# Algorithmic Analysis of Channel Machines Using Small Models

Jonathan Sharyari

22nd October 2015

**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 . . . 7
  - 2.2.1 Syntax . . . 7
  - 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 Concretization 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 Proof of lemma 1 . . . 15
- 3.8 Verification Algorithm . . . 16

**4 Naive Implementation of the Verification Method** **17**
- 4.1 Definitions and Notations . . . 17
- 4.2 Naïve implementation . . . 17
  - 4.2.1 Abstract Post-image . . . 18
  - 4.2.2 Reachable configurations, $\mathcal{R}_k$ . . . 18
  - 4.2.3 Checking for bad configurations . . . 19
- 4.3 Improving the Algorithm . . . 20
  - 4.3.1 Simplified Subword Relation . . . 20

**5 Improved Implementation** **23**
- 5.1 Set Implementation . . . 23
- 5.2 Control-State Partitioning . . . 23
  - 5.2.1 Notation . . . 23
  - 5.2.2 Abstract Post-Image . . . 24
  - 5.2.3 Post-image . . . 24
- 5.3 Advantages of Partitioning . . . 24

**6 Final Implementation** **25**
- 6.1 Reachability Analysis . . . 26
- 6.2 Specification Language . . . 27
  - 6.2.1 Example . . . 28

**7 Result** **30**

7.1 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . 31

7.2 Analysis of Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**8 Summary and Further Work** **32**

**9 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 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[22] and although these methods have been applied to infinite models[14], 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[17]. Channels systems may also represent other types of programs, such as cache coherence protocols and sensor systems[25]. 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[3][9]. 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 [2], 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 [2], 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 [2] is that they show that this property holds for any *well-structured transition systems*[4] (i.e. transitions systems monotonic with respect to a well-quasi ordering).

Building upon the work in [2], we explore the possibility of adapting the verification algorithm proposed in [2], 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[11][12], 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 other hand, 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[5] verifier translates verification problems to satisfiability problems, solvable using third party SAT-solvers, and a *backward reachability analysis* as described in [3]. The backward analysis algorithm 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 of 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. In section **??**, 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 7, 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 $w = a_1 a_2 \ldots a_n$ is a finite sequence of letters from an alphabet $\Sigma$, where $a_i$ is in $\Sigma$ for all $i$ in $\{1 \ldots \text{n}\}$. We use $\epsilon$ 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 $w = a_1 a_2 \ldots a_n$, with $a_1$, $a_2$, $\ldots$, $a_n \in \Sigma$, is n). We use $|w|$ to denote the length of the word $w$. Observe that $|\epsilon| = 0$. We use $\Sigma^*$ (resp. $\Sigma^+$) to denote the set of all words (resp. non-empty words) over the alphabet $\Sigma$. Let $k$ be a natural number. We use $\Sigma_k^*$ (resp. $\Sigma_k^+$) to denote the set of all words (resp. non-empty words) of length at most $k$. 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 $\sigma : \Sigma^* \to \Sigma^*$ be a mapping between words, s.t. $\sigma(a_1 \ldots a_n) = \tau(a_1) \ldots \tau(a_n)$, where $\tau(a_i)$ is either $a_i$ or $\epsilon$, $0 \leq i \leq n$. We call $\sigma(w) = w'$ a *subword* of $w$, denoted $w' \sqsubseteq w$.

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

### 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 *fixpoint* 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 $D \subseteq A$, such that $D$ is a fixpoint of $F$). From the Knaster-Tarski theorem [24], 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 $F$; we refer to this as the *Kleen iteration.*

## 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*, each containing a word. The control part is a labeled finite-state transition system.

Formally, a lossy channel system $LCS$ is a tuple $(S, s^0, A, Ch, \Sigma, \delta)$, 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,
- $\Sigma$ is a finite alphabet,
- $\delta$ is a finite set of transitions, each of which is a triple of the form $(s, op, s')$, 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 $m \in M$
  - $a \in A$

  were $M$ denotes the set of finite words over $\Sigma$. We use the notation $s \xrightarrow{op} s'$ to describe $(s, op, s')$.

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

> $(s, ch!m, s')$ 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 $ch \in Ch$.
>
> $(s, ch?m, s')$ 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 $\xi : Ch \to \Sigma^*$ be a function that maps the content of each channel $ch \in Ch$ to a word $w$ over $\Sigma$, then we will call $w$ the *evaluation* of the channel $ch$, and we call $\xi$ a *channel evaluation.* We define a *configuration* to be a tuple $c = (s, \xi)$, such that $s \in S$ is a global control state and $\xi$ is a channel evaluation.

Let $c = (s, \xi)$ be a configuration. We use $state(c)$ and $eval(c)$ to denote the state and the evaluation in $c$, respectively and use the shorthand $c_S = state(c)$ and $c_E = eval(c)$ for a configuration $c$.

Let $\xi^0$ denote the empty channel evaluation such that for each channel ch $\in$ Ch, we $\xi^0(ch) = \epsilon$. We define $c^0$ to be the initial configuration $(s^0, \xi^0)$, 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 $c = (s, \xi)$ to be the length of the longest word in the channels, i.e., size(c)=$max\{|\xi(ch)|, ch \in Ch\}$.

Finally, we extend the subword relation to the configuration of the lossy channel system as follows: Let c=(s,$\xi$) and c'=(s',$\xi'$) be two configurations of the lossy channel system. We say that c is a sub-configuration of c' (denoted $c \sqsubseteq c'$) if and only if s=s' and for every channel ch $\in$ Ch, we have $\xi(ch)$ is a subword of $\xi'(ch)$.

## 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 infinite-state transition system $LTS = (C, \rightarrow)$ where $C$ is the set of configurations and $\rightarrow \subseteq (C \times C)$ is the smallest transition relation defined as follows:

- For each transition $(s, a, s') \in \delta$ and each channel evaluation $\xi$, $(s, \xi) \xrightarrow{a} (s', \xi)$. This means that the control state changes with the action $a$.
- For each transmission transition $(s, ch!m, s')$ in $\delta$ and channel evaluations $\xi$ and $\xi'$, $(s, \xi) \xrightarrow{ch!m} (s', \xi')$, $\xi'(ch) = \xi(ch) \bullet m$, and $\xi'(ch') = \xi(ch')$ for all $ch' \in Ch \setminus ch$. This means that the control state and the evaluation of the configuration change, so that the message $m$ is appended to the content of the channel $ch$.
- For each reception transition $(s, ch?m, s)$ in $\delta$ and channel evaluations $\xi$ and $\xi'$, $(s, \xi) \xrightarrow{ch!m} (s', \xi')$ such that $\xi(ch) = m \bullet \xi'(ch)$ and $\xi'(ch') = \xi(ch')$ for all $ch' \in Ch \setminus ch$. This means that the control state and the evaluation of the configuration change, so that the message $m$ is removed from the content of the channel $ch$.
- For each state $s \in S$ and channel evaluations $\xi$ and $\xi'$, $(s, \xi) \xrightarrow{*} (s, \xi')$ such that $\xi'(ch)$ is a subword of $\xi(ch)$ for all $ch \in Ch$. This means that any message can be lost at any time.

**Notations for configurations** Depending on the context, different notations to describe a configuration $c \in C$ may be used. For example, $c = (s, \xi)$ can be denoted as $c = (s, w_1, w_2, ..., w_n)$ when the set of channels is ordered (e.g., $Ch = \{ch_1, ch_2, \ldots, ch_n\}$).

We refer to the act of taking a transition $c' \xrightarrow{r} c$, as *firing* the transition $r = (s, \tau, s')$ from $c$ to $c'$ such that $state(c) = s$ and $state(c') = s'$. Alternatively, we may denote $c' \xrightarrow{r} c$ as $c = r(c)$.

## 2.4 Alternating Bit Protocol

Here we present a simple protocol, the *alternating bit protocol* [8]. This protocol will serve as a running example of the theoretical concepts in this and following sections.

The Alternating Bit Protocol (ABP)[1] 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 data 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 given below.

**Algorithm 1** ABP Sender

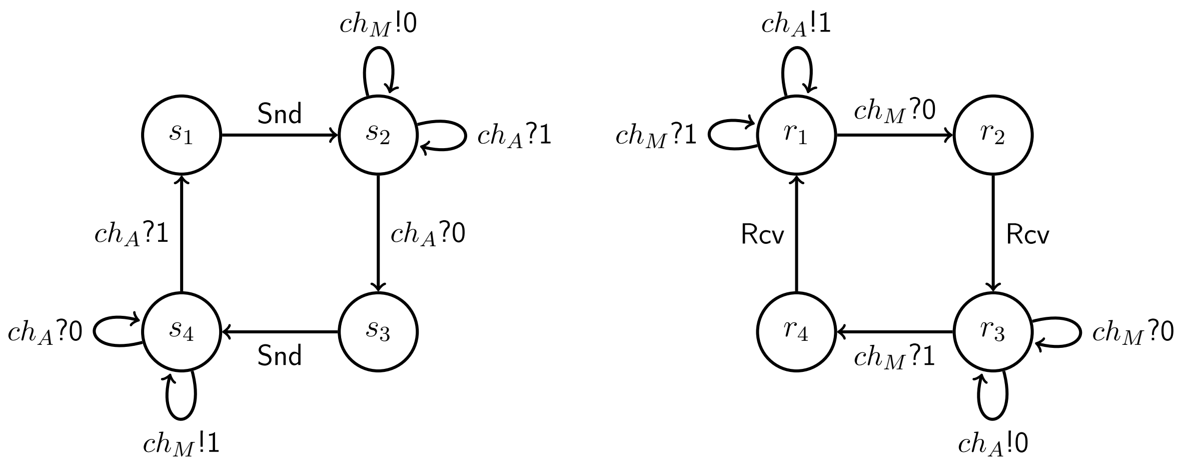
```
1: b := False                                          ▷ Initially False
2: for $True$ do
3:     msg.send(data, b);                              ▷ Send the message
4:     while ack.receive().b != b do    ▷ Wait until ack has the correct ID
5:         msg.send(data, b);                          ▷ 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 sequence number (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 as sequence number, until it receives an acknowledgement from the receiver with the same sequence number, at which point it alternates the value of $b$. The receiver behaves in the same manner, sending acknowledgements for the last received message until a new message is received.

**Syntax.** The alternating bit protocol can be described by a channel system $CS = (S, s^0, A, Ch, \Sigma, \delta)$ such that $S = \{(s_i, r_i)\}$ with $i \in \{1 \ldots 4\}$, $s^0$ is the initial state $(s_1,r_1)$, $A = \{Snd, Rcv\}$, $Ch = \{Ch_M, Ch_A\}$, $\Sigma = \{1,0\}$. The set $\delta$ of transitions is, for $1 \leq i \leq 4$ the set of transitions:



(a) Sender (b) Receiver

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

$$\begin{array}{cc}
(s_i, r_i) \xrightarrow{Snd} (s_2, r_i) & (s_i, r_1) \xrightarrow{ch_A!1} (s_i, r_1) \\
(s_2, r_i) \xrightarrow{ch_M!0} (s_2, r_i) & (s_i, r_1) \xrightarrow{ch_M?1} (s_i, r_1) \\
(s_2, r_i) \xrightarrow{ch_A?1} (s_2, r_i) & (s_i, r_1) \xrightarrow{ch_M?0} (s_i, r_2) \\
(s_2, r_i) \xrightarrow{ch_A?0} (s_3, r_i) & (s_i, r_2) \xrightarrow{Rcv} (s_i, r_3) \\
(s_3, r_i) \xrightarrow{Snd} (s_4, r_i) & (s_i, r_3) \xrightarrow{ch_A!1} (s_i, r_3) \\
(s_4, r_i) \xrightarrow{ch_M!1} (s_4, r_i) & (s_i, r_3) \xrightarrow{ch_M?0} (s_i, r_3) \\
(s_4, r_i) \xrightarrow{ch_A?1} (s_i, r_i) & (s_i, r_3) \xrightarrow{ch_M?1} (s_i, r_4) \\
(s_5, r_i) \xrightarrow{ch_A?0} (s_4, r_i) & (s_i, r_4) \xrightarrow{Rcv} (s_i, r_1)
\end{array}$$

**Notations** Let $c = (s, \xi)$ be a configuration of the alternating bit protocol, with the channels $ch_M$ and $ch_A$ containing the words 01 and 10 respectively. If the channel $ch_M$ is ordered before $ch_A$ in $Ch$, then $c$ may also be denoted as $c = (s, 01, 10)$. There are finitely many control states in this system (precisely 16), but an infinite set of channel evaluations. The initial configuration $c^0 = ((s_1, r_1), \xi^0)$.

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 $(s_2, r_1, 01, 10)$, $ch_M!1$, $(s_2, r_1, 011, 10)\rangle$.

### 2.4.1 The Verification Problem

A channel system can potentially reach a *bad* state, i.e. a state representing unintended behaviour. We call a configuration $(s, \xi)$ such that $s$ is a bad state a *bad configuration*, regardless of the channel evaluation.

Suppose *Bad* is a set of bad configurations of a lossy transition system *LTS*.

We call a sequence $t = c_0 c_1 \ldots c_n$ of configurations a *trace*, if $c_i \rightarrow c_{i+1}, 0 \leq i \leq (n-1)$ and we say that $c_n$ is reachable from $c_0$. We call a trace a *bad trace* if there exists $c^{bad} \in Bad$ such that $c^{bad}$ is a sub-configuration of $c_n$. 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 $c^0$ such that $c$ is reachable. 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 [2]. The technique is based on the use of *abstract interpretation* techniques, first formalized by Cousot and Cousot[11]. Abstract interpretation techniques are techniques for 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 may result in an infinite set of configurations. In [2] 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 a *view* $v = (s, \xi)$ to be a minimal representation of a set of configurations $C_v$, such that for any $c \in C_v$, $v \sqsubseteq c$. Note that a view is itself a configuration, as they have the same representation, and we use the same terminology and notation for them, for example $size(v)$ to denote the size. The difference is that a configuration is a single entity, whereas the view is an abstract representation of a larger set of configurations.

**Example.** Let the view $v = ((s_1, r_1), ab, cd)$, then the set $C_v$ is an infinite set, s.t. $((s_1, r_1), ab, cd) \in C_v$, $((s_1, r_1), abab, cdcd) \in C_v$ but $((s_1, r_1), ab, epsilon) \notin C_v$

## 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 the set of all configurations and views respectively of size up to $k$.

The abstraction function $\alpha_k : C \rightarrow 2^{V_k}$ maps a configuration $c$ into a set $V'$ of views of size up to $k$, such that for each $v \in V'$, $v \sqsubseteq c$ and $size(v) \leq k$.

**Example.** Suppose $c$ is a configuration $(s, ab, de)$. The configuration is of size 2 and $\alpha_2(c)$ is the set

| | | | |
|---|---|---|---|
| $(s, ab, de)$ | $(s, ab, d)$ | $(s, ab, e)$ | $(s, ab, \epsilon)$ |
| $(s, a, de)$ | $(s, a, d)$ | $(s, a, e)$ | $(s, a, \epsilon)$ |
| $(s, b, de)$ | $(s, b, d)$ | $(s, b, e)$ | $(s, b, \epsilon)$ |
| $(s, \epsilon, de)$ | $(s, \epsilon, d)$ | $(s, \epsilon, e)$ | $(s, \epsilon, \epsilon)$ |

## 3.3 Concretization function

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

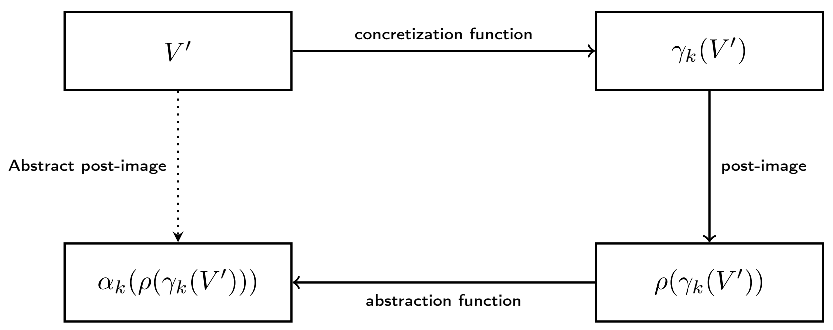


Figure 2: The application order of the $\alpha$, $\rho$ and $\gamma$

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 $V'$.

## 3.4 Post-Image

For a configuration $c \in C$ of a lossy transition system $LTS = (C, \rightarrow)$, we define the *post-image* of $c$ denoted $\rho(c) = \{c' \mid c \rightarrow c'\}$. 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 $\rho(V') = \{c' | c \rightarrow c', c \in V'\}$. 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 $\gamma_k$, $\rho$ and $\alpha_k$ in order to the set $V'$. The procedure is illustrated in figure 2.

Note that since $V' \subseteq Apost_k(V')$, $Apost_k$ is a monotonic function, thus by the Knaster-Tarski theorem, the function $Apost_k$ has a fixpoint.

## 3.6 Reachability Analysis

Let LTS be a lossy transition system, with an initial configuration $c^0$. Then the set of reachable configurations $\mathcal{R}$ of LTS can be computed inductively as follows: $A_0 = \{c^0\}, A_{i+1} = A_i \cup \rho(A_i)$. The finite set of configurations $\mathcal{R}_k$ with size at most $k$ can be similarly computed; $A_0 = \{c^0\}$, $A_{i+1} = A_i \cup (\rho(A_i) \cap C_k)$.

## 3.7 Small Models

Calculating the abstract post-image of a set of views $V' \subseteq V_k$ is essential for the verification procedure. As $\gamma_k(V')$ typically is infinite, this cannot be done straightforwardly. The main result of [2] that it suffices to consider configurations of $\gamma_k(V')$ of sizes up to $k+1$ (i.e. $\gamma_k^{k+1}$), 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 $X \subseteq V_k$, $\alpha_k(\rho(\gamma_k(X))) \cup X = \alpha_k(\rho(\gamma_k^{k+1}(X))) \cup X$.*

### 3.7.1 Proof of lemma 1

We want to show that the set $\gamma_k^{k+1}(X)$ of views of size at most $k+1$ is an abstract representation of the full set $\gamma_k(X)$. In order to do this, we want to show that for any configuration $c \in \rho(\gamma_k(X))$ of size $m > k+1$ such that there is a transition $c' \xrightarrow{r} c$, then for each view $v' \in \alpha_k(c)$, the following holds: There is a configuration $d' \in \gamma_k^{k+1}(X)$ of size at most $k+1$ and transition $d' \xrightarrow{r} d$ s.t. $v \in \alpha_k(d)$.

Note that by design (see section 2.3), transitions only change a single evaluation, therefore, although the set of evaluations may contain multiple evaluation, we direct our focus to a single evaluation in this section.

**Transmissions** Consider a configuration $c' = (s', x) \in \gamma_k(X)$ and a transition $r : c' \xrightarrow{ch!m} c = (s, x \bullet m)$. Any view $v = (s', y)$ of size at most $k$ of $c$ is on one of the following forms:

1. $y \sqsubseteq x$, $|y| \leq k$. In this case, the set $\gamma_k^{k+1}(X)$ includes all the configurations $d' = (s', y)$, as $d'_E \sqsubseteq c'_E$. Then $d' \xrightarrow{ch!m} d$ yields $d = (s, y \bullet m)$, for which $(s, y) = v$ is a view.

2. $y = z \bullet m$ with $z \sqsubseteq x$, $|z| \leq (k-1)$. The set $\gamma_k^{k+1}(X)$ includes all the configurations $d' = (s', z)$, as $d'_E \leq c'_E$. Then $d' \xrightarrow{ch!m} d$ yields $d = (s, z \bullet m)$ which is of size at most $k$, meaning it is also a view, $d = v$.

**Actions** Consider a configuration $c' = (s', x) \in \gamma_k(X)$ and a transition $r : c' \xrightarrow{a} c = (s, x)$. Since $r$ is an action, it can be fired regardless of the channel evaluation. Therefore, if $r$ can be fired from the configuration $c'$, it may also be fired from any view $v' = (s', y)$ of size at most $k$ of $c'$, resulting in a view $v = (s, y)$.

Let $\{(s', w_1), (s', w_2), \ldots, (s', w_n)\}$ be the views of $c'$, then the transition $r$ can be fired from each of these, resulting in a set $\{(s, w_1), (s, w_2), \ldots (s, w_n)\}$, which is the complete set of views of size at most $k$ of $c$, showing that $c$ is abstractly represented in $\gamma_k(X)$.

**Receptions and Message Loss** Note that receptions transitions can be seen as a special case of a message loss, where messages are lost in FIFO order, i.e. a proof of lemma 1 for message loss directly applies to the reception transitions. Consider a configuration $c' = (s', x) \in \gamma_k(X)$ and a transition

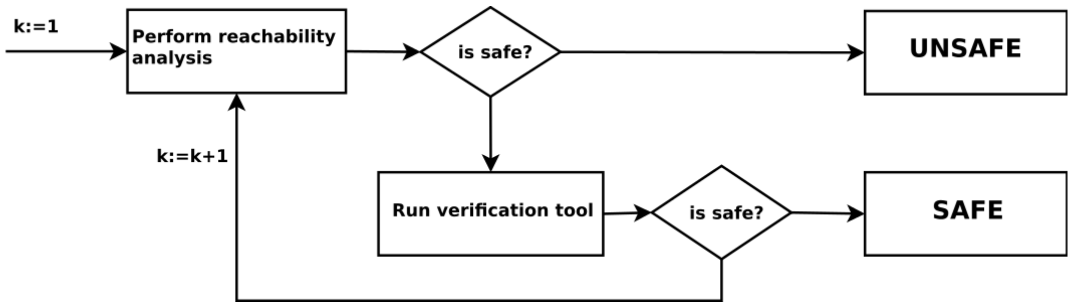


Figure 3: The general flow of the verifier.

$r : c' \xrightarrow{*} c = (s', y)$. Since $y \sqsubseteq x$, the set of views $V$ of $c$ is a subset of the set of views $V'$ of $c'$.

## 3.8 Verification Algorithm

You suggested I change from $V'$ to $V_k$ in the algorithm. I assume this applies to this entire subsection, and changed it accordingly

Suppose *Bad* is a set of bad configurations, such that $|b| = 0$ for all $b \in Bad$. As a result of lemma 1, if $\gamma_k^{k+1}(V_k) \cap Bad = \emptyset$ for any $k \geq 1$, then $\gamma_l^{l+1}(V_k) \cap Bad = \emptyset$ for any $l \geq k$. Given a set of bad configurations *Bad*, a set of initial configurations $I$ and a set of transitions, verification of a system can be done with algorithm 3, which was first presented in [2].

**Algorithm 3** General Verification algorithm

```
1: for $k := 1$ to $\infty$
2:   if $\mathcal{R}_k \cap Bad \neq \emptyset$ then return Unsafe
3:   $V_k := \mu X.\alpha_k(I) \cup Apost_k(X)$
4:   if $\gamma_k(V_k) \cap Bad = \emptyset$ then return Safe
```

The algorithm begins by performing a reachability analysis, as explained in 3.6 in order to compute $\mathcal{R}_k$ and check for bad configurations. For any buffer size $k$, if a bad configuration 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 a set of configurations $V'$, for which the set of concretizations $\gamma_k^{k+1}(V_k)$ is an *overapproximation* of reachable configurations of size $k$, reachable through configurations of size at most $k+1$ and checking for bad configurations. This is done by computing the fixpoint of $Apost_k$. If at this point no bad configuration has been found, the system can be said to be safe and the algorithm terminates. If on the other hand a bad state was found, the system is not necessarily unsafe, as $V_k$ is an overapproximation of the reachable states in $\mathcal{R}_k$. The process is then repeated with a buffer size of $k+1$. This process is abstractly described in the flowchart in figure 3.

# 4 Naive Implementation of the Verification Method

In this chapter, we show how the algorithm in 3.8 could be implemented. The implementation is naïve, it strictly follows the mathematical concepts. Then we identify the main drawbacks of this naïve implementation and suggest some improvements.

## 4.1 Definitions and Notations

You commented that the change to using $X$ rather than $V$ as in the last chapter is confusing. In my opinion, this distinction makes it more clear that $X$ denotes a programming artifact. If you think it is better, I can change to $X$ in the previous chapter as well (if not, I will atleast change it so that I don't mix them in the last chapter as it is now)

In this chapter, we introduce the term *potential concretizations*. The reason for the introduction of this term resides in the fact that no easy direct way has been found to compute the set of concretizations corresponding to a set of views, i.e. from a set $X_k$ of configurations of size at most $k$ finding $\gamma(X_k)$ of concretizations of size at most $k+1$. Instead, in order to find the concretizations, we generate all of the *potential* concretizations, such that a potential configuration is any configuration $con$, $size(v) < size(con) \leq k+1$ for some $v \in X_k$, i.e. any extension of a view already in the set with any symbol in the alphabet.

In order to compute the fixpoint $V_k$ in algorithm 3.8 we use the kleen iteration where $X_k^0$ is the set of initial configurations $I$ and we use $X_k^j$ to denote the value of the variable $X$ in the $j_th$ Kleen iteration. We assume that when the fixpoint converges, we have that $X_k^m = V_k$.

A view $v \in X_k$ may be extended with a symbol on one or more of its channels, yielding the potential concretization $con$. We say that $con$ is *accepted*, if all the views of $con$ of size up top $k$ are in $X_k$, otherwise, $con$ is *refuted*.

As an example, let $(s, a, b) \in X_k^i$, then $(s, a, bb)$ is a potential configuration. We say that $con$ is *accepted*, if $\alpha_k(con) \subseteq X_k^i$, otherwise, $con$ is *refuted*. Note that if $\alpha_k(con) \nsubseteq X_k^i$, there may be $X_k^j$, $i < j$ such that $\alpha_k(con) \subseteq X_k^j$, i.e. a potential configuration may be refuted in one iteration, but accepted in another.

We say that an accepted concretization $con$ is *reached* from the view $v$.

## 4.2 Naïve implementation

A naïve way of implementing algorithm 3.8 would be to implement it in a way that corresponds exactly to the mathematical notations. This is possible to do, as most programming languages have built-in data structures that support set operations. We maintain a *set* $X_k = \{v | size(v) \leq k\}$ of views such that all views are uniquely stored in $X_k$, and we will assume that an alphabet $\Sigma$, the set *Bad* of bad states and a set of transitions $\delta$ are known. The the implementation of the algorithm performs three tasks:

1. Compute the set $\mathcal{R}_k$ of reachable configurations of size up to k, used on line 2 of algorithm 3.8. In section 4.2.2 we show how to compute the set $\mathcal{R}_k$.

2. Compute the set $V_k$ of configurations, i.e. the fixpoint of $X_k$. We know that such a fixpoint exists, due to the Knaster-Tarski theorem[24], and we compute it by iteratively applying the Abstract post-image of $X_k$. This is used on line 3 of algorithm 3.8. The procedure is explained in section 4.2.1.

3. We need to be able to compute the intersection of a set of configurations with the set *Bad* of bad configurations. This is done on line 2 and 4 of algorithm 3.8. The procedure is explained below in section 4.2.3.

### 4.2.1 Abstract Post-image

In order to compute the abstract post-image *Apost* of a set $X_k^i$, the concretization function, the post-image and the abstraction functions below are applied in order on $X_k^i$:

1. **Concretization function**: Generate the set $\gamma_k^{k+1}(X_k^i)$. For each view $v \in X_k^i$, generate all potential concretizations *con*. Then $con \in \gamma_k^{k+1}(X_k)$ if $\alpha_k(con) \subseteq X_k^i$.

   This results in the set of accepted potential concretizations.

2. **Post-image**: Generate the set $\rho(\gamma_k^{k+1}(X_k^i))$: For every concretization *con* $\in \gamma_k^{k+1}(X_k^i)$, $r \in \delta$, compute $r(con) \in post(\gamma_k^{k+1}(X_k^i))$, i.e. the post-image of the concretization *con*.

3. **Abstraction Function**: Generate $Apost(X_k^i)$: For each $c \in \rho(\gamma_k^{k+1}(X_k^i))$, compute $\alpha_k(c)$. Let $V' = \{\alpha_k(c) | c \in \rho(\gamma_k^{k+1}(X_k^i))\}$ of views of size at most $k$. Then $X_k^{i+1} = X_k^i \cup V'$. If $X_k^i = X_k^{i+1}$, a fixpoint has been reached.

### 4.2.2 Reachable configurations, $\mathcal{R}_k$

On line 2 of algorithm 3.8, the set of all reachable configurations of size up to $k$, $\mathcal{R}_k$, is required. Computing the set $\mathcal{R}_k$ can be done in a multitude of ways. A simple way is to iteratively calculate the post-image of each configuration in the set, while taking the buffer size into account.

Let $\rho_k(X)$ be defined such that if given the set of transitions $\delta$, for a configuration $c \in X$ and each transition $r \in \delta$, $r(c) \in \rho_k(X)$ if $size(r(c)) \leq k$. Otherwise, if $size(r(c)) = l > k$, there is a buffer overflow. We assume that such an overflow is handled by removing the last $l - k$ symbols of the word, i.e., if $c = (s, w_1, w_2, ..., w_i, ..., w_m)$ with $w_i = a_1a_2...a_ka_{k+1}...a_l$ is reached, the configuration $c' = (s, w_1, w_2, ..., w_i', ..., w_n)$ with $w_i' = a_1a_2...a_k$ is a member of $\rho_k(X)$.

Let $R_0$ denote the set containing the initial configuration $(s^0, \xi^0)$, and let $R_i = R_{i-1} \cup \rho_k(R_{i-1})$. Then $R_m = \mathcal{R}_k$ for some $m$.

### 4.2.3 Checking for bad configurations

A bad configuration is any configuration which is in a *state* considered to be bad, independent of the channel evaluation. Thus, the set *Bad* can be expressed as a minimal state of configurations, such that a bad configuration $b \in Bad$ is of the format $b = (s, \xi^0)$. Checking for bad configurations can be done for each configuration $c = (s, \xi)$ check if $c' = (s, \xi^0) \in Bad$.

Note to self: this paragraph would need an explanation on how to find a bad trace leading to the bad configuration. I've written about that in chapter 6 already, and I suggest I leave it there for now.

**Pseudo-code** The pseudo-code implementing algorithm 3.8 can be seen in listing 4.

**Algorithm 4** Pseudo-code for algorithm 3.8.

```
Verifier (V, Rules, Bad):
for $True$ do
    % Set initial values
    $X_k := \{(s_0, \xi^0)\}$
    V' := $\emptyset$
    $R_0 := \{(s_0, \xi^0)\}$
    R' := $\emptyset$
    $i$ :=0

    % Calculation of $R_k$
    do
        $R' := post(R_i)$
        $R_{i+1} := R_i \cup R'$
        $i := i + 1$
    while $(R_i \neq R_{i-1} \cup R')$
    if $\mathcal{R}_i \cap Bad \neq \emptyset$ then
        return Unsafe
    end if

    % Fix-point calculation
    do
        $X_k := X_k \cup V'$
        $V' := \alpha_k(post(\gamma_k^{k+1}(X_k)))$
        while $(X_k \neq X_k \cup V')$
    if $X_k \cap Bad = \emptyset$ then
        return Safe
    end if
    k := k+1
end for
```

## 4.3 Improving the Algorithm

Evaluating the procedure described above, we find that the bottle-neck lies within the concretization function: In order to compute $\gamma(X_k)$, all potential concretizations of the set $X_k$ of size up to $k+1$ are computed by creating all extensions of the views $v \in X_k$, i.e. all potential configurations *con* created from $v$ such that at least one word of the channel evaluation is of a larger size than in $v$. For each of these potential concretizations, $\alpha_k(con)$ is created in order to determine whether the concretization should be accepted or refuted.

Although this method is correct, there is significant overlap of concretizations being considered as it is often possible to create the same concretization *con* in several ways. For example, if $X_k$ contains the views $v_1 = (s, \epsilon, ab)$ and $v_2 = (s, ab, \epsilon)$, then the concretization $(s, ab, ab)$ can, and will, be reached from both $v_1$ and $v_2$, by adding the symbols *ab* to the first and second channel of $v_1$ and $v_2$ respectively.

A rough worst-case estimate of the number of potential concretizations created in each iteration in the above algorithm can be given. Let $s$ denote the size of the alphabet, $t$ the number of channels and $n = size(X_k^i)$. A potential concretization is created for each combination of symbols with which a word of the channel evaluation can be extended. An upper bound for this is $s^k$ combinations (the number of words of size $k$ that can be created from an alphabet of $s$ symbols). Also, each combination of such extensions on the $t$ different channels is created, resulting in $(s^k)^t$ potential configurations for a single view. As this must be done for each of the $n$ existing views, the result is an upper bound of $O(n * (s^k)^t)$ potential concretizations that may be considered in a single iteration.

This leads to the conclusion that decreasing the number of potential concretizations considered may result in a faster algorithm. Below we show that this can be achieved by redefining the subword relation, originally defined in section 2.1.1. Using the new definition of the subword relation, algorithm 3.8 will still reach the fixpoint $V_k$, i.e. reachable concretizations will be found.

### 4.3.1 Simplified Subword Relation

**Definition.** We redefine the subword relation in section 2.1.1, such that a word $w' \models w = a_1 a_2 \ldots a_n$ if $w' = a_i a_{i+1} \ldots a_j$ for some $0 \leq i \leq n$, $0 \leq j \leq n$, where $\models$ denotes the new subword relation.

**Example.** Let *w*=*abc*. Then the set of subwords of $w$ is *abc, ab, bc, a, b, c,* $\varepsilon$. Note that if $w' \models w \implies w' \sqsubseteq w$, but $w' \sqsubseteq w \not\Longrightarrow w' \models w$.

**Lemma 2.** *Let the set $Y$ denote a fixpoint of algorithm 3.8 reached using the definition of the subword relation as defined in section 2.1.1, and let the set $X$ denote a fixpoint of the algorithm using the above definition of the subword relation, then $X = Y$.*

*Proof.* Consider $v$ be a view with the word $w = a_i a_{i+1} \ldots a_m$ on one of its channels, and let $v \in Y_k^p$ for some $p \leq 0$ and $v \in X_k^i$ for some $i \leq 0$.

Let the view $v'$ be a view with the word $w' = a_i a_{i+1} \ldots a_j a_l a_{l+1} \ldots a_m$ on the same channel. Then $v' \in Y_k^{p+1}$ as $w' \sqsubseteq w$, but $v' \notin X_k^{i+1}$ since $w' \not\models w$. We will show that there is a set $X_k^l$ such that $v' \in X_k^l$.

Note that there are views $v'_1, v'_2 \ldots v'_m$ with words $w'_1, w'_2 \ldots w'_m$ on their channels such that

$$\begin{aligned}
w'_1 &= a_i a_{i+1} \ldots a_j \\
w'_2 &= a_i a_{i+1} \ldots a_j a_l \\
&\ldots \\
w'_m &= a_i a_{i+1} \ldots a_j a_l a_{l+1} \ldots a_{m-1} = w'.
\end{aligned}$$

We know that $w'_1 \models w'$, thus $v'_1 \in X_k^{i+1}$, but $v'_x \notin X_k^i$ for $x > 1$. As $w'_y \models w'_{y+1}$ for each $1 \leq y \leq m-1$, $v'_2 \in X_k^{i+2}, v'_3 \in X_k^{j+3} \ldots v'_m \in X_k^{i+m} = X_k^l$. Thus the verification algorithm using the simplified subword relation is guaranteed to find each of the views of $v$ within $m$ iterations.

□

**Reducing the number of concretizations.** We can show that when creating the potential concretizations of a view $v$, we need not generate the full set of potential concretizations and instead limit the algorithm to generate all those potential concretizations *con* that can be created from $v$ by the extension of a single channel with a single symbol.

Consider a potential concretization $con = (s, w'_1, w'_2, \ldots, w'_n)$ created from the view $v = (s, w_1, w_2, \ldots, w_n)$ such that $w'_i \models w_i$ for each $i \leq n$. If the generation of potential concretizations would be limited such that only one word $w_i$ would be extended in each iteration, the concretization *con* would still eventually be reached, as the views $v \models (s, w'_1, w_2, w_3 \ldots, w_n) \models (s, w'_1, w'_2, w_3, \ldots, w_n) \models \ldots \models (s, w'_1, w'_2, \ldots, w'_{n-1}, w_n) \models con$.

Using this approach, the number of potential concretizations considered each iteration is reduced significantly. Using the same notation as above, we let $s$ denote the size of the alphabet, $t$ the number of channels and $n = size(X_k^i)$ in a certain iteration $i$, the number of potential concretizations is reduced such that each of the $n$ views is extended with at most one symbol on each of its channels, resulting in a upper bound of $O(n * s * t)$ potential concretizations. Although the number of potential configurations generated each iteration is reduced, the number of iterations required in order to create the potential configurations increases, possibly by an exponential factor.

**Reducing the number of views.** When generating potential configurations as explained above, the task of accepting or refuting the concretizations can be made more effective as well.

Suppose we want to determine whether a potential concretization *con* reached from $v$ should be accepted or refuted. We know that *con* has been genereated from $v$ by adding a single symbol $m$ on one of the channels of $v$, whereas the state of *con* and $v$ are the same, as is the content of the other channels.

**Lemma 3.** *Let $v' = (s, w'_1, \ldots, w'_n)$ be the concretization created by extending* *the view $v = (s, w_1, \ldots, w_n) \in X_k$ by adding a single message to a single channel such that $w'_p = w_p \bullet m$ for some $p \leq n$ and $w'_i = w_i$ for all $i \neq p$. Then, if $w'_p = m_1 m_2 \ldots m$, it suffices to check whether there is a view $v'' = (s, w''_1, \ldots, w''_n) \in X_k$ such that $w''_p = m_2 \ldots m$, in order to determine if con can be accepted or not.*

*Proof.* By definition, $v'$ is accepted if $\alpha_k(v') \subseteq X_k$. Although $|v'| = k + 1$, a view $v'_x \in \alpha_k(v')$ is at most of length $k$ and contains a word $w_{x_p}$ on its $p:th$ channel. If $w_{x_p}$ does not contain the added symbol $m$, $v_x \models v$ from which it follows that $v_x \in X_k$. If $w_{x_p}$ does contain the message $m$, $v_x \models v''$, thus $v_x \in X_k$ if $v'' \in X_k$. Therefore, it follows from $v'' \in X_k$ that $\alpha_k(v') \subseteq X_k$.

$\square$

**Example.** Suppose we want to determine whether the potential concretization $con = (s, abc, de)$ created from the view $v = (s, ab, de) \in X_k$. It is then sufficient to check that $v' = (s, bc, de)$ is an element of $X_k$, as any other view is either a subword of $v$ and is already in $X_k$, or a subword of $v'$, and therefore in $X_k$ if $v' \in X_k$.

# 5 Improved Implementation

In the last chapter, it was explained how redefining the subword relation could effectivize the calculation of the set $V_k$ of reachable configurations, leading to a lower computational complexity in each iteration for the cost of an increased number of iterations. In this chapter, we will show how storing the set $V_k$ and the set $\delta$ of transitions partitioned into disjunct sets of configurations and transitions respectively, we can further improve the implementation of algorithm 3.8.

## 5.1 Set Implementation

In chapter 4, the set $V_k$ of reachable configurations was computed, by iteratively calculating the abstract post-image of a set $X_k^i$. The abstraction function, and consequently , the concretization functions heavily depend on the use of standard set operations; insert, union and membership checks. The time to perform these operations is dependent on the number of elements in the set $X_k^i$.

Consider a configuration $c \in X_k^i \subseteq V_k$. By definition the views of $c = (s, \xi)$ are of type $(s, \xi')$, that is, the configuration and its views have the same control-state $s$. The time required for inserting the views of $c$ to the set $X_k$, that is, performing the set operation $\alpha_k(c) \cup X$ or conversely, checking whether $\alpha_k(c) \subseteq X_k^i$ (in order to accept or refute $c$), depends on the size of the full set $X_k$.

Due to the fact that the views of a configuration necessarily have the same control-state as the configuration itself, the time to perform these basic set operations can be greatly reduced, by partitioning the set $X_k$ into several disjoint sets, such that each set corresponds to a unique control-state.

## 5.2 Control-State Partitioning

### 5.2.1 Notation

Let $S$ denote a finite set of control states and $W_s$ denote a finite set of configurations, all of which have the control state $s$. We define a *hashmap* $H_k$, indexed by control states $s \in S$ and storing sets of configurations $W_s \subseteq V_k$. Then $H_k$ can be viewed as a functions $H_k : s \in S \rightarrow W_s \subseteq V_k$, i.e., given a control state $s$, the hashmap returns the set of configurations with the control state $s$.

In analogy with $X_k$ in the previous chapter, let $H_k^i$ denote the state of the hashmap $H$ in interation $i$.

Similarly, let $S$ denote a finite set of control states and $\delta_s$ denote a finite set of transitions, such that any $r \in \delta_s$ is of the form $r = (s, op, s')$. We define a hashmap $D$ indexed by control-state $s \in S$ and storing sets of transitions $\delta_s \subseteq \delta$. Then $D$ can be viewed as a function $D : s \in S \rightarrow \delta_s \subseteq \delta$, i.e., given a control state, the hashmap return the set of all transitions applicable on configurations with the control-state $s$.

### 5.2.2 Abstract Post-Image

The goal is to modify the implementation of the abstraction, post-image and concretization functions in section 4.2, using the hashmaps $H_k$ and $D$ as a replacement for the set $X_k$ of configuratios and $\delta$ of transitions respectively. This is done in such a manner, that by the end of and iteration $i$ $\bigcup\limits_{s \in S} H_k^i(s) = X_k^i$.

We define the concretization function, post-image and abstraction function defined in section 2, as follows:

1. **Concretization function**: Generate the set $\gamma_k^{k+1}(H_k^i)$. For each state $s \in S$, create all potential concretizations $con$ reachable from $v \in H_k^i(s)$. Then $con \in \gamma_k^{k+1}(H_k^i(s))$ if $\alpha_k(con) \subseteq H_k^i(s)$.

2. **Post-image**: Generate the set $\rho(\gamma_k^{k+1}(H_k^i))$: For every concretization $con \in \gamma_k^{k+1}(H_k^i)$, $r \in D(s)$, $r(con) \in post(\gamma_k^{k+1}(H_k^i))$, i.e. the post-image of the concretization $con$.

3. **Abstraction Function**: Generate $Apost(H_k^i)$: Let $H'$ be an empty hashmap. For each $c \in \rho(\gamma_k^{k+1}(H_k^i))$, $state(c) = s$, add $\alpha_k(c)$ to $H'(s)$. Then $H_k^{i+1} = H_k^i \cup H'$. If $H_k^i = H_k^{i+1}$, a fixpoint has been reached.

### 5.2.3 Post-image

**Example** The picture is not up-to-date any more, but it shows the general idea

Figure 4 illustrates the calculation of the post-image using the partitioned data-types. In order to clarify the procedure, the hashmaps are illustrated in the form of trees such that each leaf corresponds to a set $W_s$ of channel evaluations or $\delta_s$ of transitions, and the path from the root node to the leaf corresponds to the control-state.

## 5.3 Advantages of Partitioning

Having partitioned the set $X_k$ of configurations into a hashmap $H_k$ of sets, all set operations are made more effective. In a best-case scenario, there would be approximately the same number of configurations for every state in the system. A lower bound for the complexity of checking whether $\alpha_k(c) \in V$ is reduced to $O(log(n/|s|))$ where $s$ is the set of control-states in the system, compared to $O(log(n))$ using $X_k$. The complexity of inserting an element is naturally the same.

Additionally, the complexity of calculating the set of post-images of a set $\gamma_k^{k+1}(H_k)$ is reduced compared to $\gamma_k^{k+1}(H_k)$, as we need not perform the calculation of each $\rho = (s, op, s') \in \delta$ on each $c \in \gamma_k^{k+1}(H_k)$, but only on the subset $H_k(s)$ of transitions, which can be found in constant time.

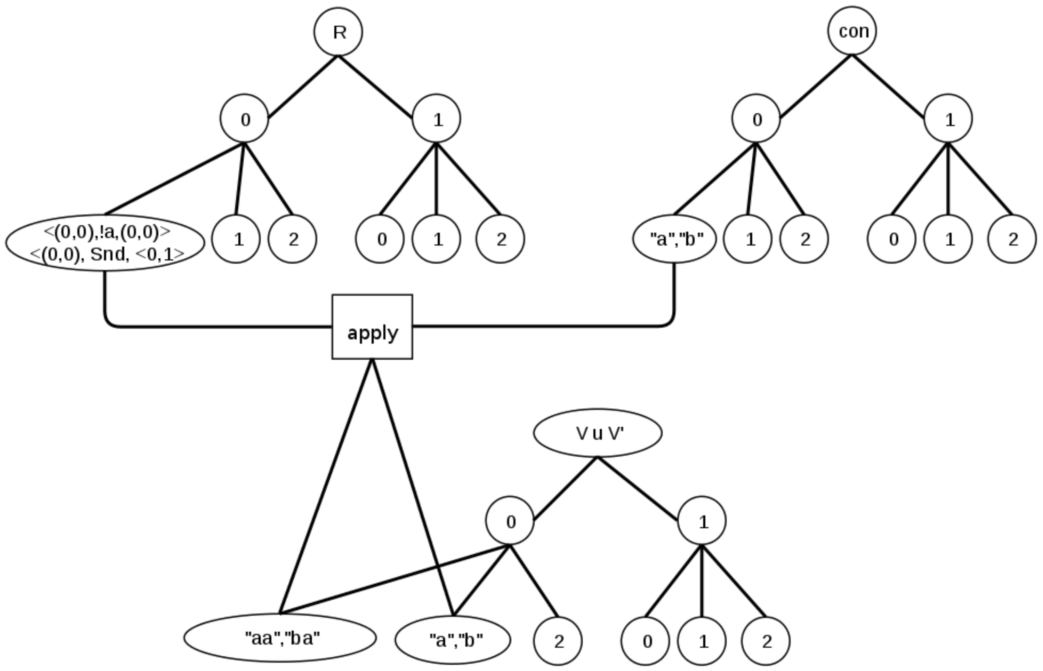


Figure 4: This picture shows the process of applying rules. For each control-state, each rules with that state are applied on each configuration with the same state. This will generate new configurations, not necessarily with the same control-state. For clarity. tree structure is used to illustrate a hashmap.

## 6 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 $\alpha$-calculation, we first check if the newly created

**Algorithm 5** The verification algorithm from section 3 in somewhat higher detail. This version includes

```
Gamma (V, Seen):
    con' := concretizations (V)
    con := c | c $\in$ con $\wedge$ c $\notin$ Seen

Step (Con, Rules):
for state $\in$ nodes(V) do
        S := r(c) | $\forall$ c $\in$ con(state) $\wedge$ $\forall$ r $\in$ Rules(state)
end for

Alpha (V, S):
    V:= V $\cup$ views(C)

Verifier (V, Rules, Bad):
for $True$ do
    if $\mathcal{R}_k \cap Bad \neq \emptyset$ then
        return Unsafe
    end if
    V := $\mu Alpha(Step(Gamma(V)))$
    if $\gamma_k(V) \cap Bad = \emptyset$ then
        return Safe
    end if
    k := k+1
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:

## 6.1 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.

## 6.2 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 [5]. 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.

### 6.2.1 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, 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.

```
<protocol name="Alternating␣Bit␣Protocol" medium="FIFO>
<messages>
␣␣␣␣␣␣<message>ack0</message>
␣␣␣␣␣␣<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>
        <first_role>RECEIVER</first_role>
        <second_role>OBSERVER</second_role>
        <action>Rcv</action>
</synchronize>
```

# 7 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 ??. 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 purposly faulty version of the ABP. |
| SW3 | The sliding window protocol, with k=3. |
| ABP_F | A purposly 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 purposly faulty version of the BRP. |

The `Backward` verifier uses *backward reachability* for verification, and follows the algorithm described in [3]. 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 6.2) 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.

## 7.1 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.

## 7.2 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.

## 8 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[2], 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[5]. Experimental results with the verifier, in comparison to that of the MPASS verifier and a verification algorithm based on backward reachability analysis[3], 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 [2]. Identifying and formally proving such guarantee of termination would be a valuable theoretical addition worth investigation.

# 9 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[6], which similar to [2] and [23] use *cut-off* points to check invariants. Such cut-off points can be found dynamicly[20] or be constant[15]. The works of Namjoshi[23] show that methods based on process invariants or cut-offs are complete for safety properties.

Small model systems have been investigated in [20], which combines it with ininite state backward coverability analysis. Such methods prevent its use on undecidable reachability problems, such as those considered in [2]. The work of [2] also show that the problem is decidable for a large class of well quasi-ordered systems, including Petri Nets[2]. 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 [16].

Abstract interpretation techniques were first described by Cousot and Cousot[11][12]. Similar work to [2] can be found in [19, 18].

Edsger Dijkstra was the first to consider the interaction between processes communicating over channels[13][7]. In [10], the authors show that verification of perfect channels systems is undecidable, by relating it to the simulation of a Turing Machine[10]. The decidability of lossy channels has been researched in[3, 9].

Similar to this thesis, [21] make use of abstract interpretation to channel systems, applying the technique to regular model checking. MPass[5] 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] The free on-line dictionary of computing.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

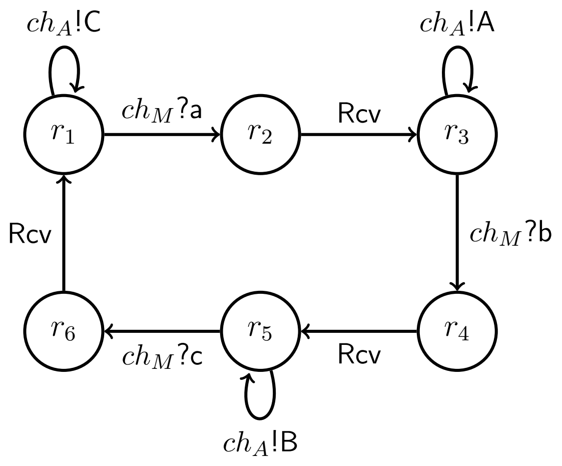
[22] McMillan, K. L. *Symbolic model checking*. Springer, 1993.

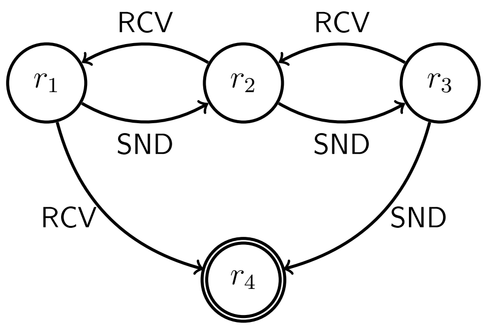
[23] Namjoshi, K. S. Symmetry and completeness in the analysis of parameterized systems. In *Verification, Model Checking, and Abstract Interpretation* (2007), Springer, pp. 299–313.

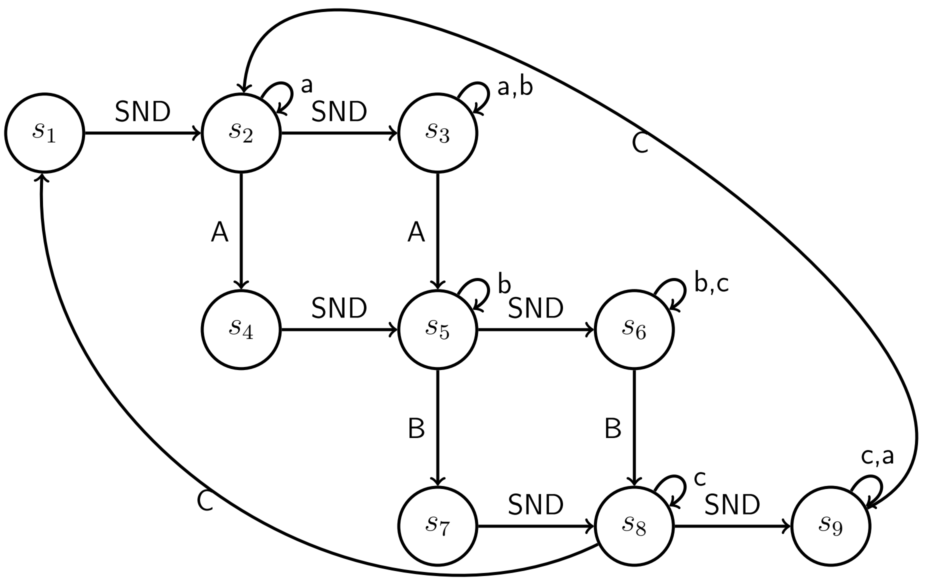
[24] Tarski, A., et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics 5*, 2 (1955), 285–309.

[25] 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


$ch_A$!C
$ch_M$?a
Rcv
$ch_A$!A
$r_1$
$r_2$
$r_3$
Rcv
$ch_M$?b
$r_6$
$r_5$
$r_4$
$ch_M$?c
Rcv
$ch_A$!B


RCV
RCV
$r_1$
$r_2$
$r_3$
SND
SND
RCV
SND
$r_4$


a
a,b
SND
SND
$s_1$
$s_2$
$s_3$
C
A
A
b
b,c
SND
SND
$s_4$
$s_5$
$s_6$
B
B
c,a
c
C
SND
SND
$s_7$
$s_8$
$s_9$