Small Model Theorems for Verification of Parameterized Systems

Tomas Sävström
Abstract

**Small Model Theorems for Verification of Parameterized Systems**

*Tomas Sävström*

The world of software is increasing and the requirements on software systems are getting harder. To ensure that these requirements are fulfilled, we use program verification. The goal of verification is to prove that the system automatically fulfills its requirements. In this thesis, we will consider parameterized systems. A parameterized system is a system that contains an arbitrary number of components (processes) organized according to a particular pattern. Such systems are heavily used to implement mutual exclusion protocols [1,2].

In this thesis we will extend parameterized verification to handle variables over unbounded data domain. In fact, there is a large number of protocols (or programs) that manipulate variable over unbounded data domain. An example is the Bakery[2] protocol which uses integer variables to decide the order in which the processes are allowed to enter their critical section. In order to handle the unbounded data domain, we use abstract interpretation. The key idea is to abstract away the variable values and only keep their internal relationships. Finally, we have constructed a prototype in C and tested it again on a number of mutual exclusion protocol.
Acknowledgements

Thanks to Parosh Abdulla, Jonathan Sharyari, Mohamed Faouzi Atig and Frédéric Haziza. For the help with my thesis. Parosh Abdulla and Frédéric Haziza for helping me when I got stuck and pointing me in the right direction and generally just discussing problems with me. Jonathan Sharyari for having someone to discuss things with and giving me feedback on the thesis from the point of view of a student. Mohamed Faouzi Atig for helping me to get this thesis to be as good as possible with all the good feedback. Without all the help this thesis would not have turned out this well.
Contents

1 Introduction 7
  1.1 Background .................................................. 7
  1.2 Contribution ................................................ 8
  1.3 Disposition ................................................ 9

2 Formal Model of Parameterized Finite-state Systems 10
  2.1 Preliminaries ................................................ 10
  2.2 Formal model ................................................ 12
  2.3 The Case of the Burns Protocol .......................... 13
    2.3.1 Pseudo-code for the Burns protocol ............... 13
    2.3.2 Formal Model for the Burns protocol ............ 15

3 Parameterized Verification of Finite State Systems 19
  3.1 Problem Definition ......................................... 19
  3.2 Verification Procedure ................................... 20
    3.2.1 View .................................................. 20
    3.2.2 Abstraction Function (\(\alpha\)) .................... 20
    3.2.3 Concretization Function (\(\gamma\)) ............... 20
    3.2.4 Post-Image (\(post\)) ................................ 21
    3.2.5 Abstract Post-Image (\(Apost_k\)) ................. 21
    3.2.6 Verification Procedure .............................. 21
  3.3 Summary .................................................. 23

4 Model for Parameterized Systems with Unbounded Data Domain 26
  4.1 Formal model ................................................ 26
  4.2 The Case of the Bakery Protocol ....................... 30
    4.2.1 Pseudo-code for the Bakery Protocol ............ 30
    4.2.2 Formal Model for the Bakery Protocol ........... 31

5 Parameterized Verification of Infinite-state Systems 35
  5.1 Subword Relation for Unbounded Parameterized Systems 35
  5.2 Problem Definition ......................................... 35
  5.3 Verification Procedure ................................... 36
Chapter 1

Introduction

This chapter introduces the background to and contributions of this thesis.

1.1 Background

With the increasing market of software products the need to ensure the correctness of such systems is bigger than ever. For example, such systems are used to handle the control of nuclear power plants, hospital equipment and train scheduling. An error in one of these systems can cause disasters, in the case of the power plant and the train scheduling, or can risk the life of a patient for the hospital equipment. The common practice in software development is to thoroughly test the product. The problem with testing is that it can not prove the absence of errors, only show their existence. An exception to this is when the input space for the product is small enough to test everything, which is normally not the case.

Because of these shortcomings of testing formal verification has become more prevalent. Formal verification means that the program is analyzed using various techniques to guarantee that the program is correct according to its specification. There are different techniques that are used to verify different classes of systems but they will try to verify that the system being checked fulfills the requirements.

Today concurrent programs are very common in the software systems, the nature of these programs makes them even harder to test as they can introduce heisenbugs\[3\]. These bugs are hard to test for and only appear in a few of the possible process interleavings in the system. As these interleavings depend on the scheduler of the operating system there is no guarantee that they will show up when running the test code. As one can not force the program to run every possible interleaving without a considerable effort in the testing process these errors are normally easy to miss. This makes formal verification even more de-
sirable as it requires less effort then testing in order to discover heisenbugs.

Concurrent programs where the number of processes running in parallel are not know a priori makes program verification tasks even harder. This because each instance needs to be proven correct. Testing each instance is not possible because the number of such instances is unbounded. Parameterized verification is used to verify these kind of systems which can be classified as parameterized systems. Parameterized systems are systems that consist of an arbitrary number of components (processes) order according to a particular pattern.

The problem that needs to be solved when doing parameterized verification is to guarantee the correctness of the system without using an enumerative approach that would check each instance of the system. One way to do this is to identify a cut-off point beyond which there is no need to verify any greater instances of the system. Because of this parameterized verification can be used to guarantee these systems correct.

Burns\cite{1} mutual exclusion protocol is an example of such a system that requires the application parameterized verification. This protocol ensures mutual exclusion properties, i.e. at most one process running its critical section at any time. The protocol is designed to work regardless of the number of processes present.

1.2 Contribution

This thesis extends the previous work presented in \cite{4} which defines a method for verifying parameterized finite-state systems. This method works by identifying a cut-off point that guarantees that all larger instances fulfill the requirements without having to check each instance individually. In this thesis, we extend the method proposed in \cite{4} to handle infinite-state systems, e.g. with variables over unbounded data domain. This extension is needed as there are many protocols that can be modeled as infinite-state systems.

A simple example of a protocol that cannot be verified is the Bakery\cite{2} protocol. The Bakery protocol uses integer variables, e.g. unbounded domain variables, to make sure that the different processes of the system can only enter their critical section one at a time. The extension presented in the thesis makes it possible to verify such protocols.

This work uses the same principles as described in \cite{4}. Our first contribution is a formal model for infinite-state systems using automata. Our second contribution is abstraction frame for such systems. The actual values of the unbounded domain variables are abstracted by only storing the relations between the variables. Applying this abstraction leads to a finite-state systems since there is a finite number of possible relation between each pair of variables. Our last
contribution is a prototype implemented in C. We have run several successful examples of mutual-exclusion protocols. This show that this approach is promising.

1.3 Disposition

The thesis is deposed as follows. In chapter 2 the formal model of a finite-state parameterized system is presented. It will presented the model and then the Burns protocol is modeled as an example. Chapter 3 presents the verification procedure for finite-state parameterized systems presented in [4]. Chapter 4 presents the formal model for parameterized systems with unbounded data domain which is part of the contribution of this thesis. Chapter 5 presents how the verification procedure presented in [4] is redefined to work with the model presented in chapter 4. Chapter 6 presents the experimental results from running the created prototype with a few protocols and discuss the results. Chapter 7 presents an extension to the model presented in chapter 4 that was not fully implemented. Chapter 8 presents related works. Chapter 9 presents the conclusions drawn from the work done in this thesis.
Chapter 2

Formal Model of Parameterized Finite-state Systems

This chapter introduces a formal model for parameterized finite-state systems. First we give some preliminaries that we will use in the rest of the thesis. Then, we define formally the class of parameterized systems and its induced transitions systems. We will use Burns protocol [1] as an instance of this class.

2.1 Preliminaries

In the following, we give some notations and concepts that are need for the rest of the thesis.

**Language Theory** Let \( \Sigma \) be a finite alphabet (a finite set of characters). We denote by \( \Sigma^* \) (resp. \( \Sigma^+ \)) the set of all words (resp. non-empty words) over \( \Sigma \), and \( \epsilon \) the empty word. A language is a (possible infinite) set of words. Let \( w = a_1 \ldots a_n \) be a word in \( \Sigma^* \), this word has a length \( n \) (i.e. \( |w| = n \)). Let \( w[i] \) denote the character at position \( i \). Given two words \( u = a_1 \ldots a_n \) and \( v = b_1 \ldots b_k \) then their concatenation is given by the word \( uv = a_1 \ldots a_n b_1 \ldots b_k \).

We denote by \( \subseteq \subseteq \Sigma^* \times \Sigma^* \) the subword relation defined as follows: for every \( u = a_1 \ldots a_n \in \Sigma^* \), and every \( v = b_1 \ldots b_k \in \Sigma^* \), \( u \subseteq v \) iff \( \exists i_1, \ldots, i_n \in \{1, \ldots, k\} \) such that \( i_1 < i_2 < \ldots < i_n \) and \( \forall j \in \{1, \ldots, n\}, a_j = b_{i_j} \). For instance ana is a subword of baaa, abbaa, ababba, .... The upward closure of a language \( L \) is the language \( L' \) defined by the set of words \( w \) such that there is word \( u \) in \( L \) and \( w \subseteq u \). A language \( L \) is called upward closed if its upward closure language \( L' \) is included in \( L \) (i.e. \( L = L' \)).
Finite State Automaton  A finite-state automaton (FSA) is a tuple $A = (Q, \Sigma, \Delta, I, F)$ where:

- $Q$ is the finite non-empty set of states
- $\Sigma$ is the input alphabet
- $\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation
- $I \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of final states

A word $w$ is accepted by $A$ iff there is a sequence of states $q_0, q_1, \ldots, q_n$ and letters $a_1, a_2, \ldots, a_n \in \Sigma \cup \{\epsilon\}$ such that (1) $w = a_1 a_2 \ldots a_n$ and (2) $(q_i, a_i, q_{i+1})$ is in $\Delta$ for all $i \in \{1, \ldots, n\}$. Then we use $L(A)$ to denote the set of words that are accepted by $A$. $L(A)$ is called the language accepted by $A$. A language is said to be regular if it is accepted by a finite-state automata. For more information on automata theory and language theory check [5].

Function Theory  Let $D$ be a set and $F$ be a function that maps any subset of $D$ to a subset of $D$. We say that $F$ is an order-preserving function if and only if for every subsets $X, Y \subseteq D$, 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 (or smallest) fix-point of $F$ with respect to the subset relation. Formally, $\mu X. F(X)$ denotes the smallest set $S \subseteq D$ such that $S$ is a fix-point of $F$. From the Knaster-Tarski fixed point theorem [6], we know that the least-fix point for order-preserving function exists. Furthermore, if $D$ is finite, then the sequence $X_0 = \emptyset, X_{i+1} = F(X_i)$ for all $i \geq 0$ converges in finitely many iterations to the least-fixpoint of $F$. An example of this would be the function that will add all numbers between 1 and 5 to the set given as input. $\mu X. F(X) = \{1, 2, 3, 4, 5\}$.

Set Theory  Let $R$ be a relation on a set $A$. The reflexive transitive closure of $R$ is then the smallest relation $R_s$ on $A$ such that

- if $(x, y) \in R$ then $(x, y) \in R_s$ for all $x, y \in A$
- $R_s$ is reflexive, i.e. $(x, x) \in R_s \forall x \in A$
- $R_s$ is transitive, i.e. for every $x, y, z \in A$, if $(x, y) \in R_s$ and $(y, z) \in R_s$ then $(x, z) \in R_s$

Transition system  A transition system $T$ is a pair $(S, \rightarrow)$ where $S$ is the set (possibly infinite) of states or configurations equipped with a partial order and $\rightarrow \subseteq S \times S$ is the set of transitions. We use $s \rightarrow s'$ to denote $(s, s') \in \rightarrow$. We use $\rightarrow^*$ to denote the reflexive transitive closure of $\rightarrow$. We say that $T$ is monotone (w.r.t. an ordering $\leq$) if whenever $s_1 \rightarrow s_2$ and $s_1 \leq s_2$ then $(s_2, s_4) \in \rightarrow^*$ for
some $s_4$ with $s_3 \leq s_4$.

An example of a transition system $T$ is given in figure 2.1 where the nodes represent the states of $T$ (i.e. $S = \{1, 2, 3, 4\}$) and the transition relation is represented by the set of edges (i.e. $\rightarrow = \{(1, 2), (2, 3), (3, 4), (4, 1)\}$).

Figure 2.1: Graphical representation of simple transition system

### 2.2 Formal model

Here we will start by giving an informal description of a parameterized finite-state system then we will provide a formal definition.

**Informal description** A parameterized finite-state system is a system that consists of an arbitrary number of processes that are ordered according to some pattern. The shape of the pattern depends on the current considered system. The most used pattern is a pipeline but trees and rings are other examples. In a parameterized system, processes run in parallel and each one follows a precise given description of its behavior (which we call the protocol). Some examples of parameterized systems include mutual exclusion protocols and cache coherence protocols. For instance the Bakery protocol, the Burns protocol and the Szymanski protocol. The Burns protocol will be explained in the next subsection and the Bakery protocol will be given in Section 4.2.1.

**Formal model** In the following, we give the formal definition for parameterized finite state systems. We will follow the definition given in [4].

A parameterized system is a pair $P = (Q, \Delta)$ where $Q$ is a finite set of local states of a process and $\Delta$ is a set of transition rules over $Q$. A transition rule is either local or global. A local rule is of the form $s \Rightarrow s'$, where $s, s' \in Q$, where the process changes its local state from $s$ to $s'$ independently of the local states of the other processes. A global rule is of the form if $\exists Q \circ i : S$ then $s \Rightarrow s'$, where $Q \in \{\exists, \forall\}$, $\circ \in \{=, >\}$ and $S \subseteq Q$. Here, the $i$th process checks also the local states of the other processes when it makes the move. For instance, the condition $\forall j < i : S$ means that "for every $j$ such that $j < i$, the $j$th process
should be in a local state that belongs to the set $S$; the condition $\forall j \neq i : S$ means that "all processes except the $i$th one should be in local states that belong to the set $S$"; etc.

A parameterized system $P = (Q, \Delta)$ induces a transition system $(TS) \mathcal{T} = (C, \to)$ where $C = Q^*$ is the set of its configurations and $\to \subseteq C \times C$ is the transition relation. We use $c[i]$ to denote the state of the $i$th process within the configuration $c$. We assume that the process are ordered according to architecture and each process has a unique id representing its position with respect to the other processes. The transition relation $\to$ contains a transition $c \to c'$ with $c[i] = s, c'[i] = s', c[j] = c'[j]$ for all $j : j \neq i$ iff either (i) $\Delta$ contains a local rule $s \Rightarrow s'$, or (ii) $\Delta$ contains a global rule $\text{if } Qj \circ i : S \text{ then } s \Rightarrow s'$, and one of the following conditions is satisfied:

- $Q = \forall$ and for all $j : 1 \leq j \leq |c|$ such that $j \circ i$, it holds that $c[j] \in S$.
- $Q = \exists$ and there exists $j : 1 \leq j \leq |c|$ such that $j \circ i$ and $c[j] \in S$.

For instance if the set of states is $Q = \{1, 2, 3\}$ then the set of possible configurations is $C = \{1, 2, 3, 11, 12, 13, 21, 22, 23, 31, 32, 33, \ldots\}$.

### 2.3 The Case of the Burns Protocol

In the following, we present the Burns protocol. First we will give its pseudo-code and provide some description of its behavior. Then, we give the formal model for the Burns protocols and its induced transition system.

#### 2.3.1 Pseudo-code for the Burns protocol

Here we will present the pseudo-code for the Burns protocol\[1\]. Burns is a mutual exclusion protocol that uses Boolean variables to ensure mutual exclusion (i.e., no two processes are executing their critical section at the same time) between a set of (possibly unbounded) processes ordered as a pipeline with respect to the process id. We assume here that each process has a unique id. The protocol works as follows: a process can not enter its critical section if a process with a smaller index is trying to enter its critical section (or a process is already in its critical section). This means that lower process id has priority. In algorithm 1 pseudo-code for the Burns protocol is given. The protocol assumes a global boolean array $\text{flag}$ of size $n$ to keep track of the pipeline pattern, where $n$ is the amount of processes in the system. In algorithm 2 and algorithm 3 an instance of the Burns protocol for two processes is given.

The protocol works as follows: Each process passes through two checking rounds (at line 2 and line 6) before getting to its critical section (that would be placed after line 10). In each round the process will restart its execution from the beginning (line 1) if he finds a process with lower id requesting the access to its critical section. Priority is given to the lower indexes, meaning that unless a
Algorithm 1: Burns

Data: integer i
1 \( flag[i] = false \)
2 if \( \exists j < i : flag[j] \) then
3 \quad goto 1
4 end
5 \( flag[i] = true \)
6 if \( \exists j < i : flag[j] \) then
7 \quad goto 1
8 end
9 while \( \exists j > i : flag[j] \) do
10 end
11 \( flag[i] = false \)
12 goto 1

Figure 2.2: Pseudo-code for the Burns protocol

Algorithm 2: Burns-Process-0

1 \( flag[0] = false \)
2 if \( \exists j < 0 : flag[j] \) then
3 \quad goto 1
4 end
5 \( flag[0] = true \)
6 if \( \exists j < 0 : flag[j] \) then
7 \quad goto 1
8 end
9 while \( \exists j > 0 : flag[j] \) do
10 end
11 \( flag[0] = false \)
12 goto 1

Figure 2.3: Pseudo-code for process 0 in the Burns protocol with two processes
Algorithm 3: Burns-Process-1

1  flag[1] = false
2  if \( \exists j < 1 : flag[j] \) then
3     goto 1
4  end
5  flag[1] = true
6  if \( \exists j < 1 : flag[j] \) then
7     goto 1
8  end
9  while \( \exists j > 1 : flag[j] \) do
10 end
11 flag[1] = false
12 goto 1

Figure 2.4: Pseudo-code for process 1 in the Burns protocol with two processes

process is already in its critical section, process 0 will always have priority. This can be seen in the code as in line 2 and line 6 were the process checks if any process with a lower index has requested access to its critical section (has its flag set to true). If it is the case, the process returns to line 1. Observe that process 0 will never go back to line 1 has there is no process with a lower index. The Burns protocol gives rise to a parameterized system as there is no restrictions on the number of processes which run in parallel. This protocol gives a set of rules and can be modeled using states. How Burns is modeled will be shown in the next subsection.

2.3.2 Formal Model for the Burns protocol

Here we will show how the Burns protocol would be modeled using the definition from section 2.2. The formal model for the Burns protocol will be \( P = (Q, \Delta) \):

\[
Q = \{1, 2, 3, 4, 5, 6\}
\]

\[
\Delta = \{1 \Rightarrow 2, \text{if} \exists j < i : \{4, 5, 6\} \text{ then } 2 \Rightarrow 1, \text{if} \forall j < i : \{1, 2, 3\} \text{ then } 2 \Rightarrow 3, \text{if} \forall j < i : \{4, 5, 6\} \text{ then } 4 \Rightarrow 1, \text{if} \forall j < i : \{1, 2, 3\} \text{ then } 4 \Rightarrow 5, \text{if} \forall j > i : \{1, 2, 3\} \text{ then } 5 \Rightarrow 6, 6 \Rightarrow 1\}
\]

Here state 1 corresponds to the process having its flag set to either true or false and is about to execute line 1. State 2 corresponds to the process having its flag set to false and have just executed line 1. State 3 corresponds to the process having its flag set to false and is about to execute line 5. State 4 corresponds
to the process having its flag set to true and have just executed line 5. State 5 corresponds to the process having its flag set to true and is about to execute line 9. State 6 corresponds to the process having its flag set to true and is about to execute line 11.

The transition "1 ⇒ 2" corresponds to executing line 1. The transition "if \exists j < i : \{4, 5, 6\} then 2 ⇒ 1" corresponds to executing the then part of the if statement on line 2. The transition "if \forall j < i : \{1, 2, 3\} then 2 ⇒ 3" corresponds to executing the else part of the if statement on line 2. The transition "3 ⇒ 4" corresponds to executing line 5. The transition "if \exists j < i : \{4, 5, 6\} then 4 ⇒ 1" corresponds to executing the then part of the if statement on line 6. The transition "if \forall j < i : \{1, 2, 3\} then 4 ⇒ 5" corresponds to executing the else part of the if statement on line 6. The transition "if \forall j > i : \{1, 2, 3\} then 5 ⇒ 6" corresponds executing the while loop at line 9. The transition "6 ⇒ 1" corresponds to executing line 11 and line 12.

In Figure 2.5 a finite state system of the created model is shown. This model is then easily used to create finite state system for an instance of the Burns protocol with two processes, these are shown in figure 2.6 and figure 2.7.

![Finite State System Diagram](image)

**Figure 2.5:** Finite state system of the parameterized system model of Burns

The created model for the Burns protocol induces a transition system. In figure 2.8 is a graphical representation of the transition system, of size two, is shown.
Figure 2.6: Finite state system of process 0 in the parameterized system model of Burns with two processes

Figure 2.7: Finite state system of process 1 in the parameterized system model of Burns with two processes
Figure 2.8: The transition system of size 2 for the Burns protocol
Chapter 3

Parameterized Verification of Finite State Systems

This chapter states the problem that we are interested in. Then describes the verification procedure proposed in [4] and all its components. In the following, we use the same notations as in the previous chapter. All definitions come from [4].

3.1 Problem Definition

In the following, we give the definition of the reachability problem that we are interested in.

An instance of the reachability problem is defined by a parameterized system \( P = (Q, \Delta) \), a regular set \( I \subseteq Q^+ \) of initial configurations, and a set \( \text{Bad} \subseteq Q^+ \) of bad configurations. Let \( \sqsubseteq \) be the usual subword relation. We assume that \( \text{Bad} \) is the upward closure of a given finite set \( B \subseteq Q^+ \) of minimal bad configurations, i.e. \( \{ c \mid \exists b \in B : b \sqsubseteq c \} \). Let \( T = (C, \rightarrow) \) be the transition system induced by \( P \) as defined in the previous chapter. We say that \( c \in C \) is reachable iff there is a sequence of configurations \( c_0, \ldots, c_l \in C \) such that \( c_0 \in I, c_l = c \) and \( c_i \rightarrow c_{i+1} \) for all \( 0 \leq i \leq l \). We use \( R \) to denote the set of all reachable configurations. We say that the system \( P \) is safe w.r.t. \( I \) and \( \text{Bad} \) if no bad configuration is reachable i.e. \( R \cap \text{Bad} = \emptyset \).

Observe that the precise definition of the initial configurations depends on the considered system. They will represent all possible configurations of the system at the beginning. For example for the Burns protocol in [2.3.1] the set of initial configuration will be all configurations were each process is in state 1, i.e. the set \( 1^+ \).
3.2 Verification Procedure

Here each part of the verification procedure will be explained and then the verification procedure will be explained.

3.2.1 View

A view is an abstract representation of a set of configurations. A view \( v \) will be represented by its unique minimal configuration \( v_c \) w.r.t. subword relation but represent the set \( \{ c | v \subseteq c, c \in C \} \). When a view is used in the verification procedure it will be handled as \( v_c \) and not as the set it represents. If we take for example the view 1221 this will represent all configurations where there are at least two processes in state 1, two processes in state 2 and are ordered in the same manner as in the view. Some examples of configurations are 121121, 1332211, 132321, ...

3.2.2 Abstraction Function (\( \alpha \))

In the following, we define the abstraction function. Let \( k \) be a natural number. Let \( C_k = \{ c \in C | |c| \leq k \} \) be the set of configurations of size less or equal to \( k \). \( 2^{C_k} \) here stands for the possible sets of views with size \( k \) or less.

For every \( k \), we abstract a configuration \( c \) by a set of views each of which is a subword of \( c \). The abstraction function \( \alpha_k : C \rightarrow 2^{C_k} \) maps a configuration \( c \) to the set \( \alpha_k(c) = \{ v \in C_k | v \subseteq c \} \) of all its views (subwords) of size up to \( k \). Finally, we can easily extend the abstraction function to a set of configurations as follows: Given a set \( S \subseteq C \), \( \alpha_k(S) = \{ \alpha_k(c) | c \in S \} \).

The subword relation is defined in 2.1. For example let \( c = 1335 \) and \( k = 3 \), then \( \alpha_3(c) = \{ 1, 3, 5, 13, 15, 33, 35, 133, 135, 335 \} \). We can also observe that the abstraction function is order preserving.

3.2.3 Concretization Function (\( \gamma \))

In the following, we define the concretization function. \( 2^C \) here stands for all possible set of configurations.

For every \( k \in \mathbb{N} \), the concretization function \( \gamma_k : 2^{C_k} \rightarrow 2^C \) inputs a set of views \( V \subseteq C_k \), and returns the set of configurations that can be reconstructed from the views in \( V \). In other words \( \gamma_k(V) = \{ c \in C | \alpha_k(c) \subseteq V \} \).
Let \( V = \{1, 3, 5, 13, 15, 33, 35, 133, 135, 333, 335\} \) be a set of views. Then the configurations of size 4 of \( \gamma_3(V) \) is \( \{1335, 1333, 3335\} \). In fact we have:

\[
\begin{align*}
\alpha_3(1335) &= \{1, 3, 5, 13, 15, 33, 35, 133, 135, 333, 335\} \\
\alpha_3(1335) &\subseteq V \\
\alpha_3(1333) &= \{1, 3, 13, 33, 133\} \\
\alpha_3(1333) &\subseteq V \\
\alpha_3(3335) &= \{3, 5, 33, 35, 333, 335\} \\
\alpha_3(3335) &\subseteq V
\end{align*}
\]

Observe that the function \( \gamma_k \) is an order preserving function. Furthermore, for any set \( S \) of configurations, we have \( S \subseteq \gamma_k(\alpha_k(S)) \).

### 3.2.4 Post-Image (\( post \))

In the following, we define the post function.

We define the post-image of a set \( X \subseteq C \) to be the set \( post(X) := \{c'|c \rightarrow c' \wedge c \in X\} \).

An example of the post-image in the case of the Burns protocol described in Sec. 2.3.2:

\[
\begin{align*}
X &= \{11, 12, 21\} \\
post(X) &= \{12, 21, 22, 21, 12\}
\end{align*}
\]

### 3.2.5 Abstract Post-Image (\( Apost_k \))

In the following, we define the abstract post-image. Let \( k \) be a natural number.

The abstract post-image of a set of views \( V \subseteq C_k \) is defined as

\[
Apost_k(V) = \alpha_k(post(\gamma_k(V))).
\]

This function will take a set of views, create all possible configurations, apply the post function to these concretizations and then create new views from the result.

### 3.2.6 Verification Procedure

We will first describe the algorithm and give some explanation. The definition of the algorithm and the lemmas come from [1]. We will then show the termination of this algorithm.
The verification procedure for solving an instance of the verification problem defined in chapter 2 is described in Algorithm 4. It performs two search procedures that can be done in parallel. Specifically, it searches, on line 2, for a bad configuration reachable from initial configurations which consist of configurations of size $k$; and it searches, on line 6, for a cut-off point, i.e. a value of $k$ where it derives a set of views $V \subseteq C_k$ such that

(i) $V$ is an invariant for instances of the system (that is, $R \subseteq \gamma_k(V)$ and $Apost_k(V) \subseteq V$), and

(ii) which is sufficient to prove that $R$ is safe (that is, $\gamma_k(V) \cap Bad = \emptyset$).

For the first search (from line 2 to line 4 of Algorithm 4), if this check finds bad configurations then it is straightforward to see that the system is unsafe. Computing the reachable configurations $R_k$ of size less or equal to $k$ is straightforward. Let $I_k = \{c | c \in I \text{ and } |c| \leq k\}$. It is easy to see that the sequence $X_0 = I_k$ and $X_{i+1} = X_i \cup (post(X_i) \cap C_k)$ converges to $R_k$ in finitely many iterations since $C_k$ is finite.

For the second search (from line 5 to line 8), the algorithm proceeds as follows:

First it computes an invariant $V$ as defined above (see the item (i)). This invariant is computed at line 5 as the least fix-point of the order-preserving function $\alpha_k(I) \cup Apost_k$. Observe that such least-fix point exists and can be effectively computed since $C_k$ is finite as described in the section "Preliminaries" of the previous chapter. We will show below that the $Apost_k(X)$ of a set of configurations $X$ can be effectively computed. The following lemma shows that this least-fix point $V$ is indeed an invariant:

**Lemma 1.** For any $k \in \mathbb{N}$ and $X \subseteq C_k$, $((\alpha_k(I) \subseteq X) \land (Apost_k(X) \subseteq X)) \implies \alpha_k(R) \subseteq X$

### Algorithm 4: Verification Procedure

1. for $k = 1$ to $\infty$ do
2.  if $R_k \cap Bad \neq \emptyset$ then
3.     return Unsafe
4.  end
5.  $V := \mu X.\alpha_k(I) \cup Apost_k(X)$
6.  if $\gamma_k(V) \cap Bad = \emptyset$ then
7.     return Safe
8.  end
9. end

The requirements on $V$ for it to be an invariant is as stated earlier $R \subseteq \gamma_k(V)$ and $Apost_k(V) \subseteq V$. $Apost_k(V) \subseteq V$ will hold as $V$ is a fix-point (i.e., $V = \alpha_k(I) \cup Apost_k(V)$). $R \subseteq \gamma_k(V)$ will be fulfilled because of Lemma 1. $\alpha_k(I) \subseteq V$ and $Apost_k(V) \subseteq V$ hold since $V$ is a fix-point (i.e., $V = \alpha_k(I) \cup Apost_k(V)$). This implies that $\alpha_k(R) \subseteq V$. Since the concretization function $\gamma_k$ is an order
preserving function, we have \( \gamma_k(\alpha_k(\mathcal{R})) \subseteq \gamma_k(V) \). Finally, we can also show \( \mathcal{R} \subseteq \gamma_k(\alpha_k(\mathcal{R})) \) and hence \( \mathcal{R} \subseteq \gamma_k(\alpha_k(\mathcal{R})) \subseteq \gamma_k(V) \).

Then, the second step is to use the computed fix-point \( V \) which is also an invariant. This means that \( \mathcal{R} \) is a subset of \( \gamma_k(V) \). At line 6, the algorithm checks if \( V \) is a cut-off point and that our system is safe (i.e., \( \gamma_k(V) \) does not contain any bad configuration and so neither does the set \( \mathcal{R} \)). If the intersection is not empty due to a too imprecise abstraction, the algorithm increases the precision of the abstraction function by increasing \( k \) and reiterating the loop.

Finally, Lemma 2 show that the function \( \text{Apost}_k(X) \) can be effectively computed for an set of configurations \( X \subseteq C_k \).

**Lemma 2.** For any \( k \in \mathbb{N} \) and \( X \subseteq C_k \), \( \alpha_k(\text{post}(\gamma_k(X))) \cup X = \alpha_k(\text{post}(\gamma_{k+1}^k(X))) \cup X \).

Where \( \gamma_{k+1}^k = \{ c | c \in \gamma_k, |c| \leq k + 1 \} \). Lemma 2 states that the \( \gamma_k \) function can be restricted to only looking at sizes up to \( k + 1 \) when it is used in the abstract post image function as it will give the same result as without the restriction.

An effective implementation of the procedure requires carrying out the following steps:

1. **Computing the abstraction** \( \alpha_k(I) \) **of initial configurations.** This step is usually easy. For instance, in the case of Burns’ protocol, all processes are initially in state 1, hence \( \alpha_k(I) \) contains only the words \( 1^i, l \leq k \). Generally, \( I \) is a (very simple) regular set, and \( \alpha_k(I) \) is computed using a straightforward automata construction.

2. **Computing the abstract post-image.** Thanks to Lemma 2, the abstract post-image can be computed by applying \( \gamma_{k+1}^k \) (which yields a finite set), \( \text{post} \) and \( \alpha_k \) (in that order) to the given input.

3. **Evaluating the test** \( \gamma_k(V) \cap \text{Bad} = \emptyset \). Since \( \text{Bad} \) is the upward closure of a finite set \( B \), the test can be carried out by testing whether there is a \( b \in B \) such that \( \alpha_k(b) \subseteq V \).

4. **Exact reachability analysis.** Line 4 requires the computation of \( \mathcal{R}_k \). Since \( \mathcal{R}_k \) is finite, this can be done using any procedure for exact state space exploration.

Since the problem is generally undecidable, existence of such a \( k \) cannot be guaranteed and the algorithm may not terminate.

### 3.3 Summary

In this chapter, we have described the verification technique developed in [4] for the verification of parametrized finite-state systems. The flow-chart of the tech-
nique is given in Figure 3.1. This technique uses two semi-decision procedures.

- An under-approximation technique which uses as parameter a natural number $k$ (denoting the size of the analyzed configurations). The parameter $k$ is set to 1 initially. Then the technique constructs all the set of reachable configurations of size $k$ from the initial configurations and checks if it contains a bad configuration. In the case of a positive answer, then the verification approach terminates by returning that the system is unsafe otherwise we increase the parameter $k$ and repeat the process.

- An over-approximation techniques which tries to compute an over-approximation of the set of reachable configurations and then checks if it does not contain any bad configuration. In the case of a positive answer, the verification technique terminates and returns that the system is safe; otherwise, the technique proceeds by increasing the precision of the over-approximation.
Figure 3.1: Flowchart of the verification method
Chapter 4

Model for Parameterized Systems with Unbounded Data Domain

In this chapter the model for a parameterized system with unbounded domain variables will be presented. This chapter will start with the formal model for a parameterized system with unbounded domain variables being presented. After that the Bakery protocol is defined. The Bakery protocol will be used as an example of how to model a protocol using this formal model.

4.1 Formal model

The protocols that we are considering in this thesis use integer variables. Therefore we use the standard comparison operation on the set of natural variables. We will have the following set \{<, >, =\} of relationships between variables. We use also - to denote that there is no relation between the two variables. This happen when a variable is set to an undefined value.

An unbounded parameterized system is a triple \( P' = (Q, V, \Delta) \), where \( Q \) is a finite set of local process states, \( V \) is a set of relationships matrices that are used to describe the relationships between the variables of the processes in the system, in this extension this will contain relationships from the set \{<, >, =, \} , and \( \Delta \) is a set of transition rules. Below \( P_i \) stands for the state of process \( i \), where \( P_i \in Q \).

- \( S \Rightarrow S' \) (A local transition of the state of a process where \( S, S' \in Q \))
- \( R_{xy} \leftarrow R \) (Set relationship at position \( x,y \) to be \( R, R \in V \) and \( R \in \{<, >, =, \} \))
\[ \forall x, \mathcal{R}_{xy} \leftarrow R, \text{ where } x < y \] (Set all relationships, \( \mathcal{R}_{xy} \), to \( R \))
\[ \forall x, \mathcal{R}_{xy} \leftarrow R, \text{ where } x > y \] (Set all relationships, \( \mathcal{R}_{xy} \), to \( R \))
\[ \forall x, \mathcal{R}_{xy} \leftarrow R, \text{ where } x = y \] (Set all relationships, \( \mathcal{R}_{xy} \), to \( R \))
\[ \forall x, \mathcal{R}_{xy} \leftarrow R, \text{ where } x \neq y \] (Set all relationships, \( \mathcal{R}_{xy} \), to \( R \))
\[ \forall x, \mathcal{R}_{xy} \leftarrow R, \text{ where } P_x \in C \] (Set all relationships, \( \mathcal{R}_{xy} \), to \( R \))

- If \( \forall x, \mathcal{R}_{xy} = R, \text{ where } x < y \) then \( S \Rightarrow S' \) (If for all \( x \), where \( x < y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))

- If \( \forall x, \mathcal{R}_{xy} = R, \text{ where } x > y \) then \( S \Rightarrow S' \) (If for all \( x \), where \( x > y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))

- If \( \forall x, \mathcal{R}_{xy} = R, \text{ where } x = y \) then \( S \Rightarrow S' \) (If for all \( x \), where \( x = y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))

- If \( \forall x, \mathcal{R}_{xy} = R, \text{ where } x \neq y \) then \( S \Rightarrow S' \) (If for all \( x \), where \( x \neq y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))

- If \( \exists x, \mathcal{R}_{xy} = R, \text{ where } x < y \) then \( S \Rightarrow S' \) (If there exist an \( x \), where \( x < y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))

- If \( \exists x, \mathcal{R}_{xy} = R, \text{ where } x > y \) then \( S \Rightarrow S' \) (If there exist an \( x \), where \( x > y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))

- If \( \exists x, \mathcal{R}_{xy} = R, \text{ where } x = y \) then \( S \Rightarrow S' \) (If there exist an \( x \), where \( x = y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))

- If \( \exists x, \mathcal{R}_{xy} = R, \text{ where } x \neq y \) then \( S \Rightarrow S' \) (If there exist an \( x \), where \( x \neq y \), \( \mathcal{R}_{xy} = R \) then do the local transition from \( S \) to \( S' \), where \( S, S' \in Q, \mathcal{R} \in V, R \in \{<,>,=,-\} \))
following cases holds:

- if(∃x, Rxy = R, where Pz ∈ C)then(S ⇒ S') (If there exist an x, where
  Pz ∈ C, Rxy = R then do the local transition from S to S', where S, S' ∈ Q; R ∈ V, R ∈ {<, >, =, =}, C ⊆ Q and C ≠ ∅)

For the if rules ∃ and ∀ statements can be combined using ∧ and ∨ creating more complex conditions for the transition. The size of R will depend on the protocol being modeled.

An unbounded parameterized system (Q, V, Δ) induces a transition system (TS) T = (Conf, →) where Conf = Q* × V is the set of configurations and → ⊆ Conf × Conf is the transition relation. Let c = (w, r) be a configuration where w ∈ Q*, r ∈ V and r is of size |w| × |w|, we use c[i] := w[i] to denote the state of the ith process and c[x, y] to be the relation stored at the position (x, y) of the matrix r if such position exists. Let c = (w, r) and c' = (w', r') be two configurations. Let δ ∈ Δ be a transition rule. We define the transition relation c → c' between the configurations which is induced by the rule δ if one of the following cases holds:

- δ is of the form S ⇒ S' such that r = r', c[i] = S, c'[i] = S' and
  c[j] = c'[j], ∀j : j ≠ i.

- δ is of the form Rij ← R' such that c'[i, j] = R', c[x, y] = c'[x, y], ∀x, y : x ≠ i ∨ y ≠ j and w = w'.

- δ is of the form ∀i, Rij ← R', where i < j such that c'[i, j] = R', ∀i : i < j, c[x, y] = c'[x, y], ∀x, y : x ≥ j ∨ y ≠ j and w = w'.

- δ is of the form ∀i, Rij ← R', where i > j such that c'[i, j] = R', ∀i : i > j, c[x, y] = c'[x, y], ∀x, y : x ≤ j ∨ y ≠ j and w = w'.

- δ is of the form ∀i, Rij ← R', where i = j such that c'[i, j] = R', ∀i : i = j, c[x, y] = c'[x, y], ∀x, y : x ≠ j ∨ y ≠ j and w = w'.

- δ is of the form ∀i, Rij ← R', where i ≠ j such that c'[i, j] = R', ∀i : i ≠ j, c[x, y] = c'[x, y], ∀x, y : x = j ∨ y ≠ j and w = w'.

- δ is of the form ∀i, Rij ← R', where Pz ∈ C such that c'[i, j] = R', ∀i : Pz ∈ C, c[x, y] = c'[x, y], ∀x, y : Pz ∈ C ∧ y ≠ j and w = w'.

- δ is of the form if(∀i, Rij = R, where i < j)then(S ⇒ S') such that c[i, j] = R, ∀i : i < j, r = r' and c[z] = S, c'[z] = S' and c[k] = c'[k], ∀k : k ≠ z.

- δ is of the form if(∀i, Rij = R, where i > j)then(S ⇒ S') such that c[i, j] = R, ∀i : i > j, r = r' and c[z] = S, c'[z] = S' and c[k] = c'[k], ∀k : k ≠ z.

- δ is of the form if(∀i, Rij = R, where i = j)then(S ⇒ S') such that c[i, j] = R, ∀i : i = j, r = r' and c[z] = S, c'[z] = S' and c[k] = c'[k], ∀k : k ≠ z.
\[ \delta \text{ is of the form } \text{if} (\forall i, \mathcal{R}_{ij} = R, \text{ where } i \neq j) \text{then} (S \Rightarrow S') \text{ such that } c[i, j] = R, \forall i : i \neq j, r = r' \text{ and } c[z] = S, c'[z] = S' \text{ and } c[k] = c'[k], \forall k : k \neq z. \]

\[ \delta \text{ is of the form } \text{if} (\forall i, \mathcal{R}_{ij} = R, \text{ where } P_i \in C) \text{then} (S \Rightarrow S') \text{ such that } c[i, j] = R, \forall i : P_i \in C, r = r' \text{ and } c[z] = S, c'[z] = S' \text{ and } c[k] = c'[k], \forall k : k \neq z. \]

\[ \delta \text{ is of the form } \text{if} (\exists i, \mathcal{R}_{ij} = R, \text{ where } i < j) \text{then} (S \Rightarrow S') \text{ such that } \exists i : c[i, j] = R, \text{ where } i < j, r = r' \text{ and } c[z] = S, c'[z] = S' \text{ and } c[k] = c'[k], \forall k : k \neq z. \]

\[ \delta \text{ is of the form } \text{if} (\exists i, \mathcal{R}_{ij} = R, \text{ where } i > j) \text{then} (S \Rightarrow S') \text{ such that } \exists i : c[i, j] = R, \text{ where } i > j, r = r' \text{ and } c[z] = S, c'[z] = S' \text{ and } c[k] = c'[k], \forall k : k \neq z. \]

\[ \delta \text{ is of the form } \text{if} (\exists i, \mathcal{R}_{ij} = R, \text{ where } i = j) \text{then} (S \Rightarrow S') \text{ such that } \exists i : c[i, j] = R, \text{ where } i = j, r = r' \text{ and } c[z] = S, c'[z] = S' \text{ and } c[k] = c'[k], \forall k : k \neq z. \]

\[ \delta \text{ is of the form } \text{if} (\exists i, \mathcal{R}_{ij} = R, \text{ where } P_i \in C) \text{then} (S \Rightarrow S') \text{ such that } \exists i : c[i, j] = R, \text{ where } P_i \in C, r = r' \text{ and } c[z] = S, c'[z] = S' \text{ and } c[k] = c'[k], \forall k : k \neq z. \]

We are using the macro notation \( T_0; T_1; \ldots; T_n \) where each \( T_i \) is \( \delta \) to denote the application of these rules from left to right. Thus we can extend the definition of \( \Rightarrow \) such that \( c \Rightarrow c' \), if there is a rule of the form \( T_0; T_1; \ldots; T_n \) and a sequence of transitions \( c_0, c_1, \ldots, c_n \) such that \( c_0 = c, c_{n+1} = c' \) and \( c_i \Rightarrow c_{i+1} \) holds \( \forall i \in \{0, \ldots, n\} \) and is induced by the rule \( T_i \).

An example of how to use the model would be a simple mutual exclusion protocol that would be defined as follows:

\[ \mathcal{P}' = (Q, V, \Delta) \]

\[ Q = \{ \text{Idle}, \text{Waiting}, CS \} \]

\[ V = \{ (\leq), (\geq), (\neq), (\leq \leq), (\geq \geq), (\leq \geq), (\geq \leq), \ldots \} \]

And \( \Delta \) contains the following rules, where \( i \) is the index of the processes performing the rule:

- \( \text{Idle} \Rightarrow \text{Waiting} ; \forall j, \mathcal{R}_{ij} \leftarrow<, \text{ where } P_j \in \{ \text{Idle} \} ; \forall j, \mathcal{R}_{ij} \leftarrow>, \text{ where } P_j \in \{ \text{Waiting}, CS \} \)
• if(∀j, R_{ij} = <, where \( i \neq j \)) then (Waiting ⇒ CS)

• CS ⇒ Idle; ∀j, R_{ji} ← <, where \( P_j \in \{Waiting\} \); ∀j, R_{ij} ← −, where \( i \neq j \)

This would give the following transition system:

\[ T = (C, \rightarrow) \]
\[ C = \{\{Idle, (-)\}, \{Waiting, (-)\}, \{CS, (-)\}, \ldots\} \]
\[ \rightarrow = \{\{Idle, (-)\} → \{Waiting, (-)\}, \{Waiting, (-)\} → \{CS, (-)\}, \{CS, (-)\} → \{Idle, (-)\}, \{Idle.Idle, (-\ldots)\} → \{Idle.Waiting, (-\ldots)\}, \ldots\} \]

4.2 The Case of the Bakery Protocol

In the following, we present the Bakery protocol. First we will give its pseudo-code and provide some description of its behavior. Then, we give the formal model for the Burns protocols and its induced transition system.

4.2.1 Pseudo-code for the Bakery Protocol

To give an easier understanding of the method we will use Bakery[2] as an example, here we will define Bakery in pseudo code. The protocol is divided up into two algorithms, one to get the lock, see algorithm 5, and one to release the lock, see algorithm 6. The protocol assume two global arrays of size \( n \), where \( n \) is the amount of processes in the system, \textbf{Getting Ticket} and \textbf{Ticket}. \textbf{Getting Ticket} is a boolean array and \textbf{Ticket} is an integer array.

The protocol works as follows: When a process wants to enter its critical section it will need to get a ticket number. This ticket number will need to be strictly greater then all other numbers that are used. Because of this the new ticket number will be max of the numbers in \textbf{Ticket}, which keeps track of the ticket numbers, plus one. When the process has the ticket number it will wait for its turn, which will be when the process has the lowest ticket number. If two or more processes has the lowest ticket number ties are broken by using the process id, meaning the the process with the lowest id will get priority. When the process is done with its critical section it will set its ticket value to be
undefined, which in a practical case is defined as a specific number.

Algorithm 5: Bakery lock algorithm

1 Algorithm: Bakery-Aquire-Lock
   Data: integer \( i \)
2 \( \text{Getting.} \text{Ticket}[i] := \text{true} \)
3 \( \text{Ticket}[i] := 1 + \max(\text{Ticket}[0], \text{Ticket}[1], \ldots, \text{Ticket}[n-1]) \)
4 \( \text{Getting.} \text{Ticket}[i] := \text{false} \)
5 for \( k = 0 \) to \( n-1 \) do
   6 while \( \text{Getting.} \text{Ticket}[k] \) do
   7 end
   8 while \( (\text{Ticket}[k] \neq 0) \land ((\text{Ticket}[k] < \text{Ticket}[i]) \lor ((\text{Ticket}[k] = \text{Ticket}[i]) \land (k < i))) \) do
   9 end
10 end

Algorithm 6: Bakery unlock algorithm

1 Algorithm: Bakery-Release-Lock
   Data: integer \( i \)
2 \( \text{Ticket}[i] := 0 \)

4.2.2 Formal Model for the Bakery Protocol

As presented in 4.2.1 the Bakery protocol uses one variable for each process to keep track of the ticket number. One variable is also used to keep track if the process is getting a ticket number. Since this is a Boolean variable we can model this using the process states. The protocol works by using the values of the ticket variables to decide who is allowed to enter the critical section. Because of this the relationships between these variables is something important and is saved. Since we will need to know the exact place in the queue that process \( i \) has we will save the relation between the ticket variable in process \( i \) and the ticket variables in all other processes in process \( i \). This will give us the information needed to calculate the place of process \( i \) in the queue. Below is how Bakery is defined using this model.

\[
P’ = (Q, V, \Delta) \\
Q = \{1, 2, 3, 4\} \\
V = \{(\prec), (\succ), (\preceq), (\succeq), (\lessgtr), (\gtrless), \ldots\}
\]

And \( \Delta \) contains the following rules, where \( i \) is the index of the processes performing the rule:

- Rule 1: \( 1 \Rightarrow 2; \forall j, R_{ij} \leftrightarrow \), where \( P_j \in \{3, 4\} \); \( \forall j, R_{ij} \leftarrow = \), where \( P_j \in \{2\} \); \( \forall j, R_{ji} \leftarrow = \), where \( P_j \in \{2\} \)
• Rule 2: $2 \Rightarrow 3; \forall j, R_{ij} \leftarrow <$, where $P_j \in \{1\}$

• Rule 3: if ($\forall j, R_{ij} = <$, where $j < i \land \forall j, R_{ij} \leq$, where $j > i$) then ($3 \Rightarrow 4$)

• Rule 4: $4 \Rightarrow 1; \forall j, R_{ij} \leftarrow -$, where $P_j \neq i$; $\forall j, R_{ji} \leftarrow <$, where $P_j \in \{3, 4\}$; $\forall j, R_{ji} \leftarrow -$, where $P_j \in \{2\}$; $R_{ii} \leftarrow -$.

In this explanation index $i$ will refer to the process performing the rule. State 1 here represents that the process is before starting to run the acquire lock function. State 2 represents that it currently executing line 3. State 3 represents that the process has executed line 4 and is currently executing the for loop on line 5. State 4 means that the process has finished executing the acquire lock function, i.e. has acquired the lock. As can be seen here the first rule is that the process goes from state 1 to state 2, i.e. executed line 2. It will also in this rule set that variable $i$ have a larger value then all processes that are currently in state 3 or 4, i.e. has already acquired a ticket value. This is because we will know that these variables will have a smaller value then the process that the rule is performed on. This as a process without a ticket value will get a value strictly greater then all other processes currently have a ticket value. The rule will also set that variable $i$ was the same ticket value as all processes that are currently in state 2. This as they are choosing ticket value at the same time and will therefore get the same value as no lock is placed on reading the values. It will also update the relationships in all processes already in state 2 so the the relationships are consistent.

In the second rule process $i$ transitions from state 2 to state 3, i.e. has acquired a ticket value and is waiting for its turn. The rule will then also set that all processes that are in state 1 will have a greater ticket value then process $i$ has for the same reason that was present in rule 1. All new ticket values will be strictly greater. The third rule is an if rule, it says that process $i$ can only transition to state 4 if it is first in the queue. In other words has a ticket value that is the smallest among all processes $j$, where $j < i$, and the smallest or equal to the values of all processes $k$, where $k > i$. This is to make sure that smaller indexes have priority when the ticket value is equal. The last rule is then a transition from state 4 to state 1, i.e. leaving the critical section, this will be equal to executing the unlock function. Here the other processes are updated to show that the process $i$ now will have a larger ticket value then all processes with a value. This as process $i$ has left the critical section and would need to get a new value to enter again. It also makes sure that all relationships in process $i$ are set to undefined so they will be set to correct values the next time the process tries to enter the critical section. In figure 4.1 a finite state system of the created model is shown. For the transitions the number of the rule is put instead of the whole rule as this would take to much space. In figure 4.2 a graphical representation of the induced transition system of size 2 is shown where $T_0$ is the ticket value in process 0 and $T_1$ is the ticket value in process 1. The first number is the state of process 0 and the second number is the state of process 1.
Figure 4.1: Finite state system of the parameterized system model of Bakery
Figure 4.2: The transition system of size 2 for the Bakery protocol.
Chapter 5

Parameterized Verification of Infinite-state Systems

In this chapter we will present the changes to the verification procedure presented in chapter 3 to allow for parameterized verification of infinite-state systems. First a revised definition of the reachability problem will be defined. Then the modification for each part of the verification procedure will be presented.

5.1 Subword Relation for Unbounded Parameterized Systems

In the following we extend the definition of the subword relation to configurations of unbounded parameterized systems.

Let \( c = (w, r) \) and \( c' = (w', r') \) be configurations. \( c' \subseteq c \) iff there exist a function \( h : \{1, 2, \ldots, |w'|\} \rightarrow \{1, 2, \ldots, |w|\} \) such that: \( w'[i] = w[h(i)], \forall i \in \{1, 2, \ldots, |w'|\} \) and \( r'[i, j] = r[h(i), h(j)], \forall i, j \in \{1, 2, \ldots, |w'|\} \).

5.2 Problem Definition

In this section, we give a revised definition of the reachability problem.

An instance of the reachability problem is defined by a unbounded parameterized system \( P' = (Q, V, \Delta) \), a regular set \( I \subseteq (Q^+ \times V) \) of initial configurations, and a set \( Bad \subseteq (Q^+ \times V) \) of bad configurations. We assume that \( Bad \) is the upward closure of a given finite set \( B \subseteq (Q^+ \times V) \) of minimal bad configurations, i.e. \( Bad = \{c | \exists b \in B : b \subseteq c\} \). Let \( T = (Conf, \rightarrow) \) be the transition system induced by \( P' \) as defined in the previous chapter. We say that \( c \in Conf \) is reachable iff there is a sequence of configurations \( c_0, \ldots, c_l \in Conf \) such that \( c_0 \in I, c_l = c \) and \( c_i \rightarrow c_{i+1} \) for all \( 0 \leq i \leq l \). We use \( R \) to denote the set of all reachable configurations.
configurations. We say that the system $P'$ is safe w.r.t. $I$ and $Bad$ if no bad configuration is reachable i.e. $R \cap Bad = \emptyset$.

5.3 Verification Procedure

5.3.1 View

Here a revised definition of a view is presented.

A view is an abstract representation of a set of configurations. A view $v$ will be represented by its unique minimal configuration $v_c$ w.r.t. subword relation but represent the set $\{v \mid v \subseteq c, c \in Conf\}$. When a view is used in the verification procedure it will be handled as $v_c$ and not as the set it represents.

5.3.2 Abstraction Function ($\alpha'$)

Here a revised definition of the abstraction function is presented.

Let $Conf_k = \{(w, r) \in Conf | \|w\| \leq k\}$. The abstraction function $\alpha'_k : Conf \rightarrow 2^{Conf_k}$ maps a configuration $c$ into the set $\alpha'_k(c) = \{v \in Conf_k | v \subseteq c\}$ of all its views (subwords) of size up to $k$. Finally, we can easily extend the abstraction function to a set of configurations as follows: Given a set $S \subseteq Conf, \alpha'_k(S) = \{\alpha'_k(c) | c \in S\}$.

Given a view $v = (w', r')$ with process $i$ abstracted away from configuration $c = (w, r)$, $r'$ is created by removing column $i$ and row $i$ from $r$.

5.3.3 Concretization Function ($\gamma'$)

Here a revised definition of the concretization function is presented.

For every $k \in \mathbb{N}$, the concretization function $\gamma'_k : 2^{Conf_k} \rightarrow 2^{Conf}$ inputs a set of views $V \subseteq Conf_k$, and returns the set of configurations that can be reconstructed from the views in $V$. In other words $\gamma'_k(V) = \{c \in Conf | \alpha'_k(c) \subseteq V\}$.

5.3.4 Post-Image ($post'$)

Here a revised definition of the post-image function is presented.

We define the post-image of a set $X \subseteq Conf$ to be the set $post'(X) := \{c' | c \rightarrow c' \land c \in X\}$. 
5.3.5 Abstract Post-Image ($Apost'_k$)

Here a revised definition of the abstract post-image function is presented.

The abstract post-image of a set of views $V \subseteq Conf_k$ is defined as $Apost'_k(V) = \alpha'_k(post'(\gamma'_k(V)))$.

5.3.6 Verification Procedure

Here a revised definition of the verification procedure is presented.

<table>
<thead>
<tr>
<th>Algorithm 7: Verification Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for $k = 1$ to $\infty$ do</td>
</tr>
<tr>
<td>2   if $R_k \cap \text{Bad} \neq \emptyset$ then</td>
</tr>
<tr>
<td>3       return Unsafe</td>
</tr>
<tr>
<td>4   end</td>
</tr>
<tr>
<td>5 $V := \mu X.\alpha'_k(I) \cup Apost'_k(X)$</td>
</tr>
<tr>
<td>6   if $\gamma'_k(V) \cap \text{Bad} = \emptyset$ then</td>
</tr>
<tr>
<td>7       return Safe</td>
</tr>
<tr>
<td>8   end</td>
</tr>
<tr>
<td>9 end</td>
</tr>
</tbody>
</table>

Because of the change in the presented functions termination of the verification procedure can no longer be guaranteed but termination was achieved for all tested protocols.
Chapter 6
Experimental section

In this chapter the results of the experiments are first presented and then discussed.

6.1 Numerical results

To get the following times algorithm 7 described in chapter 5 were implemented using the programming language C[8]. The following times were recorded when running the program in 64-bit Windows 7 using Cygwin[9] on a 3.30 GHz 6 core AMD processor with 8 GB of RAM. The prototype is not optimized and not made to run on more than one core. The formulation of the algorithms can be seen in Appendix A. The first column gives the name of the tested algorithm, the second column gives the run time for each algorithm in seconds, the third column gives the value of $k$, current size of the system, when the procedure terminated and the fourth column has a check mark if the tested algorithm were considered safe and a cross if the tested algorithm were considered unsafe.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time(Sec)</th>
<th>Size when stopped</th>
<th>Safe?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burns</td>
<td>1.54</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>Bakery</td>
<td>0.05</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>Simple Mutual Exclusion</td>
<td>0.02</td>
<td>2</td>
<td>✓</td>
</tr>
</tbody>
</table>

6.2 Result Discussion

Our tool can verify the correctness of all the algorithms in the used benchmark in few seconds while the procedure of algorithm 7 stops after at most two iterations ($k = 2$) for all the examples. The running times for the used examples are obtained for an unoptimized implementation of the verification procedure described. The Burns algorithm takes the longest to verify, this is because this algorithm leads to the most unique configurations that can be reached. This conclusion is not hard to draw by just looking at algorithm as the algorithm
contains 7 process states and 9 rules, where Bakery contains 4 process states and 4 rules and Simple Mutual Exclusion 3 process states and 3 rules.

The Simple Mutual Exclusion algorithm is a simpler version of the Bakery algorithm. It will behave in the same way with the only difference that it cannot model two processes getting the same ticket value, this is something that can happen in Bakery. Because of this it is easy to come to the conclusion that Simple should be faster to verify which is also shown by the results. One more thing that can be seen by doing these tests is that Burns that does not use integer variables can be tested. Burns uses Boolean variables as presented in chapter [2]. To see how this is modeled in the extension see Appendix [A]. This is also something that can be modeled using process states and can be verified using the procedure presented in [4] but was tested here to see if it could also work with this model.
Chapter 7

Extension to protocols with multiple variables

In this thesis we have presented a framework to handle protocols where each process is allowed to have at most one variable. In this chapter we will discuss how to modify our modeling part in order to be able to handle the general case where each process can handle multiple variables. We recall first the abstraction and its extension that we have used in chapter 4.4 and which consists of a matrix which size depends on the number of current processes. This is trivially extended to keep a matrix which size depends on the number variables in the system. Furthermore this matrix keeps the relationships between each pair of variables in the system using the domain \{\langle,\rangle,=,-\}. Then we present its disadvantages. Finally we present our second abstraction which we believe to provide a better precision and scalability.

7.1 Abstraction 1

Given a set of variables for all processes, \(V_{\text{ars}}\), this abstraction will consist of a \(|V_{\text{ars}}| \times |V_{\text{ars}}|\) matrix, \(R\), where \(R[i,j] = R\), where \(R\) comes from \(v_iRv_j\), \(R \in \{\langle,\rangle,=,-\}\). The \(-\) relation means that there is no relationship defined, this could happen if one or both of the variables in the relation are undefined. This means that if \(R[i,j] = \langle'\rangle\) then \(R[j,i] = \rangle'\) and vice-versa. If \(R[i,j] = '='\) then \(R[j,i] = '='\). This abstraction will then keep track of the relationship between each pair of variables \(v_i,v_j\), where \(0 \leq i,j < |V_{\text{ars}}|\). Each matrix here will represent an infinite amount of value assignments, a matrix will represent those assignments that fulfill all relationships given by the matrix.

As a consequence of this, the diagonal will always contain only \('=\)' if the variable has a value, as \(R[i,i] = '='\) because a variable can not have a different value from itself. By doing this there is a finite set of views, here the matrices together with the process states. This holds as long as the amount of variables is
a finite number which should be the case for any protocol. This means that the verification procedure can be run on these matrices together with the process states as they will represent all the value assignments that can happen. One problem would be if the protocol relies on using concrete values to work. Below follow some examples on how these matrices could look.

$$\begin{bmatrix}
= & < \\
> & =
\end{bmatrix}$$

Figure 7.1: Example 1

Figure 7.1 is just a simple example with 2 variables, \( v_0 \) and \( v_1 \). The matrix gives the order \( v_1 > v_0 \) because \( R[0,1] = ') \) and \( R[1,0] = ' \). This matrix now represents all value assignments where \( v_1 > v_0 \). A few examples are \( \{ v_0 = 100, v_1 = 1000 \} \), \( \{ v_0 = -4, v_1 = 4 \} \) and \( \{ v_0 = -33, v_1 = -32 \} \).

$$\begin{bmatrix}
= & > & < & < \\
< & = & < & < \\
> & > & = & > \\
> & > & < & =
\end{bmatrix}$$

Figure 7.2: Example 2

Figure 7.2 is a bit more complicated example with 4 variables, \( v_0, v_1, v_2 \) and \( v_3 \). The matrix gives the order \( v_2 > v_3 > v_0 > v_1 \). This matrix now represents all value assignments where \( v_2 > v_3 > v_0 > v_1 \). A few examples of configurations satisfying this view are \( \{ v_0 = 100, v_1 = 10, v_2 = 102, v_3 = 101 \} \), \( \{ v_0 = -1, v_1 = -6, v_2 = 62, v_3 = 16 \} \) and \( \{ v_0 = -46, v_1 = -54, v_2 = -13, v_3 = -33 \} \).

$$\begin{bmatrix}
= & < & > & = & < & > \\
> & = & > & = & > \\
< & < & = & < & < & > \\
= & < & > & = & < & > \\
> & = & > & > & = & > \\
< & < & < & < & < & =
\end{bmatrix}$$

Figure 7.3: Example 3

As can be seen in Figure 7.3 the relationships in this matrix gives the order of the variables are as follows: \( v_1, v_4 > v_0, v_3 > v_2 > v_5 \), where \( v_1 = v_4 \) and \( v_0 = v_3 \). This matrix now represents all value assignments to the variables that fulfill this ordering. A few examples are \( \{ v_0 = 67, v_1 = 100, v_2 = 5, v_3 = 67, v_4 = 100, v_5 = -3 \} \), \( \{ v_0 = -5, v_1 = 1, v_2 = -35, v_3 = -5, v_4 = 1, v_5 = -39 \} \) and \( \{ v_0 = 95, v_1 = 111, v_2 = 3, v_3 = 95, v_4 = 111, v_5 = 0 \} \).
7.1.1 Advantages

The advantage with this abstraction is that it solves the problem of the infinite aspect. It will keep track of all relationships between variables so all algorithms that do not need specific values on the variables should be able to be verified.

7.1.2 Disadvantages

The main disadvantage with this abstraction is that it has relationships that are not needed. This as the algorithm normally only need to know specific relationships to work. This leads to the verification procedure being less accurate because the $\alpha$ function will remove relationships that can not be reconstructed from the parts that are created, this happens when the amount of variables is dependent on the size of the system. This means that $\gamma$ will not be able to reconstruct the exact configuration that the views that are used come from. This problem comes from the fact that the information that is needed to create a larger configuration will not be present. This is because all information regarding the extra variables is removed when the configuration is divided. This problem is only present if the amount of variables is dependent on the amount of processes in the system. To look at an example, say each process in the algorithm will have $n$ variables, where $n$ is the current size of the system.

This will then mean that if the current size of the system is 2 then the matrix would be 4 x 4, as each process would have 2 variables, which is 16 relationships but if we increase the size of the system to 3 then the matrix would be 9 x 9 which is 81 relationships. As $\gamma$ takes views of size $k$ and builds configurations of size $k + 1$ this would mean in this example that $\gamma$ would take three 4 x 4 matrices and try to create one 9 x 9 matrix, as three 4 x 4 give at most 48 relationships, some will overlap, and we need 81 relationships there were some relationships that were forgotten when the $\alpha$ function was performed. These would be the relationships were the extra variables are involved. This means that $\gamma$ can not recreate the configuration that created the views to start with which leads to that the $\gamma$ function will be inaccurate.

If we take a concrete example, in Figure 7.4, Figure 7.5 and Figure 7.6 we can see three views that could be created when running the verification procedure on a protocol. Each process has an amount of variables equal to the size of the system and we have two processes represented in the views. View 1 represents the pair $P_0P_1$, view 2 represents the pair $P_1P_2$ and view 3 represents $P_0P_2$. In Figure 7.7 we can see what we get when we combine the three views and try to create a configuration, all places that has a ‘-’ symbol has no information to get from the views. In the example when the views are combined, each process will get a new variable so $P_0$ will have $v_0$ and $v_1$ first and then get $v_2$ which means that $P_1$ which had $v_2$ and $v_3$ will be moved and now have $v_3, v_4$ and $v_5$. The same is true for $P_2$ it had $v_4$ and $v_5$ and get $v_6, v_7$ and $v_8$. This conversion is what is used to determine where in the bigger matrix that the old relationships
will end up. Because of this problem the second abstraction was thought out to handle this problem.

\[
\begin{bmatrix}
= & > & > & > \\
< & = & < & < \\
< & > & = & < \\
< & > & > & =
\end{bmatrix}
\]

Figure 7.4: View 1, \(v_1 < v_2 < v_3 < v_0\)

\[
\begin{bmatrix}
= & < & < & < \\
> & = & > & > \\
> & < & = & < \\
> & < & > & =
\end{bmatrix}
\]

Figure 7.5: View 2, \(v_2 < v_4 < v_5 < v_3\)

\[
\begin{bmatrix}
= & > & > & > \\
< & = & < & < \\
< & > & = & < \\
< & > & > & =
\end{bmatrix}
\]

Figure 7.6: View 3, \(v_1 < v_4 < v_5 < v_0\)

\[
\begin{bmatrix}
= & > & - & > & - & > & > & - \\
< & = & - & < & - & < & < & - \\
< & > & - & = & < & - & < & - \\
< & > & - & > & = & - & > & - \\
< & > & - & > & < & - & = & < \\
< & > & - & > & < & - & > & = \\
- & - & - & - & - & - & - & -
\end{bmatrix}
\]

Figure 7.7: Result of \(\gamma_3\) function
7.2 Abstraction 2

This abstraction works in the same way as abstraction 1 in that it will only save relationships between variables and not any actual values. The main difference between this abstraction and abstraction 1 is that this abstraction will only save relationships that are defined in the model of the system and not all available relationships between variables. This means that if the model is good then only the needed relationships are saved. This will create a matrix that will be \( k \times n \), where \( k \) is the amount of relationships saved from each process and \( n \) is the number of processes in the system. What relationships are saved in each spot will depend on the algorithm being modeled but an example could be that it has the relationships to one variable that is used to make decisions in the algorithm as these relationships most likely are important. Below are some examples.

\[
\begin{bmatrix}
= & < \\
> & =
\end{bmatrix}
\]

Figure 7.8: Example 1

Figure 7.8 could for example represent that in \( p_0 \) \( v_0 \) is the biggest, i.e. bigger then \( v_2 \). And in \( p_1 \) \( v_2 \) is the smallest, i.e. smaller then \( v_3 \). This example would then assume that each process has at least two variables.

\[
\begin{bmatrix}
= & > & > \\
< & = & > \\
< & < & =
\end{bmatrix}
\]

Figure 7.9: Example 2

Figure 7.9 could for example represent that in \( p_0 \) \( v_0 \) is the smallest, in \( p_1 \) \( v_1 \) is bigger then \( v_0 \) and smaller then \( v_2 \) and in \( p_2 \) \( v_2 \) is biggest. So each process will have the relation to one variable in the other processes.

\[
\begin{bmatrix}
= & > & > \\
< & = & > \\
< & < & < \\
= & < & < \\
> & = & < \\
> & > & =
\end{bmatrix}
\]

Figure 7.10: Example 3

Figure 7.10 could for example represent that in \( p_0 \) \( v_0 \) is the smallest, in \( p_1 \) \( v_1 \) is bigger then \( v_0 \) and smaller then \( v_2 \) and in \( p_2 \) \( v_2 \) is biggest. And that in \( p_0 \) \( v_3 \) is the biggest in \( p_1 \) \( v_4 \) is smaller then \( v_3 \) and bigger then \( v_5 \) and in \( p_2 \) \( v_5 \) is smallest.
In this example each process has at least 2 variables and the relations that are important are the order of $v_0, v_1$ and $v_2$ as well as $v_3, v_4$ and $v_5$. Where $v_0$ and $v_3$ belong to process 0, $v_1$ and $v_4$ belong to process 1 and $v_2$ and $v_5$ belong to process 2.

### 7.2.1 Advantages

The main advantage with using the abstraction is that only the important relationships are saved. This means that one can use abstraction 2 even though the amount of variables for each process are dependent on the size of the system because not all relationships are saved. So the example that was given in 7.1.2 with each process having $n$ variables where $n$ is the size of the system could be modeled using $n$ relationships for each process which means $n \times n$ relationships instead of the $n^2 \times n^2$ relationships that were present in abstraction 1. This is if the relationships that are saved are the relationships to one variable in each process. Below is an example of a model that saves $n$ relations per process and how three views are combined into a configuration. View 1 is considered as the pair $P_0P_1$, view 2 is considered as the pair $P_0P_2$ and view 3 is considered as the pair $P_1P_2$. The relationships saved are $v_iRv_j$ where $i$ is the row and $j$ is the column.

\[
\begin{bmatrix}
= & > \\
< & = 
\end{bmatrix}
\]

Figure 7.11: View 1, $v_1 < v_0$

\[
\begin{bmatrix}
= & < \\
> & = 
\end{bmatrix}
\]

Figure 7.12: View 2, $v_0 < v_2$

\[
\begin{bmatrix}
= & < \\
> & = 
\end{bmatrix}
\]

Figure 7.13: View 3, $v_1 < v_2$

\[
\begin{bmatrix}
= & > & < \\
< & = & < \\
> & > & = 
\end{bmatrix}
\]

Figure 7.14: Result of $\gamma_3$ function, $v_1 < v_0 < v_2$
7.2.2 Disadvantages

By not keeping track of each pair of variables this abstraction will need more work to use. This as the algorithm being verified needs to have certain relationships and these need to be identified so that there is room for them in the matrix. And what relationships is where will also be needed to be kept track of by the rules that are applied when running the verification procedure. This means that the amount of work needed to use this abstraction is greater then abstraction 1 as in abstraction 1 all possible relationships were saved in specific spots.
Chapter 8

Related Work

The field of parameterized verification is quite large. A lot of work has been done into regular model checking, e.g. [10, 11], and into augmenting regular model checking with techniques. Some of these are widening [12, 13] and abstraction [14]. Widening is the technique used to create upper approximations when working with regular model checking. Abstraction works by looking at abstract systems that fulfill the same predicates as the concrete one. This abstract system is then used as the abstract system will fulfill all properties that the concrete system does. If a property is found that is only fulfilled by the abstract system then the abstraction is refined to not fulfill this property. All these techniques work by iterating transducers of regular languages or computing the transitive closure of transducers, the procedure presented here is much simpler. A technique that is interesting for parameterized systems is counter abstraction. This idea is to keep track of how many processes that fulfill a certain property [15, 16, 17, 18, 19]. This technique is in general designed for systems with clique or unstructured architectures. There are also some techniques that reduce the verification of a parameterized system to the verification of finite-state models. Among these techniques some [20, 21, 22] exploit cut-off properties to check invariants for mutual exclusion protocols.

In [23] the authors use predicate abstraction, which abstracts away the exact contents of the memory and just keeps track of if certain properties hold or not. They combine this with counter abstraction which abstracts away the ordering of the processes just keeping track of how many processes are in each state. This technique is called environment abstraction. In [24] a model is presented that contains a more restricted form of global conditions and less features then the model present in [4]. A recent technique that has been introduced is monotonic abstraction [24]. This technique combines regular model checking with abstraction. A model that is similar to [4] can be found in [25, 26, 27] but it targets only well-quasi ordered systems which means the scope of systems that can be verified is smaller.
The method found in [28] is close to [4] but the class of systems considered basically corresponds to Petri nets. This means that it can not deal with systems that have a linear or tree-like topology. The method also works by finding a dynamic cut-off point as is done in [4]. To verify, [28] works by performing a backward reachability analysis on the underlying transition system.
Chapter 9

Conclusion

We have extended the techniques presented in [4] to handle parameterized system where each process can have a variables over unbounded data domain. We have first proposed a formal model (called unbounded parameterizes systems) to model this new class of protocols. Then, we have extended the verification algorithm proposed in [4] in order to work for this new model. Finally, we have implemented a prototype in C [8]. We have run our prototype successfully on a certain number of non-complicated examples.

As future work we plan to slightly extend our formal model with the Abstraction 2 proposed in chapter 6 in order to handle programs with multiple variables. Then the second step is to extend our prototype with this future and run it on more complicated examples. One example of this would be the Lamport Distributed Mutex [29] algorithm which is quite complicated but we believe that our technique will succeed to handle it. This requires to extend our current model with some new rules in order to be able to model the Lamport Distributed Mutex algorithm. Some other algorithms that can not be handled by the current work are Dijkstra [30] and Szymanski [7]. For these examples we need to add more rules that are using existential and universal quantifiers over more than one relationship (e.g., a Boolean combination of several relations).
Bibliography


Appendix A

Algorithms

A.1 Bakery

The Bakery algorithm\textsuperscript{2} is quite simple. It uses one variable for each process part of the system this variable will contain the ticket number for that process. The algorithm works like follows, where process $i$ is the process performing the step, $n$ is the size of the system and CS = critical section:

- 1.(Go in queue for CS) Set process $i$’s variable to be strictly greater than all the variable for process 0-$(i - 1)$ and process $(i + 1) - n$.
- 2.(Enter the CS) If the variable of process $i$ is less or equal than that of process 0-$(i - 1)$ and less then process $(i + 1) - n$ then process $i$ is allowed to enter the CS.
- 3.(Leaving the CS) When process $i$ is done in the critical section it will set its variable to 0.

Below will follow how Bakery is defined in the extension described in this thesis where each process will save $n$ relationships, where $n$ is the size of the system. 4 corresponds to CS:

\[
P' = (Q, V, \Delta)
\]
\[
Q = \{1, 2, 3, 4\}
\]
\[
V = \{(\leq), (\geq), (\leq\leq), (\geq\geq), ..., \}
\]

And $\Delta$ contains the following rules, where $i$ is the index of the processes performing the rule:

- $1 \Rightarrow 2; \forall j, R_{ij} \leftarrow \rightarrow$, where $P_j \in \{3, 4\}; \forall j, R_{ij} \leftarrow =$, where $P_j \in \{2\}; \forall j, R_{ji} \leftarrow =$, where $P_j \in \{2\}$
• $2 \Rightarrow 3; \forall j, R_{ij} \leftarrow <$, where $P_j \in \{1\}$

• if($\forall j, R_{ij} = <$, where $j < i \land \forall j, R_{ij} = \leq$, where $j > i$) then $(3 \Rightarrow 4)$

• $4 \Rightarrow 1; \forall j, R_{ij} \leftarrow -$, where $j \neq i; \forall j, R_{ji} \leftarrow <$, where $P_j \in \{3, 4\}; \forall j, R_{ji} \leftarrow -$, where $P_j \in \{2\}; R_{ii} \leftarrow -$

A.2 Burns

The Burns algorithm[1] is a bit more complicated and uses one boolean variable, $b$, and the state of the process. The algorithm works like follows, where process $i$ is the process performing the step, $n$ is the size of the system and $b_i$ means the boolean variable belonging to variable $i$:

• 1. Go from state 1 to state 2 and set $b_i$ to false.
• 2. Go from state 2 to state 1 if there is a $b_k$ that is true where $k \in \{1, 2, 3, \ldots, i-1\}$.
• 3. Go from state 2 to state 3 if all $b_k$ are false where $k \in \{1, 2, 3, \ldots, i-1\}$.
• 4. Go from state 3 to state 4 and set $b_i$ to true.
• 5. Go from state 4 to state 1 if there is a $b_k$ that is true where $k \in \{1, 2, 3, \ldots, i-1\}$.
• 6. Go from state 4 to state 5 if all $b_k$ are false where $k \in \{1, 2, 3, \ldots, i-1\}$.
• 7. Go from state 5 to state 6 if all $b_k$ are false where $k \in \{i+1, i+2, i+3, \ldots, n\}$.
• 8. Go from state 6 to state 7 and set $b_i$ to false.
• 9. Go from state 7 to state 1.

Here state 6 represents the critical section.

Below will follow how Burns is defined in the extension described in this thesis. Since Burns works on boolean values the 2 relationships are used to represent true and false, $<$ represents false and $>$ represents true. 6 corresponds to CS:

$$P' = (Q, V, \Delta)$$

$$Q = \{1, 2, 3, 4, 5, 6, 7\}$$

$$V = \{( < ), ( > ), \ldots \}$$

And $\Delta$ contains the following rules, where $i$ is the index of the processes performing the rule:
A.3 Simple Mutual Exclusion

This protocol is a simple mutual exclusion protocol. It is quite similar to Bakery, the main difference is that this does not model two processes getting the same ticket value which can happen in Bakery. So the protocol would be the same as for Bakery but with the added condition that there are locks to make sure only one process can get a ticket value at a time.

Below will follow how Simple Mutual Exclusion is defined in the extension described in this thesis where each process will save $n$ relationships, where $n$ is the size of the system. 4 corresponds to CS:

\[ P' = (Q, V, \Delta) \]

\[ Q = \{1, 2, 3\} \]

\[ V = \{(<), (>), (=), (\preceq), (\preceq\sim), (\preceq\succeq), (\preceq\gg)\ldots\} \]

And $\Delta$ contains the following rules, where $i$ is the index of the processes performing the rule:

- $1 \Rightarrow 2; \forall j, R_{ij} \leftrightarrow$, where $P_j \in \{2, 3\}$; $\forall j, R_{ij} \leftarrow=$, where $P_j \in \{1\}$; $R_{ii} \leftarrow$
- if($\forall j, R_{ij} =<, \text{where } i \neq j$)then($2 \Rightarrow 3$)
- $3 \Rightarrow 1; \forall j, R_{ij} \leftarrow=$, where $i \neq j$; $\forall j, R_{ji} \leftarrow<$, where $P_j \in \{2\}$; $R_{ii} \leftarrow$