DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Approaching Safety for Parameterized Systems via View Abstraction

Philip Offtermatt

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Approaching Safety for Parameterized Systems via View Abstraction

Ein Ansatz zum Nachweis von Sicherheitseigenschaften für parametrisierte Systeme durch Zustandsabstraktion

Author:	Philip Offtermatt
Supervisor:	Univ.-Prof. Dr. Dr. h.c. Javier Esparza
Advisor:	M.Sc. Christoph Welzel
Submission Date:	15.11.2019

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, .11.2019 Philip Offtermatt

Acknowledgments

I want to thank my advisor Christoph Welzel for his encouraging advice and all the time we spent discussing problems and brainstorming solutions on many monday afternoons.

Furthermore, I want to thank my supervisor Prof. Javier Esparza for stoking my interest in theoretical computer science, and for the opportunity to work on this thesis.

Abstract

Parameterized systems are models for interactions between many participants or processes. While there are infinitely many instances of the system, one for each number of interaction partners, the problem of verifying such a system becomes possible to tackle by viewing the system as a family of systems, parameterized by some variables. In this thesis, we examine one method that is able to verify safety properties for such infinite families. This method is called view abstraction, and it relies on an abstraction function, which enables us to extract patterns, which we call views, from the states of the system. These patterns are then used to compute an overapproximation of the reachable states of the system, together with a so-called small model property. This property means that small, fixed-size instances of the system exhibit all the patterns found in all instances of the system. The accuracy of the overapproximation is determined by a parameter, and when it becomes accurate enough, we can prove whether the system is safe without examining the whole state space. We present our reimplementation of this method. In two casestudies, we benchmark the new and original implementations, and apply the method to population protocols, an established subclass of parameterized systems. We examine how view abstraction can be used to compute a certain property of population protocols that can help humans make less errors while designing them, and showcase our prototype implementation of tool-assisted manual generation of population protocols.

Contents

1. Introduction

For many years, the number of computers and the interdependence between them has steadily increased. New technological paradigms like the Internet of Things [TW10; AIM10] are emerging. In these paradigms, not only humans but everyday objects like lamps or cars are connected with each other. This means more and more participants interact. One consequence of this development is that it is increasingly important to ensure that algorithms that govern systems with many interaction partners are thoroughly verified.

There are many examples of the fact that it is hard to write software that behaves as expected. For example, the crash of the Ariane 5 launcher in 1996 was due to an undetected software bug [JM97], and there are many more similar examples of comparable scope. Consequently, there is much literature dedicated to verifying programs and software, i.e. via model checking [BK08; Vis+03]. Extending this to distributed programs means that the problem becomes more complex. For example, the 2003 northeast blackout that left large parts of the eastern United States and Canada without power for over ten days was among other reasons the result of a software bug in a distributed system [And+05].

In this thesis, we focus on *safety* properties. This means we are given some initial and some bad states. Then, the question we want to answer is: Can the system reach a bad state when it starts at an initial state? Many interesting properties can be formulated as such safety properties. For example, the mutual exclusion problem means that a certain resource, for example write-access to a database entry, can only be used by one process at a time. We can view this problem as a safety property by denoting as bad states all those where two processes access the resource at the same time.

When we look at a system with only one process with finitely many states, we can exhaustively perform a forward reachability search starting at its initial states to determine whether the system can reach a bad state. However, when we are considering parameterized systems with arbitrarily many interacting processes, there are infinitely many instances of the system, one per number of processes.

Such a parameterized system can be viewed as a family of systems, characterized by some parameters. For example, a system could be a model of a network, where there is no bound on how many devices are part of the network, and the topology in which the devices interact is not a priori known. The parameters might then be the device number and the topology of the network. Naturally, for a fixed number of devices and one fixed topology, the instance of the system for these parameters has finitely many states. Still, exhaustive forward reachability over the system as a whole is bound to fail, since there are infinitely many instances that need to be checked. Therefore, we need other methods.

One promising approach is to infer properties of the full system by looking at the behaviour of instances for some fixed parameter value. In essence, we want to utilize a so-called *small model property*: Small instances of the system suffice to explain behaviour of all its instances.

This thesis is dedicated to studying a recently proposed method for parameterized verification of safety properties called *view abstraction*, that is given by Abdulla et al. in [AHH16]. It relies on an abstraction that maps larger instances of the system into their smaller patterns. The method then uses the abstraction to construct an overapproximation of the reachable configurations, thereby deriving an abstract state space. If we can prove the abstract model to be safe then the original model must also be safe, since that abstract state space is an overapproximation of the actual state space.

We present our implementation of the core part of the view abstraction algorithm. The original paper contained evaluations of a prototype implementation, that is however to our knowledge not made public under a license that allows reuse and modification. We use our implementation to verify some results from [AHH16].

Additionally, we use view abstraction to verify properties of a type of parameterized systems called population protocols, that were originally introduced by Angluin et al. in [Ang+06]. Population protocols, and especially their verification, have recently been examined in the literature. Additionally, view abstraction has not previously been applied to them, even though they satisfy the criteria for applicability. We introduce the consensus stability property, which is a property that can be helpful for humans to judge correctness of population protocols and spot errors. In this context, we present our implementation of a small prototype that provides capability for tool-assisted, manual generation of population protocols, and utilizes view abstraction to compute consensus stability and provide feedback to the user. It has recently been shown that automatically generating population protocols can be done in polynomial time, but the resulting protocols are very slow [Blo+19]. On the other hand, manual generation is prone to errors, but humans can produce fast protocols. This prototype seeks to support humans in this tedious task and can help them avoid errors.

The structure of the rest of this thesis is detailed in the following. Chapter 2 introduces notation and concepts that are used throughout the thesis. Chapter 3 is dedicated to presenting some other work in the area of parameterized systems and their verification. Chapter 4 introduces the view abstraction algorithm, as well as correctness and completeness results, and follows the structure of [AHH16] closely. Chapter 5 introduces population protocols. Chapter 6 is dedicated to presenting our prototype implementation for tool-assisted manual protocol generation and reproducing some of the benchmark results from [AHH16]. Lastly, Chapter 7 draws conclusions and describes opportunities for future work.

2. Preliminaries

In the following, $\mathbb{N}$ denotes the natural numbers including 0, while $\mathbb{Q}$ denotes the rational numbers.

2.1. Multisets

A *multiset* is given by a mapping $M : E \to \mathbb{N}$, where E is its finite underlying set. Intuitively, a multiset is a set that allows for multiple repetitions of the same element. The semantics are as follows: For an element $e \in E$, $M(e)$ denotes how often the element occurs in M. We call $M(e)$ the multiplicity of e in M. The *support* of a multiset M over a set E is given by $\{e \in E | M(e) > 0\}$, denoted as $[[M]]$.

The *size* of a multiset is given as the sum of the multiplicities of its elements, i.e. $\sum_{e \in E} M(e)$, denoted as $||M||$. For two multisets M, M', we define a *subset* relation: M is a subset of M' if for all $m \in [[M]]$, it holds that $m \in [[M']]$ and $M(m) \leq M'(m)$. Intuitively, M' needs to contain at least all elements that M contains, and they need to have at least the same (or greater) multiplicities.

We sometimes use the notation $\{|m_1, m_2, ...|\}$ to denote a multiset containing the elements $m_1, m_2, ...$, where the multiplicity of an element is how often the element occurs.

We define some useful standard operations on multisets:

- For two multisets M and M' over a set E, we define their union $U = M \cup M'$ as follows: For all $m \in E$, if $m \in [[M]] \cup [[M']]$, then $U(m) = M(m) + M'(m)$, otherwise $U(m) = 0$. Intuitively, we simply add together the multiplicities of all elements of both multisets. For example, $\{|0,1,2|\} \cup \{|0,1,3|\} = \{|0,0,1,1,2,3|\}$.

- For two multisets M and M' over a set E, we define the difference between M and M', denoted as $U = M \setminus M'$, as follows: For all $m \in E$, if $m \in [[M]]$, then it holds that $U(m) = max(M(m) - M'(m), 0)$, otherwise $U(m) = 0$. Intuitively, we subtract the multiplicites of the elements in M' from those in M. If this would yield a negative multiplicity, we simply set the multiplicity of the element to 0.

2.2. Words

An *alphabet* is given as a potentially infinite set of elements. We call the elements of an alphabet *letters*. A *word* over an alphabet $A = \{a_1, a_2, a_3, ...\}$ is given as a sequence

$w = w_1 w_2 w_3 ... w_n$ of letters of A. We sometimes use the notation $w[i]$ to refer to the i-th letter of w.

Note that in the context of this thesis, words consist only of a finite number of letters. We call the number of letters in the sequence of a word w the length of w, denoted by $|w|$. We denote the *frequency* of a letter l in w as $|w|_l$, and formally define $|w|_l = |\{n \in \mathbb{N} | w_n = l\}|$. The empty word is denoted by ϵ and has a length of 0.

In the following, we define two relations over words. Both relations describe that a word is a subword of another word, but their semantics differ slightly.

For two words $w = w_1 w_2 ... w_n, v = v_1 v_2 ... v_m$ we define the *ordered subword relation* $\sqsubseteq$ as follows: $w \sqsubseteq v$ if there exist $i_1 < i_2 < i_3 ... < i_n$ s.t. $w = v_{i_1} v_{i_2} v_{i_3} ... v_{i_n}$. Intuitively, the ordered subwords of a word are those that are derived by deleting some letters from it, while keeping the other letters in the same order. For example, the non-empty subwords of the word *aba* are $\{a, b, aa, ab, ba, aba\}$.

In contrast to the ordered subword relation, we define the *permuted subword relation* $\trianglelefteq$ as follows: For two words $w = w_1 w_2 ... w_n, v = v_1 v_2 ... v_m$ over an alphabet A, $w \trianglelefteq v$ if and only if for all letters $l \in A$, $|w|_l \leq |v|_l$. Intuitively, w is a permuted subword of v if it can be constructed from v by leaving zero or more letters out, and reordering the remaining letters. For example, the permuted subwords of the word *aba* are $\{a, b, aa, ab, ba, aba, aab, baa\}$.

Next, to go along with the permuted subword relation, we also define what it means for two words to be *equal under permutation*. We denote this relation as $\sim$. For two words $w = w_1 w_2 w_3 \ldots, v = v_1 v_2 v_3 \ldots$ over an alphabet A, $w \sim v$ if and only if for all letters $l \in A, |w|_l = |v|_l$. Intuitively, w is equal to v under permutation if w can be constructed from v by only reordering, not adding or removing, letters.

2.3. Regular Expressions

We define regular expressions as a way of reasoning about words. The following definitions are adapted from [Nip+19]. For an alphabet A, we define the syntax of regular expressions over A inductively:

- $\varnothing$ and ϵ are regular expressions,
- for all $a \in A$, a is a regular expression, and
- for two regular expressions α, β, then also
  - $\alpha | \beta$,
  - $\alpha\beta$,
  - α^*, and
  - α^+

  are regular expressions.

A regular expression defines a language L as follows:

- $L(\varnothing) = \varnothing$,
- $L(\epsilon) = \{\epsilon\}$,
- $L(a) = \{a\}$,
- $L(\alpha\beta) = L(\alpha)L(\beta)$ (i.e. concatenate words from $L(\alpha)$ and $L(\beta)$),
- $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$,
- $L(\alpha^*) = L(\alpha)^*$, and
- $L(\alpha^+) = L(\alpha\alpha^*)$.

As an example, consider the alphabet $A = \{a, b\}$. The regular expression for all possible words over the alphabet is given as $(a|b)^*$. As another example, consider $L((aa|b)(ab)^*) = \{aaab, bab, aaabab, babab, \dots\}$.

2.4. Well-Quasi-Orderings

Well-quasi-orderings describe a useful subclass of relations. They have been studied extensively in the literature, and a short overview of their history and the basic definitions can be found in [Kru72]. In the following, we introduce the definitions and notations as they are used throughout this thesis.

Let $\preccurlyeq \subseteq Q \times Q$ be a relation over elements of some potentially infinite set Q. For $a, b \in Q$, if $(a, b) \in \preccurlyeq$, we can denote this fact alternatively as $a \preccurlyeq b$. We define two properties of relations:

- *Reflexivity*: $\preccurlyeq$ is reflexive if and only if for all $x \in Q$ it holds that $x \preccurlyeq x$.
- *Transitivity*: $\preccurlyeq$ is transitive if and only if for all $x, y, z \in Q$, it holds that if $x \preccurlyeq y$ and $y \preccurlyeq z$, then also $x \preccurlyeq z$.

There are some examples for relations over the natural numbers $\mathbb{N}$ that fulfill neither, one, or both of the properties. Consider for example the relation $<$. This relation satsifies transitivity - if $x < y$ and $y < z$, then naturally it must also hold that $x < z$. On ther other hand, $x < x$ never holds. On the other hand, the relation $=$ on the natural numbers satisfies both transitivity as well as reflexivity. Observe that $x = x$ holds for any natural number, and if $x = y$ and $y = z$, then naturally, $x = z$ must also hold. Note that the same holds for the "less-or-equal" relation $\leq$. An example of a relation that satisfies neither reflexivity nor transitivity is $\neq$. Reflexivitiy does not hold since $x \neq x$ does not hold for any natural number x. Transitivity does not hold either - consider the case where we choose three numbers x, y, z such that $x = z$, but $x \neq y$ and $y \neq z$. In order for $\neq$ to be transitive, $x \neq z$ has to hold, but we chose numbers such that $x = z$, therefore, $\neq$ cannot be transitive.

A *quasi-ordering* is a tuple $(Q, \preccurlyeq)$, where

- Q is a potentially infinite set of elements, and
- $\preccurlyeq \subseteq Q \times Q$ is a reflexive and transitive relation over Q.

Some examples of quasi-orderings are given by the natural numbers under equality $(\mathbb{N}, =)$, the natural numbers under "less-or-equal" $(\mathbb{N}, \leq)$ or the rational numbers under "less-or-equal" $(\mathbb{Q}, \leq)$.

For two elements $a, b \in Q$, the pair (a, b) is called *increasing* if $a \preccurlyeq b$, and it is called *strictly increasing* if additionally $b \preccurlyeq a$ does not hold. We also say that a is *smaller* than b with respect to $\preccurlyeq$. One can define *decreasing*, *strictly decreasing* and *greater* similarly. The set of *minimal elements* of Q is defined as $\{b \in Q|$ There is no $e \in Q$ such that $e \preccurlyeq b$, but not $b \preccurlyeq e\}$, i.e. there is no element e such that (e, q) is strictly increasing. The set of *maximal elements* can be defined analogously. To illustrate the concepts of minimal and maximal elements, we consider the previously given examples of quasi-orderings. The quasi-order $(\mathbb{N}, =)$ has infinitely many minimal and maximal elements, since any given element is not equal to any element except for itself. Therefore, every element is minimal as well as maximal. If we replace $=$ by $\leq$ and consider the case of $(\mathbb{N}, \leq)$, there exists no maximal element. Intuitively, the natural numbers do not have any upper bound, and therefore no element can be maximal. However, there is a unique minimal element, namely 0. If we consider the rational numbers instead of the natural numbers, i.e. $(\mathbb{Q}, \leq)$, no maximal or minimal elements exist - the rational numbers have neither an upper bound nor a lower bound.

Given an arbitrary set $S \subseteq Q$, we can also identify minimal resp. maximal elements of S by simply swapping Q for S in the definition: The set of minimal elements of S is given as $\{b \in S|$ There is no $e \in S$ such that $e \preccurlyeq b$ but not $b \preccurlyeq e\}$. We can similarly define the set of maximal elements. For example, even though $(\mathbb{Q}, \leq)$ has no minimal or maximal elements, if we consider the set $S = \{\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \dots\}$, we can identify the maximal element 1, but there are still no minimal elements.

For a set S and a quasi-order $(Q, \preccurlyeq)$, we denote by $\uparrow S$ the *upward-closure* of S, defined as $\{q \in Q|$ There exists $s \in S$ such that $s \preccurlyeq q\}$. The *downward-closure* can be defined analogously as $\downarrow S = \{q \in Q|$ There exists $s \in S$ such that $q \preccurlyeq s\}$. When $S = \downarrow S$ (resp. $S = \uparrow S$), we say that S is *upward-closed* (resp. *downward-closed*). As an example, consider again the quasi-order $(\mathbb{N}, \leq)$. An example of a set that is downward-closed, but not upward-closed are intervals starting at 0, i.e. the set $\{0, 1, ..., n\}$ is downward-closed for every n. On the other hand, infinite intervals starting at some number n and containing all larger numbers, e.g. the infinite set $\{n, n+1, n+2, ...\}$, are upward closed.

A quasi-order $(Q, \preccurlyeq)$ is called a *well-quasi-order* if every infinite sequence $a_0, a_1, a_2, ...$ contains an infinite subsequence $a_{i_0}, a_{i_1}, a_{i_2}, ...$ with $i_0 < i_1 < i_2 < ...$ such that $a_{i_0} \preccurlyeq a_{i_1} \preccurlyeq a_{i_2} \preccurlyeq ....$ Another equivalent definition says that a quasi-order $(Q, \preccurlyeq)$ is a *well-quasi-order* if for every $P \subseteq Q$ there is a finite set of minimal elements $P' \subseteq P$ such that $P = \uparrow P'$. Of the presented examples, only $(\mathbb{N}, \leq)$ is a well-quasi-order. Intuitively, any infinite sequence is not decreasing forever, since it reaches the minimal element 0 from which it cannot decrease further, and therefore has to contain an infinite ascending subsequence. Note that since $x \leq x$, this can also be an infinite sequence that consists of the same element repeated infinitely often. The other two examples, namely $(\mathbb{N}, =)$ and $(\mathbb{Q}, \leq)$ are not well-quasi-orders. For $(\mathbb{N}, =)$, we consider as a counterexample any infinite sequence of inequal elements, i.e. $0, 1, 2, ...$, while for $(\mathbb{Q}, \leq)$, we consider the sequence $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, ...$, where every pair of two successive elements of the sequence is strictly decreasing.

2.5. Monotonicity

We adapt the definitions of this section from [AHH16, Page 10].

Let S be a set. Then, for a well-quasi-order $\preccurlyeq$ and a relation $R \subseteq S \times S$ over elements of S, we say that R is monotonic w.r.t. to $\preccurlyeq$ if for all $s_1, s_2, s_1' \in S$, if $(s_1, s_2) \in R$ and $s_1 \preccurlyeq s_1'$, then there exists $s_2' \in S$ such that $(s_1', s_2') \in R$ and $s_2 \preccurlyeq s_2'$.

Given a monotonic relation $R \subseteq S \times S$ w.r.t. to some well-quasi-order $\preccurlyeq$, and a set $A \subseteq S$, we define the image of A under R as $R(A) = \{b |$ There exists $a \in A$ such that $(a, b) \in R\}$. Then if it holds that $R(A) \subseteq A$, then $R(\downarrow A) \subseteq \downarrow A$.

2.6. Petri Nets

Petri nets are a widespread model used to describe distributed systems, business processes and chemical reactions. There is a large amount of literature concerned with Petri nets and their properties. For a rather comprehensive introduction, see [Esp17]. In the following, we present some of the definitions from this introduction, and describe the notation that is used in the context of this thesis.

A *Petri net* is given by a tuple (S, T, F), where:

- S is a finite set of places,
- T is a finite set of transitions, and
- $F \subseteq S \times T \cup T \times S$ is a finite set of arcs.

In this thesis, we use standard Petri net notation, in which we illustrate places as circles, transitions as boxes, and arcs as arrows between places and transitions.

For a given place or transition $a \in S \cup T$, we denote by ${}^\bullet a = \{b | (b, a) \in F\}$ the *pre* of a. Similarly, we define the *post* of a as $a^\bullet = \{b | (a, b) \in F\}$. Intuitively, the pre of an element are those elements that have an arc to it, while its post are the elements it has an arc to.

A *marking* is a multiset M over the set of places S, where for each place, its multiplicity in the marking denotes how many tokens are in that place. If $M(s) > 0$, i.e. there is atleast one token in s, we say that s is *marked*. Tokens flow through the net via transitions. A transition t is enabled at marking M if all places in the pre of t are marked. We say

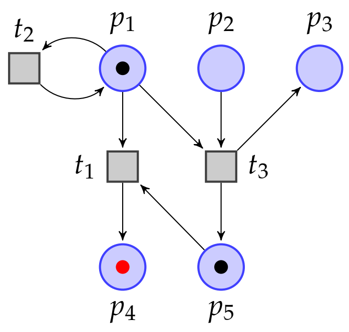


Figure 2.1.: A Petri net where p_1 and p_5 are marked (represented by black tokens), and where therefore transitions t_1 and t_2 are enabled. while transition t_3 is disabled. After firing transition t_1, the resulting marking has one token in p_4 (represented by a red token), and no tokens anywhere else.

that transitions that are not enabled are *disabled*. From a marking M where a transition t is enabled, we can derive a successor marking M' by *firing* t such that for all $s \in S$:

$$M'(s) = \begin{cases} M(s) - 1 & \text{if } s \in {}^{\bullet}t \setminus t^{\bullet} \\ M(s) + 1 & \text{if } s \in t^{\bullet} \setminus {}^{\bullet}t \\ M(s) & \text{otherwise.} \end{cases}$$

Intuitively, firing a transition takes one token out of each input place of the transition, and puts one token into each output place of the transition. Additionally, no place can have negative tokens, so transitions can only be fired when there is one token in each input place. Note that a transition can have the same place as an input and as an output place - this means the transition can only be fired when there is a token in the place, but it is put back after the transition has been fired. Using the notation $M \xrightarrow{t} M'$, we denote that M' can be reached from M by firing t. If there exists a transition t such that $M \xrightarrow{t} M'$, we can denote this fact without specifying the transition as $M \to M'$. We can extend this notion to multi-step reachability: $M \xrightarrow{n} M'$ if there are $M_1, M_2, M_3, \ldots, M_{n-1}$ such that $M \to M_1 \to M_2 \to \cdots \to M_n \to M'$. We write $M \xrightarrow{*} M'$ to mean that there exists n such that $M \xrightarrow{n} M'$. Note this means $\xrightarrow{*}$ is the reflexive and transitive closure of $\to$.

As an example, we consider the Petri net given in Figure 2.1. We denote its markings as vectors of the form $(m_1, m_2, m_3, m_4, m_5)$, where m_i is the number of tokens in place p_i in the marking. The black tokens represent the marking $(1,0,0,0,1)$, which means that only transitions t_1 and t_2 are enabled. When transition t_1 is fired, this leads to the marking given by the red token, which can be represented as $(0,0,0,1,0)$. From this marking, no further transitions are enabled.

One extension of the main model consists of allowing arcs to have *weights*. We define a Petri net with weights as a tuple (S, T, M), where

- S and T are defined as in the main model, while

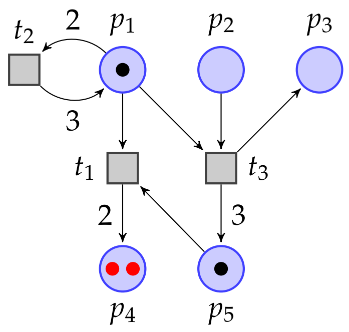


Figure 2.2.: A similar Petri net to the one presented in Figure 2.1, where some arcs where modified to have weights. We simply denote the weight of an arc by writing it next to it. Note that only transition t_1 is enabled in the given marking, which again has one token in p_1 and one token in p_5 (represented by black tokens). After t_1 is fired, the resulting marking has two tokens in p_4 (represented by red tokens), and no tokens anywhere else.

- the weight function $W : (S \times T) \cup (T \times S) \to N$ is a mapping from arcs to weights.

Note that we do not draw arcs with a weight of 0 in illustrations.

Intuitively, the weight of an arc denotes how many tokens flow along that arc when it's corresponding transition is fired. A transition t is then only enabled if every input place has atleast as many tokens as the weight of the arc from that place to t, i.e. t is enabled in a marking M if and only if for all $s \in S$, it holds that $M(s) \geq W(s, t)$. We redefine the successor marking M' obtained by firing a transition t at marking M as follows:

$$\text{For all states } s \in S : M'(s) = M(s) + W(t, s) - W(s, t)$$

All other notions are defined as in the main model. As an example, we consider the Petri net of Figure 2.2, which is the net from Figure 2.1, extended with arc weights.

3. Related Work

Parameterized systems and algorithms to verify their properties are widely investigated in the literature. The *view abstraction* method covered in Chapter 4 is originally introduced by Abdulla et al. in [AHH14]. Some additional clarification about the used model and proposed algorithm can be found in [AHH16].

In [AHH14; AHH16], Abdulla et al. propose an extension of the view abstraction method that deals with systems that do not satisfy all criteria of applicability of view abstraction, mainly systems that do not allow good downward-closed invariants. They extend the model using so-called contexts, where the smaller patterns that make up the abstraction of a large instance of the system are extended with additional information about the original system instance. In the context of this thesis, we do not need contexts, as the systems that we want to analyze, i.e. population protocols, generally provide these downward-closed invariants.

When one drops the assumption that the system is parameterized, there are many methods that attempt to prove correctness of such systems, for example model checking [BK08; Vis+03]. However, they usually require major adaptions to cope with parameterized systems where the state space is infinite.

In the literature, many related classes of parameterized systems have been introduced, with algorithms for the verification of safety properties for some of them, and we mention some of these classes and how they relate to the class covered in this thesis in the following.

In [Abd+96], a class of infinite-state systems called well-structured-systems is introduced. They are closely related to the class of parameterized systems we study here. The main condition for a system to be well-structured is that the transition relation is monotonic, which means that the system does not necessarily need to model a distributed system. The main difference lies in the fact that in this thesis, we restrict ourselves to infinite-state systems that consist of the parallel execution of an arbitrary number of processes that all run the same program. This means we can treat the system as a family of systems parameterized by the number of processes, while well-structured-systems do not have this restriction of uniformity. However, some of the proof details we use in later chapters to prove that the view abstraction method is correct resemble closely the observations made in [Abd+96] about this more general class.

A recent survey of the current state-of-the-art regarding the complexity of computing safety properties for different types of parameterized systems is given in [Esp14]. There, the systems are classified by their communication capabilities, and the classes defined there are those with broadcast communication, global stores with locking, rendez-vous communication and global stores without locking. In our case, the view abstraction method can handle systems that support both broadcast and rendez-vous transitions.

A related model for parameterized systems is used by Bozga et al. in [BIS19]. As in the model used for view abstraction, the system is parameterized over the number of processes, and processes perform parallel executions with the possibility of interactions between processes. However, while in this thesis we only consider the case where all processes are copies of each other and use the same transitions, the model by Bozga et al. allows one to have different types of so-called components, which can be imagined as different types of processes. Each type uses a different transition relation, and instead of parameterizing over the total number of processes, there is one parameter per component type. This allows for greater flexibility in systems that can be modelled.

A paradigm for verification of a related class of parameterized systems is used by Bouajjani et al. in [Bou+00]. This paradigm is called regular model checking, and works on infinite-state systes where the states are given as finite words over a finite alphabet. The goal there is to compute the set of reachable states of the system and the transitive reflexive closure of the transition relation. Therefore, the used model and the goal are very close to the underlying model and goal of view abstraction. However, the differences lie in the methodology and its limiations. In [Bou+00], two approaches are presented.

The first approach is based on constructing the transitive closure of the transition relation using techniques from automata theory. The limitation here is that the algorithm is not guaranteed to terminate if the set R^+, i.e. words generated by the repeated concatentation of one or more words from R, is not regular.

The second approach uses a technique called widening, and is based on guessing the result of repeated iteration of a given relation from a starting set. Here, the limitation is that the procedure is only guaranteed to be exact for so-called simple relations.

4. Parameterized Systems

Parameterized systems are models for systems that can be viewed as a family of infinitely many finite-state systems, characterized by one or more parameters. For every fixed evaluation of the parameters, the system only has finitely many states, however, since there are infinitely many values for the parameters, e.g. because they can take on all values from the natural numbers, the family as a whole has infinitely many members. An example for such a parameterized system is given by a utility that performs an operation on entries in two tables of a database. Such a system could be modelled as determined by two parameters, which denote the number of entries in the two tables of the database respectively.

The infinite size of the state space makes automatic verification of such systems hard. Verification in this context means that we want to verify wether the system satisfies some correctness speficiation for all parameter evaluations. This means that techniques like a full exploration of the state space of the system are impossible to use, and must be adapted to cope with the infinitely many states.

In the context of this thesis, we focus on a subclass of parameterized systems, which is introduced in [AHH16]. Systems in this subclass consist of many copies of identical processes. The only parameter is the number of processes. These processes act in parallel without any form of synchronization, and may communicate with each other. As such, an instantiation of the system for the number of processes n consists of the parallel execution of n processes.

Processes can act without interacting with another process and simply update their own states, which is what we call local transitions. On the other hand, when a process executes a global transition, it can first check the states of some other processes before deciding its next state.

Which processes can interact with each other is determined by the topology of the system. In this thesis, we focus on two topologies:

The first is a *linear* topology, where a process can perform checks over all other processes, but can only distinguish between its right and left neighbours. That is, processes can be thought of as organized from left to right on a line, and a process can either check if any process, or only any process on its left (resp. right), is in a given state.

The other topology we consider is a *multiset* topology, which we define in this context as a topology where a process can check all other processes, but cannot distinguish between any of them - it is only possible to check whether there is another process that is in a given state. This can be thought of as a weaker version of a linear topology, where processes cannot even distinguish whether other processes are to the left or to the right of them.

We will consider Burns' mutual exclusion algorithm as an example of a parameterized system, which we use as a running example when we give more formal definitions in the following. The algorithm is introduced in [Bur78].

4.1. Burns' Mutual Exclusion Algorithm

In distributed computing, a frequent problem is managing a resource that is shared among a number of parallel processes, but may only be used by a single processes at once. For example, a program may contain a critical section that must be fully executed by one process before another process can start executing it. Access to such a critical section may be such a resource. We call problems like this *mutual exclusion problems*.

The dining philosophers problem, as it appears for example in [Hoa78], is a more abstract example that is nonetheless widely referenced in the literature. In this problem n philosophers sit at a round table, with a fork between each philosopher and each of his neighbours. The philosophers can either think or eat. For thinking, there are no additional requirements, but in order to eat, a philosopher needs to have two forks. However the philosophers can only pick up their neighbouring forks one after another, not both at once. This can lead to some undesirable situations. Imagine each philosopher picking up the fork to their left at the same time, which means the philosophers are stuck. All of them have a fork in their left hand, but since the fork to each philosophers right is already in use, noone can start eating, and eventually, the philosophers starve. We call such a situation where the system is stuck a *deadlock*.

Another problem can occur when the system is not stuck, but a philosopher that wants to eat never is allowed to. For example, imagine our solution to the problem is such that when a philosopher is hungry, he first takes the left fork, and then tries to take the right fork. However if the right fork is already in use he puts the left fork down, so someone else can use it, and tries again in five minutes. Then in the case that all philosophers get hungry at the exact same time, they pick up the left fork first, then try to take the right fork. Upon realizing that the right fork is already in use by the philosopher to the right, they all simultaneously put down their left fork, and try again in five minutes. However since the philosophers are in perfect sync, the same situation will arise again. Therefore, even though the philosophers want to eat, and the system is not in a deadlock, they never get to eat. On the contrary, if our algorithm for solving this problem is designed in a way such that when a philosopher wants to eat, he eventually gets to do so, we say that the algorithm satisfies *starvation freedom*.

Burns' mutual exclusion algorithm is one way to ensure mutual exclusion, while satisfying freedom of deadlocks and of starvation. The algorithm is due to Burns [Bur78], and has been covered extensively in the literature [Lyn96; JL98; AHH16]. A representation of the algorithm from point-of-view of process i can be seen in Algorithm 1. Figure 4.2 illustrates the transitions as a state diagram, and is adapted from the literature [Haz15].

Note that the algorithm works on a linear topology - process i needs to know which processes have smaller and greater process numbers than itself. Communication happens

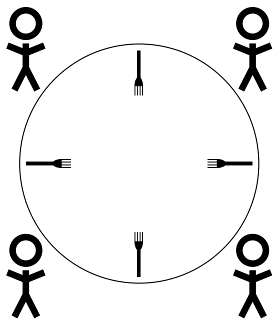


Figure 4.1.: An illustration of the dining philosophers problem, with $n = 4$ philosophers and forks.

Algorithm 1 Pseudocode for Burns' mutual exclusion algorithm, from the point of view of process i. Adapted from [Lyn96]. The critical section that only one process at a time should execute is line 6 (marked in red).

```
1: BEGIN: $flag[i] := 0$
2: If there exists $j < i$ such that that $flag[j] = 1$ then goto BEGIN
3: $flag[i] := 1$
4: If there exists $j < i$ such that that $flag[j] = 1$ then goto BEGIN
5: WAIT: If there exists $j > i$ such that $flag[j] = 1$ then goto WAIT
6: *Critical Section*
7: $flag[i] := 0$
8: goto BEGIN
```

$$\begin{tikzcd}
 & \text{start} \arrow[r] & 1 \arrow[r] & 2 \arrow[l, bend right, "\exists j < i : flag[j] = 1"'] \arrow[r, "\nexists j < i : flag[j] = 1"] & 3 \arrow[d] \\
8 \arrow[urr] & 7 \arrow[l] & 6 \arrow[l] & 5 \arrow[l, "\nexists j > i : flag[j] = 1"'] \arrow[loop below, "\exists j > i : flag[j] = 1"] & 4 \arrow[l, "\nexists j < i : flag[j] = 1"'] \arrow[ull, "\exists j < i : flag[j] = 1"']
\end{tikzcd}$$

Figure 4.2.: A state diagram of Burns' mutual exclusion algorithm. Adapted from [Haz15]. The critical section is represented by node 6 (marked in red).

through an array of shared variables called *flag*, where $flag(j) = 1$ signals that process j wants to enter the critical section.

Initially, all flags are 0. From the point of view of process i, the first loop checks whether there is any process with a smaller process number that has its flag set to 1. If there is, that process first gets the right to enter the critical section, so process i returns to the beginning. If there is not, process i sets its flag to 1 and rechecks all processes with smaller flags, again moving back to the beginning if any have their flag set to 1. This second check is important, since a process might have set its flag to 1 after it was checked by process i, but before process i set its own flag to 1. After both checks have shown no processes to the left have their flag set to 1, process i can now wait until all processes with a larger process number have left the critical region, and signalled this by setting their flag to 0. Now, process i can enter the critical section, and after leaving it sets its flag back to 0. In our algorithm, the actual code of the critical section is omitted, and we simply treat its end, i.e. setting the flag back to 0, as the entire critical section.

For a proof that Burns' mutual exclusion algorithm ensures mutual exclusion, and satisfies both deadlock-freedom and starvation-freedom, see [Lyn96].

In the following chapters, we review this example, and use it to illustrate the concepts.

4.2. Formal Definition of Parameterized Systems

A *parameterized system* is defined as a tuple (Q, Δ) where:

- Q is a finite set of states, and
- Δ is a finite set of transition rules.

A configuration $c = c_1c_2...c_n$ is a word over the alphabet Q, and characterizes the state of a system of n processes. The set of all configurations is denoted as $\mathcal{C}$. Transition rules can be of one of two possible forms:

- *Local* rules are rules that depend only on a single process, and they are of the form $src \rightarrow dst$ where $src, dst \in Q$.
- *Global* rules are rules that depend not only on a single process. They are instead *existential* or *universal* rules that depend on other processes. Therefore, global rules are of the following form:

  If $\mathbb{Q}\ j \circ i : c[j] \in S$ **then** $src \rightarrow dst$ **else** $src \rightarrow dst'$,

  where

  – $c = (c_1, ..., c_n)$ is the *current configuration* of the system,

  – $i \in \{1, ..., n\}$ is the *current process*,

  – $\mathbb{Q} \in \{\forall, \exists\}$ is the *quantifier*,

  – $\circ \in \{<, >, \neq\}$ denotes the *range* of the rule,

- $S \subseteq Q$ is the *condition*,
- $src \in Q$ is the source, and
- $dst, dst' \in Q$ are the destinations.

Intuitively, a global rule means the i-th process inspects other processes of the configuration in the range to check whether they satisfy the condition. For example, the condition $\exists j > i : c[j] \in \{q_1, q_2\}$ checks whether there exists a process j with a greater process number than the current process i that is in state q_1 or q_2.

Checks are assumed to be instantaneous and atomic, i.e. one process checks all other processes and then makes the move specified by the rule, without interruption from other processes. Nonatomic checks are out of the scope of this thesis.

For Burns' mutual exclusion algorithm, we can formalize the algorithm by using eight rules, of which three (one per loop in the algorithm) are global rules, and five are local rules. Note that we denote the state of the processes as in the state diagram of Figure 4.2, i.e. one state per line of the algorithm. Therefore, the states of Burns' algorithm are given by the finite set $Q = \{1, 2, 3, 4, 5, 6, 7, 8\}$. We do not need to consider the value of the flag, since the line number determines the value of the flag - initially, the value of the flag is 0. If a process is in one of lines $1, 2, 3$ or 8, its flag is 0, otherwise it is 1. We could include the flag in the process states by using pairs of lines and flag values. For example, the process being in line 5 with its flag set to 1 is then represented as the tuple $(5, 1)$. However, for ease of notation we use only line numbers for our states.

The rules are as follows:

- $1 \to 2$,
- **If** $\exists j < i : flag[j] = 1$ **then** $2 \to 0$ **else** $2 \to 3$,
- $3 \to 4$,
- **If** $\exists j < i : flag[j] = 1$ **then** $4 \to 0$ **else** $4 \to 5$,
- **If** $\exists j > i : flag[j] = 1$ **then** $5 \to 5$ **else** $5 \to 6$,
- $6 \to 7$,
- $7 \to 8$,
- $8 \to 0$.

For a configuration $c = c_1 ... c_n$, we denote as $\delta(c, t, i)$ the successor with respect to a process index i and a transition rule t as applying t with process $c[i]$ as current process. Formally, this means:

- If t is a local rule of the form $src \rightarrow dst$, then $\delta(c,t,i) = c_1 \dots c_{i-1}\, dst\, c_{i+1} \dots c_n$ if $c[i] = src$. Informally, this simply means applying the local rule to process i and replacing that process by the result of the transition rule.
- If t is an existential rule of the form **If** $\exists\ j \circ i\ :\ c[j] \in S$ **then** $src \rightarrow dst$ **else** $src \rightarrow dst'$, then if there exists some j such that $j \circ i$ and $j \in S$, then $\delta(c,t,i) = c_1 \dots c_{i-1}\, dst\, c_{i+1} \dots c_n$ else $\delta(c,t,i) = c_1 \dots c_{i-1}\, dst'\, c_{i+1} \dots c_n$. Note that this only holds if $c[i] = src$.
- If t is a universal rule of the form **If** $\forall\ j \circ i\ :\ c[j] \in S$ **then** $src \rightarrow dst$ **else** $src \rightarrow dst'$, then if for all j such that $j \circ i$, it also holds that $j \in S$, then $\delta(c,t,i) = c_1 \dots c_{i-1}\, dst\, c_{i+1} \dots c_n$ else $\delta(c,t,i) = c_1 \dots c_{i-1}\, dst'\, c_{i+1} \dots c_n$. Again, this only holds if $c[i] = src$.

Note that for both local and global rules if the current process is not in the source state, i.e. $c[i] \neq src$, then $\delta(c,t,i)$ is undefined. By $\delta(c,i)$ we denote the set of successors obtainable from c by any transition rule applied with i as the current process. For configurations c, c' we write $c \rightarrow c'$ if there exist i, t such that $c' = \delta(c,t,i)$. We also say that c is one-step-reachable from c'. We extend this notion to multi-step reachability, which we denote as $c \xrightarrow{*} c'$. Informally, this means that c' is reachable from c in zero or more steps. More formally, $c \xrightarrow{*} c'$ if and only if $c = c'$ or $c \rightarrow c'$ or there exists a sequence $c_1, c_2, ..., c_j$ s.t. $c \rightarrow c_1 \rightarrow c_2 \rightarrow ... \rightarrow c_j \rightarrow c'$.

For a configuration c, we define the set of all successors obtainable from c, independent of transition rule or current process, as $post(c) = \{c' |$ There exist t, i such that $c' \in \delta(c,t,i)\}$. We can lift $post$ to sets of configurations. For a set of configurations $C = \{c_1, c_2, \dots\}$, we define $post(C) = \bigcup_{c \in C} post(c)$.

Note that for a linear topology, all types of transition rules outlined in this section are possible - intuitively, global rules where we quantify over $j > i$ resp. $j < i$ check whether any process to the right resp. left of process i satisfies the condition, while those where we quantify over $j \neq i$ check processes both to the left and to the right. Linear topologies allow us to make all these checks. However for multiset topologies we can only quantify over $j \neq i$ in global rules, since processes can not distinguish between left and right neighbours, but are instead only allowed to check whether *any* other process satisfies a condition. Therefore we constrain global rules to this form in multiset topologies and disallow quantifying over $j < i$ and $j > i$.

4.3. Reachability Problem in Parameterized Systems

A problem frequently studied on parameterized systems is that of safety. To prove safety for parameterized systems, one needs to ensure that certain bad configurations cannot be reached for any number of processes. Therefore, in the context of this thesis, reachability problems are equivalent to safety problems. A formal definition of the *reachability problem* is given in the following.

Given

- a parameterized system (Q, Δ),
- a set of initial configurations $I \subseteq Q^+$, and
- a set of bad configurations $B \subseteq Q^+$,

determine whether from any initial configuration $c_i \in I$ we can reach any of the bad configurations. More formally, the system is unsafe if there exist $c_i \in I, c_b \in B$ such that $c_i \xrightarrow{*} c_b$.

In this context,we define the set of *reachable configurations* of size k as $R_k = \{c \in \mathcal{C} |$ There exists $c_i \in I$ such that $c_i \xrightarrow{*} c$ and $|c| = k\}$. The set of reachable configurations of all sizes is given as $R = \bigcup_{k=1}^{\infty} R_k$. We say that a parameterized system is *safe* with respect to I and B if no bad configuration is in the set of reachable configurations. More formally, $R \cap B = \varnothing$.

Since there can be many bad configurations, which can make checking the intersection for emptiness costly, we do not want to use the set B directly. Instead, in the context of this thesis we assume that B is the upward closure of a finite set of *minimal bad elements* B_{min}.

For our example of Burns' mutual exclusion algorithm, we want to ensure that the algorithm maintains mutual exclusion. We defined the corresponding parameterized system in Section 4.2. Now, we define a reachability problem on it. Recall that initially, all processes start in state 1 in Burns' algorithm, so our set of initial configurations is given by the simple regular expression 1^+. Bad configurations are those where two states are simultaneously in the critical section. As a regular expression, our set of bad configurations is therefore given by $(S^*)6(S^*)6(S^*)$, where S denotes the set of states of the algorithm. Noticeably, this is an infinite set, and storing the whole set is superfluous. It is easy to see that with respect to the ordered subword relation, we can denote this set by its unique minimal element 66.

4.4. View Abstraction

This section is dedicated to examining an algorithm for solving the reachability problem in parameterized systems, as presented in Section 4.3. We call this the *view abstraction algorithm*. Note that this algorithm is originally due to Abdulla et al. [AHH16]. Where the description in this chapter differs from the original, we note this fact.

The basic idea behind the method lies in the use of an abstraction. When we apply the abstraction to a configuration, it breaks it down into the patterns it contains. We call these patterns *views*. When we use these views to reconstruct possible configurations, i.e. those that produce a subset of these views when abstracted, we get at least the configuration that we abstracted before. In addition, we might get other configurations that happen to generate the same views. Therefore, when abstracting, then reconstructing, we arrive at an overapproximation of our input. By making steps on the configurations in this overapproximation, we can reconstruct an overapproximation of the reachable configurations.

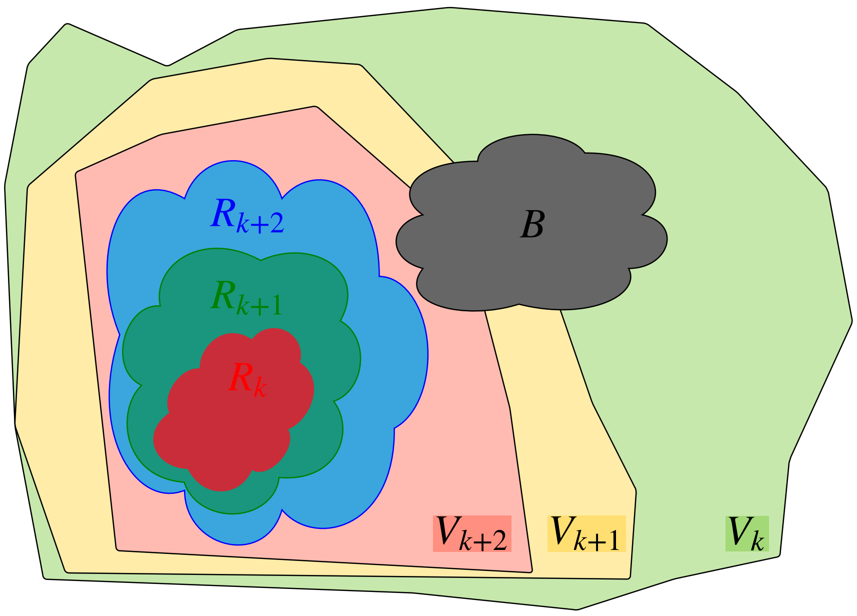


Figure 4.3.: An illustration of how view abstraction is able to prove safety. As k increases, the set of reachable configurations R_k grows larger, while the overapproximation V_k becomes smaller. At some point, either the reachable configurations intersect with the bad configurations and the system is proven unsafe, or the overapproximation does not intersect with the bad configurations and the system is proven safe.

.

The basic intuition is that we parameterize the abstraction and reconstruction using a parameter k. As mentioned, we generate an overapproximation by repeatedly abstracting, performing a step on, and reconstructing a configuration until we reach a fixpoint. This overapproximation becomes tighter as we increase k, and becomes closer and closer to the set of reachable configurations. We call this fixpoint that is an overapproximation of the reachable configurations V_k, where k denotes the parameter of the abstraction/reconstruction. This fixpoint can help prove safety. On the other hand, for one fixed size k of initial configurations, we can simply perform exhaustive forward reachability from them, which gives as a result the set of reachable configurations up to that size. This helps prove unsafety. As we increase k, the set of reachable configurations grows, while the overapproximation shrinks (but always remains an overapproximation). At some point, when we consider configurations of sufficient size, either a bad configuration is found by the forward reachability analysis (in which case the system can be deemed unsafe), or no more bad configurations are present in the overapproximation (in which case we know that the system is safe). See Figure 4.3 for an illustration of this rough intuition.

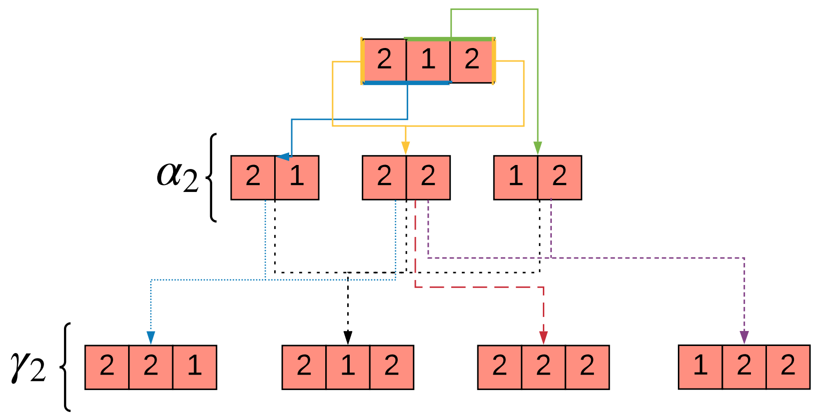


Figure 4.4.: An illustration of abstraction and reconstruction where $k = 2$. Note that for reconstruction, we only show the configurations of size 3, and for abstraction, only those of size 2, to illustrate that the reconstruction can contain different configurations even of the same size as an input configuration. The colored lines illustrate how patterns from the input configuration relate to the views. The dashed and dotted lines illustrate how the views are recombined to form the reconstruction.

4.4.1. Abstraction and Reconstruction

As noted in the last section, the underlying abstraction allows us to construct an overapproximation of the reachable configurations. This abstraction is called *view abstraction*, and it is parameterized by the *size* of the abstraction, which we denote as k. We also refer to the abstraction with size k as the k-abstraction.

For a configuration c, we denote its k-abstraction as $\alpha_k(c)$, and define it as follows:

Definition 1. *$\alpha_k(c) = \{e \in \mathcal{C} | e \sqsubseteq c$ and $|e| \leq k\}$, where $\sqsubseteq$ denotes the ordered subword relation as defined in Section 2.2.*

We call the elements of $\alpha_k(c)$ the views of c up to size k. Intuitively, the views of a configuration are patterns of size k or less that can be found in it. We denote as $\mathcal{V}$ the set of all possible views, and as $\mathcal{V}_k$ the set of all views up to size k.

We define the k-abstraction of a set of configurations C as follows:

Definition 2. $\alpha_k(C) = \bigcup_{c \in C} \alpha_k(c)$.

For a given set of views V, we define the *k-reconstruction* $\gamma_k(V)$ as follows:

Definition 3. $\gamma_k(V) = \{c \in \mathcal{C} | \alpha_k(c) \subseteq V\}$.

Less formally, the k-reconstruction of a set of views is a set containing those configurations such that their k-abstraction is a subset of V.

Note that since views of a configuration are again words over the states of the parameterized systems, these views resemble smaller configurations.

Figure 4.4 illustrates abstraction and reconstruction, and can give some intuition regarding the meaning of the views in the abstraction. Note that the figure only contains subsets of the views in α_2 and the configurations in γ_2 due to size constraints. To give a more complete example for abstraction and reconstruction, consider the configuration $c = 31223$. In the following, we list the k-abstraction and respective reconstruction for some possible values of k:

$$\begin{aligned}
\alpha_1(c) =& \{1, 2, 3\} \\
\gamma_1(\alpha_1(c)) =& L((1|2|3)^*) \setminus \{\epsilon\} \\
\alpha_2(c) =& \alpha_1(c) \cup \{31, 32, 33, 12, 13, 22, 23\} \\
\gamma_2(\alpha_2(c)) =& L((3^*)(1|\epsilon)(2^*)(3^*)) \setminus \{\epsilon\} \\
\alpha_3(c) =& \alpha_2(c) \cup \{312, 313, 322, 323, 122, 123, 223\} \\
\gamma_3(\alpha_3(c)) =& \alpha_3(c) \cup \{3123, 31223, 3122, 3223, 1223\} \\
\alpha_4 =& \alpha_3(c) \cup \{3122, 3223, 3123, 1223\} \\
\gamma_4(\alpha_4(c)) =& \alpha_4(c) \cup \{31223\}
\end{aligned}$$

Note that the reconstruction always contains at least the views it was reconstructed from, and that the k-reconstruction of the k-abstraction becomes smaller as k increases, but it always contains at least the initial configuration c.

There are sets of views that do not make much sense to reconstruct from - namely, those such that even the views of size up to k themselves are not in the k-reconstruction. Consider for example the set of views $V = \{112, 121\}$. It is clearly impossible that V was constructed by abstracting a configuration - for example, since the view 112 is contained in V, the set would also need to contain the view 11 if it was constructed by abstracting a set of views. This leads to unintended behaviour, where γ_k is empty for any k, even though the k-reconstruction should always contain at least the input views up to size k. To circumvent this, we introduce a notion of *admissibility*. This notion does not appear in the original definition of the view abstraction method in [AHH16]. However for ease of explanation, we include it here.

Definition 4. *V is admissible if and only if for all $k \in \mathbb{N}, v \in V$, it holds that if $|v| \leq k$, then $v \in \gamma_k(V)$.*

Intuitively, we call a set of views V admissible if for all k, the k-reconstruction of V contains the views of V up to size k. We now prove that admissibility is equivalent to requiring that for all elements $v \in V$, V contains the $|v|$-abstraction of v, and introduce some related lemmata first.

Lemma 1. *For all $x \in \mathcal{C}, k \in \mathbb{N}$, it holds that $\alpha_k(x) \subseteq \alpha_{k+1}(x)$.*

Proof. Consider the k-abstraction of a configuration x. This abstraction contains all ordered subwords of x of length up to k. For a larger value $k' > k$, the k'-abstraction then contains all the subwords of length up to k', therefore also those up to length k, i.e. those in the k-abstraction. $\square$

Lemma 2. *For all $x \in \mathcal{C}, k \in \mathbb{N}$, it holds that if $k > |x|$, then $\alpha_k(x) = \alpha_{|x|}(x)$.*

Proof. Recall that the k-abstraction of contains all ordered subwords of x up to size k. Since all subwords of x are at most as long as x, the abstraction already contains all ordered subwords of x when we are considering the $|x|$-abstraction, so larger abstractions cannot add new subwords. $\square$

Lemma 3. *V is admissible if and only if for every view $v \in V$, V contains the $|v|$-abstraction of v.*

Proof. Consider an admissible set V. Then admissibility requires that for all k, for all $v \in V$ such that $|v| \leq k$, it holds that $v \in \gamma_k(V)$. Therefore it must hold that $\alpha_k(v) \subseteq V$, therefore, for all k, the k-abstractions of all views up to size k must be contained in V. For a view v, we include all the views of its k-abstraction for every k if we include $\alpha_{|v|}(v)$, as shown in Lemma 2.

On the other hand, if we consider a set V such that for every view $v \in V$, it holds that $\alpha_{|v|}(v) \subseteq V$, then that means that for every view $v \in V$ it holds that $v \in \gamma_k(V)$ for all k. Therefore, V must be admissible. $\square$

Since we always start with a configuration (or set of configurations) that we then abstract, it is easy to see that the resulting set is admissible. Therefore, from now on we only reason about admissible sets of configurations when we apply reconstruction.

As mentioned before, we can obtain an overapproximation of a set of configurations by abstracting and then reconstructing it. An important property of view abstraction is that this becomes smaller as the size of the abstraction and reconstruction is increased.

Lemma 4 ([AHH16, Lemma 2])**.** *For all $C \subseteq \mathcal{C}$, it holds that $C \subseteq \cdots \subseteq \gamma_3(\alpha_3(C)) \subseteq \gamma_2(\alpha_2(C)) \subseteq \gamma_1(\alpha_1(C))$.*

The post of a set of configuration X, as defined in Section 4.2, denotes the set of successor configurations of the configurations in X. Recall that these successor configurations are generated by applying possible transitions to each process of each configuration individually. Similarly, we can define the *abstract post* function A_{post_k}, parameterized by the size of the underlying abstraction k. For a set of views V, we define:

Definition 5. $A_{post_k}(V) = \alpha_k(post(\gamma_k(V)))$.

Intuitively, the abstract post first reconstructs larger configurations from the input configurations, then performs one step on them. Lastly, the resulting configuirations are abstracted again, which means only configurations up to size k remain.

We observe that α_k and γ_k are complementary operations. We list some properties that these two operations satisfy in the following.

Lemma 5 ([AHH16, Lemma 1]). *For all $k \in \mathbb{N}$, $A, B \subseteq \mathcal{C}$, $V, W \subseteq \mathcal{V}_k$, the following properties hold:*

1. *If $V \subseteq W$, then also $\gamma_k(V) \subseteq \gamma_k(W)$.*
2. *$A \subseteq \gamma_k(\alpha_k(A))$.*
3. *If $A \subseteq B$, then also $\alpha_k(A) \subseteq \alpha_k(B)$.*
4. *$\alpha_k(\gamma_k(V)) \subseteq V$.*

These properties can all be derived in a straightforward manner from the definitions of γ_k, α_k and $\sqsubseteq$.

Lemma 6 ([AHH16, Lemma 1]). *For all $k \in \mathbb{N}$, $A \subseteq \mathcal{C}$, $B \subseteq \mathcal{V}_k$, it holds that $\alpha_k(A) \subseteq B$ if and only if $A \subseteq \gamma_k(B)$.*

Proof. For the proof, we reference the properties of Lemma 5.

Assume we have A, B such that $\alpha_k(A) \subseteq B$. By property 1 we derive that $\gamma_k(\alpha_k(A)) \subseteq \gamma_k(B)$ must hold. Finally, by property 2, it then holds that $A \subseteq \gamma_k(\alpha_k(A)) \subseteq \gamma_k(B)$, therefore $A \subseteq \gamma_k(B)$.

Now assume that we have A, B such that $A \subseteq \gamma_k(B)$. Then by property 3, it holds that $\alpha_k(A) \subseteq \alpha_k(\gamma_k(B))$. By property 4, it follows that $\alpha_k(A) \subseteq \alpha_k(\gamma_k(B)) \subseteq B$, therefore $\alpha_k(A) \subseteq B$. □

4.4.2. Algorithm

Algorithm 2 A scheme for the view abstraction algorithm. Adapted from [AHH16, Algorithm 1].

```
1: for $k := 1$ to $\infty$ do
2:     if $R_k \cap B \neq \varnothing$ then return Unsafe
3:     $V := \mu X.\alpha_k(I) \cup A_{post_k}(X)$
4:     if $\gamma_k(V) \cap B = \varnothing$ then return Safe
```

Having defined the view abstraction and its corresponding reconstruction, we can now start defining the remaining components of the algorithm. The first step consists of giving a schema of the algorithm, that we then expand into a concrete algorithm. Pseudocode for the schema can be found in Algorithm 2. In the following, we explain the algorithm line-by-line.

Line 1 initiates the outer loop, by which a parameter k is gradually increased until it is sufficiently large to determine safety. Line 2 encompasses a forward reachability analysis, since R_k are the reachable configurations of size k. Note that we have previously defined the set R_k more formally in Section 4.3. If the reachable configurations of size k contain a bad configuration, this obviously means that the system is unsafe, so we can return. Otherwise, the algorithm progresses to line 3. This line contains the computation of a least fixpoint. We initialize this computation with the k-abstraction of the initial configurations I, i.e. $\alpha_k(I)$. In each iteration, we perform one abstract step, i.e. A_{post_k}, and add the resulting configurations to the fixpoint. The computation stops when no more change occurs and the fixpoint has been found. The resulting fixpoint V is stable, i.e. $A_{post_k}(V) \subseteq V$ and it covers the views of size k generated from the initial configurations, i.e. $\alpha_k(I) \subseteq V$.

Next, we prove that the reconstruction of such a fixpoint V covers all reachable views.

Lemma 7 **([AHH16, Lemma 3]).** *For all $k \in \mathbb{N}$, $V \subseteq \mathcal{V}$ such that $A_{post_k}(V) \subseteq V$ and $\alpha_k(I) \subseteq V$, it holds that $R \subseteq \gamma_k(V)$.*

Proof. Choose k, V such that it holds that $A_{post_k}(V) \subseteq V$ and $\alpha_k(I) \subseteq V$. By the definition of A_{post_k}, it holds that $\alpha_k(post(\gamma_k(V))) \subseteq V$. By Lemma 6, we can follow that $I \subseteq \gamma_k(V)$ and $post(\gamma_k(V)) \subseteq \gamma_k(V)$. Since $post$ is monotonic w.r.t. $\subseteq$, we can follow that $post(I) \subseteq post(\gamma_k(V)) \subseteq V$. Note that $R = post^*(I)$. Because $\gamma_k(V)$ covers I, i.e. $I \subseteq \gamma_k(V)$, and $\gamma_k(V)$ is a fixpoint of $post$, i.e. $post(\gamma_k(V)) \subseteq \gamma_k(V)$, it follows that $post^*(I) \subseteq \gamma_k(V)$. Therefore, it also holds that $R \subseteq \gamma_K(V)$. □

Informally, Lemma 7 means that the reconstruction of the fixpoint, i.e. $\gamma_k(V)$, is an overapproximation of the reachable configurations. Therefore, if $\gamma_k(V)$ does not contain bad configurations, then neither does R, which means we know that the system is safe. This check occurs in line 4 of the algorithm in Algorithm 2.

What remains to be shown is that the reconstruction of the fixpoint from line 3 approaches the set of reachable configurations as k increases. In the following, we denote the fixpoint obtained in iteration k as V_k.

Lemma 8 **([AHH16, Lemma 5]).** *For all $k \in \mathbb{N}$, it holds that $\gamma_{k+1}(V_{k+1}) \subseteq \gamma_k(V_k)$.*

Proof. We define for a fixed $k \in \mathbb{N}$ the i-th iteration of the fixpoint of A_{post_k} as V_k^i. For $i = 0$, we define $V_k^0 = \alpha_k(I)$ and for all $i \geq 0$, we define $V_k^{i+1} = \alpha_k(post(\gamma_k(V_k^i)))$.

Now, we introduce an intermediate lemma regarding V_k^i for all i.

Lemma 9 **([AHH16, Lemma 5]).** *For all $k \in \mathbb{N} \setminus \{0\}$ it holds that $\gamma_{k+1}(V_{k+1}^i) \subseteq \gamma_k(V_k^i)$.*

Proof. We proceed by induction over i. Consider the base case where $i = 0$. Then $V_k^0 = \alpha_k(I)$, and $V_{k+1}^0 = \alpha_{k+1}(I)$. By substitution in our statement, we now need to prove that $\gamma_{k+1}(\alpha_{k+1}(I) \subseteq \gamma_k(\alpha_k(I))$. By Lemma 4, which claims that for larger k, the overapproximation generated by abstraction and subsequent reconstruction becomes smaller, this holds.

For the induction step, consider that for fixed $i \geq 0$, it holds that $\gamma_{k+1}(V_{k+1}^i) \subseteq \gamma_k(V_k^i)$. Consider now the case for $i + 1$.

$$\begin{aligned}
\gamma_{k+1}(V_{k+1}^{i+1}) &= \gamma_{k+1}(\alpha_{k+1}(post(\gamma_{k+1}(V_{k+1}^{i})))) && (4.1)\\
&\subseteq \gamma_{k+1}(\alpha_{k+1}(post(\gamma_{k}(V_{k+1}^{i})))) && (4.2)\\
&\subseteq \gamma_{k}(\alpha_{k}(post(\gamma_{k}(V_{k+1}^{i})))) && (4.3)\\
&= \gamma_{k}(V_{k+1}^{i}) && (4.4)
\end{aligned}$$

The equalities in 4.1 and from 4.3 to 4.4 are simply by definition. The subset relation from 4.1 to 4.2 uses the induction hypothesis together with the fact that *post* is monotonic w.r.t. $\subseteq$, while the step from 4.2 to 4.3 uses Lemma 4. □

Now, we can return to the main proof.

$$\begin{aligned}
\gamma_{k+1}(V_{k+1}) &= \gamma_{k+1}(\bigcup_{i\geq 0} V_{k+1}^{i})\\
&= \gamma_{k+1}(V_{k+1}^{0} \cup \bigcup_{i\geq 0} V_{k+1}^{i+1})\\
&= \gamma_{k+1}(\alpha_{k+1}(I) \cup \bigcup_{i\geq 0} \alpha_{k+1}(post(\gamma_{k+1}(V_{k+1}^{i}))))\\
&\subseteq \gamma_{k+1}(\alpha_{k+1}(I) \cup \bigcup_{i\geq 0} \alpha_{k+1}(post(\gamma_{k}(V_{k}^{i})))) && \text{By Lemma 9}\\
&= \gamma_{k+1}(\alpha_{k+1}(I \cup \bigcup_{i\geq 0} post(\gamma_{k}(V_{k}^{i}))))\\
&\subseteq \gamma_{k}(\alpha_{k}(I \cup \bigcup_{i\geq 0} post(\gamma_{k}(V_{k}^{i})))) && \text{By Lemma 4}\\
&= \gamma_{k}(\alpha_{k}(I) \cup \bigcup_{i\geq 0} \alpha_{k}(post(\gamma_{k}(V_{k}^{i}))))\\
&= \gamma_{k}(V_{k}^{0} \cup \bigcup_{i\geq 0} V_{k}^{i+1})\\
&= \gamma_{k}(V_{k})
\end{aligned}$$

□

Therefore, the fixpoint becomes more precise as k increases. However, note that this is not enough yet. Since for a set of configurations X and a constant k, $\gamma_k(X)$ can in general be infinite, we cannot ensure that there is necessarily a k such that $\gamma_k(V_k) \cap B = \varnothing$, even if $R \cap B = \varnothing$ and the fixpoint becomes smaller for larger k.

Therefore, we need to prove that there is a k for which the overapproximation is tight enough such that $\gamma_k(V_k) \cap B = \varnothing$ if $R \cap B = \varnothing$.

Lemma 10 ([AHH16, Lemma 6]). *If the set of reachable configurations R is downward-closed with respect to the ordered subword relation, then there is a $k \in \mathbb{N}$ such that $R = \gamma_k(V_k)$.*

Proof. Recall that by Lemma 7 it holds that for all $k \in \mathbb{N}$, $V \subseteq \mathcal{C}$ it holds that $R \subseteq \gamma_k(V)$ if V is a fixpoint of A_{post_k} that covers $\alpha_k(I)$.

Let us show that there exists k such that for any set $X \subseteq \mathcal{C}$ such that if X is a fixpoint of *post* and X covers I, i.e. $post(X) \subseteq X$ and $I \subseteq X$, and X is downward-closed w.r.t. $\sqsubseteq$, then it holds that $\gamma_k(V_k) \subseteq X$.

Note that X is downward-closed, and can therefore be characterized by a set of maximal elements X_{max}. Consider now the complement of X, i.e. $\overline{X}$. This must be an upward-closed set, and can therefore be characterized by a set of finite minimal elements $\overline{X}_{min}$ w.r.t. the well-quasi-order $\sqsubseteq$. Let k be the maximal length of a minimal element of $\overline{X}$, i.e. $k = max_{m \in \overline{X}_{min}} |m|$.

We now want to show that $\gamma_k(\alpha_k(X)) = X$. By property 2 of Lemma 5, we know that $X \subseteq \gamma_k(\alpha_k(X))$. It remains to show that $X \supseteq \gamma_k(\alpha_k(X))$. Assume for contradiction that there is an element e that is not contained in X, but $e \in \gamma_k(\alpha_k(X))$ holds. Then e must be contained in $\overline{X}$, and therefore be contained in $\uparrow\overline{X}_{min}$. It follows that there must exist $m \in \overline{X}_{min}$ such that $m \sqsubseteq e$. Additionally, since m must be at most as long as the longest minimal element of $\overline{X}$, we know that $|m| \leq k$. Because X is upward closed, m can not be contained in the k-abstraction of X, since otherwise e must also be contained in X. Therefore, $m \notin \alpha_k(X)$ and by definition of reconstruction $e \notin \gamma_k(\alpha_k(X))$. This is a contradiction, since we assumed $e \in \gamma_k(\alpha_k(X))$.

Therefore, it holds that $\gamma_k(\alpha_k(X)) = X$. Because X is a fixpoint of *post*, i.e. $post(X) \subseteq X$, it holds that $post(X) \subseteq X$. Then also $post(\gamma_k(\alpha_k(X))) \subseteq X$ holds. Because of monotonicity of α_k w.r.t. $\subseteq$, it holds that $\alpha_k(post(\gamma_k(\alpha_k(X)))) \subseteq \alpha_k(X)$. Recall the definition $A_{post_k} = \alpha_k \cdot post \cdot \gamma_k$. Then it becomes clear that $\alpha_k(X)$ is a fixpoint of A_{post_k}. Because V_k is by definition the least fixpoint of A_{post_k} that covers $\alpha_k(I)$, it must hold that $V_k \subseteq \alpha_k(X)$, and by Lemma 6 we follows that $\gamma_k(V_k) \subseteq X$. Finally, we observe that R satisfies the constrains outlined for X initially: It is easy to see that R is a fixpoint of *post* that covers I, i.e. $post(R) \subseteq R$ and $I \subseteq R$. □

There are two important propositions regarding termination and correctness of the algorithm that are given in [AHH16, Corollary 1, Corollary 2], but are left without a proof. We state these propositions in the following and provide proofs for them.

The first proposition regards termination and correctness of the algorithm when R is downward-closed.

Proposition 1 ([AHH16, Corollary 1][1]). *If the set of reachable configurations R is downward-closed with respect to the ordered subword relation, then the view abstraction algorithm is sound and complete.*

Proof. We consider two cases.

In the first case, a bad configuration is reachable, and the algorithm should return unsafe. Then this means that there must be a bad configuration $b \in B$ such that there is

[1]The proposition is slightly extended here. In [AHH16], the corollary only claims that the algorithm terminates, and leaves soundness and completeness for another part of the paper.

an initial configuration $c_i \in I$ such that $c_i \xrightarrow{*} b$. Therefore, for $k = |c_i|$, this means that $b \in R_k$, which means the intersection check in line 2 of the algorithm in Algorithm 2 returns a nonempty set containing at least b, and the algorithm returns *Unsafe* in line 2.

In the second case, no bad configuration is reachable, therefore the algorithm should return unsafe. By Lemma 10 and because R is downward-closed, there exists a k such that $\gamma_k(V_k) = R$. Therefore, if the system is safe, i.e. $R \cap B = \varnothing$, and when we consider a sufficiently large k, then also $\gamma_k(V_k) \cap B = \varnothing$ holds, and the algorithm returns *Safe* in line 4. □

The second proposition is concerned with the case when R is not necessarily downward-closed, as can happen for some parameterized systems where some configurations of size 1 are unreachable because more than one process is required to enter certain states of the system.

Proposition 2 ([AHH16, Corollary 2][2])**.** *If B is upward-closed with respect to the ordered subword relation $\sqsubseteq$, and the transition relation $\rightarrow$ is monotonic with respect to $\sqsubseteq$, then it holds that $R \cap B = \varnothing$ if and only if $\downarrow R \cap B = \varnothing$.*

Proof. Recall the definition of monotonicity from Section 2.5. Because $\rightarrow$ is a monotonic relation, we know that for all sets of configurations C, C' it holds that $\rightarrow (C) \subseteq C'$ implies that $\rightarrow (\downarrow C) \subseteq \downarrow C'$.

Now assume that we have R and B such that $R \cap B = \varnothing$ and B is upward-closed. Assume for contradiction that $\downarrow R \cap B \neq \varnothing$. This means there must exist some e s.t. $e \in \downarrow R$ and $e \in B$, but $e \notin R$. Since e is in the downward closure of R, there must exist $r \in R$ s.t. $e \sqsubseteq r$. Because B is upward closed and $e \in B$, it must hold that $r \in B$. Therefore, the intersection $R \cap B \neq \varnothing$, which is a contradiction, therefore $\downarrow R \cap B = \varnothing$.

Now we prove the other direction. It holds that $R \subseteq \downarrow R$, therefore if $\downarrow R \cap B = \varnothing$ then $R \cap B = \varnothing$. □

This means in the case where R is not downward closed but B is upward-closed, we can simply use the downward-closure of R to check for the intersection with B. We make use of this fact when we introduce the pseudocode for the view abstraction algorithm later.

From the two propositions, we conclude that the view abstraction method is guaranteed to terminate, and is both sound and complete.

However, note that there are still several problems with implementing the algorithm on a machine. These problems mostly stem from the use of the reconstruction γ_k, which can in general be infinite. Therefore, it cannot be expressed directly in memory. In line 3, we use γ_k when computing A_{post_k}, i.e. in the fixpoint iteration.

Note that in this fixpoint iteration, we only reconstruct the larger configurations in order to perform a step on them. Since the length of configurations in the initialization of the fixpoint is at most k, and the fixpoint iteration does not add larger configurations, $\gamma_k(V)$ always covers at least the configurations from V. Therefore, the successors of

[2]We remove part of the corollary here, since in the original, it contains two separate statements.

the reconstruction of V covers at least the successors of V, i.e. $post(V) \subseteq post(\gamma_k(V))$. Intuitively, by reconstructing, we do not disable transitions, but can only enable more transitions by adding additional witness processes to the configurations. This is the case because some configuration in the reconstruction may fulfill additional preconditions of existential rules. Note that for existential rules, as defined in Section 4.2, we can satisfy the precondition by adding a single process. This can give an intuition that since the fixpoint only contains configurations of a finite size, and enabling transitions only requires adding a finite number of witnesses, computing the infinite set $\gamma_k(V)$ is not necessary. To circumvent this infinite set, we restrict ourselves to the configurations in $\gamma_k(V)$ up to a certain size, and define a corresponding new operation.

We call this operation *extension*, and denote it as $\oint_k^l$, where k is the size of the underlying abstraction, and l is the length of the extension. Intuitively, configurations in the k-extension up to length l of the set of configurations V are those configurations in the k-reconstruction of V, up to length l.

Definition 6. $\oint_k^l(V) = \{c | \alpha_k(c) \subseteq V \wedge |c| \leq l\}$

Naturally, since configurations in $\oint_k^l$ are limited in length, the extension can only contain finitely many configurations.

Next, we want to prove that for a given k, in the computation of A_{post_k}, we can replace γ_k by $\oint_k^l$ for some l. We already hinted at only needing to add a single witness process, therefore, we choose $l = k + 1$.

We give the following lemma without a proof, which can be found in the original presentation.

Lemma 11 ([AHH16, Lemma 4]). *For all $k \in \mathbb{N}, V \subseteq \mathcal{V}_{\|}$, it holds that $\alpha_k(post(\gamma_k(V)) = \alpha_k(post(\oint_k^{k+1}(V))$.*

Next, we modify lines 1 and 4 of the algorithm. First, note line 4, where we compute the intersection of two potentially infinite sets, $\gamma_k(V)$ and B. This intersection cannot be computed in an exhaustive manner directly. However, recall that we assume that the set of bad configurations B is the upward closure of a set of minimal elements B_{min} with respect to the ordered subword relation. One can imagine these minimal elements as patterns of processes, such that when a configuration contains this pattern, it is bad. Note that these minimal elements have similarities to the views up to size k of a configuration, which are simply its ordered subwords of size k or less. Then it becomes clear that if k is at least as large as the length of elements of B_{min}, we can simply check whether the intersection of V and B_{min} is nonempty. However, note that this means we need to start at a potentially large k. Figure 4.5 illustrates the relation between bad configurations and minimal bad configurations.

We also need to modify the intersection check in line 2. This modification also appears in [AHH16], but there is no proof that it is equivalent to the unmodified test. Therefore, we add such a proof in the following.

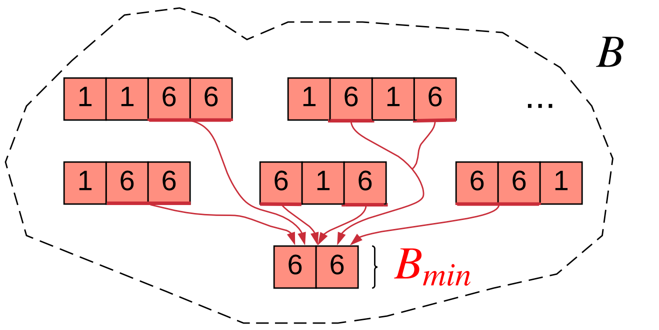


Figure 4.5.: An illustration of bad configurations and their relation to the minimal bad configurations. This closely relates to view abstraction, which also uses the ordered subword relation for generating views. This can give some intuition as to why we only need to check the intersection between V and B_{min}.

Algorithm 3 The view abstraction algorithm. Adapted from [AHH16, Algorithm 2].[3]

```
1: for $k := max_{b \in B_{min}} |b|$ to $\infty$ do
2:     if $\alpha_k(R_k) \cap B_{min} \neq \varnothing$ then return Unsafe
3:     $V := \mu X. \alpha_k(R_k) \cup \alpha_k(post(\oint_k^{k+1}(X))$
4:     if $V \cap B_{min} = \varnothing$ then return Safe
```

The modification is necessary because B, as an infinite set, does not lend itself to exhaustive representation in memory. Since we test for a nonempty intersection here, we do not run into the same problem as with the check in line 4, where we would erronously terminate early with a potentially wrong result when starting with a k that is not large enough.

Lemma 12. *A system S, such that R is the set of reachable configurations in S, where R is downward-closed with respect to the ordered subword relation, and B is the set of bad configurations, is unsafe if and only if there is $k \in \mathbb{N}$ such that $\alpha_k(R_k) \cap B_{min} \neq \varnothing$.*

Proof. Assume S is unsafe, i.e. $R \cap B \neq \varnothing$. Therefore, there is a b such that $b \in R \wedge b \in B$. Since B is upward-closed with respect to the ordered subword relation $\sqsubseteq$, there must be a minimal decomposition $b^1_{min}, b^2_{min}, \ldots b^n_{min} \in B_{min}$ of b such that for all $i \leq n$, it holds that $b^i_{min} \sqsubseteq b$. We know that the k-abstraction of b contains its ordered subwords up to size k. Consider an element of the minimal decomposition of b with minimal size, and denote this element as b^{min}_{min}. Then for $k = |b^{min}_{min}|$, it must hold that $b^{min}_{min} \in \alpha_k(b)$, since

[3]Note that in [AHH16, Algorithm 2], the algorithm starts k at 1. In iteration k, V then contains only views up to size k, and if all minimal bad configurations are of size larger than k, the intersection $V \cap B_{min}$ is the empty set, even if the system is not safe when starting from larger initial configurations. There is no proof given for the correctness of the algorithm without this adjustment in the original presentation, and we believe this to be an oversight. Therefore, we adapt this here and start at the size of the largest minimal bad configuration.

both $b_{min}^{min} \sqsubseteq b$ and $|b_{min}^{min}| \leq k$. Therefore, $b_{min}^{min} \in B_{min}$, and because we assumed $b \in R_k$ it must also hold that $b_{min}^{min} \in \alpha_k(R_k)$. We conclude that if the system is unsafe, then there must be a k such that $\alpha_k(R_k) \cap B_{min} \neq \varnothing$.

For the other direction, suppose for the sake of contradicion that there is $b \in \alpha_k(R_k) \cap B_{min}$ but the system is safe, i.e. $R \cap B = \varnothing$. If $b \in \alpha_k(R_k)$, there must be $r \in R_k$ such that $b \sqsubseteq r$. Since R is downward closed with respect to the ordered subword relation, and $r \in R$, it follows that $b \in R$. This is a contradiction, since we assumed the system is safe. □

The modified algorithm can be found in Algorithm 3.

4.4.3. Rendez-vous Transitions

Note that in our current model of parameterized systems, interactions are constrained to one-way interactions. It is easy to see that for all global transitions, the current process i observes the state of one or more other processes, and bases its behaviour on the presence (or absence) of certain states. The observed processes do not change their state based on the fact that they were observed. However, we sometimes want interactions between multiple processes to simultaneously change the state of two or more of the interaction partners. Consider for example Petri nets where each transition has the same incoming and outgoing degree. The restriction on transitions means that no tokens are created or destroyed, but tokens can change between states. Then we can model tokens of the Petri net as processes of the parameterized system, and transitions of the net as interactions between multiple processes, where each involved process may simultaneously change its state.

More formally, we define a *rendez-vous transition* r as a tuple of local rules.[4] In terms of semantics, we define for a configuration $c = c_1, \dots, c_n$, a rendez-vous transition $r = (src_1 \to dst_1, src_2 \to dst_2, \dots, src_m \to dst_m)$ and mutually distinct process ids $p_1, p_2, \dots, p_m \in [1, n]$ the successor of c w.r.t. the vector of process ids $p = (p_1, p_2, \dots, p_m)$ and the transition rule r as $c' = c'_1, \dots, c'_n$, where for all i, $c'_{p_i} = dst_i$, and $c'_i = c_i$ if $i \notin p$. Note that this only holds when for all i, $c_{p_i} = src_i$, otherwise the successor c' is undefined. Intuitively, for each local transition rule $src \to dst$ in r, there needs to be a process that is in state src. The transition updates the chosen processes simultaneously to their respective dst-states. Note that in r, there could be multiple local rules with the same src. For each such rule, there needs to be a different process in the src-state, i.e. the same process cannot be chosen twice.

Also, note that existential rules that quantify over all processes $j \neq i$ can also be viewed as rendez-vous transitions.

We adapt the following lemma:

[4] In [AHH16], this type of transitions is called a *simple rendez-vous transition*. However, since in the context of this thesis we do not introduce the more powerful *general rendez-vous transitions*, we do not make this distinction here.

Lemma 13 ([AHH16, Lemma 7][5]). *For a parameterized system $P = (Q, \Delta)$, where Δ contains local, global and rendez-vous transitions, let m be the largest arity of a rendez-vous rule in Δ. Then for all $k \in \mathbb{N}$, for all sets V of configurations of size up to k, it holds that $\alpha_k(post(\gamma_k(V))) \cup V = \alpha_k(post(\oint_k^{k+m-1}(V))) \cup V$.*

Intuitively, this lemma claims that rendez-vous transitions can be taken into account by changing the extension operation used in the computation of the abstract post. Instead of extending configurations by a single witness process, we need to add enough processes to enable every rendez-vous transition where one of the *src*-states is already present in the configuration, i.e. we extend the views to size $k + m$ instead of to $k + 1$. A proof can be found in [AHH16, Lemma 7], but is omitted here.

4.4.4. Global Process Pointers

Some mutex algorithms, such as Dijkstra's algorithm [Dij65], make use of a global variable called *Process Pointer*. This process pointer ranges over the process indices, and can indicate for example which process is allowed to move into the critical section next.

Integrating such a process pointer into the method presents some obstacles, and we examine those and their solutions in the following. The first problem is that we do not have the capability to store anything outside the states of the processes in our current model. We can circumvent this by adding an additional boolean variable to each process. This variable is *True* when the process is the target of the process pointer, and *False* otherwise. To ensure that there is always one target for each process pointer, changing the pointer target happens via a rendez-vous transition that sets the pointer variable in the old target to *False*, and at the same time sets it to *True* in the new target.

However, note that storing the process pointer implicitly in the variables of processes means that when we abstract configurations, they could end up without the process that is being pointed to. In that case, there is a configuration without a valid value for the process pointer, which does not occur in the real system. Therefore, we need to always keep the process that is the current pointer target when abstracting. We can do so by performing a normal abstraction on the configuration of all processes except for the process pointer, and then inserting the process pointer back into the views at the correct position afterwards.

More details as to the implementation of process pointers can be found in Section 4.5.1.

4.4.5. Completeness for well-quasi-ordered systems

This section closely follows [AHH16, Section 3.6]. It is dedicated to examining for which parameterized systems the view abstraction algorithm is complete.

We say that a transition system $(\mathcal{C}, \rightarrow)$ is a well-quasi-orered (WQO) system with respect to a well-quasi-order $\preccurlyeq$ if $\rightarrow$ is monotonic with respect to $\preccurlyeq \subseteq \mathcal{C} \times \mathcal{C}$.

[5]We omit parts of the proof about *broadcast* transitions, which we do not introduce in this thesis.

Definition 7. *A* discrete measure *over a set S is a function $||.|| : S \to \mathbb{N}$ such that the set $\{s \in S | |s| = k\}$ is finite for every k.*

Intuitively, a discrete measure is something akin to the size of elements in S, under the constraint that there can only be finitely many elements of any given size. Note that this means the size of configurations of a parameterized system as defined in Section 4.2 is a discrete measure. There are often infinitely many configurations because they can be of unbounded size. However, for a given size, there are only finitely many configurations, because there are only finitely many ways to build a configuration from the finitely many states of processes. However, the size of configurations is only one possible discrete measure. Therefore, one can think of discrete measures as generalizations of the size of configurations. Indeed, the definitions of abstraction, reconstruction and abstract post can be adjusted to use such a generalized discrete measure instead of configuration size. We show this more formally in the following.

For a well-quasi-ordered system $(\mathcal{C}, \rightarrow)$ w.r.t. $\preccurlyeq$, and a discrete measure $||.||$, we define $\alpha_k(c) = \{c' \in \mathcal{C} | c' \preccurlyeq c \land ||c'|| \leq k\}$. Similar adjustments to the definitions of γ_k and A_{post_k} are fairly trivial, as their definitions are based on α_k.

Note that the completeness and soundness results of Proposition 1 follows with only minor adjustments when using this generalized discrete measure.

Theorem 1 ([AHH16, Theorem 1])**.** *Let $P = (\mathcal{C}, \rightarrow)$ be a well-quasi-ordered system with respect to a measure $||.||$. For $I, B \subseteq \mathcal{C}$ such that B is upward-closed with respect to $\preccurlyeq$, then if P is safe with respect to the initial configurations I and the bad configurations B, then there must be $k \in \mathbb{N}$ s.t. $B \cap \gamma_k(V_k) = \emptyset$, where $V_k = \mu X.\alpha_k(I) \cup A_{post_k}(X)$.*

Together with the results in Section 4.4.2, this means that the view abstraction algorithm is complete for a large variety of systems. Among others, this includes Petri nets (see Section 2.6) and population protocols (see Chapter 5).

4.5. Implementation

In [AHH16], Abdulla et al. mention that a prototype implementation of the view abstraction method has been implemented in OCaml, and they show several benchmarks on classical distributed algorithms. However, this prototype implementation is not publicly available under a license that allows its use or modification at the time of writing.

In order to provide a second implementation for the verification and reconstruction of some of the results from the benchmark in [AHH16], we implemented the view abstraction algorithm in Python. The implementation is available under the MIT license at `https://gitlab.lrz.de/philip_offtermatt/viewabstraction-protocolassist`.

Over the course of the implementation, some implementation details were ambigious, and the literature did not give clarification as to how these details should be implemented. In order to clarify and justify our decisions in the implementations of these details, the rest of this section is dedicated to presenting parts of the implementation, especially where it is not straightforward to derive it from the algorithm in Algorithm 3.

4.5.1. Extension

Recall that we defined the extension of a set of views $V \subseteq \mathcal{V}$ as $\oint_k^l(V) = \{c \in \mathcal{C} | \alpha_k(c) \subseteq V \wedge |c| \leq l\}$. The naive way of computing this would be by simply enumerating all finitely many configurations up to size l, and checking for each configuration whether its k-abstraction is a subset of V. Since we cannot know the configurations a priori, we need to check all possible configurations, which means all words over states of the system. Assuming there are s states each process can assume, this means there are $s^l + s^{l-1} + s^{l-2} + \cdots + s^1$ possible words, and for each we need to compute its abstraction and check if it is covered by V. It is easy to see that even for systems of modest size and for comparatively small l, the naive approach is not feasible.

For this reason, we present in the following an algorithm for computing the extension $\oint_k^l(V)$. The main intuition behind the algorithm relies on the fact that V must be an admissible set. In view abstraction, we need the extension operation for the fixpoint iteration, which is seeded with $\alpha_k(I_k)$, i.e. an admissible set, and each iteration of the fixpoint applies $A_{post_k} = \alpha_k \cdot post \cdot \gamma_k$. Because the last operation is again abstraction, the result is admissible.

The intuition is that if V admissible and k is greater than the length of elements in V, to compute $\oint_k^l(V)$ it is sufficient to look at the views of size k and larger in V. Because in view abstraction, V is always the result of applying k-abstraction, there are no views larger than size k.

We prove first that the configurations of size $\leq k$ in the extension are those that are already in V. Afterwards, we tackle the case for configurations of size $> k$.

Lemma 14. *For an admissible set $V \subseteq \mathcal{V}$, and constants $k > 0$, $l > 0$, it holds that $\{e \in \oint_k^l(V) |\, |e| \leq k\} = \{v \in V |\, |v| \leq k\}$.*

Proof. We first show that $\{e \in \oint_k^l(V) |\, |e| \leq k\} \subseteq \{v \in V |\, |v| \leq k\}$. Choose e with $|e| \leq k$ such that $e \in \oint_k^l(V)$. By definition of $\oint_k^l$, it must hold that $\alpha_k(e) \subseteq V$. Because $|e| \leq k$, it must hold that $e \in \alpha_k(e)$, since e is a subword of length at most k of itself. Therefore, it must hold that $e \in V$.

Now we show that $\{e \in \oint_k^l(V) |\, |e| \leq k\} \supseteq \{v \in V |\, |v| \leq k\}$.

Choose e with $|e| \leq k$ such that $e \in V$. Because V is admissible it holds that $\alpha_{|e|}(e) \subseteq V$, and because $|e| \leq k$ it must then also hold that $\alpha_k(e) \subseteq V$. Since $|e| \leq k \leq l$ it follows that $e \in \oint_k^l(V)$ by definition of $\oint_k^l$. □

Therefore, for all elements of size at most k, we simply take the elements from V, and do not need to generate new ones. The remaining case is therefore when looking at elements of size greater than k.

We prove that in order to determine whether an element c with $|c| \geq k$ is in $\oint_k^l(V)$, it suffices to determine whether the views of size k in the k-abstraction of c are a subset of the views of size k in V, i.e. we can discard the views of smaller sizes when computing the configurations of size larger than k in the extension.

Lemma 15. *For an admissible set V and constants $k > 0$, $l > 0$, for all configurations c such that $l \geq |c| > k$, it holds that $c \in \oint_k^l(V)$ if and only if $\{v \in \alpha_k(c) \,|\, |v| = k\} \subseteq \{v \in V \,|\, |v| = k\}$.*

Proof. We first prove that if $c \in \oint_k^l(V)$, then also $\{v \in \alpha_k(c) \,|\, |v| = k\} \subseteq \{v \in V||v| = k\}$.

Take any c such that $c \in \oint_k^l(V)$ and $|c| > k$. Then it must hold that $\alpha_k(c) \subseteq V$. This implies that $\{v \in \alpha_k(c) \,|\, |v| = k\} \subseteq \{v \in V \,|\, |v| = k\}$.

Now we prove the other direction. Choose any c such that $\{v \in \alpha_k(c) \,|\, |v| = k\} \subseteq \{v \in V \,|\, |v| = k\}$ and $l \geq |c| > k$. Now assume for contradiction that $c \notin \oint_k^l(V)$. Therefore, it must hold that $\alpha_k(c) \nsubseteq V$. Therefore, there exists $v \in \alpha_k(c)$ such that $v \notin V$. Since $\{v \in \alpha_k(c) \,|\, |v| = k\} \subseteq \{v \in V \,|\, |v| = k\}$, it must hold that $|v| < k$. By definition of abstraction, there exists $v' \in \{v \in \alpha_k(c) \,|\, |v| = k\}$ such that $v \sqsubseteq v'$. Because V is admissible and by assumption, $v' \in V$, it must also hold that $v \in V$. This is a contradiction, since we assumed $v \notin V$. □

Therefore, it becomes clear that regardless of l, we only need to check whether V contains all views of length k in the k-abstraction of c. This gives rise to the algorithm from Figure 4.

Algorithm 4 The extension algorithm.

Input:

- An admissible set of views V,
- the alphabet of possible letters A,
- the abstraction parameter k,
- the maximal length of generated configurations l.

Output: $\oint_k^l(V)$

```
1: procedure Extension
2:     $W := \{v | v \in V \wedge |v| = k\}$
3:     $R := \varnothing$
4:     while $W \neq \varnothing$ do
5:         $w := W.pop()$
6:         if $|w| < l$ then
7:             for $a \in A$ do
8:                 if For all $w'$ such that $w' \sqsubseteq w$ and $|w'| = k-1$, $w'a \in V$ holds then
9:                     $W = W \cup \{wa\}$
10:                    $R = R \cup \{wa\}$
11:    return $R \cup \{v | v \in V \wedge |v| \leq k\}$
```

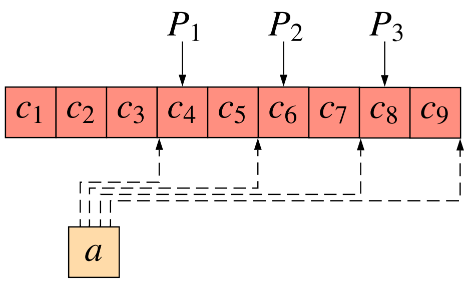


Figure 4.6.: An illustration of extension for systems with process pointers. The words generated by inserting letter a before each process pointer target and at the end of the word are candidates for configurations of the extension.

The intuition behind the algorithm relies heavily on Lemma 15. Since it suffices to check the largest views, we start with each of them as a baseline. Now, we extend the view to the right, one letter at a time with all possible letters and check whether the addition we just made introduces any new views of size k and checks whether these views also appear in V. For this, it is sufficient to check the subwords of length $k-1$ of the existing view, together with the new addition. This procedure is wrapped into a classical workset algorithm, where we simply keep extending these newly generated views until we reach length l.

It is not necessary to extend the views to the left or in the middle. This is because the resulting configuration needs to again satisfy the condition that all its subwords of length k must be contained in V, therefore its subword from index 1 to index k must be contained in V. Then, for any configuration in $\oint_k^l(V)$, we can generate it by starting with a view of size k of V and extending it to the right.

Extending this procedure to systems where we allow process pointers requires some adjustments. The problem is that every configuration contains the states that are targets of process pointers, but we do not include them as possible extensions, since otherwise we would have multiple pointer targets. It does not suffice to extend the views to the right in that case, since this does not allow for the word to the left of a given process pointer to become longer. Therefore, when we use process pointers, there are more possible points in the views where we need to consider extending them. Namely, we try to extend the word to the left of each process pointer, and at the end of the word. Figure 4.6 illustrates this concept.

4.5.2. Input Language

To enable users of the prototype to verify safety properties parameterized systems, there needs to be a way to specify them. In the following, we present three different input languages, each for a different class of parameterized system. Having multiple input languages, instead of one unified language that can handle all cases, may seem wasteful.

```
flag[i] := 0 # begin
goto (exists j < i: flag[j] = 1) begin
flag[i] := 1
goto (exists j < i: flag[j] = 1) begin
goto (exists j > i: flag[j] = 1) wait # wait
flag[i] := 0
goto (True) begin

arrays: flag
```

Figure 4.7.: The program specification for Burns' mutual exclusion algorithm.

However, since Petri nets and population protocols (which we introduce in Chapter 5) are widely used, there already exist input formats for these classes of systems that are used in various tools. Supporting these input formats with no modification required by the user is therefore a priority.

Program specification format

This language is used when the parameterized system in question follows a linear topology, and there are no rendez-vous transitions. The pseudocode for Burns' mutual exclusion algorithm in Figure 1 can give an idea of the type of system that this language supports.

In this language, we assume that each process has one or more local variables. These variables are declared at the bottom of the specification as *arrays*. Note that specifications are written from the point of view of process i. There are then two main operations: Either process i can modify the values of one of its variables, or it performs a (conditional) jump that depends on one or more other processes. Variable modifications and jumps where the condition does not include other processes are then local rules, while conditional jumps that depend on other processes correspond to global rules.

Figure 4.7 shows as an example Burns' mutual exclusion algorithm specified in the language.

Petri Net Markup Language

Because the program specificatio format only offers support for systems without rendez-vous transitions. When we include these types of transitions and drop the assumption of a linear topology for the weaker multiset topology, the parameterized systems essentially are Petri nets.

Because Petri nets are widely used not only in the literature but also in many analysis tools designed for industrial applications, there have been attempts to develop a standard file format for their interchange. One of the most widely used formats is the *Petri Net*

Markup Language (PNML), which is supported by many state-of-the-art-tools. Further information about the format can be found in [Bil+03].

Population Protocol Specification

Population protocols, which are introduced in Chapter 5, can be viewed as Petri nets that are extended with an output function that maps states to outputs. Even though the PNML format is extendable and would allow for such a modification, such an extension has not yet been introduced, and the available tools do not support it. Instead, the tool Peregrine [BEJ18] uses a format based on the JavaScript Object Notation Data Interchange Format [Bra17]. As Peregrine is the state-of-the-art tool for manipulation and verification of population protocols, we offer support for that same format in our implementation of the view abstraction algorithm.

5. Population Protocols

Population protocols are models for describing distributed systems with an arbitrary number of agents. Each agent only has a limited amount of memory, which stores its current state. Population protocols are introduced in [Ang+06], and have since been studied extensively in the literature. For an introduction to population protocols and their basic definitions, as well as some additions to the base model, see [AR09].

In the context of this thesis, population protocols are interesting because they are a parameterized system, where the parameter is the number of agents. As such, they face the same problems that made us turn to view abstraction as a way of verifying parameterized systems in Chapter 4. In addition, several recent advancements have given rise to literature about them. For a survey of recent results, see [Blo+18].

To our knowledge, view abstraction has not been applied to population protocols yet, which appear as a natural candidate for testing and benchmarking the view abstraction method on. This chapter is dedicated to establishing the basic definitions of population protocols, and define a well-known correctness problem for them.

5.1. Definition

In a population protocol, there is an arbitrary number of agents, each in one of finitely many possible states. In each step, a pair of agents is chosen uniformly at random, and the agents of the pair interact according to some transition function, changing their states.

Formally, a population protocol is defined as a tuple $P = (Q, \Sigma, \iota, \delta, Y, \omega)$, where:

- Q is a finite set of states,
- Σ is a finite input alphabet,
- $\iota : \Sigma \to Q$ is a mapping from letters of the input alphabet to states,
- $\delta \subseteq Q^2 \times Q^2$ is the transition relation, where a transition of the form (q_1, q_2, q_1', q_2') means that when two agents interact, and the first agent is in state q_1, while the second agent is in state q_2, they update their states to q_1' and q_2' respectively,
- Y is a finite output alphabet, and
- $\omega : Q \to Y$ is a mapping from states to letters of the output alphabet.

For many protocols, Y is given by a simple truth-indicator from $\{0,1\}$. We call the states of Q that are the image of some input letter of Σ under ι *initial states*. Note that transitions can be nondeterministic, i.e. there could be two transitions (q_1, q_2, q_1', q_2') and (q_1, q_2, q_1'', q_2'') that both describe the behaviour when agents in states q_1 and q_2 meet. Additionally, transitions can depend on order, i.e. the two transitions (q_1, q_2, q_1', q_2') and (q_2, q_1, q_1', q_2') are not the same. We often do not explicitly give Y, Σ and ι, but instead give the output function ω and a set of initial states I.

We use a well-known protocol as an example. This protocol is called *4-state majority* or *simple majority* protocol, and is referenced frequently in the literature [AR09; Blo+18]. The underlying problem definition of the protocol is as follows: Given an initial population of agents, where each agent has either opinion *Yes* or opinion *No*, are there more agents with opinion *Yes* than with opinion *No*?

More formally, our input alphabet is then given as $\Sigma = \{Yes, No\}$. The protocol uses the states $Q = \{Y, N, y, n\}$, and the input function ι defines $\iota(Yes) = Y$ and $\iota(No) = N$. The population protocol uses 4 transitions, which are given as follows:

$$\begin{aligned} Y, N &\rightarrow y, n \\ Y, n &\rightarrow Y, y \\ N, y &\rightarrow N, n \\ y, n &\rightarrow n, n \end{aligned}$$

The output alphabet is given by $\{0,1\}$, where 0 indicates that the agent believes that *No* has the majority, while 1 indicates that the agent believes *Yes* has the majority. The output function ω maps Y and y to 1, while N and n are mapped to 0.

One can imagine the protocol working in two phases: Initially, all agents are in states Y and N. By applying the first transition $Y, N \rightarrow y, n$ until there are no more agents with state Y or no more agents with state N (or both), we can end the first phase. For the second phase, there are three cases:

- Case 1: There are only agents in the states Y, y, n, which means transition $Y, n \rightarrow Y, y$ can be used to convert the remaining n agents to y until only agents with output 1 remain.
- Case 2: There are only agents in the states N, y, n, where similarly to Case 1, transition $N, y \rightarrow N, n$ can be used to convert all agents to states with output 0.
- Case 3: There are only agents in the states y, n, which means the transition $y, n \rightarrow n, n$ can be used until only agents in state n remain.

A *configuration* $C = \{|c_1, c_2, ...|\}$ is a multiset over the set of states of the protocol. The multiplicity of a state denotes how many agents are in the state at the configuration. Note that this means that in population protocols, agents are anonymous - two agents in the same state are not distinguishable.

Now, we can more formally define how transitions work. Similarly to Petri nets, we define for a transition $t = (q_1, q_2, q_1', q_2')$ and a configuration C that t is enabled at C if and only if either $q_1 = q_2$ and $C(q_1) \geq 2$ or $q_1 \neq q_2$ and $C(q_1) \geq 1, C(q_2) \geq 1$. Intuitively, a transition is enabled if there is a pair of agents in C that has the states needed for t to occur.

Note that if for a pair (q_1, q_2), there is no transition $(q_1, q_2, q_1', q_2') \in \delta$, i.e. no transition specifies which behaviour should occur when two agents with these states interact, we assume there is a *silent transition* (q_1, q_2, q_1, q_2) that does not change the states of the agents.

Firing a transition $t = (q_1, q_2, q_1', q_2')$ at a configuration C where t is enabled leads to a successor configuration $C' = (C \setminus \{|q_1, q_2|\}) \cup \{|q_1', q_2'|\}$. We denote the fact that firing t leads from C to C' by writing $C \xrightarrow{t} C'$. We define some standard notions: One-step reachability $C \rightarrow C'$ holds if and only if there is a t such that $C \xrightarrow{t} C'$, and multi-step reachability $C \xrightarrow{*} C'$ denotes the reflexive transitive closure of $\rightarrow$.

5.2. Using population protocols for computation

In the last section, we introduced population protocols, and we already hinted at what types of problems one might apply them to. This section formalizes what it means for a protocol to compute a function, and how one can define correctness of a protocol. In the following, $P = (Q, \Sigma, \iota, \delta, Y, \omega)$ denotes a population protocol.

An execution or run of P is an infinite sequence $C_0 C_1 C_2 \ldots$ such that for all $n \in \mathbb{N}$ it holds that $C_n \rightarrow C_{n+1}$. We denote the i-th element of an execution π as $\pi(i)$.

We say a configuration C is in a *consensus* if there exists $y \in Y$ such that for all $e \in [[C]]$ it holds that $\omega(e) = y$. Therefore, a configuration is in a consensus if all its agents have the same output. We can also overload the output function ω for configurations: if C is in a consensus, and its agents have output y, then $\omega(C) = y$, while a configuration that is not in a consensus has no output. This is sometimes denoted as $\omega(C) = \perp$. We revisit the the example of the 4-state majority protocol to illustrate these concepts. In that protocol, the configurations $\{|N, n, n|\}$ and $\{|Y, Y, y, y|\}$ are in consensus with output 0 and 1 respectively, while the configuration $\{|N, n, y|\}$ is not in a consensus, and therefore has output $\perp$.

After assigning outputs to configurations, we now want to do the same for configurations. However, for this it does not simply suffice that the execution contains a configuration with an output, since the execution might leave the configuration with that output again, and even enter a configuration with a different output. Additionally, executions are infinite, and therefore do not have a "final" configuration that one could use to determine the output.

Instead, we use a *convergence* property. For a configuration $\pi = C_0 C_1 C_2 ...$, we say that π converges to an output y, denoted as $\omega(\pi) = y$, if and only if there is an index $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, it holds that $\omega(C_{n_0}) = y$. Informally, the execution has an output if there is a configuration after which all subsequent configurations have the same output. We say that π has reached a *stable* consensus with value y. Otherwise, we reuse the notation from configurations, and we write $\omega(\pi) = \perp$ if π does not have an output. For example, in the 4-state majority protocol, the execution $\{|N, N, Y, Y|\}\{|N, Y, n, y|\}\{|n, n, y, y|\}\{|n, n, n, y|\}\{|n, n, n, n|\} \dots$ has output 0. On the other hand, the execution $\{|N, N, Y, Y|\}\{|N, Y, n, y|\}\{|N, Y, n, n|\}\{|N, Y, n, y|\}\{|N, Y, n, n|\} \dots$, where by using the transitions $Y, n \to Y, y$ and $N, y \to Y, n$, the execution alternates between the configurations $\{|N, Y, n, y|\}$ and $\{|N, Y, n, n|\}$, does not reach a configuration in a consensus, and has output $\perp$.

By considering these two executions, we can see that the output is not determined by the initial configuration - in the second case, even though one could fire the transition $N, Y \to n, y$ to make progress towards a consensus, by simply never firing it, one can avoid a configuration with a consensus forever. In order to prohibit such behaviour, we define a notion of *fairness*.

A configuration π is *fair* if a configuration that is reachable infinitely often also occurs infinitely often. More formally, π is fair if and only if for all configurations C, C' such that $C \to C'$ and C occurs infinitely often in π, then C' occurs infinitely.

Related to fairness and convergence is the notion of *well-specification*. We say that P is well specified if for every configuration only consisting of initial states, all fair executions starting from that configuration converge to the same output.

Now, we can define what it means for a protocol to be correct, and to perform a meaningful computation. We say that P *computes* a function $f : \Sigma^* \to Y$ if and only if for every possible input word $w \in \Sigma^*$, all fair executions starting at the initial configuration C_0 corresponding to w, i.e. where it holds for all x that $C_0(\iota(x)) = |w|_x$, have output $f(w)$.

Deciding for a given population protocol and predicate whether the protocol computes the predicate is at least as hard as Petri net reachability [Esp+17]. Recent results on this problem have shown that it has non-elementary complexity [Cze+19].

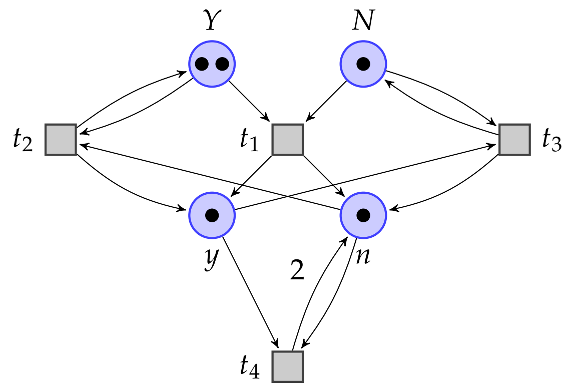
5.3. Population Protocols and Petri Nets

One might notice similarities between population protocols and Petri nets. In both, transitions and states are used in order to define how agents (or tokens) change between states. In fact, population protocols characterize a certain subclass of Petri nets with arc weights, namely where each transition has a total weight of 2 on its incoming arcs, and a total weight of 2 on its outgoing arcs. This models the pairwise interactions between agents in population protocols, and means that the number of agents in the Petri net does not change.

As an example, Figure 5.1 shows on one side the transitions of the 4-state majority protocol, while the oder side shows the corresponding Petri net. Intuitively, the transformation simply involves having one place in the Petri net per state of the population protocol, and having for each transition $t = (q_1, q_2, q_1', q_2')$ of the population protocol a transition in the petri net with incoming arcs from q_1 and q_2, and outgoing arcs to q_1'

$$t_1 : Y, N \to y, n$$
$$t_2 : Y, n \to Y, y$$
$$t_3 : N, y \to N, n$$
$$t_4 : y, n \to n, n$$

(a) The transitions of the 4-state majority protocol.



(b) The Petri net corresponding to the 4-state majority protocol.

Figure 5.1.: The transitions of the 4-state majority protocol, together with its representation as a Petri net with arc weights. The marking of the Petri net corresponds to the configuration $\{Y, Y, N, y, n\}$.

and q'_2.

We make use of this representation of population protocols as Petri nets later, when we apply the view abstraction method to verify a specific property of population protocols.

6. Case Study

To test our implementation and ensure that its performance is in an acceptable range, we benchmarked it on several examples of parameterized systems. One subclass we focus on are population protocols, as defined in Chapter 5. This class of parameterized systems lends itself to analysis, as it has recently been studied in the literature (see Chapter 5 for a mention of some of the recent literature). Additionally, we implemented a small prototype for tool-assisted manual generation of population protocols, using the view abstraction algorithm.

All measurements of this chapter were run on an Intel Core i5-8250U CPU @ 1.60GHz and 8GB of ram.

6.1. Interactive protocol generation

As mentioned in Chapter 5, it is a problem of non-elementary complexity to decide whether a protocol computes a given predicate [Esp+17; Cze+19]. In general, this means computing this on-the-fly can be impossible for large protocols.

Although there exist methods for generating a protocol for a given predicate, and it has recently been shown that this construction takes only polynomial (w.r.t. size of the predicate) time and generates a protocol with a polynomial number of states, these protocols are not guaranteed to have fast convergence time, and are in general quite slow [Blo+19]. Therefore, in order to find fast protocols, one might still need to involve human input in the generation of the protocol.

While automatic generation of protocols is inefficient, manual generation is a tedious and unintuitive process - population protocols do not lend themselves to sequential composition, which means it is hard to separate the state space so that one can work on one part of the protocol without disrupting parts that already function as intended. In theorem proving, another similarly formal area, the widespread tool Isabelle [Pau94] provides interactive feedback to the user. This section is dedicated to presenting a rough prototype implementation of an interactive tool for manual generation of protocol, using the view abstraction method to on-the-fly compute reachability properties on population protocols, while a user generates them, and providing feedback based on that. In the following, we introduce one property of population protocols called *consensus stability*, which we argue can be useful in constructing well-specified population protocols to compute given predicates, and describe how the view abstraction method can be used to verify this property.

6.1.1. Consensus Stability

For a population protocol $P = (Q, \Sigma, \iota, \delta, Y, \omega)$ and a set of states $s \subseteq Q$, we define:

Definition 8. *The set of states s is consensus-stable with output $y \in Y$ if for all $c \in s^*$, for all configurations c' such that $c \xrightarrow{*} c'$, it holds that $\omega(c') = y$.*

Intuitively, a set of states s is consensus stable with output y if, starting from a configuration that consists only of states from s, only configurations with output y are reachable. This means when an execution reaches a configuration such that only states from s appear, it is certain that the execution has output y, even though the states of agents may still change. We say that a population protocol is y-consensus stable for $y \in Y$ when the set of all states with output y is consensus stable with that same output.

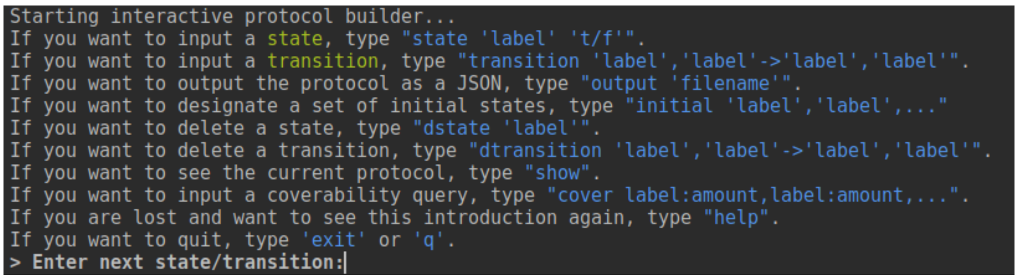
Determining the consensus stable sets of states of a protocol can help catch errors. For example, consider the protocols that we benchmarked, where the results can be seen in Table 6.2. The protocols that compute the same predicate very often have the same behaviour w.r.t. consensus stability, i.e. both protocols from our sample that compute majority satisfy both *True* and *False* consensus stability. Naturally, it is possible to design a population protocol that computes the majority protocol without satisfying consensus stability with respect to both outputs. However since many examples from the literature do so, it can suggest that when one designs a new protocol for the majority predicate, and it is not consensus-stable with respect to both *True* and *False*, it can be warrented to look either for a typo in the protocol or for some superfluous states that could be removed to simplify the protocol.

6.1.2. Checking Consensus Stability using View Abstraction

As introduced in the previous section, determining consensus stable sets of a population protocol can be useful in order to facilitate manual, interactive, tool-assisted generation of population protocols. This section is dedicated to presenting the details of how the view abstraction method can be used to determine whether for a population protocol $P = (Q, \Sigma, \iota, \delta, Y, \omega)$, given set of states of a protocol $s \subseteq Q$, all with the same output $y \in Y$, it holds that s is consensus-stable w.r.t. y.

In Section 5.3, we present how a population protocol can be transformed into a Petri net, where configurations of the protocol correspond to markings of the Petri net. Note that the two representations are equivalent with respect to reachability, i.e. if a marking is reachable in the Petri net from the initial markings, then the corresponding configuration is reachable in the population protocol from the initial configurations. We can easily see that consensus stability can be viewed as a reachability problem as defined in Section 4.3. Given a $P = (Q, \Sigma, \iota, \delta, Y, \omega)$ and a set of states $s \subseteq Q$ with the output $y \in Y$, we want to determine whether s is consensus-stable w.r.t. y. We define the corresponding reachability problem as follows:

- The underlying parameterized system is given as Q, δ, where δ contains rendezvous transitions, as defined in Section 4.4.3,



```
Starting interactive protocol builder...
If you want to input a state, type "state 'label' 't/f'".
If you want to input a transition, type "transition 'label','label'->'label','label'".
If you want to output the protocol as a JSON, type "output 'filename'".
If you want to designate a set of initial states, type "initial 'label','label',..."
If you want to delete a state, type "dstate 'label'".
If you want to delete a transition, type "dtransition 'label','label'->'label','label'".
If you want to see the current protocol, type "show".
If you want to input a coverability query, type "cover label:amount,label:amount,...".
If you are lost and want to see this introduction again, type "help".
If you want to quit, type 'exit' or 'q'.
> Enter next state/transition:|
```

Figure 6.1.: The start screen of the interactive protocol generation prototype.

- the initial configurations are given by the regular expression s^*,
- and the bad configurations are given as an upward-closed set of a set of minimal elements $b_{min} = \{x \in Q | \omega(x) \neq y\}$.

Intuitively, our initial configurations are all possible combinations of states from s, and we determine whether we can reach any configuration that contains a state with an output other than y. Since there is a Petri net corresponding to the population protocol such that they are equivalent with respect to reachability, we can apply the view abstraction algorithm to this parameterized system using a multiset topology, extended with rendez-vous transitions.

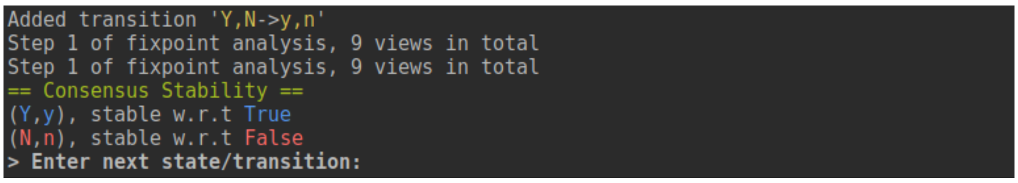
6.1.3. Prototype Implementation

Even though designing population protocols is prone to errors, few tools exist for designing them manually with some form of assistance by the tool, i.e. integrated simulation or verification. We are aware of the following tools:

- Peregrine [BEJ18] provides an intuitive user interface for the construction of protocols, as well as built-in capabilities for experimental simulation and automatic verification. However, this verification is not on-the-fly, which means errors while constructing a protocol frequently occur, but can often be corrected afterwards.
- NETCS [Ama+15] is a tool for construction of population protocols, and provides simulation capabilities. As of September of 2019, the tool is not in active development, and there is very little documentation.

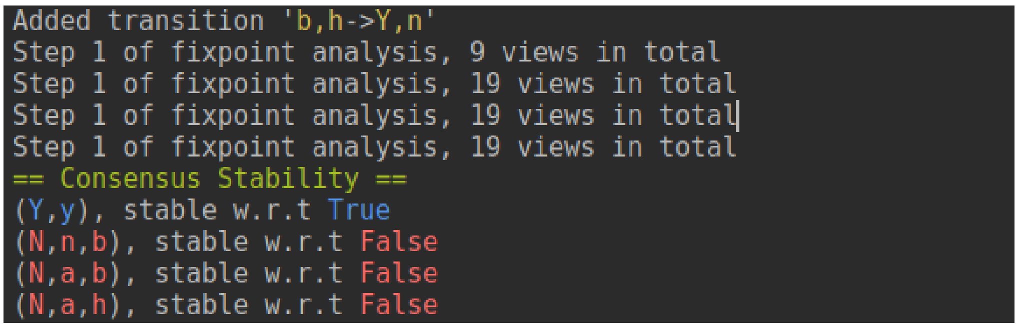
Because existing tools provide little in terms of on-the-fly feedback we have implemented a prototype in Python, based on the presented approach for interactive tool-assisted protocol generation using consensus stability. It is available on `https://gitlab.lrz.de/philip_offtermatt/viewabstraction-protocolassist` under the MIT license.

The prototype uses a simple command line interface to allow generation of protocols. Figure 6.1 shows the initial screen after startup, and possible commands are listed here.



```
Added transition 'Y,N->y,n'
Step 1 of fixpoint analysis, 9 views in total
Step 1 of fixpoint analysis, 9 views in total
== Consensus Stability ==
(Y,y), stable w.r.t True
(N,n), stable w.r.t False
> Enter next state/transition:
```

Figure 6.2.: The interactive protocol generation prototype offers feedback after adding a transition.



```
Added transition 'b,h->Y,n'
Step 1 of fixpoint analysis, 9 views in total
Step 1 of fixpoint analysis, 19 views in total
Step 1 of fixpoint analysis, 19 views in total
Step 1 of fixpoint analysis, 19 views in total
== Consensus Stability ==
(Y,y), stable w.r.t True
(N,n,b), stable w.r.t False
(N,a,b), stable w.r.t False
(N,a,h), stable w.r.t False
```

Figure 6.3.: The interactive protocol generation prototype can list multiple largest consensus stable sets.

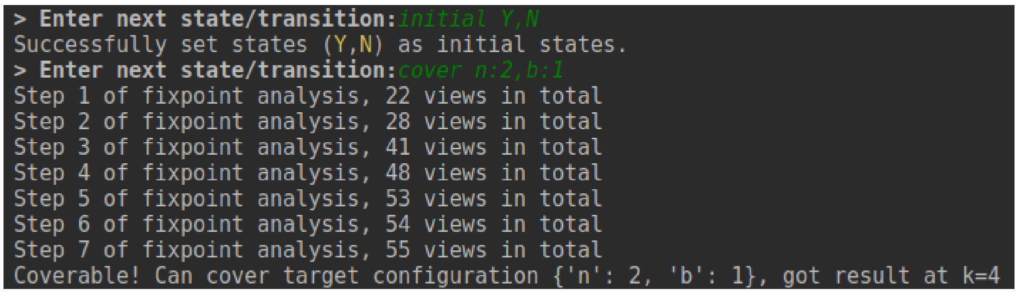
Note that the tool currently only supports boolean outputs, i.e. *True* and *False*, since they are almost exclusively used among the literature.

We can manually generate a protocol step-by-step via the *transition* and *state* commands. After adding a transition, the tool gives as feedback the consensus stable sets of largest size w.r.t. both possible outputs. Naturally, after adding a state with no transition involving that state, it is part of all largest consensus stable sets with the same output as that state, so feedback is only shown when new transitions are created. The form of this feedback can be seen in Figure 6.2.

We only show the consensus stable sets of largest size, since otherwise, when there are many states, there are many consensus stable sets that are merely a subset of some larger consensus stable set. However, when there are multiple such sets of largest size, they are all listed, as can be seen in Figure 6.3.

When the user specifies a set of initial states via the command *initial*, the tool also allows evaluating simple safety queries for upward-closed sets of bad configurations, which are invoked using the command *cover*. This coverability analysis utilizes the view abstraction method. The underlying methodology is similar to that presented in Section 6.1.2, where we utilize view abstraction to check consensus stability in population protocols. We merely need to adapt initial and bad configurations according to the initial states and target configuration specified by the user. An example of how states can be marked as initial, and then used for coverability queries, can be seen in Figure 6.4.

Lastly, the *show* command can be used to inspect the protocol generated so far, as well as to rerun the computation of largest consensus stable sets. Figure 6.5 shows an



```
> Enter next state/transition:initial Y,N
Successfully set states (Y,N) as initial states.
> Enter next state/transition:cover n:2,b:1
Step 1 of fixpoint analysis, 22 views in total
Step 2 of fixpoint analysis, 28 views in total
Step 3 of fixpoint analysis, 41 views in total
Step 4 of fixpoint analysis, 48 views in total
Step 5 of fixpoint analysis, 53 views in total
Step 6 of fixpoint analysis, 54 views in total
Step 7 of fixpoint analysis, 55 views in total
Coverable! Can cover target configuration {'n': 2, 'b': 1}, got result at k=4
```

Figure 6.4.: The tool allows the user to designate initial states and evaluate coverability queries.

examplary result of this command.

The tool is generally fast enough to use for on-the-fly verification of small protocols (<10 states with output *True* and *False* each). However, since this prototype does not utilize caching of previously computed largest consensus stable sets, rerunning these computations each time a new transition is added is fairly time intensive once there are many states for at least one of the possible outputs. However, we believe that this prototype still serves as a proof-of-concept that interactiveness can assist humans in generating population protocols with fewer errors. Chapter 7 describes some areas where further optimization might be possible, and suggests more types of useful interactive feedback that could be added.

6.2. Benchmarks

To compare our implementation of the view abstraction method to the original prototype, we ran both on two sets of benchmarks.

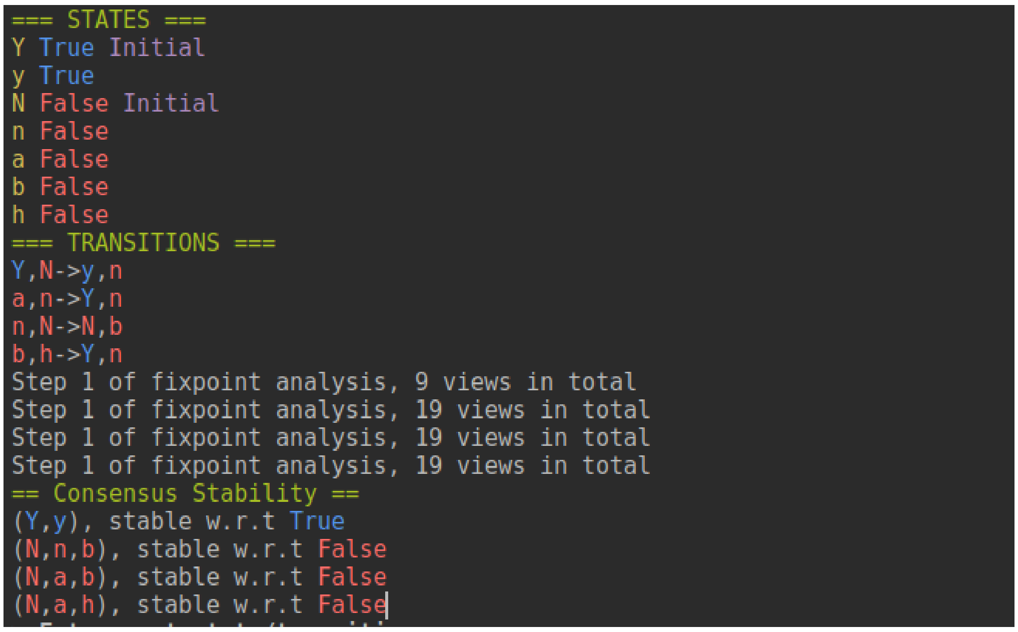
The first set of benchmarks are those from [AHH16, Section 6]. We restrict ourselves to a subset of the atomic versions of protocols that use the linear topology[1], and that exhibit strong downward-closed invariants. Namely we are able to verify several algorithms for mutual exclusion: Burns', Dijkstra's, and Bakery. The input specifications into our program, as described in Section 4.5.2, can be found in Appendix A. The results reported for the original prototype implementation were obtained by running the original source code, obtained privately for evaluation purposes.

Table 6.1 shows the results for this first set of benchmarks. The times were obtained by running each benchmark instance 40 times and taking the average of those times. All times are given in seconds.

For these benchmarks, the two implementations perform relatively comparable. For Bakery and Burns, our implementation is slower by a factor of 2. On the other hand, our implementation is faster for Dijkstra's aglorithm.

We suspect the difference in speed for Bakery and Burns is due to our implementation

[1]In [AHH16, Section 6], these are denoted as *Array* protocols.



```
=== STATES ===
Y True Initial
y True
N False Initial
n False
a False
b False
h False
=== TRANSITIONS ===
Y,N->y,n
a,n->Y,n
n,N->N,b
b,h->Y,n
Step 1 of fixpoint analysis, 9 views in total
Step 1 of fixpoint analysis, 19 views in total
Step 1 of fixpoint analysis, 19 views in total
Step 1 of fixpoint analysis, 19 views in total
== Consensus Stability ==
(Y,y), stable w.r.t True
(N,n,b), stable w.r.t False
(N,a,b), stable w.r.t False
(N,a,h), stable w.r.t False
```

Figure 6.5.: The *show* command lists the current states and transitions, as well as the largest consensus stable sets.

Protocol	*Time*	*Time*$_{Original}$	k	*Safe*
Bakery	0.01072	0.00576	2	True
Dijkstras	0.08581	1.39866	2	True
Burns	0.07966	0.04908	2	True

Table 6.1.: Benchmark results for verifying safety of mutex algorithms. Note that for Dijkstras, the time reported in [AHH16, Section 6] differs from the one we measured by around two orders of magnitude. We suspect this is due to differences between the version that the paper is evaluated on and the version of the original prototype that was made available to us.

Protocol	$Time^{True}$	$Time^{False}$	$Time^{True}_{Original}$	$Time^{False}_{Original}$	k^{True}	k^{False}	True Stable	False Stable
Threshold	0.17515	0.20509	0.00034	0.0003	2	2	False	False
Threshold (Large constant)	2.10702	3.67426	0.00092	0.00046	2	2	False	False
Threshold (Negative coefficients)	0.44961	0.60939	0.00043	0.00036	2	2	False	False
Remainder	0.01163	0.00931	0.00028	0.00027	2	2	False	False
Flock-of-birds	0.00029	0.00624	0.00128	0.00026	1	2	True	False
Flock-of-birds (C=10)	0.00031	0.11886	0.00139	0.0003	1	2	True	False
Flock-of-birds (Tower)	0.00031	0.01224	0.00134	0.00028	1	2	True	False
Flock-of-birds (Alternative)	0.00033	0.00762	0.00138	0.00028	1	2	True	False
Simple Majority	0.0008	0.00067	0.00176	0.00168	1	1	True	True
Approximate Majority	0.00086	0.00022	0.00172	0.00128	1	1	True	True

Table 6.2.: Benchmark results for checking consensus stability of population protocols.

of forward reachability, which is not extensively optimized in our prototype and can take as much as 80% of the analysis time. For Dijkstras, since it is the only example that makes use of process pointers, we suspect that the original implementation does not handle those as fast as the case without them.

The second set of benchmarks consists of population protocols. For those, we want to determine whether they satisfy consensus stability, as defined in Section 6.1.1. We check for each output whether the set of all states of that output is consensus stable. As all protocols in question use the outputs True and False, we need two such checks for each protocol.

Table 6.2 shows the results for this second set of benchmarks. Note that all times are given in seconds. Again, all results were obtained by averaging 40 runs. The columns $Time^{True}$ resp. $Time^{False}$ show the average time needed to determine *True* resp. *False* consensus-stability using our prototype implementation in Python, while the columns $Time^{True}_{Original}$ and $Time^{False}_{Original}$ give those values for the original prototype implementation from [AHH16]. The two columns k^{True} and k^{False} show the value of k for which the result could be determined. Note that the original prototype implementation does not readily output the final value for k in some cases, so these values are obtained from our implementation.

The Threshold, Remainder and Flock-of-birds protocols are given in [Ang+06]. The Flock-of-birds (Tower) protocol is given in [Ang+07]. The Flock-of-birds (Alternative) protocol is given as *Threshold* protocol in [Clé+11]. The approximate majority protocol is given in [AAE08]. In order to see the exact protocols used, consult the benchmarks available in the tool repository at `https://gitlab.lrz.de/philip_offtermatt/`

`viewabstraction-protocolassist`.

The results show that except for the very simple examples like approximate and simple majority, the original prototype implementation runs faster by a large margin. Especially for the very large examples like Threshold with large constants, our implementation is slower by a factor of hundreds to thousands. We suspect that this difference stems again from the implementation of forward reachability. Our implementation uses a fairly naive approach without much optimization, and forward reachability becomes increasingly important as the size of the statespace grows.

One interesting detail can be noticed when comparing the time difference between cases where the answer is True and False for each protocol. The original implementation tends to be slow when the answer is True, and fast when the answer is False. On the other hand, our implementation is fast when the answer is True, and slow when it is False. We suspect this lies again in the implementation of forward reachability. Note that when the answer is True, this can be concluded after running forward reachability and fixpoint iteration k times. However, when the answer is False, forward reachability is run k times, while fixpoint iteration is run $k-1$ times. Since k is fairly small for all our benchmarks, this means the effect of the difference in optimization between forward reachability and fixpoint iteration becomes more pronounced.

7. Conclusion

In this thesis, we examine the view abstraction algorithm for verification of safety properties in parameterized systems, originally introduced in [AHH16]. In this context, we give proofs for some statements that were given without one in the original introduction. In addition, we have reimplemented and verified some of the original benchmarks, and present a publicly accessible implementation for modification and evaluation. We prove correctness of our proposed approaches for some implementation details. Although our implementation does not have the same level of optimization as the original, it presents reasonable performance on the samples we tested.

In addition to the reimplementation of the algorithm, we use view abstraction to check properties of population protocols, which constitute a class of parameterized systems that has been in widespread use in the literature recently. In this context, we introduce a prototype for manual tool-assisted generation of population protocols, which utilizes on-the-fly feedback as a way of enabling humans to make fewer errors when manually designing such protocols.

The property we check on population protocols, and that is used provide feedback to the user, is called consensus stability. We introduce a definition for this property, and perform additional benchmarks for the view abstraction algorithm on it.

In the following, we present some avenues for future work:

Implement contexts and non-atomic transitions. In the original introduction of the view abstraction algorithm, several extensions are presented. Notably, there is an extension of the algorithm to cope with non-atomic transitions, as well as systems that do not have strong downward-closed invariants. Even though these extensions are not necessary in the scope of this thesis, since the algorithm can be applied to the subclass of parameterized systems we focus on in this thesis without them, implementing them is a necessary step of fully verifying and reimplementing the results of [AHH16], so that the full method with its extensions is publicly available in an implementation.

Optimization of interactive protocol generation prototype. Since the tools implemented over the course of this thesis were meant to be protoypes, the focus was not on optimization, but mainly on developing a proof-of-concept of the fundamentals. Therefore, there are ample opportunities for optimization. One approach that seems especially promising would be to use some caching techniques in order to accelerate the forward reachability step of the view abstraction algorithm. This might allow the prototype for interactive protocol generation to handle larger protocols, and could remove delays.

Integration of interactive protocol generation into state-of-the-art tools. In particular, the tool Peregrine, which is used for simulation, automatic verification and generation of population protocols, already provides automatic verification, but lacks capability for on-the-fly feedback. A logical next step in order to enhance usability of the prototype implemented in the context of this thesis would be to integrate it into such an already existing tool.

Integrate other useful forms of feedback. Even though consensus stability might give some hints regarding the correctness of a population protocol, it is not the only type of feedback one might want. Other considerations are to compute on-the-fly the (or one possible) predicate that a protocol is computing. It is possible to verify whether a protocol computes a given predicate, but even this is not something that is fast enough for use as on-the-fly feedback. Deciding which predicate a protocol is computing seems even less feasible in this context. However, it does not seem unreasonable to hope for heuristics that could speed such a verification up in many cases, even though they might not reduce the actual complexity bounds.

Complexity analysis. Even though the view abstraction algorithm has been benchmarked experimentally, there are currently no further results regarding its complexity. This remains an open question, and it might be interesting to find subclasses of parameterized systems where the algorithm can be proven to be fast.

Limitations of view abstraction. In [AHH16], it is claimed that even for classes of systems where the view abstraction is not in general sound and complete, it can often still produce invariants of the system that are strong enough to prove safety. This opens up the question of whether there is some larger class of parameterized systems where the algorithm is not guaranteed to return a correct result, but where there is a weaker guarantee that the algorithm returns an invariant of the system. This invariant might then be useful as an initial input for a different invariant analysis, where starting with more invariants can lead to finding results earlier.

A. Mutual Exclusion Algorithms

This appendix is dedicated to presenting input specifications for the mutual exclusion algorithms evaluated in Section 6.2.

A.1. Burns

```
flag[i] := 0 # begin
goto (exists j < i: flag[j] = 1) begin
flag[i] := 1
goto (exists j < i: flag[j] = 1) begin
goto (exists j > i: flag[j] = 1) wait # wait
flag[i] := 0
goto (True) begin

arrays: flag
```

A.2. Dijkstra

```
flag[i] := 1 # begin
goto ($P = i) endfirstif
goto (flag[$P] != 0) wait # wait
$P := i
goto (exists j != i: flag[j] = 1) begin # endfirstif
flag[i] := 0
goto (True) begin

arrays: flag
process_pointers: $P
```

A.3. Bakery

```
flag[i] := 0 # begin
flag[i] := 1
goto (exists j<i: flag[j]=1) begin # begin
goto (exists j>i: flag[j]=1) waittwo # waittwo
goto (True) begin

arrays: flag
```

List of Figures

List of Tables

Bibliography

[AAE08] D. Angluin, J. Aspnes, and D. Eisenstat. “A simple population protocol for fast robust approximate majority.” In: *Distributed Computing* 21.2 (2008), pp. 87–102.

[Abd+96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. “General decidability theorems for infinite-state systems.” In: *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 1996, pp. 313–321.

[AHH14] P. A. Abdulla, F. Haziza, and L. Holık. “Block Me If You Can!” In: *International Static Analysis Symposium*. Springer. 2014, pp. 1–17.

[AHH16] P. Abdulla, F. Haziza, and L. Holık. “Parameterized verification through view abstraction.” In: *International Journal on Software Tools for Technology Transfer* 18.5 (2016), pp. 495–516.

[AIM10] L. Atzori, A. Iera, and G. Morabito. “The internet of things: A survey.” In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[Ama+15] D. Amaxilatis, M. Logaras, O. Michail, and P. G. Spirakis. “NETCS: A new simulator of population protocols and network constructors.” In: *arXiv preprint arXiv:1508.06731* (2015).

[And+05] G. Andersson, P. Donalek, R. Farmer, N. Hatziargyriou, I. Kamwa, P. Kundur, N. Martins, J. Paserba, P. Pourbeik, J. Sanchez-Gasca, et al. “Causes of the 2003 major grid blackouts in North America and Europe, and recommended means to improve system dynamic performance.” In: *IEEE transactions on Power Systems* 20.4 (2005), pp. 1922–1928.

[Ang+06] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. “Computation in networks of passively mobile finite-state sensors.” In: *Distributed computing* 18.4 (2006), pp. 235–253.

[Ang+07] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. “The computational power of population protocols.” In: *Distributed Computing* 20.4 (2007), pp. 279–304.

[AR09] J. Aspnes and E. Ruppert. “An introduction to population protocols.” In: *Middleware for Network Eccentric and Mobile Applications*. Springer, 2009, pp. 97–120.

[BEJ18] M. Blondin, J. Esparza, and S. Jaax. “Peregrine: A Tool for the Analysis of Population Protocols.” In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 604–611.

[Bil+03] J. Billington, S. Christensen, K. Van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. "The Petri net markup language: concepts, technology, and tools." In: *International Conference on Application and Theory of Petri Nets*. Springer. 2003, pp. 483–505.

[BIS19] M. Bozga, R. Iosif, and J. Sifakis. "Checking deadlock-freedom of parametric component-based systems." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2019, pp. 3–20.

[BK08] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.

[Blo+18] M. Blondin, J. Esparza, S. Jaax, and A. Kučera. "Black Ninjas in the Dark: Formal Analysis of Population Protocols." In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM. 2018, pp. 1–10.

[Blo+19] M. Blondin, J. Esparza, B. Genest, M. Helfrich, and S. Jaax. *Succinct Population Protocols for Presburger Arithmetic*. 2019. arXiv: `1910.04600 [cs.DC]`.

[Bou+00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. "Regular model checking." In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 403–418.

[Bra17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor, Dec. 2017.

[Bur78] J. E. Burns. "Mutual exclusion with linear waiting using binary shared variables." In: *ACM SIGACT News* 10.2 (1978), pp. 42–47.

[Clé+11] J. Clément, C. Delporte-Gallet, H. Fauconnier, and M. Sighireanu. "Guidelines for the verification of population protocols." In: *2011 31st International Conference on Distributed Computing Systems*. IEEE. 2011, pp. 215–224.

[Cze+19] W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki. "The reachability problem for Petri nets is not elementary." In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. ACM. 2019, pp. 24–33.

[Dij65] E. W. Dijkstra. "Solution of a problem in concurrent programming control." In: *Communications of the ACM* 8.9 (1965), p. 569.

[Esp+17] J. Esparza, P. Ganty, J. Leroux, and R. Majumdar. "Verification of population protocols." In: *Acta Informatica* 54.2 (2017), pp. 191–215.

[Esp14] J. Esparza. "Keeping a crowd safe: On the complexity of parameterized verification (invited talk)." In: *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2014.

[Esp17] J. Esparza. *Petri Nets Lecture Notes*. Accessed on 30.09.2019. June 2017.

[Haz15] F. Haziza. *Parameterized Verification through View Abstraction - Experiments*. Accessed on 30.09.2019. Mar. 2015.

[Hoa78] C. A. R. Hoare. "Communicating sequential processes." In: *The origin of concurrent programming*. Springer, 1978, pp. 413–443.

[JL98] H. E. Jensen and N. A. Lynch. "A proof of burns n-process mutual exclusion algorithm using abstraction." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1998, pp. 409–423.

[JM97] J.-M. Jazequel and B. Meyer. "Design by contract: The lessons of Ariane." In: *Computer* 30.1 (1997), pp. 129–130.

[Kru72] J. B. Kruskal. "The theory of well-quasi-ordering: A frequently discovered concept." In: *Journal of Combinatorial Theory, Series A* 13.3 (1972), pp. 297–305.

[Lyn96] N. A. Lynch. *Distributed algorithms*. Elsevier, 1996. Chap. 10.6.

[Nip+19] T. Nipkow, H. Seidl, J. Esparza, and J. Křetınsk. *Lecture Notes Einführung in die Theoretische Informatik*. `https://www21.in.tum.de/teaching/theo/SS19/folien-handout.pdf`. Apr. 2019.

[Pau94] L. C. Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[TW10] L. Tan and N. Wang. "Future internet: The internet of things." In: *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*. Vol. 5. IEEE. 2010, pp. V5–376.

[Vis+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. "Model checking programs." In: *Automated software engineering* 10.2 (2003), pp. 203–232.