systems, that restates the verification problem to an equivalent first-order logic, which is then solved by an SMT solver. Its input language is roughly the same as this thesis, although the tool is more general, being applicable also on *stuttering* and unordered channels.

References

[1] Abdulla, P., Haziza, F., and Holík, L. 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*. Springer Berlin Heidelberg, 2013, pp. 476–495.

[2] Abdulla, P., and Jonsson, B. Verifying programs with unreliable channels. In *Logic in Computer Science, 1993. LICS '93., Proceedings of Eighth Annual IEEE Symposium on* (Jun 1993), pp. 160–170.

[3] Abdulla, P. A. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic 16*, 4 (12 2010), 457–515.

[4] Abdulla, P. A., Atig, M. F., Cederberg, J., Modi, S., Rezine, O., and Saini, G. Mpass: An efficient tool for the analysis of message-passing programs.

[5] Arons, T., Pnueli, A., Ruah, S., Xu, Y., and Zuck, L. Parameterized verification with automatically computed inductive assertions? In *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 221–234.

[6] Baier, C., Katoen, J.-P., et al. *Principles of model checking*, vol. 26202649. MIT press Cambridge, 2008.

[7] Bartlett, K. A., Scantlebury, R. A., and Wilkinson, P. T. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM 12*, 5 (1969), 260–261.

[8] Bertrand, N., and Schnoebelen, P. Model checking lossy channels systems is probably decidable. In *Foundations of Software Science and Computation Structures*, A. Gordon, Ed., vol. 2620 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 120–135.

[9] Brand, D., and Zafiropulo, P. On communicating finite-state machines. *Journal of the ACM (JACM) 30*, 2 (1983), 323–342.

[10] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1977), ACM, pp. 238–252.

[11] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1979), ACM, pp. 269–282.

[12] Dijkstra, E. W. Information streams sharing a finite buffer. *Information Processing Letters 1*, 5 (1972), 179 – 180.

[13] Emerson, E., and Namjoshi, K. On model checking for non-deterministic infinite-state systems. In *Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on* (Jun 1998), pp. 70–80.

[14] Emerson, E. A., and Namjoshi, K. S. Reasoning about rings. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), ACM, pp. 85–94.

[15] Finkel, A., and Schnoebelen, P. Well-structured transition systems everywhere! *Theoretical Computer Science 256*, 1 (2001), 63–92.

[16] Fredlund, L.-Å., and Svensson, H. Mcerlang: a model checker for a distributed functional programming language. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 125–136.

[17] Ganty, P., Raskin, J.-F., and Van Begin, L. A complete abstract interpretation framework for coverability properties of wsts. In *Verification, Model Checking, and Abstract Interpretation* (2006), Springer, pp. 49–64.

[18] Geeraerts, G., Raskin, J.-F., and Van Begin, L. Expand, enlarge, and check. In *Proceedings of MOVEP06: MOdeling and VErifying parallel processes* (2006), p. 4.

[19] Kaiser, A., Kroening, D., and Wahl, T. Dynamic cutoff detection in parameterized concurrent programs. In *Computer Aided Verification* (2010), Springer, pp. 645–659.

[20] Le Gall, T., Jeannet, B., and Jéron, T. Verification of communication protocols using abstract interpretation of fifo queues. In *Algebraic Methodology and Software Technology*. Springer, 2006, pp. 204–219.

[21] McMillan, K. L. *Symbolic model checking*. Springer, 1993.

[22] Namjoshi, K. S. Symmetry and completeness in the analysis of parameterized systems. In *Verification, Model Checking, and Abstract Interpretation* (2007), Springer, pp. 299–313.

[23] Zuck, L., and Pnueli, A. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures 30*, 3–4 (2004), 139 – 169. Analysis and Verification.

A Automata for the Sliding Window Protocol