Algorithmic Analysis of Infinite-State Systems

by

Naghmeh Hassanzadeh Ghaffari

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Many important software systems, including communication protocols and concurrent and distributed algorithms generate *infinite* state-spaces. Model-checking which is the most prominent algorithmic technique for the verification of concurrent systems is restricted to the analysis of finite-state models. Algorithmic analysis of infinite-state models is complicated—most interesting properties are undecidable for sufficiently expressive classes of infinite-state models.

In this thesis, we focus on the development of algorithmic analysis techniques for two important classes of infinite-state models: *FIFO Systems* and *Parameterized Systems*.

FIFO systems consisting of a set of finite-state machines that communicate via unbounded, perfect, FIFO channels arise naturally in the analysis of distributed protocols. We study the problem of computing the set of reachable states of a FIFO system composed of *piecewise* components. This problem is closely related to calculating the set of all possible channel contents, i.e. the *limit language*. We present new algorithms for calculating the limit language of a system with a single communication channel and important subclasses of multi-channel systems. We also discuss the complexity of these algorithms. Furthermore, we present a procedure that translates a piecewise FIFO system to an abridged structure, representing an expressive abstraction of the system. We show that we can analyze the infinite computations of the more concrete model by analyzing the computations of the finite, abridged model.

Parameterized systems are a common model of computation for concurrent systems consisting of an arbitrary number of homogenous processes. We study the reachability problem in parameterized systems of *infinite-state* processes. We describe a framework that combines Abstract Interpretation with a backward-reachability algorithm. Our key idea is to create an abstract domain in which each element (a) represents the lower bound on the number of processes at a control location and (b) employs a numeric abstract domain to capture arithmetic relations among variables of the processes. We also provide an extrapolation operator for the domain to guarantee sound termination of the backward-reachability algorithm.

Acknowledgements

I would like to thank my advisor Prof. Richard Trefler for encouraging me to do research in the field of software verification. I am deeply grateful for his encouragement, understanding, and constant support during the course of my studies. His insightful comments improved the content and presentation of this thesis.

I would like to express my gratitude to Prof. Tevfik Bultan, Prof. John Thistle, Prof. David Toman, and Prof. Grant Weddell who graciously agreed to be on my thesis committee and read and evaluated this thesis. I am very grateful for their insightful suggestions for improving the content of this thesis. I would like to thank Prof. Jo Atlee for her support and discerning feedback at the early stages of my research.

I wish to thank Dr. Arie Gurfinkel who significantly influenced much of the work in this thesis. His enthusiasm, wisdom, and knowledge have been a great source of inspiration for me. I have been very fortunate to benefit from his knowledge and intuition and also to have him as a supporting friend throughout my studies. I would like to thank Dr. Nils Klarlund who taught me how to formulate definitions and theoretical cocepts precisely and clearly. He has provided invaluable suggestions and wise feedback on many of my early ideas for my research on FIFO systems.

I would like to express my deep gratitude to Prof. Alan Hu for hosting me as a visiting student in his lab at the University of British Columbia. Discussions with him have always served to broaden my perspective on both technical and non-technical matters.

My research has partly been supported by the Doctorate scholarship from Natural Sciences and Engineering Research Council (NSERC) of Canada and Ontario Graduate Scholarship (OGS). I am grateful to both of these organizations.

I have learned a lot from discussions with my fellow graduate students, Shoham Ben-David and Zarrin Langari. I am thankful for their support and friendship. I am also thankful for my friends Alma Juarez and Panthea Sepehrband at the University of Waterloo and Dr. Domagoj Babic, Flavio de Paula, and Zvonimir Rakamaric at the University of British Columbia. I am also indebted to my friends Dr. Azadeh Riahi and Delaram Fakhrai for their love and support.

I am greatly thankful to my brother, Dr. Shahab H. Ghaffari, and my sister-in-law, Bahar Taravati, who have encouraged and supported me throughout my education. I am deeply grateful to my mother, Dr. Minoo Rafiee, and my father, Mehdi H. Ghaffari, for the encouragement and continuous love and support over the years. The last but the most, my everlasting gratitude goes to my husband, Peyman, for always believing in me and for his enduring love and support during the highs and lows of the past few years. It is very difficult and a bit silly to thank your family formally; they provide something that goes beyond what can be "thanked". It is for this reason that this thesis is dedicated to Peyman and to my parents.

To my parents

To Peyman

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Current hardware and software systems are highly complex, yet their use in safety critical areas from network communications to aerospace, from healthcare to e-commerce, and others, requires an exceptionally high level of reliability. These systems are now used ubiquitously, and thus the consequences of their failures have become more and more severe. Several system failures and outages caused by an error in hardware or software systems have been documented such as the Arian-5 crash [50], the Therac-25 malfunctioning [85], the North-East blackout of 2003, which to a large extend was attributed to a software failure, and others. A survey conducted by Department of Commerce's National Institute of Standards and Technology (NIST) claims that software errors cost approximately $59.5 billion dollars annually to the US economy alone[1].

Clearly, the need for reliable hardware and software systems is critical. As the dependency of our lives to these systems increases, it will become even more important to develop methods that increase our confidence in their correctness.

The principal validation methods for complex software and hardware systems are based on either extensive *simulation* and *testing*, or *formal verification*. Simulation and testing both involve providing certain inputs and observing the corresponding outputs. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. These methods can be a cost-efficient way to find many

[1] http://www.nist.gov/public_affairs/releases/n02-10.htm

errors. However, checking *all* of the possible interactions and potential pitfalls using simulation and testing is rarely possible. On the other hand, formal verification methods, based on mathematical principles, perform an analysis of all possible behaviors and can find some significant errors that may be missed during simulation and testing.

The main topic of this thesis is development of formal verification techniques for analysis of software systems. In this chapter, first, we give a brief overview of formal verification techniques in general. Then, we present the main contributions of this thesis.

1.1 Formal Verification

The goal of formal verification is to decide whether an artifact, such as a program P, is correct with respect to its specification (or a correctness property) ϕ. Approaches to formal verification are typically classified as either *deductive* or *algorithmic*. The term *deductive verification* normally refers to the use of axioms and proof rules to establish the correctness of a system. In a deductive approach, both the system and its specification are described as logical formulae, and the verification problem is reduced to establishing whether the formula $P \implies \phi$ is valid. Despite the generality of the approach, deductive verification is a time-consuming process that can be effectively performed by experts who are educated in logical reasoning. Automated theorem provers (cf. [22, 68]) may be used to assist with simplification and proof management but it still requires substantial human effort to carry out a proof with a theorem prover.

In algorithmic methods, the system is represented as a finite-state machine and the correctness property as a logical formula; the verification problem is reduced to an algorithmic decision procedure. Algorithmic verification is far less general than deductive approaches; for instance, there cannot be an algorithm that decides whether an arbitrary program terminates. On the positive side, algorithmic approaches are highly automated, which make them applicable to very large systems.

One of the most prominent algorithmic verification techniques is *model-checking* [35, 96]. It is an automated technique for verifying finite-state concurrent systems. It arose

from the insight that many systems have a finite number of states; hence, the truth of a correctness formula over such a system can be determined by an exhaustive exploration of the global state transition graph of the system based on the structure of the correctness formula.

In a landmark paper, Pnueli [94] proposed the use of *temporal logic* for reasoning about the correctness of systems that maintain an ongoing interaction with their environment, so called *reactive systems*. Temporal logic and its variants can assert how the behavior of the system evolves over time. In the early 1980's, Clarke and Emerson [35], and independently Queille and Sifakis [96], introduced a model-checking algorithm for branching time logic CTL (Computational Tree Logic) that is polynomial in both the size of the model and the length of the correctness formula. Later, Clarke, Emerson, and Sistla [36] improved this algorithm to be linear in the product of the length of the formula and the size of the state transition graph. Sistla and Clarke [100, 101] analyzed the model-checking problem for a variety of temporal logics and showed, in particular, that for Linear Temporal Logic (LTL), the problem was PSPACE-complete.

An interesting and useful feature of most model-checking algorithms is the generation of counter-example execution traces if the desired property is not satisfied. A counter-example is an execution of the model whose presence invalidates the correctness property. Model-checking has been successfully applied to verification of hardware designs and telecommunication protocols, e.g. [32, 87]. Several industrial-strength model-checkers are available from such companies as IBM [11], Cadence [87], and Microsoft [9].

Over the years, three major approaches for implementing model-checking algorithms have emerged: explicit state, symbolic, and SAT-based model-checking. In the explicit approach, exemplified by SPIN [70], transition relations are represented explicitly and the model-checking algorithm is based on a combination of graph exploration techniques to allow (typically a partial) exploration of the system state graph. This approach is quite practical for concurrent systems with a small number of processes, where the number of states is usually small. However, in systems with many concurrent processes, the num-

ber of states is usually too large to be handled by exploration algorithms. The symbolic approach exemplified by SMV [87] and SLAM [9] uses a *symbolic* representation for the state transition graphs in which transition relations are represented implicitly by *ordered binary decision diagrams* (OBDDs) [25]. The main advantage of this approach is that it makes it possible to verify systems with very large state spaces since it does not depend directly on the number of states but rather on an efficient representation of the transition relation and sets of states occurring during the analysis. The explicit algorithm presented in [35] is able to check state transition graphs with 10^4 to 10^5 states, while implementing the same algorithm using a symbolic approach, it becomes possible to verify some examples with more than 10^{20} states [32]. SAT-based approaches, as exemplified by CBMC [34, 61], efficiently explore a state transition graph of the system up to a bounded-depth by encoding the model-checking problem by a propositional formula and using a SAT-solver for the actual analysis.

There are still many challenges facing wide-spread adoption of model-checking as a practical software verification technique. Although model-checking algorithms have time complexity that is linear in the size of the structure, the size of the structure (i.e. the number of states) may itself be exponential in the size of its description as a program. For instance, a program with n Boolean variables may have a reachable state space of size 2^n. This phenomenon referred to as the *state explosion problem*, is the main obstacle to the application of model-checking to the analysis of realistic systems. In addition, the restriction to the systems with a finite number of states excludes a large number of interesting reactive systems, such as distributed protocols, from the range of systems to which model-checking is applicable directly.

Many important concurrent and distributed software systems have infinite state spaces. Thus, it is crucial to be able to analyze infinite-state models. Indeed, even though all physically constructible systems are finite in some sense, their size is often an implementation parameter that is typically left unspecified. Modeling such systems as infinite-state systems or systems with unbounded state spaces is often more appropriate. Another reason is that the techniques developed for the analysis of infinite-state models are also

applicable to the analysis of systems with finite but very large state spaces.

1.2 Contributions of This Thesis

A variety of systems, including communication protocols, concurrent algorithms, and real-time automata generate infinite state spaces. In addition, infinite-state models provide a useful abstraction for the analysis of realistic protocols in that they simplify the semantics of specification languages and free the protocol designers from implementation details. However, algorithmic analysis of infinite-state models is very complicated–most interesting properties are undecidable for sufficiently expressive classes of infinite-state models. For instance, the result of their reachability analysis cannot be expressed as the explicit enumeration of all their reachable states. The undecidability of infinite-state models has led naturally to two forms of analysis: (i) identifying practically useful subclasses with decidable properties, and (ii) investigating applicable semi-algorithms based on induction and abstraction that scale to realistic examples.

The main contribution of this thesis is to address some challenges facing model-checking of infinite-state systems. The focus of the thesis is on algorithmic analysis of two important classes of infinite-state systems: *FIFO systems* and *parameterized systems*, which are described below.

FIFO Systems A FIFO system consists of several finite-state machines that communicate via unbounded perfect First-In First-Out (FIFO) channels. FIFO systems are a common model of computations for describing concurrent and distributed protocols such as IP-telecommunication protocols, composite web services, and System on Chip (SoC) architectures (e.g., [24, 14, 1, 93, 33, 16, 105, 60, 29]). While unboundedness of communication channels provides a useful modeling abstraction, it complicates the analysis–a single unbounded channel is sufficient to simulate the tape of a Turing machine [24]. Hence, verification of any non-trivial property such as reachability is undecidable.

In this thesis, we show that by restricting attention to systems composed of *piecewise* components automated system analysis is possible even when the components communi-

cate over unbounded perfect FIFO channels. We study decidability of several verification problems for the class of piecewise FIFO systems. We provide model-checking algorithms based on techniques that reduce large structures to smaller ones while preserving a number of properties of interest.

Parameterized Systems A parameterized system is a family of systems in which n processes execute the same program concurrently. The problem of parameterized verification is to decide whether for all values of n, the system with n processes is correct. Parameterized systems are of significant interest as they can describe a wide range of concurrent protocols such as mutual-exclusion and leader election, to distributed systems such as web-services, to cache coherence, resource sharing, transactional memory protocols and others (e.g. [55, 51, 56, 43, 54, 53, 18, 3, 49]. Parameterized systems have arbitrarily large (or unbounded) state spaces from the unbounded number of instances, each with a fixed number of processes.

The current practice is to use model-checking to determine correctness of a few instances of a parameterized protocol. This approach has a strong similarity to testing, and all of the disadvantages that go with it. On the other hand, automated verification of parameterized systems is undecidable in general [7]. The approach put forward in this thesis is to provide sound, automated, and terminating model-checking algorithms for the analysis of parameterized systems. Since this is an undecidable problem, such algorithms would be incomplete.

The contributions of this thesis are discussed in more detail in the following sections.

1.2.1 Decidability Results on Piecewise FIFO Systems

A part of this thesis studies the class of *piecewise FIFO systems*. These systems can be used for modeling distributed protocols such as IP-telecommunication protocols and interacting web services [77, 66]. A piecewise FIFO system is composed of components whose behaviors can be expressed by *piecewise languages*, a subclass of regular languages. Intuitively, a language is piecewise if it is accepted by a non-deterministic

finite-state automaton whose only non-trivial strongly connected components are states with self-loops.

Previous work has shown that piecewise languages play an important role in the study of FIFO systems [77]. In this thesis, we study verification problems for piecewise FIFO systems. We show that verification of non-trivial properties of FIFO systems is closely related to calculating all possible channel contents that may arise from an initial state, i.e. the *limit language*. This problem is undecidable in general. Moreover, the limit language is not necessarily regular, even if the initial language is [33], and even when the limit language is known to be regular, determining it may still be undecidable [33].

For single-channel piecewise FIFO systems we show that the limit language is regular (piecewise) if the initial channel language is regular (piecewise). For multi-channel piecewise system, we show that the limit language is not regular in general. However, we are able to establish the regularity of the limit language by excluding the *conditional actions* from the actions allowed by piecewise components. Conditional actions increase the expressive power of the model by allowing a message being written on a channel only if a message first being read. We further show that in the presence of conditional actions, the limit language is piecewise if the initial channel language is piecewise. However, the construction of the limit language may not always be effective. These results were first reported in [64].

1.2.2 Algorithmic Analysis of Piecewise FIFO Systems

The problem of computing the set of reachable states of a piecewise FIFO system can be reduced to calculating the limit language at each control location.

We present two new algorithms for computing the limit language of a system with a single communication channel and a class of multi-channel system in which messages are not passed around in cycles through different channels. In both algorithms, we use automata to represent and manipulate the set of possible channel configurations. These algorithms were first reported in [63].

The algorithm for single-channel systems requires that components be piecewise, and it applies to any regular initial channel content. We show that the worst case complexity of the algorithm is, at most, exponential in the size of the automaton that represents the language of the initial channel content.

The algorithm for the multi-channel systems requires that both the components and the initial contents of the channels be piecewise, and that the *communication graph* be acyclic. A communication graph is a graph with channels as vertices and conditional actions as edges indicating which channels are connected by these actions. To aid the presentation, we develop the algorithm incrementally by restricting the topology of the communication graphs to *star*, *tree*, *inverted tree*, and *directed acyclic graph* (*DAG*) topologies. We study the worst case complexity of the algorithm for each topology. We show that for the *star* and *tree* topologies the worst case complexity of the algorithm is exponential in the size of the automaton that represents the language of the initial content of the channel in the origin of the star and the root of the tree, respectively.

We note that, in general, limit languages do not represent system computations but rather the reachable state set. In order to reason about computations of piecewise FIFO systems, we present a procedure that, given a piecewise FIFO systems, constructs an abridged structure, representing an expressive abstraction of the system. We show that the construction procedure of the abridged model terminates in piecewise FIFO systems with acyclic communication graphs. Furthermore, we show that we can analyze the infinite computations of the more concrete model by analyzing the computations of the finite, abridged model. The preliminary results of this work was published in [66].

1.2.3 Reachability in Parameterized Systems

It is well-known that parameterized systems are undecidable in general; in fact, it has been shown that verification of parameterized systems of *finite-state* processes is undecidable [7].

In this thesis, we focus on the analysis of parameterized systems of *infinite-state* processes. This setting is common in practice. For example, in the Lamport's bakery

protocol [81], each process maintains an integer ticket, and therefore, has an infinite state-space. We require a sound, automated, and terminating procedure. Since this is an undecidable problem, such a procedure would be incomplete.

We present a new technique for the analysis of parameterized systems of *infinite-state* processes. We describe a framework that combines Abstract Interpretation with a backward-reachability algorithm for verifying safety properties of parameterized systems [65].

Our key idea is to create an abstract domain in which each element (a) represents the lower bound on the number of processes at a control location and (b) employs a numeric abstract domain to capture arithmetic relations among variables of the processes. We also provide an extrapolation operator for the domain to guarantee sound termination of the backward-reachability algorithm. Our abstract domain is sufficiently generic to be instantiated by different well-known numeric abstract domains such as octagons [89] and polyhedra [48]. This makes the framework applicable to a wide range of parameterized systems. We illustrate an implementation of our algorithm on a variant of Lamport's bakery mutual-exclusion protocol (Alg. 2 in [88]).

1.3 Organization

The rest of the thesis is structured as follows.

In Chapter 2, we present our notation and provide the definitions of automata and regular languages. We introduce piecewise languages and describe their properties. We also give a brief overview of Abstract Interpretation. In Chapter 3, we describe piecewise FIFO systems and present decidability results. Chapter 4 describes the algorithms for computing the limit language in single-channel and multi-channel piecewise FIFO systems. It also describes a procedure for constructing an abridged model of piecewise FIFO systems. Chapter 5 describes the parameterized systems considered in this thesis and presents our approach for verification of their safety properties. Finally, we conclude in Chapter 6 with a summary of the thesis and an outline of directions for future research.

Chapter 2

Background

This chapter contains definitions of various concepts that are used throughout this thesis. They include the syntax and semantics of regular languages and finite automata, piecewise languages and their properties, and a brief overview of Abstract Interpretation [46].

2.1 Regular Languages and Finite Automata

An alphabet, denoted by Σ, is a nonempty finite set of symbols. A word (string) w is a finite sequence of symbols whose length is denoted by $|w|$. The set Σ^* is the set of all finite words on Σ. The empty word is denoted by ϵ. Let w_1 and w_2 be two words in Σ^*. In the sequel, $w_1 + w_2$ denotes the non-deterministic choice between w_1 and w_2 and $w_1 \cdot w_2$ denotes concatenation of the elements of w_1 and w_2. We sometimes omit '$\cdot$', i.e. we may write $w_1 w_2$ instead of $w_1 \cdot w_2$.

A *finite-state automaton* is a mathematical model of a system with discrete inputs and outputs and is formally defined as follows:

Definition 1 (FSA) *[71] A* finite-state automaton (FSA) A *is a tuple* $(\Sigma, Q, q^0, \delta, F)$*, where* Σ *is a finite alphabet;* Q *is a finite set of states;* $q^0 \in Q$ *is the initial state;* $\delta : Q \times \Sigma \to \mathbf{2}^Q$ *is the transition relation; and* $F \subseteq Q$ *is a set of accepting (or final) states. When* F *is omitted, it is assumed that* $F = Q$*.*

A finite-state automaton as defined above is also called a *nondeterministic finite automaton* (NFA). A finite automaton is *deterministic* (DFA) if for each state $q \in Q$ and symbol $a \in \Sigma$, $|\delta(q,a)| \leq 1$. Note that every DFA is an NFA [71].

For $a \in \Sigma$ we write $\delta(q,a,q')$ or $q \xrightarrow{a} q'$ to mean that $q' \in \delta(q,a)$. We write $q \rightarrow q'$ when we do not distinguish the specific symbol on the transition of q to q'. Given $q \in Q$, and $w \in \Sigma^*$, $\delta(q,w)$ is defined as usual: $\delta(q,\epsilon) \triangleq \{q\}$, and $\delta(q,wa) \triangleq \{p \mid \exists r \in \delta(q,w),\, p \in \delta(r,a)\}$. We say that a word w is accepted by A if and only if $(\delta(q^0,w) \cap F) \neq \emptyset$. The language of A is defined as $\mathcal{L}(A) \triangleq \{w \in \Sigma^* \mid \delta(q^0,w) \cap F \neq \emptyset\}$. We define the *size* of an FSA A as: $|A| \triangleq |Q| + |\delta|$. A *run* in A is a finite or infinite sequence of states denoted $P = q_0 \rightarrow q_1 \rightarrow \ldots$, where q_0 is the initial state and for all i, $q_i \rightarrow q_{i+1} \in \delta$. It is worth noting that for every NFA we can construct an equivalent DFA, i.e. one that accepts the same language.

The languages accepted by finite automata are described by expressions called *regular expressions*. A regular expression (RE) over Σ is defined by the following grammar $R ::= a \in \Sigma \mid R \cdot R \mid R + R \mid R^* \mid \mathbf{0} \mid \mathbf{1}$. The symbol $\mathbf{0}$ denotes the empty language, and $\mathbf{1}$ denotes the language $\{\epsilon\}$; in particular, we have $\mathbf{1} = \mathbf{0}^*$. We sometimes write ϵ instead of $\mathbf{1}$.

The language $\mathcal{L}(R)$ of a RE R is defined in the usual way. We sometimes write R to mean $\mathcal{L}(R)$. In a further abuse of notation, we often regard a set $M \subseteq \Sigma \cup \{\epsilon\}$ as an RE, namely the sum of elements in M. For a language $L \subseteq \Sigma^*$, we use $\complement L$ to denote the complement of L: $\Sigma^* \backslash L$. The expression $test(R)$ is $\mathbf{1}$ if $\mathcal{L}(R) \neq \emptyset$ and $\mathbf{0}$ if $\mathcal{L}(R) = \emptyset$.

We often use RE notation with automata. For example, $A_1 \cdot A_2$ stands for concatenation of two automata, $A_1 + A_2$ for an automaton with language $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.

Regular languages are closed under Boolean operators union, intersection and complement, concatenation and Kleen star operators, substitution, homomorphism and inverse homomorphism[1]. Moreover, there exist efficient algorithms to decide whether the

[1] A homomorphism h is a substitution such that $h(a)$ contains a single string for each a. The inverse homomorphic image of a language $\mathcal{L}$ is

$$h^{-1}(\mathcal{L}) \quad = \quad \{x \mid h(x) \text{ is in } \mathcal{L}\}.$$

language accepted by a regular expression is empty, finite, or infinite. It is also decidable to determine if two finite automata accept the same language.

Another operator that is defined on regular expressions is called *left residual operation* or *derivative* [26]:

Definition 2 (Derivative of RE) *Given a regular expression R and a finite word s, the* derivative *of R with respect to s is denoted $s^{-1}R$ and is $\mathcal{L}(s^{-1}R) = \{t \mid s \cdot t \in \mathcal{L}(R)\}$.*

Let $a, b \in \Sigma$ and R, S be regular expressions. Then,

$$\begin{aligned} a^{-1}\mathbf{0} &\triangleq \mathbf{0} \\ a^{-1}\mathbf{1} &\triangleq \mathbf{0} \\ a^{-1}b &\triangleq test(a \cap b) \\ a^{-1}(R \cdot S) &\triangleq ((a^{-1}R) \cdot S) + (test(R \cap \mathbf{1}) \cdot (a^{-1}S)) \\ a^{-1}(R + S) &\triangleq (a^{-1}R) + (a^{-1}S) \\ a^{-1}(R^*) &\triangleq (a^{-1}R) \cdot R^* \end{aligned}$$

Similarly, we may define a residual operation for M^*, where $M \subseteq \Sigma$:

$$\begin{aligned} (M^*)^{-1}\mathbf{0} &\triangleq \mathbf{0} \\ (M^*)^{-1}\mathbf{1} &\triangleq \mathbf{1} \\ (M^*)^{-1}a &\triangleq a + test(a \cap M) \\ (M^*)^{-1}(R \cdot S) &\triangleq (((M^*)^{-1}R) \cdot S) + (test(R \cap M^*) \cdot ((M^*)^{-1}S)) \\ (M^*)^{-1}(R + S) &\triangleq ((M^*)^{-1}R) + ((M^*)^{-1}S) \\ (M^*)^{-1}(R^*) &\triangleq (((M^*)^{-1}R) \cdot R^*) + \mathbf{1} \end{aligned}$$

Then, it can be verified that

$$\mathcal{L}((M^*)^{-1}R) \;=\; \{v \mid \exists u \in \mathcal{L}(M^*),\, u \cdot v \in \mathcal{L}(R)\}\,.$$

We conclude this section with a review of recognizable (or regular) relations.

Definition 3 (Recognizable Relation) *[107] A relation $\rho \subseteq (\Sigma^*)^K$ is* recognizable *(or regular) if and only if*

$$\rho \;=\; \bigcup_{0 \le i < I} \mathcal{L}(R_0^i) \times \cdots \times \mathcal{L}(R_{K-1}^i)$$

for some natural number I and regular expressions R_j^i over Σ.

Proposition 1 *[107] Let ρ be a K-ary relation over Σ^*. Define $\mathcal{L}^{\#}(\rho) \triangleq \{w_0 \cdot \# \cdots \# \cdot w_{K-1} \mid (w_0, \ldots, w_{K-1}) \in \rho\}$. Then $\mathcal{L}^{\#}(\rho)$ is a regular language over $\Sigma \cup \{\#\}$ if and only if ρ is recognizable.*

It is easy to see that regular relations are closed under finite unions and intersections.

2.2 Piecewise Languages

Piecewise languages[2] [77, 21] are a subclass of regular languages that restrict the Kleen star operation to be applied to sets of letters only. More simply, piecewise languages are those recognized by nondeterministic automata whose only nontrivial, strongly-connected components are states with self-loops. Piecewise languages and their characteristics were first presented in [77, 21]. Piecewise languages are defined formally as follows:

Definition 4 (Piecewise Languages) *A language is* simply piecewise *if it can be expressed by an* RE *of the form $M_0^* a_0 \cdots a_{n-1} M_n^*$, where each $M_i \subseteq \Sigma$ and $a_i \in \Sigma \cup \{\epsilon\}$. A* piecewise *language is a finite (possibly empty) union of simply piecewise languages. A language L is* repetition piecewise *if for all i, a_i is ϵ.*

For example, $(a + b)^* c$ is simply piecewise, where $M_0 = \{a, b\}$ and $a_0 = c$, but $(ab)^* c$ is not piecewise.

Definition 5 (PO-FSA) *A* partially-ordered automaton (PO-FSA) *is a tuple $(A, \preceq)$, where $A = (\Sigma, Q, q^0, \delta, F)$ is an automaton, and $\preceq \subseteq Q \times Q$ is a partial order on states such that $q' \in \delta(q, a)$ implies that $q \preceq q'$.*

[2]The name "Alphabetic Pattern Constraints" has been suggested [21].

The following proposition can be used to decide whether a given regular language is piecewise; an efficient algorithm is provided in [21]. We construct a new proof for this proposition.

Proposition 2 *A language is piecewise if and only if it is recognized by a PO-FSA.*

Proof: ($\Leftarrow$) Consider the PO-FSA $A = ((\Sigma, Q, \delta, q^0, F), \preceq)$. Consider all acyclic runs $P = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{k-2}} q_{k-1} \xrightarrow{a_{k-1}} q_k$, where any $q_i \in Q$ for $i \in [0..k]$ occurs at most once and $q_0 = q^0$ is initial and $q_k \in F$ is an accepting state. The number of such runs is finite. For each q_i we can associate M_i, the set of a's such that $\delta(q_i, a) = q_i$. Let $L_P = M_0^* a_0 M_1^* ... M_k^*$. Then, $L_P \subseteq \mathcal{L}(A)$. Let $L' = \bigcup_P L_P$, where the union is over all appropriate runs P as just considered. We clearly have that $L' \subseteq \mathcal{L}(A)$. To see that $\mathcal{L}(A) \subseteq L'$, we use that automaton A is partially ordered. Consider $w \in \mathcal{L}(A)$. Thus, w defines a run P from which an acyclic run P' for a word w' can be constructed by deleting letters a_i in w for which $\delta(q_i, a_i) = q_i$. Then, w' is a scattered subword of $w : w = u_0 b_0 \ldots b_n u_{n+1}$, where $w' = b_0 \ldots b_n$ and u_i, for $0 \leq i \leq n+1$, is in M_i^*. Hence, $w \in L_{P'}$. As a result, $\mathcal{L}(A) \subseteq L'$.

($\Rightarrow$) A piecewise language is a finite union of simply piecewise languages. Each simply piecewise language is recognized by a totally-ordered automaton. Consider the simply piecewise language $M_0^* a_0 \cdots M_{k-1}^* a_{k-1} M_k^*$. This language is recognized by an automaton $A = (\Sigma, Q, \delta, q^0, F)$, where $Q = \{q_0, q_1, \ldots, q_k\}$, $q^0 = q_0$, and $F = \{q_k\}$. The transition relation δ is defined as follows. For $i \in [0..k]$,

$$\begin{aligned}(q, a, q') \in \delta \quad \Leftrightarrow \quad & (q = q_{i-1} \wedge q' = q_i \wedge a = a_{i-1}) \vee \\ & (q = q' = q_i \wedge a \in M_i)\,.\end{aligned}$$

By construction, for $i, j \in [0..k]$, the relation $q_i \preceq q_j \Leftrightarrow i \leq j$ is a total order satisfying the constraint of Definition 5. Let $L = L_1 + L_2$ be a piecewise language, where L_1 and L_2 are simply piecewise languages recognized by PO-FSAs $A_1 = ((\Sigma, Q_1, \delta_1, q_1^0, F_1), \preceq_1)$ and $A_2 = ((\Sigma, Q_2, \delta_2, q_2^0, F_2), \preceq_2)$, respectively. Then, L is recognized by a PO-FSA $A = ((\Sigma, Q, \delta, q^0, F), \preceq)$, where $Q = Q_1 \cup Q_2 \cup q^0$, and q^0 is a new state not appearing

in $Q_1 \cup Q_2$, $F = F_1 \cup F_2$, and δ is defined as follows:

$$\begin{aligned}(q, a, q') \in \delta \quad \Leftrightarrow \quad & (q, q' \in Q_1 \wedge (q, a, q') \in \delta_1) \vee \\ & (q, q' \in Q_2 \wedge (q, a, q') \in \delta_2) \vee \\ & (q = q^0 \wedge ((q_1^0, a, q') \in \delta_1 \vee (q_2^0, a, q') \in \delta_2))\end{aligned}$$

It is easy to see that A is a PO-FSA with the partial order $\preceq$ defined as follows:

$$q \preceq q' \Leftrightarrow \begin{cases} \text{true} & \text{if } q = q^0 \\ q \preceq_1 q' & \text{if } q, q' \in Q_1 \\ q \preceq_2 q' & \text{if } q, q' \in Q_2 \end{cases}$$

□

The following proposition summarizes the properties of the piecewise languages.

Proposition 3 *Piecewise languages are closed under finite unions* $(+)$*, finite intersections* $(\cap)$*, concatenation* $(\cdot)$*, shuffle* $(||)$ [1]*, letter-to-letter mappings, and inverse homomorphisms, but not under complementation and homomorphisms.*

Proof: **Finite unions, intersections, and concatenation.** Closure under finite unions and concatenation follows immediately from Definition 4. Closure under finite intersections is shown in [21], Proposition 1.

Shuffle. To show that piecewise languages are closed under shuffle, we show that PO-FSAs are closed under shuffle. Let L_1 and L_2 be two piecewise languages recognized by PO-FSAs, $A_1 = ((\Sigma, Q_1, \delta_1, q_1^0, F_1), \preceq_1)$ and $A_2 = ((\Sigma, Q_2, \delta_2, q_2^0, F_2), \preceq_2)$, respectively. Let $L = L_1||L_2$. Then, L is recognized by a PO-FSA, $A = ((\Sigma, Q, \delta, q^0, F), \preceq)$, where $Q = Q_1 \times Q_2$, $q^0 = (q_1^0, q_2^0)$, and $F = F_1 \times F_2$. The transition relation δ is defined as follows:

$$((q_1, q_2), a, (q_1', q_2')) \in \delta \quad \Leftrightarrow \quad ((q_1, a, q_1') \in \delta_1 \wedge q_2 = q_2') \vee ((q_2, a, q_2') \in \delta_2 \wedge q_1 = q_1')$$

[1] The shuffle of two words w and w', $w \,||\, w'$, is the set of words that are obtained by braiding w and w'; for example $ab \,||\, cd = \{abcd, acbd, acdb, cabd, cdab, cadb\}$. In the sequel, $\mathcal{L} \,||\, \mathcal{L}' = \bigcup_{\forall w \in \mathcal{L}, w' \in \mathcal{L}'} w \,||\, w'$.

It is easy to see that A is a PO-FSA, with the partial order $\preceq$ defined as follows:

$$(q_1, q_2) \preceq (q_1', q_2') \Leftrightarrow \begin{cases} q_1 \preceq_1 q_1' & \text{if } q_2 = q_2' \\ q_2 \preceq_2 q_2' & \text{if } q_1 = q_1' \end{cases}$$

Therefore, by Proposition 2, the language of the shuffled automaton A is piecewise.

Letter-to-letter mappings. Let $T : \Sigma \to \Sigma$ be a letter-to-letter mappings over a finite alphabet Σ. Consider simply piecewise language $M_0^* a_0 M_1^* \dots M_k^*$, where $M_0 = \{b_0, \dots, b_i\}, M_1 = \{c_0, \dots, c_j\}$, and so on. Applying T on this language results in the simply piecewise language $M_0'^* a_0' M_1'^* \dots M_k'^*$ where $M_0' = \{T(b_0), \dots, T(b_i)\}$, $a_0' = T(a_0), M_1' = \{T(c_0), \dots, T(c_j)\}$ and so on. Clearly, this is a simply piecewise language. Since T distributes over union, the result follows for arbitrary piecewise languages as well.

Inverse homomorphisms. Let $A = ((\Sigma, Q, \delta, q^0, F), \preceq)$ be a partially-ordered automaton accepting piecewise language L. Let Δ be an alphabet, and h a homomorphism from Δ to Σ^*. We construct automaton A' over Δ that accepts $h^{-1}(L)$. Intuitively, A' works by reading a symbol a in Δ and simulating PO-FSA A on $h(a)$. Formally, let $A' = ((\Delta, Q, \delta', q^0, F), \preceq)$, and define $\delta'(q, a)$, for $q \in Q$ and $a \in \Delta$ to be $\delta(q, h(a))$. Since $h(a)$ may be a long string or ϵ, δ is defined on all strings by extension. It is easy to show by induction on $|x|$ that $\delta'(q^0, x) = \delta(q^0, h(x))$. Therefore, A' accepts x if and only if A accepts $h(x)$. That is , $\mathcal{L}(A') = h^{-1}(\mathcal{L}(A))$. The transition relation of A', δ', simulates the transition relation of A on $h(x)$ for any symbol $x \in \Delta$, thus it respects the partial order relation on states of A. Hence, $\mathcal{L}(A')$ is also piecewise.

Homomorphism. Piecewise languages are not closed under homomorphisms. For example, the piecewise language c^*ab under the homomorphisms $[a \mapsto c, b \mapsto b, c \mapsto (ab)]$ is $(ab)^*cb$ that is not piecewise.

Complementation. Piecewise languages are not closed under complementation. For example, consider a piecewise language $L = \Sigma^* aa \Sigma^* + \Sigma^* bb \Sigma^*$, with $\Sigma = \{a, b\}$. The complement of L is the set of sequences where a's and b's alternate — which is not

piecewise. □

We have not found the following properties of repetition piecewise languages in the literature.

Proposition 4 *A language is repetition piecewise if and only if it is recognized by a PO-FSA* $A = ((\Sigma, Q, \delta, q^0, F), \preceq)$*, where* $F = Q$ *and* δ *satisfies the following two conditions. Let* $q_i, q_j, q_l \in Q$ *and* $a, b \in \Sigma$*. Then,*

(I) $(q_i, a, q_j) \in \delta \implies (q_j, a, q_j) \in \delta$*, and*

(II) $(q_i, a, q_j) \in \delta \wedge (q_j, b, q_l) \in \delta \implies (q_i, b, q_l) \in \delta$.

Proof: ($\Rightarrow$) Let L be a simply repetition piecewise language and $L = M_0^* M_1^* \dots M_k^*$. Let $A = ((\Sigma, Q, \delta, q^0, F), \preceq)$ be a PO-FSA with $k+1$ states where $Q = \{q_0, \dots, q_k\}$, $q^0 = q_0$, and $F = Q$. For $i, j \in [0..k]$, δ is defined as follows:

$$(q_i, a, q_j) \in \delta \Leftrightarrow i \leq j \wedge a \in M_j$$

The transition relation δ satisfies the conditions (I) and (II). The partial ordering is defined as follows: $q_i \preceq q_j \Leftrightarrow i \leq j$. We show that A recognizes L, i.e., $\mathcal{L}(A) = L$.

Let w be a word in L. Then, $w = P_0 \cdot P_1 \cdots P_k$, where $P_i \in M_i^*$. We use this partitioning to define an accepting run $\rho = \rho(0) \rightarrow \rho(1) \rightarrow \dots \rightarrow \rho(n)$ of A on w as follows:

$$\rho(i) = q_j \Leftrightarrow \Sigma_{t=0}^{j-1} |P_t| \leq i < \Sigma_{t=0}^{j} |P_t|$$

Intuitively, the automaton goes to state q_i when reading a letter from partition P_i. It is easy to see that the run is well-defined. It is accepting since every state of A is accepting. Thus, $L \subseteq \mathcal{L}(A)$.

To show $\mathcal{L}(A) \subseteq L$, assume $\rho = q_0 \rightarrow \dots \rightarrow q_n$ is an accepting run of A on a word w, where $q_0 = q^0$. Then, ρ induces a partitioning $P_0, \dots, P_k$ on w, such that $P_i \in M_i^*$. Hence, $w \in L$. Thus, $\mathcal{L}(A) \subseteq L$.

A repetition piecewise language is a finite union of simply repetition piecewise languages. Consider a repetition piecewise language $L = L_1 + L_2$, where L_1 and L_2 are two simply repetition piecewise languages that are recognized by PO-FSAs A_1 and A_2, respectively, satisfying conditions (I) and (II). Similarly to the proof of Proposition 2, we construct PO-FSA A that recognizes L. It is easy to show that the construction satisfies conditions (I) and (II).

($\Leftarrow$) Let A be a PO-FSA satisfying conditions (I) and (II). For each state $q_i \in Q$, let $M_{q_i} = \{a \mid (q_i, a, q_i) \in \delta\}$. Let $\rho = q_0 \rightarrow \ldots \rightarrow q_n$ be an acyclic run of A, where every $q_i \in Q$ for $i \in [0..k]$ occurs at most once, and $q_0 = q^0$. The number of such runs is finite. Let the language L_ρ be defined as $M^*_{\rho(0)} \cdots M^*_{\rho(n)}$. It is easy to see that $L_\rho \in \mathcal{L}(A)$. Similarly, let $L' = \bigcup L_\rho$ over all such acyclic runs. Then, $L' \subseteq \mathcal{L}(A)$. Since L' is repetition piecewise, we only need to show that $\mathcal{L}(A) \subseteq L'$. Let w be a word in $\mathcal{L}(A)$, and ρ an accepting run of A on w. Let ρ' be a maximal subsequence of ρ in which every state in Q appears at most once. For example, if ρ is $q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1$, then ρ' is $q_0 \rightarrow q_1$. Then, ρ' is acyclic, and $w \in L_{\rho'}$. Hence, $\mathcal{L}(A) \subseteq L'$. □

The following proposition summarizes the properties of the repetition piecewise languages.

Proposition 5 *Repetition piecewise languages are closed under finite unions and intersections, concatenation, shuffle, and letter-to-letter mappings, but not under homomorphisms or inverse homomorphisms.*

Proof: **Finite unions, intersections, concatenation, shuffle.** Closure under finite unions and concatenation follows immediately from Definition 4. To show closure under finite intersections, let L_1 and L_2 be two repetition piecewise languages. By Proposition 4, they are recognized by PO-FSAs A_1 and A_2, respectively, such that both A_1 and A_2 satisfy conditions (I) and (II) of the proposition. It is easy to check that conditions (I) and (II) are preserved by intersection and shuffle. Thus, the automata $A_1 \cap A_2$ and $A_1 || A_2$ are PO-FSAs satisfying conditions (I) and (II). Hence, by Proposition 4 their languages are repetition piecewise.

Letter-to-letter mapping. The proof is similar to that of piecewise languages (Proposition 3).

Homomorphisms. Repetition piecewise languages are not closed under homomorphisms. For example, repetition piecewise language $(a+b)^*c^*$ under the homomorphisms $[a \mapsto a, b \mapsto b, c \mapsto (ab)]$ is $(a+b)^*(ab)^*$ which is not piecewise.

Inverse homomorphisms. Repetition piecewise languages are not closed under inverse homomorphisms. For example, let $\Sigma = \{0\}$, and $\Sigma' = \{a, b\}$, and h be a homomorphism from Σ to Σ'^* such that $h(0) = ab$. Then, the repetition piecewise language $L = a^*b^*$ under the inverse homomorphism is $h^{-1}(L) = \{\epsilon, 0\}$ which is not repetition piecewise. □

We conclude this section by defining piecewise relations:

Definition 6 (Piecewise Relation) *A relation $\rho \subseteq (\Sigma^*)^K$ is* piecewise *if and only if*

$$\rho \;=\; \bigcup_{0 \leq i < I} \mathcal{L}(R_0^i) \times \cdots \times \mathcal{L}(R_{K-1}^i)$$

for some natural number I and piecewise languages $\mathcal{L}(R_j^i)$ over Σ.

We say that a relation is *repetition piecewise* if and only if R_j^i above are repetition piecewise. Similarly, proposition 1 is extended to piecewise relations, i.e. $\mathcal{L}^{\#}(\rho)$ is piecewise if and only if ρ is a piecewise relation.

2.3 Abstract Interpretation

One of the main limiting factors in feasibility of application of model-checking for the analysis of reactive systems is the size of the model that results in the state explosion problem. One of the most effective techniques to tackle the state explosion problem is *abstraction*. The goal of abstraction is to build an approximation of a model which is smaller than the original one in such a way that if a property holds true for the abstract model, it also holds for the original model.

Such abstraction techniques are formalized in the framework of *Abstract Interpretation* (AI) [45, 46]. Abstract interpretation provides a collection of tools for systematic design and analysis of semantic approximations. The framework is very flexible and can be applied in various ways. In this section, we give a brief overview of the abstract interpretation framework and summarize its main results.

The correctness proof of an AI requires the existence of *standard semantics* that describes the possible behaviors of programs, i.e. a sequence of states. The abstract interpretation focuses on a class of properties of program executions that is defined by *collecting semantics*. The collecting semantics is the semantics of the language formally defining the program execution properties of interest to be analyzed by abstraction. It can be viewed as an instrumented version of standard semantics reduced to essentials in order to ignore irrelevant details about program execution. Collecting semantics can be used as a reference semantics for proving the correctness of all other approximate semantics (for a particular class of properties). The *abstract semantics* is an approximate semantics that considers effectively computable properties of programs and its soundness is proved with respect to the collecting semantics.

Being abstract or concrete is a relative concept and simply means that abstract semantics is an approximation of the concrete semantics. For example, collecting semantics is abstract with respect to standard semantics but is concrete for subsequent abstract interpretation.

Concrete Semantics The *concrete semantics* describes properties of the possible executions of a program and is represented by the concrete semantic properties (or elements) c in a given set C called the *concrete domain*.

A semantics definition associates with each program its *concrete semantics* which is a concrete semantic property c chosen in C representing some characteristics of its possible executions. For example, element c in C can be a function, a set of states, etc.

Abstract Semantics The first basic choice to be made in an abstract interpretation is to design an *abstract domain* A which is an approximate version of the concrete domain C.

We assume that A is just a set without any restriction on its structure.

Inputs to an AI framework are collections of concrete elements, C, and abstract elements, A. The goal of an abstract interpretation is to find an abstract element a, if any, in the abstract domain A that is a *correct* approximation of the concrete semantics $c \in C$ of the program. Thus, the next basic choice in an abstract interpretation is to design a method for associating abstract elements $a \in A$ to programs. Then, we have to specify a correspondence between the concrete and abstract elements. This correspondence is formalized by a *soundness relation*

$$\rho \subseteq C \times A\,.$$

Here, $\langle c, a\rangle \in \rho$ means that the concrete property c of the program has the abstract property a.

It is worth noting that the abstract interpretation problem would have no solution for a program with semantics c when the set $\{a \mid \langle c, a\rangle \in \rho\}$ is empty. Therefore, a common assumption is that every concrete property has an abstract approximation.

A *concretization function* $\gamma : A \rightarrow \mathbf{2}^C$ maps each abstract element a to a set of concrete elements corresponding to it: $\gamma(a) \triangleq \{c \mid \langle c, a\rangle \in \rho\}$. The elements of A can be thought of as properties such as "positive" or "odd" and $\gamma(a)$ as a collection of concrete elements satisfying a. The dual of the γ is an *abstraction function* $\alpha : C \rightarrow A$.

Interval Example An example of an abstract interpretation is the correspondence between sets of integers and intervals. The concrete domain consists of the set of integers. The abstract domain, on the other hand, has intervals of integers. We define $\alpha(X)$ as $[min(X), max(X)]$, and $\gamma([a, b])$ as $\{x \mid x \in \mathbb{Z} \text{ and } a \leq x \leq b\}$ (refer to Figure 2.1). The concrete element $\{1, 3, 5\}$ is therefore mapped to $[1, 5]$ through α, and the abstract element $[1, 5]$ is mapped to the set of concrete elements $\{1, 2, 3, 4, 5\}$ via γ. We can already see that there is loss of precision in the way α has been defined. It is evident that the information that certain number of integers are missing in the set is lost while taking the α. Consider function $f : x := x + 1$ in a program. Let x be in the set $\{1, 3, 5\}$.

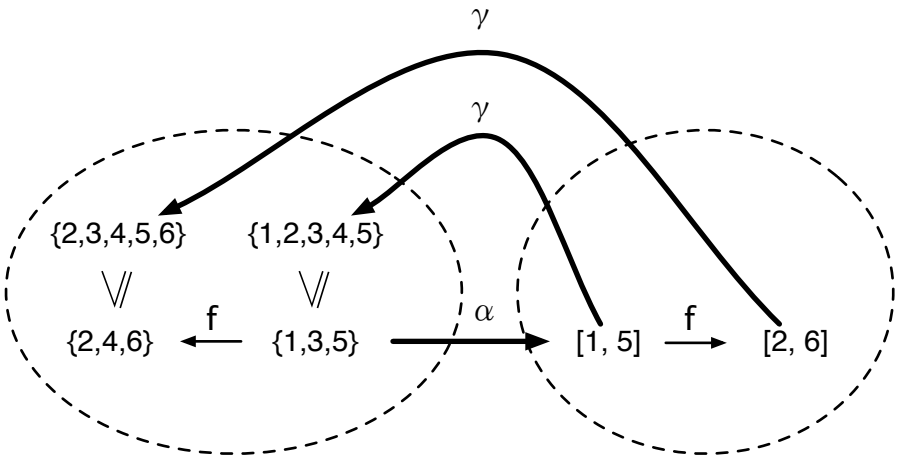

Figure 2.1: Abstract interpretation for interval example.

We want to compute the value of x after execution of the line $f : x := x + 1$. Assume that the implementation of function f in abstract domain (i.e. domain of intervals) is less complex than its implementation in concrete domain (i.e. domain of sets of integers). Thus, in order to compute the value of x after execution of the line $f : x := x + 1$, first, we apply α on $\{1, 3, 5\}$, which results in $[1, 5]$. Next, we perform the operation in the abstract domain on $[1, 5]$, which results in $[2, 6]$. Then, we concretize back to the concrete domain by applying γ on $[2, 6]$, which results in $\{2, 3, 4, 5, 6\}$. This is a superset of the actual value, i.e. $\{2, 4, 6\}$, if we performed the operation in the concrete domain on $\{1, 3, 5\}$. $\square$

As illustrated in the above example, the soundness relation framework is very general and does not take into account the relative precision of abstract properties in the approximation of concrete properties. Thus, we need to introduce a notion of *precision* in order to be able to compare the abstract elements.

2.3.1 Approximation

In general, a program may satisfy many concrete properties c and each one may be approximated by many abstract elements a (according to the soundness relation $\langle c, a\rangle \in \rho$).

In order to distinguish the more precise ones, a notion of *approximation* on the abstract domain can be introduced. Since there exists no metric distance specifying closeness of elements, often their relative precision is indicated using a preorder relation $\sqsubseteq$ which is reflexive and transitive: $a \sqsubseteq a'$ means that a is more precise than a'. The preorder $\sqsubseteq$ can be called the *approximation* or *ordering relation*.

The ordering relation is defined such that less precise abstract properties approximate greater sets of concrete properties, i.e.

$$\forall a, a' \in A, \;\; a \sqsubseteq a' \;\; \Rightarrow \;\; \gamma(a) \subseteq \gamma(a') \, .$$

Intuitively, $a \sqsubseteq a'$ means that a' is less informative or precise than a. When viewed as a property, then a' is weaker than a. For example, knowing that an element is "positive" is less informative than knowing it is both "positive" and "odd".

One potential problem is that α maybe equal to ρ which is the case for example when α satisfies $\langle c, a \rangle \in \alpha \wedge a \sqsubseteq a' \Rightarrow \langle c, a' \rangle \in \alpha$. To avoid this redundancy, we require α to select minimal elements, that is the most precise ones:

$$\alpha \;\; = \;\; \{ \langle c, a \rangle \in \rho \mid \forall a' \in A, \, (\langle c, a' \rangle \in \rho \wedge a' \sqsubseteq a) \Rightarrow (a \sqsubseteq a') \} \, .$$

The intent is that ρ specifies the abstract properties which can be used to approximate a concrete property, α specifies the preferred ones, and $\sqsubseteq$ specifies their relative precision.

However, finding the best abstract property may not be always possible. There may exist many (incomparable) minimal elements for a concrete property. For example, 'positiveness' and 'evenness' of the value of a variable are examples of incomparable properties. This situation represents a lack of expressiveness of A in that the best or more precise property of programs cannot be stated within the set of abstract properties in A.

One of the reasonable assumptions in abstract interpretation is the existence of best

abstract approximation:

$$\forall c \in C,\, \exists a \in A,\, \langle c, a\rangle \in \rho \wedge \forall a' \in A,\, \langle c, a'\rangle \in \rho \;\;\Rightarrow\;\; a \sqsubseteq a'\,.$$

In this case, α is a function such that

$$\forall c \in C,\, \forall a \in A,\, \langle c, a\rangle \in \rho \;\;\Leftrightarrow\;\; \alpha(c) \sqsubseteq a\,.$$

Join *Join* is a binary operation on a partially-ordered set that returns the least upper bound of its arguments, provided the least upper bound exists. Most of the abstract domains are equipped with a join operator, denoted by $\sqcup$. For abstract domain A, $\sqcup_A : A \times A \to A$. According to the soundness relation

$$a \sqcup_A b = c \;\;\Longrightarrow\;\; (\gamma_A(a) \cup \gamma_A(b)) \subseteq \gamma_A(c)\,.$$

2.3.2 Widening

Computing fixpoints of increasing sequences of sets is an important problem in many areas of computer science including algorithmic verification, program analysis, inductive inference, etc. A *fixpoint* of an operator $f : S \to S$ on a poset $\langle S, \sqsubseteq\rangle$ is an element $x \in S$ such that $f(x) = x$. The set of fixpoints of f is the set *Fixpoints*$(f) = \{x \in S \mid f(x) = x\}$. The *least fixpoint* of f, denoted *lfp*(f) is the least element of *Fixpoints*(f) and the greatest fixpoint of f, denoted *gfp*(f) is the greatest element of *Fixpoints*(f).

In Abstract Interpretation, the collecting semantics of a program is expressed as a least fixpoint of a set of equations. The equations are solved over some abstract domain chosen based on desired precision and cost. Typically, the equations are solved iteratively; that is, successive approximations of the solution are computed until they converge to a least fixpoint. However, for many useful abstract domains (particularly those for analyzing numeric properties, such as intervals, octagons [89], and polyhedra [48]) such chains of approximations can be very long or even infinite.

A choice of two methods exists to accelerate a fixpoint computation in the abstract domain. If a finite abstract domain is used, the abstract fixpoint computation converges in finite time to an abstract fixpoint, which is guaranteed to exist. Then, the abstract fixpoint is an approximation of the concrete fixpoint. If the abstract domain is infinite and a fixpoint computation requires infinite steps, Cousot and Cousot [45, 46] define an extrapolation technique called *widening* to accelerate convergence to (a possibly over-approximation of) the abstract fixpoint and if possible enforce termination of the computation.

A widening operator is used to detect and generalize an increment between sets in a fixpoint computation. The extrapolation introduced by a widening step is usually larger than the difference between two sets in a fixpoint computation, so convergence to the fixpoint is accelerated. If the set obtained after a widening step is an over-approximation of the fixpoint, the computation also terminates. Typically, widening degrades the precision of the analysis; i.e., the obtained solution is a fixpoint, but not necessarily the least fixpoint.

Definition 7 (Widening) *A widening operator on an abstract domain A, denoted by $\bigtriangledown_A$, is an operator that satisfies two conditions:*

- *it over-approximates join: $\forall x, y \in A$, $x \sqcup y \sqsubseteq x \bigtriangledown y$,*
- *for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_n \ldots$ in A, the increasing chain $y_0 = x_0, \ldots, y_{n+1} = y_n \bigtriangledown x_{n+1}, \ldots$ is not strictly increasing (i.e. converges after a finite number of terms).*

Consider the example given in Section 2.3.1 concerning intervals. We now expand on that example to show how widening can be used. Consider the program in Figure 2.2 annotated with the intervals that x lies in before and after every statement execution.

The join of two intervals is the interval formed by taking the minimum of the lower bound and the maximum of the upper bound. The intersection of two intervals is the interval formed by taking the maximum of the lower bound and the minimum of the

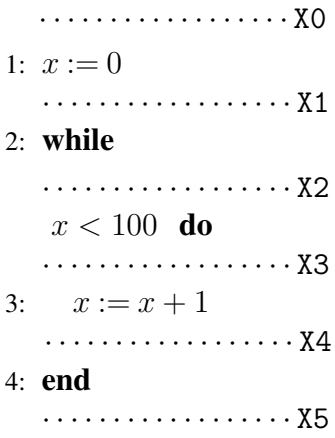

Figure 2.2: An example program annotated with intervals.

upper bound. The program in Figure 2.2 can be converted into one involving basic operations on intervals as shown in Figure 2.3.

According to Figure 2.3, X2 can be calculated recursively:

$$\texttt{X2} \quad = \quad [0,0] \sqcup ((\texttt{X2} \sqcap [-\infty, 99])\,[x := x+1])$$

If we were to compute the value of X2 exactly, we would have to recurse 100 times to finally converge. Instead, we try to use a widening operator for intervals as defined below:

$$[a,b] \bigtriangledown_{\text{INT}} [c,d] \quad = \quad [\text{if } c < a \text{ then } -\infty \text{ else } a,\ \text{if } d > b \text{ then } \infty \text{ else } b]$$

For example $[0,0] \bigtriangledown_{\text{INT}} [0,1] = [0,\infty]$. Let $\perp$ be the $\sqsubseteq_{\text{INT}}$-smallest element in interval domain, then $\perp \bigtriangledown_{\text{INT}} [0,0] = [0,0]$. We now compute the value of X2 using this widening operator. Below, on the left, we show the exact computation of X2 and on the right, the same calculation using interval widening operator. It is clear after computing the interval $[0,\infty]$ we converge.

$\cdots\cdots\cdots\cdots\cdots\cdots \texttt{X0} = [-\infty, +\infty]$

1: $x := 0$

$\cdots\cdots\cdots\cdots\cdots\cdots \texttt{X1} = \texttt{X0}\,[x := 0]$

2: **while**

$\cdots\cdots\cdots\cdots\cdots\cdots \texttt{X2} = \texttt{X1} \sqcup \texttt{X4}$

$x < 100$ **do**

$\cdots\cdots\cdots\cdots\cdots\cdots \texttt{X3} = \texttt{X2} \sqcap [-\infty, 99]$

3: $x := x + 1$

$\cdots\cdots\cdots\cdots\cdots\cdots \texttt{X4} = \texttt{X3}\,[x := x + 1]$

4: **end**

$\cdots\cdots\cdots\cdots\cdots\cdots \texttt{X5} = \texttt{X2} \sqcap [100, +\infty]$

Figure 2.3: An example program annotated with basic operations on intervals.

$$
\begin{array}{rcl}
\texttt{X2}^{(0)} & = & \bot \\
\texttt{X2}^{(1)} & = & [0,0] \sqcup \bot \\
 & = & [0,0] \\
\texttt{X2}^{(2)} & = & [0,0] \sqcup [1,1] \\
 & = & [0,1] \\
\texttt{X2}^{(3)} & = & [0,0] \sqcup [1,2] \\
 & = & [0,2] \\
 & & \cdots
\end{array}
\qquad
\begin{array}{rcl}
\texttt{X2}^{(0)} & = & \bot \\
\texttt{X2}^{(1)} & = & \bot \bigtriangledown_{\text{INT}} ([0,0] \sqcup \bot) \\
 & = & \bot \bigtriangledown_{\text{INT}} [0,0] \\
 & = & [0,0] \\
\texttt{X2}^{(2)} & = & [0,0] \bigtriangledown_{\text{INT}} ([0,0] \sqcup [1,1]) \\
 & = & [0,0] \bigtriangledown_{\text{INT}} [0,1] \\
 & = & [0,\infty] \\
 & & \text{convergence!}
\end{array}
$$

Note that $[0, \infty]$ is an over-approximation of the exact computation result, i.e. $[0, 99]$. Successive applications of a widening operator result in imprecision, thus, a *narrowing* operator is used to improve precision.

For any given program a finite abstract domain that provides the same precision as an infinite abstract domain with a widening operator can be found. However, for a family of programs, there may be no single finite domain that provides the same precision as an infinite domain with a widening operator.

It is worth noting that the widening and narrowing are not dual concepts. A widening operator is used to over-approximate the fixpoint of an increasing sequence. A narrowing

operator is used to over-approximate the limit of a decreasing sequence, thereby, ensuring that an unknown fixpoint is not overshot.

In 1992, Cousot and Cousot [47] observed that in comparison to the other techniques defined in abstract interpretation, "the design of widenings and narrowings is often thought to be more difficult since it appears as a heuristic to cope with induction". Over a decade later, in 2006, Halbwachs [67] remarked that "widening is 'still' often considered as a kind of dirty heuristic in the model-checking community."

2.4 Summary

In this chapter, we presented the basic concepts from finite automata, regular languages, and Abstract Interpretation. We introduced the notion of piecewise languages and their characteristics.

We will use the main concepts on automata and regular languages in Chapter 3 and 4 for the analysis of FIFO systems. In these chapters, we will restrict our attention to the analysis of *piecewise* FIFO systems. Later we will use the main concepts of Abstract Interpretation in Chapter 5 where we will develop abstract domains for the analysis of parameterized systems.

Chapter 3

Piecewise FIFO Systems

In this chapter, we describe FIFO systems and the reachability problem for them. We show that piecewise languages play an important role in the analysis of FIFO systems. In particular, we focus on computing the set of reachable states of a piecewise FIFO system. The main insight in this chapter is that the problem of computing the set of reachable states of a piecewise FIFO system is closely related to calculating the set of all possible channel contents, i.e. the *limit languages*. Our key contribution is that the limit language of a piecewise FIFO system is piecewise if the initial channel language is piecewise.

3.1 Introduction

Concurrent systems consisting of a set of finite-state machines that communicate via unbounded First-In First-Out (FIFO) channels are a common model of computation for describing distributed protocols such as IP-telecommunication protocols, interacting web services, and System-on-Chip (SoC) architectures (e.g., [24, 14, 1, 93, 33, 16, 105, 60, 29]). Even though all physically constructible systems have finite size channels, their size is often an implementation parameter that is typically left unspecified. Modeling such systems with unbounded channels often makes reasoning about them simpler. The abstraction may of course fail to reveal certain deadlock situations that occur if the chan-

nels fill up, but the abstract system behaves otherwise essentially the same as the system with finite size channels.

Unboundedness of communication channels provides a useful modeling abstraction, but it does in a theoretical sense complicate analysis if compared to a system of a given fixed size, say with channels of length 1024. In fact, Brand and Zafiropulo [24] showed that a single unbounded channel is already sufficient to simulate the tape of a Turing machine. Hence, verification of any non-trivial property, such as reachability, is undecidable. Despite these results, a substantial effort has gone into identifying subclasses of FIFO systems for which the verification problem is decidable (e.g., [1, 4, 12, 13, 14, 19, 21, 33, 93]).

In this thesis, we study the class of *piecewise* FIFO systems. These systems can be used for modeling distributed protocols such as IP-telecommunication protocols and interacting web services. A piecewise FIFO system is composed of components whose behaviors can be expressed by piecewise languages (refer to Definition 4). As explained in Section 2.2, a language is piecewise if it is accepted by a non-deterministic finite-state automaton whose only non-trivial strongly connected components are states with self-loops. Formally, a piecewise language is a union of sets of strings, where each set is given by a regular expression of the form $M_0^* a_0 M_1^* \cdots a_{n-1} M_n^*$, where each M_i is a subset of the alphabet Σ and each a_i is an element of Σ.

The ability to calculate all possible channel contents that may arise from an initial state, i.e. the *limit language*, plays a central role for automated verification of non-trivial properties of FIFO systems. This problem is undecidable in general. Moreover, the limit language is not necessarily regular, even if the initial language is [33], and even when the limit language is known to be regular, determining it may still be undecidable [33]. We focus on computing the limit languages in piecewise FIFO systems. In this chapter, our main contribution is that the limit languages of piecewise FIFO systems remain regular even if simultaneous read and write actions (i.e. *conditional actions*) are added to a set of transitions iterated on piecewise FIFO systems. In particular, we show, for multi-channel piecewise FIFO systems, the limit language is piecewise if the initial channel language

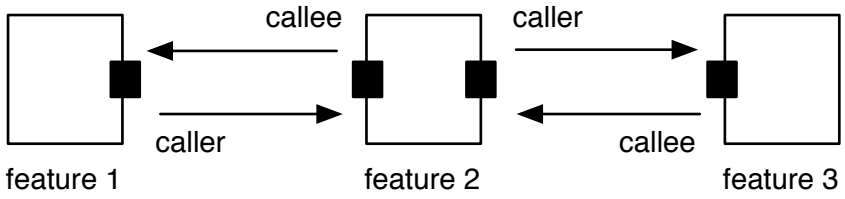

Figure 3.1: BoxOS call structure.

is piecewise.

The rest of this chapter is organized as follows. Section 3.2 explains the motivation behind piecewise FIFO systems. Section 3.3 provides the definition of piecewise FIFO systems and the reachability problem for them. It describes how the reachability problem in such systems can be reduced to computing the limit languages. Section 3.4 describes the conditions under which the limit language of piecewise FIFO systems is piecewise. We review the related work in Section 3.5 and conclude in Section 3.6 with a summary of our contributions in this chapter.

3.2 Motivating Example

Although piecewise languages may look restrictive, they can be used to express descriptions of IP-telephony features [66] and seem amenable to describing composite web services specified in Business Process Execution Language (BPEL) [72]. For example, [66] studied the behavior of the telephony features in BoxOS which is the next generation telecommunication service over IP developed at AT&T Research [16, 74]. As shown in Figure 3.1, an active call is represented by a graph of telephony features (referred to as *boxes*) while communication between neighboring boxes is handled via unbounded perfect FIFO channels. Boxes at the end points represent telephones, intermediate boxes represent call features, for example call-forwarding-on-busy. At a sufficient level of abstraction, boxes may all be viewed as finite-state transducers. Communication in these protocols begins with an initiator trying to reach a given destination. A call is built recur-

sively. The current endpoint, the *caller*, begins the call initiation protocol with a chosen neighbor, the *callee*. If this initiation results in a stable connection, the *callee* becomes the new endpoint and the call construction continues. Call termination is required to proceed in a reverse order and, in general, is required to begin at a call endpoint.

In order to manage inter-feature communication, it is desirable that communication between features follows a certain pattern [16]. Thus, all of the feature boxes implement a communication template that consists of three phases (cf. [16]): *setup* phase, *transparent* phase, and *teardown* phase. Figure 3.2 describes a *transparent box* that represents such a communication template. The transparent box communicates with two neighbors across four separate channels. Messages to/from the upstream (initiating), *caller*, are sent/received via *ro/ri* channels. Messages to/from the downstream (receiving), *callee*, are sent/received via *eo/ei* channels. A message is received with the '?' symbol and sent with the '!' symbol. For example *ri?setup* indicates a *call setup message* received from the *ri* channel. Interestingly, this communication template can be expressed by piecewise languages. To achieve piecewiseness, we have abstracted the transparent box by replacing the original LINKED state and its left and right neighbors, shown in shaded rectangle on the top right corner of Figure 3.2, by the LINKED state, shown in the shaded rectangle in the middle of the figure. Both of these states have the same functionality. The difference is the addition of *conditional actions* of the form $ri?status \rightarrow eo!status$, where the *status* message is sent to the callee only if the *status* message has been received from the caller first.

It is crucial to be able to reason about safety and deadlock properties of BoxOS implementations having multiple features communicating over unbounded perfect FIFO channels, somethings that the techniques in [16] fell short to address.

3.3 FIFO Systems and the Reachability Problem

In this section, we review the definition of FIFO systems and the reachability problem for them.

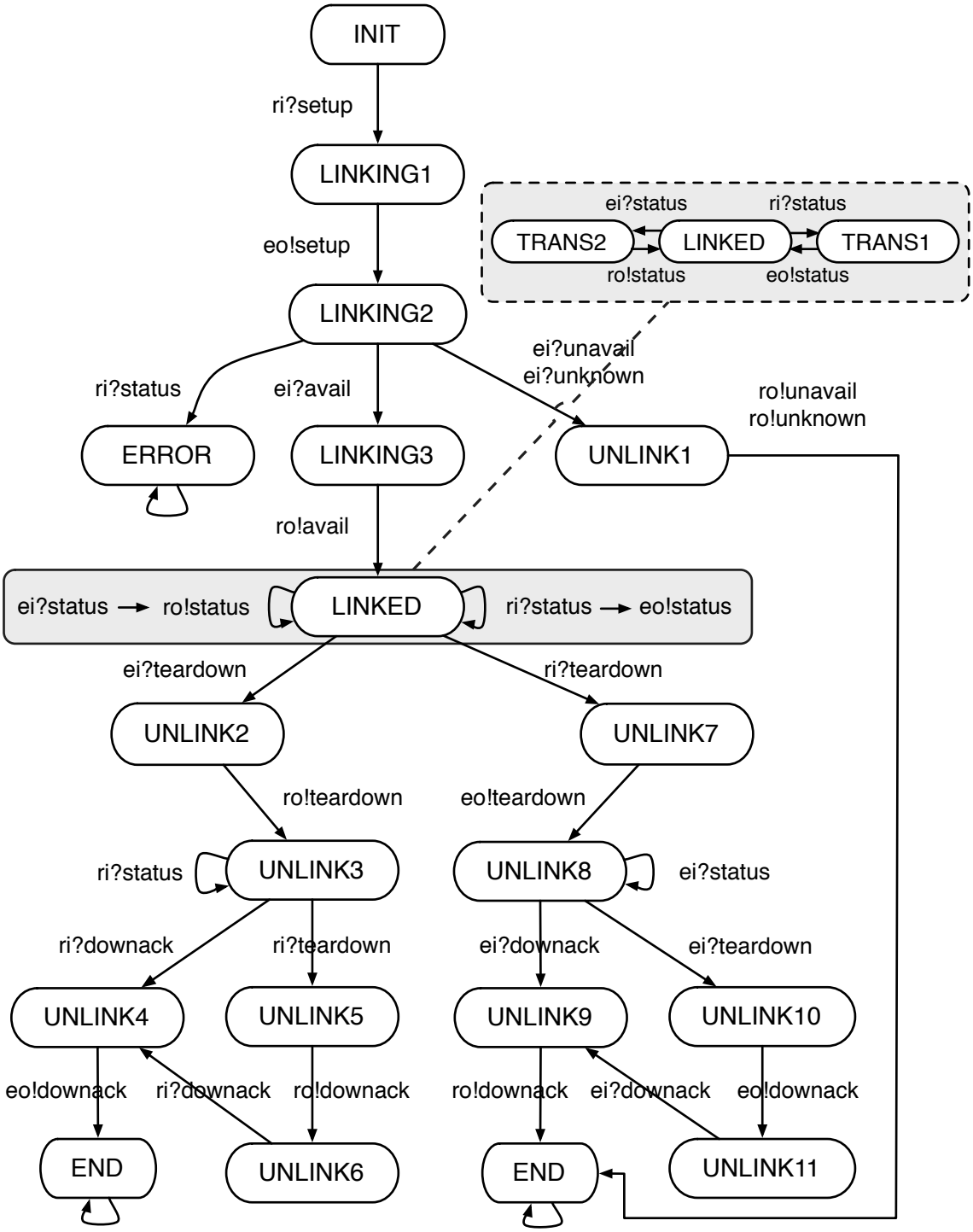

Figure 3.2: Transparent feature box.

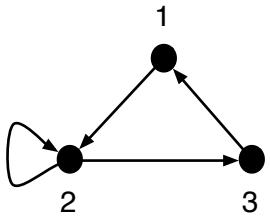

Figure 3.3: An example of a communication graph for a set of actions $Act = \{1?a \rightarrow 2!a, 2?b \rightarrow 3!b, 3?b \rightarrow 1!a, 2?b \rightarrow 2!b\}$.

3.3.1 Action Languages and Semantics

A channel over an alphabet Σ is a FIFO queue whose contents is given by a word $w \in \Sigma^*$. We define two types of channel actions: read a, denoted by $?a$, and write a, denoted by $!a$, that stand for reading and writing a letter a from/to a channel, respectively. We use $f : w$ to denote the application of an action f to a word w. For example, $?a : abb = bb$ and $!a : bb = bba$.

Let $\Sigma_{rw} \triangleq \{?, !\} \times \Sigma$ denote read/write(rw)-alphabet over Σ. For a set of channels $C = \{c_1, \ldots, c_k\}$ this alphabet is extended as follows: $\Sigma_{rw}(C) \triangleq [1..k] \times \Sigma_{rw}$. Thus, an action $4?a$ corresponds to reading a from channel c_4, and $6!b$ corresponds to writing b to channel c_6. In the sequel, we drop C from the notation when it is clear from the context. We call Σ_{rw} an *action alphabet*, and any subset of Σ_{rw}^* an *action language*.

A *channel configuration* for a system with k channels is a k-tuple $\mathbf{w} \in (\Sigma^*)^k$. We use $\langle w_1, \ldots, w_k \rangle$ to denote a tuple, where w_i is the content of channel i. In single-channel systems, a configuration is just the content of the single channel. We use bold fonts to differentiate between channel configurations in multi-channel and single-channel systems. Let $\mathbf{w}[i]$ denote the content of channel i in $\mathbf{w}$ and $\mathbf{w}[i \mapsto y]$ denote a channel configuration obtained from $\mathbf{w}$ by replacing the content of channel i with y.

In the single-channel case, for $X \subseteq \Sigma_{rw}^*$ and $W \subseteq \Sigma^*$, we use $X : W$ to denote the result of applying all sequences of actions in X to the words in W. This is called the concrete semantics of actions and is defined as follows:

Definition 8 (Action Language Semantics) *Let $W \subseteq \Sigma^*$ be a set of words over Σ, and X an action language, then $X : W$ is defined as follows:*

$$?a : W \triangleq (a^{-1})W \qquad !a : W \triangleq W \cdot a$$

$$\{x \cdot y\} : W \triangleq y : (x : W) \qquad X : W \triangleq \bigcup_{x \in X} (x : W)$$

For example, $(\{?a!b\ ,\ ?a!c\} : a) = \{b\ ,\ c\}$.

Definition 8 is extended to a k-channel system as follows. Given $\mathbf{w} \in (\Sigma^*)^k$ and an action language X, then $X : \mathbf{w}$ for a single action is defined as shown below:

$$i?a : \mathbf{w} \triangleq \mathbf{w}[i \mapsto (?a : \mathbf{w}[i])] \qquad i!a : \mathbf{w} \triangleq \mathbf{w}[i \mapsto (!a : \mathbf{w}[i])]$$

and is extended to words identical to Definition 8. For example, given a 2-channel system, $(\{1?a\ 2!b, 1?a\ 2!c\} : \langle ab, b\rangle) = \{\langle b, bb\rangle, \langle b, bc\rangle\}$.

We write $?a \rightarrow !b$ for a *conditional action* that means "b is written only if a is read first". In other words, $?a \rightarrow !b$ is an abbreviation for a sequence of simple actions: $?a!b$. Given an action alphabet $\Sigma_{rw}(C)$ over a set of channels C, we define a conditional action alphabet $\Sigma_{rwc}(C)$ that treats conditional actions as letters:

$$\Sigma_{rwc}(C) \triangleq \Sigma_{rw}(C) \cup ((C \times \{?\} \times \Sigma) \cdot (C \times \{!\} \times \Sigma))\,.$$

For example, given $\Sigma = \{a\}$ and $C = \{1\}$, then $\Sigma_{rwc}(C) = \{1?a, 1!a, 1?a \rightarrow 1!a\}$.

For a set of actions $Act \subseteq \Sigma_{rwc}(C)$, a *communication graph* of Act, $CG(Act)$, is a digraph (C, E), with an edge $(i, j) \in E$ if and only if there are a and b in Σ such that $i?a \rightarrow j!b$ is in Act. For example, given $Act = \{1?a \rightarrow 2!a, 2?b \rightarrow 3!b, 3?b \rightarrow 1!a, 2?b \rightarrow 2!b\}$, $CG(Act)$ is a digraph with 3 nodes and 4 edges one for each conditional action in Act (see Figure 3.3).

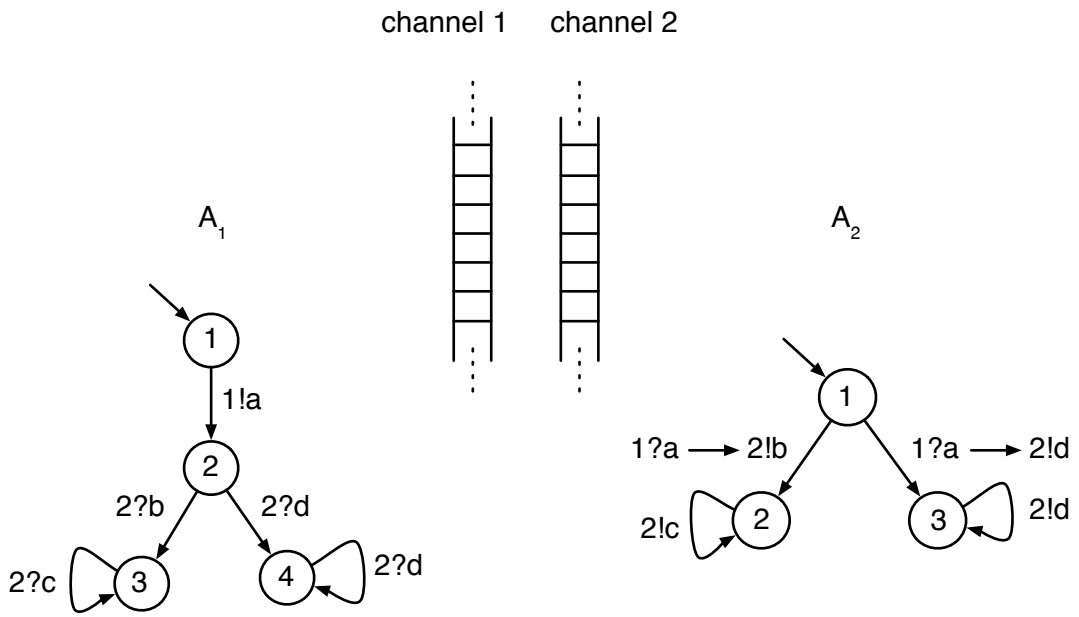

Figure 3.4: An example of a FIFO system consisting of two processes and two channels.

3.3.2 FIFO Systems

A FIFO system is formally defined as follows:

Definition 9 (FIFO System) *A FIFO system is a tuple $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$, where Σ is a finite alphabet; $C = \{c_1, \ldots, c_k\}$ is a finite set of channels; Q is a finite set of control locations; $q^0 \in Q$ is the initial control location; and $\delta \subseteq Q \times \Sigma_{rwc} \times Q$ is a set of transition rules.*

Note that in Definition 9, a FIFO system is defined with respect to a conditional action alphabet Σ_{rwc}. A *global state* of $\mathcal{S}$ is a pair $(q, \mathbf{w})$ where q is a state in Q and $\mathbf{w}$ is a channel configuration. The transition relation of $\mathcal{S}$, Δ, is a set of triples of the form $((q, \mathbf{w}), op, (q', \mathbf{w}'))$, where $op \in \Sigma_{rwc}$, $(q, op, q') \in \delta$, and $\mathbf{w}' \in (op : \mathbf{w})$.

A FIFO system $\mathcal{S}$ is *piecewise* if there exists a partial order $\preceq$ on Q such that $q' \in \delta(q, op)$ implies that $q \preceq q'$.

Most often a FIFO system is represented as a set $\{A_i\}_i^n$ of n processes communicating through a set of channels, C. Each process is a finite-state automaton, $A_i =$

$(\Sigma, Q_i, \delta_i, q_i^0)$. The corresponding FIFO system, $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$, is constructed by computing the cross product of these automata. Thus, $Q \triangleq \Pi_{i=1}^{n} Q_i$ and the transition relation Δ of $\mathcal{S}$ is built up from the transition relations of the A_i's such that every transition in Δ corresponds to exactly one transition in some δ_i. Formally,

$$(((q_1, \ldots, q_n), \mathbf{w}), op, ((q'_1, \ldots, q'_n), \mathbf{w}')) \in \Delta \text{ if and only if}$$
$$\exists i,\ \forall j \neq i,\ q_j = q'_j \ \wedge \ q'_i \in \delta_i(q_i, op) \ \wedge \ \mathbf{w}' \in (op : \mathbf{w})\,.$$

Figure 3.4 shows an example of a piecewise FIFO system consisting of two processes, A_1 and A_2, and two channels. Initially, we assume that both of the processes are in their initial states and both of the channels are empty. Thus, the initial global state of the system is $((1,1), \langle \epsilon, \epsilon \rangle)$. Then, process A_1 writes a on channel 1 and moves from state 1 to 2. The new global state of the system is $((2,1), \langle a, \epsilon \rangle)$. Therefore,

$$(((1,1), \langle \epsilon, \epsilon \rangle), 1!a, ((2,1), \langle a, \epsilon \rangle)) \ \in \ \Delta\,.$$

Then, in a possible execution path, process A_2 reads a from channel 1 and writes b on channel 2 and moves to state 2. The new global state of the system is $((2,2), \langle \epsilon, b \rangle)$. While on state 2 process A_2 can write zero or more c's on channel 2, or, process A_1 can read b from channel 2 and move to state 3. Therefore, a possible next global state of the system could be $((3,2), \langle \epsilon, \epsilon \rangle)$.

3.3.3 Reachability Problem

We are interested in the reachability problem in FIFO systems:

FIFO Systems Reachability Problem. Given a FIFO system $\mathcal{S}$ and a set of channel configurations **I** (called initial), find the set of all global states reachable from **I**.

The set of all reachable global states of a FIFO system can be partitioned, based on the control locations. Each partition represents the set of all reachable channel configurations

at a particular control location. Thus, in order to calculate the set of all reachable global states, we need to calculate the set of all reachable channel configurations at each control location. This problem can be reduced to computing the semantics (Definition 8) of a regular action language.

Proposition 6 *Let $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$ be a FIFO system, $q \in Q$ some control location, and* $\mathbf{I}$ *a set of configurations. Then, the set of all reachable configurations of $\mathcal{S}$ at control location q is $(\mathcal{L}(A_q) : \mathbf{I})$, where $A_q = (\Sigma_{rwc}, Q, q^0, \delta, \{q\})$ is a finite automaton with accepting state q.*

Figure 3.5 is an example illustrating this reduction. On the left, we show an example of a FIFO system and on the right the set of all reachable configurations at control locations 1, 2, 3, and 4. For example, in order to compute the set of all reachable configurations at control location 2, we construct automaton A_2 in which control location 2 is the only accepting state. The language of this automaton is described by a regular expression $\mathcal{L}(A_2) = (?d)^*?a(?c!a)^*$. If $\mathbf{I}$ denotes the set of initial channel configurations, we can compute the set of all reachable channel configuration at control location 2 by computing $((?d)^*?a(?c!a)^*) : \mathbf{I}$.

Finally, computing the semantics of a regular action language is itself reducible to the *limit language problem*: given a regular language of actions L_a and a regular language of channel content $\mathbf{W}$, compute the language of $(L_a^* : \mathbf{W})$. In our running example, in order to compute the set of reachable configurations at control location 2, we need to be able to compute the result of repeated application of $?c$ followed by $!a$ on some channel configurations $\mathbf{W}$, i.e we need to compute $(?c!a)^* : \mathbf{W}$.

In the particular case of piecewise FIFO systems, since the only non-trivial, strongly-connected components are states with self-loops, L_a is further restricted to subsets of Σ_{rwc}. This is the problem we study in the rest of this chapter and the following chapter.

Proposition 7 *For regular (piecewise) language L, it holds that $(?a : L)$, $(!a : L)$, and $(?a \rightarrow !b : L)$ are regular (piecewise).*

Proof: For a single write action, clearly $(!a : L) = L \cdot a$. For a read action, we have

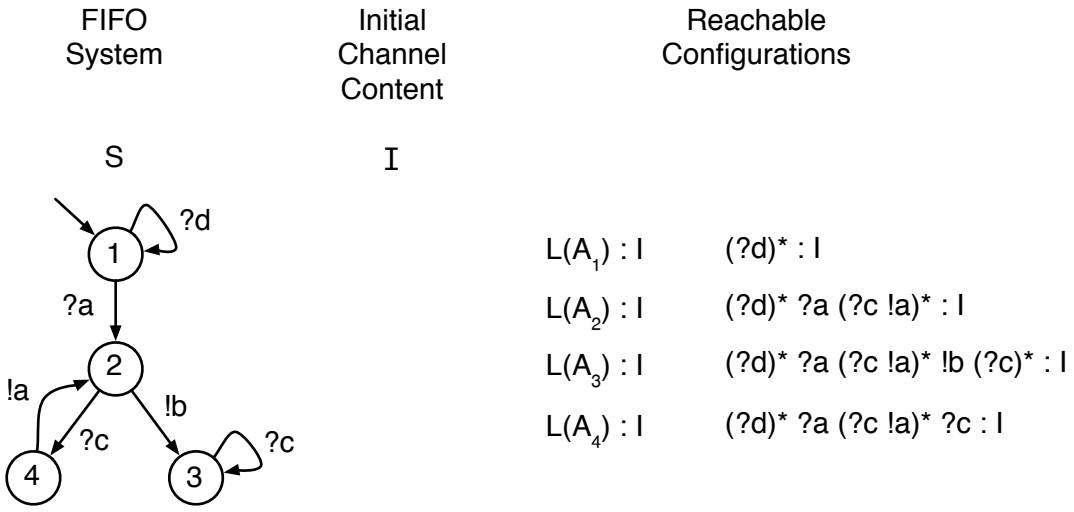

Figure 3.5: An example illustrating the calculation of all reachable global states by computing the semantics of a regular action language.

$(?a : L) = a^{-1}L$ following from the definition of derivative. For the conditional action, we have $(?a \rightarrow !b : L) = (a^{-1}L) \cdot b$. □

3.4 Decidability Results

In this section, we focus on the limit language problem for a set of actions, Act, on a k-channel system, and a set of channel configurations $\mathbf{L}$. A configuration $\langle w_1, \ldots, w_k \rangle$ of a k-channel system is represented by a word of the form $w_1 \cdot \# \cdots \# \cdot w_k$, where $\#$ is a fresh letter not in Σ. Thus, a channel configuration can be seen as an element of a relation. In the sequel, a set of channel configurations correspond to a relation over Σ. A *regular configuration* is a set of channel configurations that correspond to a regular relation and a *piecewise configuration* is a set of channel configurations that correspond to a piecewise relation (refer to Definition 6).

We show that the limit languages of multi-channel systems are not regular in general. However, regularity can be achieved by either restricting Act to exclude conditional actions, or restricting $\mathbf{L}$ to piecewise configurations.

Proposition 8 *There exists a set of actions $Act \subseteq \Sigma_{rwc}$ and a regular configuration* $\mathbf{L}$*, such that the limit language $(Act^* : \mathbf{L})$ is not regular.*

Proof: Consider a 2-channel action set $Act = \{1?a \rightarrow 2!a, 1?b \rightarrow 1!b', 1?b' \rightarrow 2!b\}$ and a regular configuration $\mathbf{L} = \langle (ab)^*, \epsilon \rangle$. The idea is this: first, all the a's are transferred to channel 2, then, all the b's (after each b has temporarily been renamed to b'). In configuration $\mathbf{L}$, channel 1 contains exactly the same number of a's and b's. After applying Act^*, there should be equal number of a and b in channel 2, and all b's should follow a's. Thus, we have

$$(Act^* : \mathbf{L}) \cap \langle \epsilon, \Sigma^* \rangle = \{ \langle \epsilon, a^n b^n \rangle \mid n \geq 0 \} .$$

Hence, $(Act^* : \mathbf{L})$ is non-regular. □

In the rest of this section, we show how to achieve regularity in the limit languages of piecewise FIFO systems. First, we show that restricting the action set to unconditional actions is sufficient to make the limit language regular. However, excluding the conditional actions significantly restricts the expressiveness of the system model. We then show that to achieve regularity in the presence of conditional actions, we need to restrict the initial channel configuration to only piecewise configurations.

3.4.1 Limit Languages by Restricting the Action Set

In this section, we present results on recognizability of limit languages in piecewise multi-channel systems where conditional actions are excluded from the action set.

Proposition 9 *Let $Act \subseteq \Sigma_{rw}$ be a set of unconditional actions in a multi-channel system and* $\mathbf{L}$ *a regular (piecewise) configuration. Then,*

(a) $(Act^* : \mathbf{L})$ is regular (piecewise).

(b) For a piecewise expression T over Act, $(T : \mathbf{L})$ is regular (piecewise).

Proof: Without loss of generality, we consider a 2-channel system. Generalization to systems with more channels is trivial.

(a) Since $\mathbf{L}$ is regular (piecewise), it is equivalent to $\sum_{0 \leq i < I} R_1^i \# R_2^i$ for some I, where each R_j^i is regular (piecewise), and $\#$ is a fresh letter not present in Σ (Proposition 1). Let $M_k = \{a \mid k?a \in Act\}$ and $N_k = \{a \mid k!a \in Act\}$ for $k = 1$ and $k = 2$ represent the set of all letters that are read and written, respectively. Then,

$$(Act^* : \mathbf{L}) = \sum_{0 \leq i < I} ((M_1^*)^{-1} R_1^i) \cdot N_1^* \# ((M_2^*)^{-1} R_2^i) \cdot N_2^* \,.$$

Hence, $(Act^* : \mathbf{L})$ is regular (piecewise) if $\mathbf{L}$ is regular (piecewise).

(b) By definition, T is a sum of simply piecewise expressions of the form

$$Act_0^* \; act_0 \cdots act_\ell \; Act_{\ell+1}^* \,,$$

on which (a) can be applied inductively.

□

3.4.2 Limit Languages by Restricting the Initial Channel Language

In this section, we present results on recognizability of limit languages in multi-channel systems with initial piecewise channel configurations. First, we review some necessary definitions such as well-quasi-ordering and upward and downward closed sets. Then, we establish our main result by showing that an *arbitrary* union of repetition piecewise relations is, in fact, piecewise.

Preliminaries

In the rest of this section, we assume a finite alphabet Σ and use x and y to denote elements of Σ^*.

Definition 10 (WQO) *[78] A binary relation $\preceq$ on a set X is a* well-quasi-ordering (wqo) *on X if $\preceq$ is reflexive, transitive, and any infinite sequence of elements $x_0 x_1 x_2 \cdots$*

from X contains an increasing pair $x_i \preceq x_j$ with $i < j$. The set X is said to be well-quasi-ordered *by $\preceq$.*

Note that if X is well-quasi-ordered then it does not contain an infinite descending chain, nor an infinite set of pairwise incomparable elements.

Definition 11 (Subword Ordering) *The subword relation $\leq \subseteq \Sigma^* \times \Sigma^*$ is defined such that $x \leq y$ if and only if x can be obtained by deleting some letters of y.*

For example, $ac \leq abc$, but $abc \not\leq abd$. The relation $\leq$ is reflexive and transitive. Furthermore, it follows from Higman's Lemma [69] that the subword ordering relation is a wqo. Intuitively, this means that any *infinite* subset of Σ^* contains at least two words x and y such that x is a subword of y.

Definition 12 (Upward/Downward Closure) *For a set of words A, the* upward closure of A, *denoted $A^{\leq}$, is the set of all words y such that $x \leq y$ for some $x \in A$. Dually, the* downward closure *of A, denoted $A^{\geq}$, is the set of all words y such that $y \leq x$ for some $x \in A$.*

For example, an upward closure of the singleton set $\{abc\}$ is:

$$\begin{aligned}\{abc\}^{\leq} &= \{y \mid abc \text{ is a subword of } y\} \\ &= \Sigma^* a \Sigma^* b \Sigma^* c \Sigma^*\end{aligned}$$

The downward closure of the same set is:

$$\begin{aligned}\{abc\}^{\geq} &= \{y \mid y \text{ is a subword of } abc\} \\ &= \{abc, ab, ac, bc, a, b, c, \epsilon\}\end{aligned}$$

We say that a set A is upward closed or an upset if $A^{\leq} = A$, and that it is downward closed or a downset if $A^{\geq} = A$. For example, $\Sigma^* a \Sigma^* b \Sigma^* c \Sigma^*$ is an upset, $a^* b^* c^*$ is a downset, and $\{abc\}$ is neither an upset nor a downset.

An upward closed set is uniquely determined by its minimal elements. For an upward closed set A, let $\text{MIN}(A) \triangleq \{x \in A \mid \nexists y \in A,\, y \leq x\}$, then $A = (\text{MIN}(A))^{\leq}$. Note that all elements of $\text{MIN}(A)$ are pairwise incomparable and since $\leq$ is a wqo, $\text{MIN}(A)$ is finite.

The subword ordering is extended pointwise to tuples of words. Let $\mathbf{w}, \mathbf{u} \in (\Sigma^*)^K$ be such that $\mathbf{w} = (w_1, \ldots, w_K)$ and $\mathbf{u} = (u_1, \ldots, u_K)$. Then, $\mathbf{w} \leq \mathbf{u}$ if and only if $\forall i,\, w_i \leq u_i$. This ordering is still a wqo, since it is a cross product of well-quasi-orderings.

The notions of upward and downward closures extend to sets of tuples in a natural way. In particular, an upward closure of a set containing a single tuple is:

$$\{(w_1, \ldots, w_n)\}^{\leq} = \{w_1\}^{\leq} \times \{w_2\}^{\leq} \times \cdots \times \{w_n\}^{\leq}$$

Repetition Piecewise Relations

Repetition piecewise languages are downward closed with respect to the subword ordering. For example, let $\Sigma = \{a, b, c\}$, and $L = a^*b^*$. The downward closure of L is the set of all words that have an arbitrary number of a's followed by an arbitrary number of b's, which is L itself.

For any downset $L \subseteq \Sigma^*$, its complement, $\complement L \subseteq \Sigma^*$, is upward closed. Since subword ordering $\leq$ is a wqo, there exists a finite set $A = \text{MIN}(\complement L)$ such that $\complement L = A^{\leq}$. For example, for the language L above, $\complement L = \{c, ba\}^{\leq}$.

The same is true of repetition piecewise relations: they are downward closed, and allow for a finite representation of their complements.

Lemma 1 *Let $R \subseteq (\Sigma^*)^n$ be a relation that is upward closed with respect to the subword ordering. Then, there exists a finite set $\{r_1, \ldots, r_k\} \subseteq (\Sigma^*)^n$ such that $R = \{r_1, \ldots, r_k\}^{\leq}$.*

Proof: Since R is upward closed, $R = \text{MIN}(R)^{\leq}$, and all elements of $\text{MIN}(R)$ are pairwise incomparable. Since $\leq$ is a wqo, $\text{MIN}(R)$ is finite. □

The following lemma shows that the language of the complement of the upward closure of a tuple of strings is piecewise.

Lemma 2 *Let* $\mathbf{w} \in (\Sigma^*)^n$ *be a tuple of strings, and* $\{\mathbf{w}\}^{\leq}$ *be its upward closure. Then,* $\complement\{\mathbf{w}\}^{\leq}$ *is piecewise.*

Proof: Assume that $\mathbf{w}$ is of the form $\mathbf{w} = (w_1, \ldots, w_n)$. Then $\{\mathbf{w}\}^{\leq} = \{w_1\}^{\leq} \times \cdots \times \{w_n\}^{\leq}$. The complement of this set can be expressed as:

$$\begin{aligned} \complement\{\mathbf{w}\}^{\leq} \;=\; & (\complement\{w_1\}^{\leq} \times \Sigma^* \times \cdots \times \Sigma^*) \quad \cup \\ & (\Sigma^* \times \cdots \times \complement\{w_i\}^{\leq} \times \cdots \times \Sigma^*) \quad \cup \\ & \cdots \quad \cup \\ & (\Sigma^* \times \cdots \times \Sigma^* \times \complement\{w_n\}^{\leq})\,. \end{aligned}$$

Here, Σ^* is trivially piecewise. Thus, it is sufficient to show that $\complement\{w_i\}^{\leq}$ is piecewise for any word w_i.

Let $w_i = w_i(1)w_i(2)\cdots w_i(k)$, where $w_i(j)$ is the j-th letter in w_i. The upward closure of w_i is the set of all words y that contain w_i as a subword. That is

$$\{w_i\}^{\leq} \;=\; \mathcal{L}(\Sigma^* w_i(1)\Sigma^* w_i(2)\cdots\Sigma^* w_i(k)\Sigma^*)\,.$$

Thus, $\complement\{w_i\}^{\leq}$ is the union of a set of languages as follows:

$$\begin{aligned} & (\Sigma - \{w_i(1)\})^* \quad \cup \\ & (\Sigma - \{w_i(1)\})^* w_i(1)(\Sigma - \{w_i(2)\})^* \quad \cup \\ & \cdots \quad \cup \\ & (\Sigma - \{w_i(1)\})^* w_i(1)(\Sigma - \{w_i(2)\})^* \cdots w_i(k-1)(\Sigma - \{w_i(k)\})^*\,. \end{aligned}$$

$\complement\{w_i\}^{\leq}$ is a finite union of a set of simply piecewise expressions, and therefore it is piecewise.

□

For example, consider a (trivial) tuple $w = ab$ and $\Sigma = \{a, b, c\}$. The upward closure

of w is $\{ab\}^{\leq} = \Sigma^* a \Sigma^* b \Sigma^*$. The complement $\complement\{ab\}^{\leq}$ is represented by a piecewise expression $(b+c)^* + (b+c)^* a (a+c)^*$.

The following Lemma extends Lemma 1 to all downward closed relations.

Lemma 3 *A relation that is downward closed with respect to the subword ordering is piecewise.*

Proof: Let R be a downward closed relation. Its complement $\complement R$ is upward closed. By Lemma 1, $\complement R = \{r_1, \ldots, r_k\}^{\leq} = \bigcup_{i=1}^{k} \{r_i\}^{\leq}$, and $R = \bigcap_{i=1}^{k} \complement\{r_i\}^{\leq}$. By Lemma 2, R is a finite intersection of piecewise relations and is therefore a piecewise relation. □

For example, (a^*b^*, a^*c^*) is downward closed and piecewise. The following proposition establishes the piecewiseness of an *arbitrary* union of repetition piecewise relations.

Proposition 10 *An arbitrary union of a family of repetition piecewise relations is a piecewise relation.*

Proof: Repetition piecewise relations are downward closed and an arbitrary union of downward closed sets is downward closed. By Lemma 3, this union is a piecewise relation. □

Main Results

Our main result follows directly from Proposition 10. However, first we need to introduce the notion of an *anchor sequence*.

Definition 13 (Anchor Sequence) *The set of* anchor sequences *of a piecewise expression*

$$R \;=\; \textstyle\sum_{0 \leq i < I} M_1^{i*} a_1^i \cdots M_k^{i*} a_{k(i)}^i M_{k(i+1)}^{i*}$$

is the set $\{a_1^i \cdots a_{k(i)}^i | 0 \leq i < I\}$. *The* anchor length *of* R *is* $\max_{0 \leq i < I} k(i)$ *and the anchor length of piecewise* L *is the minimum anchor length of an* R *such that* $\mathcal{L}(R) = L$.

For example, given piecewise $R = (a+b)^*cd^*ba^* + d^*ba^*$, the anchor sequences of R is $\{cb, b\}$, and the anchor length of R is 2.

The following proposition shows that an arbitrary union of piecewise languages with bounded anchor length is piecewise.

Proposition 11 *The union of any (possibly infinite) family of piecewise languages of bounded anchor length is piecewise.*

Proof: Note that for any bound, there are finitely many different anchor sequences. By a rearrangement of the simply piecewise expressions of the languages of the family, it suffices to consider a finite union of languages L_ℓ, where L_ℓ is a union of simply piecewise languages all having the same anchor sequence. Since piecewise languages are closed under finite union, it is sufficient to show that L_ℓ is piecewise. Consider $L_\ell = \bigcup_{i\geq 0} \mathcal{L}(R^i)$, where R^i for example has the following form:

$$R^i = M_1^{i*} a_1^i \cdots {M_{k-1}^i}^* M_k^{i*} a_{k(\ell)}^i M_{k(\ell+1)}^{i*}$$

By renaming a_j^i's to $\#$, L_ℓ can be viewed as a union of repetition piecewise relations. According to Proposition 10, L_ℓ is a piecewise relation. By restoring $\#$ with a_j^i's, we get that the language L_ℓ is piecewise. □

Recall that a piecewise configuration $\mathbf{L}$ can be seen as a piecewise language; hence, it has an anchor sequence. The following proposition shows that the repeated application of a single read, write, or a conditional action to a piecewise configuration does not increase its anchor length.

Proposition 12 *For an action $act \in \Sigma_{rwc}$ and a piecewise channel configuration $\mathbf{L}$ with anchor length l, the anchor length of $(act^* : \mathbf{L})$ is less than or equal to l.*

Proof: Without loss of generality, we consider only a 2-channel system and conditional actions. Generalization to systems with more channels and unconditional reads and writes is trivial. The proof proceeds by induction on the anchor length of $\mathbf{L}$. The base case is trivial (and is omitted), the inductive cases are shown below. We assume

$\mathbf{L} = \langle U, V \rangle$, where U and V are simply piecewise expressions, and act is a conditional action of the form $1?a \rightarrow 2!b$.

- **Case 1:** $U = (a + a_1 + \cdots + a_n) \cdot W$, then

$$((1?a \rightarrow 2!b)^* : \mathbf{L}) = \langle U, V \rangle \ \vee ((1?a \rightarrow 2!b)^* : \langle W, V \cdot b \rangle)$$

- **Case 2:** $U = (a + a_1 + \cdots + a_n)^* \cdot W$, then

$$((1?a \rightarrow 2!b)^* : \mathbf{L}) = \langle U, V \cdot b^* \rangle \ \vee ((1?a \rightarrow 2!b)^* : \langle W, V \cdot b^* \rangle)$$

- **Case 3:** $U = c \cdot W$ or $U = \epsilon$, then

$$((1?a \rightarrow 2!b)^* : \mathbf{L}) = \epsilon$$

It is easy to see that, in all of the cases above, the anchor length has not increased. □

The following proposition shows that in multi-channel piecewise FIFO systems, the limit language is piecewise if the initial channel language is piecewise.

Proposition 13 *Let Act be a set of actions in a multi-channel system and $\mathbf{L}$ a set of channel configurations. Then, $(Act^* : \mathbf{L})$ is piecewise if $\mathbf{L}$ is piecewise.*

Proof: Let $Act = \{act_1, \ldots, act_n\}$. Then,

$$\begin{aligned} Act^* : \mathbf{L} &= (act_1 + \cdots + act_n)^* : \mathbf{L} \\ &= ((act_1^* \cdot act_2^* \cdot \cdots \cdot act_n^*)^*) : \mathbf{L} \\ &= \textstyle\bigcup_{w \in Act^*} (\textsc{starall}(w)) : \mathbf{L}\,, \end{aligned}$$

where for $w = w(1)w(2)\cdots$, $\textsc{starall}(w) = w(1)^* w(2)^* \cdots$.

In a k-channel system, we may assume that $\mathbf{L} = \sum_{0 \leq i < I} L(i)$ for some I, where $L(i) = R_0^i \# R_1^i \# \cdots \# R_{k-1}^i$ and R_j^i for $0 \leq j < k$ is simply piecewise. Then,

$$(Act^* : \mathbf{L}) \ = \ \textstyle\sum_{0 \leq i < I} (\bigcup_{w \in Act^*} (\textsc{starall}(w)) : L(i))\,.$$

Let l_i denote the anchor length of $L(i)$. By Proposition 12 $(\text{STARALL}(w) : L(i))$ is a union of piecewise expressions with anchor length bounded by l_i.

Let $L_w(i)$ denote $(\text{STARALL}(w) : L(i))$. Then,

$$(Act^* : \mathbf{L}) \;\; = \;\; \textstyle\sum_{0 \leq i < I} (\bigcup_{w \in Act^*} L_w(i)) \, .$$

Therefore, $(Act^* : \mathbf{L})$ is a (possibly infinite) union of piecewise expressions with anchor length bounded by $\max\{l_i \mid 0 \leq i < I\}$. Thus, by Proposition 11 it is piecewise. □

The proof of Proposition 13 is non-effective: it provides us with no algorithm for calculating the limit language in the general case. As explained, in order for the limit language to be regular, we either need to exclude conditional actions or consider only piecewise initial channel configurations. In the next chapter we will provide algorithms to compute the limit language in important subclasses of piecewise multi-channel systems with piecewise initial channel configurations.

3.5 Related Work

FIFO systems play key roles in the description and analysis of distributed systems. It is well-known that most non-trivial verification problems for FIFO systems are undecidable [24]. However, a substantial effort has gone into analysis of these systems. In general, two main approaches have been followed for the analysis of FIFO systems. The first approach, and the one taken in this thesis, is to identify practically useful subclasses of FIFO systems with decidable properties (e.g., [93, 59, 102, 77]). The second approach is to look for efficient semi-algorithms that scale to realistic examples, but do not guarantee to always terminate (e.g., [13, 14, 83]). Although this approach may look promising, in many cases finding a good boundary between scalability and termination is very challenging.

These two approaches may be combined, as illustrated in the analysis of lossy channel systems in which channels may lose messages. In these systems, the problem of

reachability of a given state is decidable [4, 57, 33, 1]; however, calculating the set of all reachable states is impossible.

In [57], Finkel considered a model of errors, called *completely specified protocols*, in which messages from the front of the channels can be lost. He showed that the termination problem is decidable for this class. In [4, 5], Abdulla and Jonsson consider a slightly more general notion of message lossiness: they assume that messages from anywhere in the channel can be lost. They show that the reachability problem is decidable and that the model-checking problem is undecidable in such a model. In [33], two other sources of errors in lossy channels are considered: duplication and arbitrary insertion of messages. It is shown that the systems with duplication errors are equivalent to Turing machines. On the other hand, in the case of systems having arbitrary insertion errors, it is shown that the reachability problem is decidable.

A recent research approach focuses on probabilistic lossy channels. In these systems, the losses of messages are seen as faults occurring with some given probability. This idea led to the introduction of a Markov chain model for lossy channel systems [73]. The preliminary results on probabilistic lossy channels reported for example in [8] prove decidability of qualitative model-checking under several limitations – it is shown whether a linear time property holds *almost surely*, i.e. with probability 1. The recent work of [2] shows that the decidability holds in more realistic models compared to [8]. The authors also consider other types of faulty behavior, such as corruption and duplication of messages and insertion of new messages, and show that the decidability results extend to these models.

The system considered in this thesis is not lossy; all channels are perfect, i.e., they do not lose any messages.

Pachl [93] introduces assertions for proving FIFO systems properties. These assertions are in the form of recognizable or rational descriptions of the channel configurations. Pachl proves that for FIFO systems if the set of reachable channel configurations (the limit language in our terminology) is recognizable then it is decidable to check for reachability of any given state. In other words, if recognizable assertions are associated

with control locations, it is decidable whether the assertions hold across transitions (and whether the assertion associated with the initial control location holds). It suffices to interleave these two algorithms: one that generates the recognizable relations for all control locations (the limit languages) and one that verifies whether a recognizable relation is included in another. Of course, this algorithm is unusable in practice. It was later shown in [33] that even though the reachability set might be recognizable, determining it may still be undecidable.

Pachl also notes that the decidability of the reachability problem for FIFO systems, where the limit language is rational, is still an open problem. The same approach used to prove the decidability in FIFO systems where the limit language is recognizable, is not applicable to the rational case: it is not decidable whether a given rational relation is contained in another.

An appealing general model to distributed systems with channels is that of *FIFO nets*, which are formulated as Petri nets except that places are replaced by FIFO channels. The survey [59] contains several decidability results, but they depend on the channel languages being *bounded*, i.e. a subset of some language $w_0^* \dots w_{n-1}^*$, where the w_i's are words.

A special kind of regular expression, called *semilinear*, was introduced in [58] as a symbolic presentation of regular, bounded languages describing channel contents. Unfortunately, a bounded language L has a polynomial density: there are at most $P(n)$ words of size n for some polynomial P. This may be a severe restriction: for example, it precludes sending a's and b's that are arbitrarily interspersed.

Boigelot *et al.* [13, 14, 12] describe a data structure, called Queue content Decision Diagrams (QDDs), for representing sets of queue contents, and a QDD-based semi-algorithm to compute a set of reachable states. The algorithm generates sets of reachable states from a single reachable state using *meta-transitions* [15]. Given a loop that appears in the system description and a control location c in that loop, a *meta-transition* is a transition that generates all global states that can be reached after repeated executions of the body of the loop. This is equivalent to computing the limit languages in our terminology.

The reachability algorithm is based on standard state exploration algorithms where meta-transitions are prioritized over transitions among different control locations. The resulting algorithm is called *loop-first search*. The loop-first search algorithm associates a QDD with each reachable control location. The termination of this algorithm depends on handling iterations of arbitrary sequences of actions. In [13], automata-theoretic algorithms are given to calculate $f : L$ and $f^* : L$ for a *single* read, write, or conditional action f. Boigelot's Ph.D. thesis [12] and [14] extend that to those action sequences that preserve recognizability of channel contents. For single-channel systems the authors show that the iteration of any sequence of single actions preserves the recognizability of the channel contents. For multi-channel systems, they show that the recognizability is preserved by the iteration of a sequence of actions if and only if the sequence of actions does not have more than one projection that is *counting* messages. Let σ be a sequence of actions involving only one channel. The sequence σ is counting if and only if there exists a recognizable set $U \subseteq \Sigma^*$ of channel contents such that $\sigma^{k_1}(U) \neq \sigma^{k_2}(U)$ for all $k_1, k_2 \in \mathbb{N}$, where $k_1 \neq k_2$ and $\sigma^k(U)$ denotes k repeated application of σ on U.

The key difference between [13, 14] and our work is that we allow iteration of multiple conditional actions. Conditional actions are used in practice in modeling and describing the communication protocols. However, sequences of conditional actions are harder to handle since in general they do not preserve recognizability of channel contents.

3.6 Conclusion

In this chapter, we described how the problem of computing the set of reachable states of a piecewise FIFO system can be reduced to calculating the set of all possible channel contents, i.e. the limit languages. We studied the problem of computing the limit languages in piecewise FIFO systems. We showed that the limit languages of multi-channel piecewise FIFO systems are not regular in general. However, the regularity can be achieved by excluding conditional actions or restricting the initial channel language to piecewise languages. Our results can be summarized in the following table:

	Channel Configuration	
	multiple w/o conditionals	multiple
piecewise initial channel	effectively piecewise	piecewise
regular initial channel	effectively regular	non-regular

In the next chapter, we will provide algorithms for computing the limit languages in single and multi-channel piecewise FIFO systems.

Chapter 4

Algorithmic Analysis of Piecewise FIFO Systems

In Chapter 3, we showed that the problem of computing the set of reachable states of a FIFO system is closely related to calculating the set of all possible channel contents, i.e. the limit language. In this chapter, we provide algorithms to compute the limit languages in single-channel and multi-channel piecewise FIFO systems. We also discuss the complexity of these algorithms. In addition, we demonstrate that important subclasses of multi-channel piecewise FIFO systems can be described by a finite-state, abridged structure, representing an expressive abstraction of the system. We present a procedure for building the abridged model. We show that we can analyze the computations of the more concrete model by analyzing the computations of the finite, abridged model.

4.1 Introduction

Piecewise languages play an important role in the study of FIFO systems. In this chapter, we present two algorithms for computing the set of reachable states of a piecewise FIFO system. As explained in the previous chapter, the problem of computing the set of reachable states of such a system is closely related to calculating the set of all possible channel

contents, i.e. the limit language. We present new algorithms for computing the limit language of a system with a single communication channel and a class of multi-channel systems with acyclic communication graphs. In these algorithms, we use automata to represent and manipulate the set of possible channel configurations. We also discuss the complexity of these algorithms.

The algorithm for single-channel systems requires that components be piecewise, and applies to any regular initial channel content. We show that the worst case complexity of the algorithm is at most exponential in the size of the automaton that represents the language of the initial channel content.

The algorithm for the multi-channel systems requires that both the components and the initial contents of the channels be piecewise, and that the communication graph be acyclic. As described in Section 3.3.1, a communication graph is a graph with channels as vertices and conditional actions as edges indicating which channels are connected by these actions. For ease of presentation, we develop the algorithm incrementally by restricting the topology of the communication graphs to *star*, *tree*, *inverted tree*, and *directed acyclic graph* (*DAG*) topologies. We study the worst case complexity of the algorithm for each topology. We show that for the *star* and *tree* topologies the worst case complexity of the algorithm is exponential in the size of the automaton that represents the language of the initial content of the channel in the origin of the star and the root of the tree, respectively.

Besides reachability properties, it is also crucial to be able to reason about computations of FIFO systems. We note that, in general, limit languages do not represent system computations but rather the reachable state set. In this chapter, we present a procedure that translates a piecewise FIFO system into an abridged structure, representing an expressive abstraction of the system. The abridged model is closely related to the concrete model; however, it differs in that the contents of the unbounded channels are represented by simply piecewise expressions. We show that the computation procedure of the abridged model terminates in piecewise FIFO systems with acyclic communication graphs. The representation of channel contents by simply piecewise expressions in

the context of global system transitions has allowed us to group together sets of actions that may be executed from a given global state.

The main reason for abridging a piecewise FIFO system is to be able to reason about its infinite behavior by analyzing the behavior of the finite, abridged model. We consider system properties expressed by a restricted, but expressive, class of Büchi automata. Here, the states of the Büchi automata represent the finite set of control locations and we require that the language of the automata be stuttering closed. We can then show that there is a computation of abridged model that satisfies the automaton, if and only if, there is a computation of the concrete model that satisfies the automaton. This procedure allows one to check a general class of temporal properties of piecewise FIFO systems.

The rest of this chapter is organized as follows. The algorithm for single-channel systems is presented in Section 4.2, and the one for multi-channel systems in Section 4.3. We present the procedure for translating a piecewise FIFO system to an abridged structure in Section 4.4 and conclude in Section 4.5 with a summary of our contributions in this chapter.

4.2 Analysis of Single-Channel Systems

In this section, we focus on the analysis of a single-channel piecewise FIFO system. We present an algorithm for calculating the limit language, show its correctness, and discuss its worst case complexity.

Figure 4.1 shows the algorithm SINGLELIMIT for calculating the limit language. The inputs to the algorithm are an automaton $\mathtt{A_I}$ representing a set of initial single-channel configurations $I \subseteq \Sigma^*$, and a set $Act \subseteq \Sigma_{rwc}$ of actions; the output is an automaton that accepts the limit language $(Act^* : I)$. For notational convenience, in the examples we use regular expressions instead of automata to represent channel configurations.

The algorithm has two phases. In the first phase, called PHASE1 (lines 3 – 6 of the SINGLELIMIT), the algorithm iteratively computes all configurations reachable by (i) reading the current channel content completely, and (ii) writing the result of conditional

1: **Aut** SINGLELIMIT (**Aut** $\mathtt{A_I}$, **Set** Act)
2: $\mathtt{R} := \epsilon,\ \mathtt{F} := \mathtt{A_I}$
3: **while** $\mathcal{L}(\mathtt{F}) \not\subseteq \mathcal{L}(\mathtt{R})$ **do**
4: $\mathtt{R} := \mathtt{R} + \mathtt{F}$
5: $\mathtt{F} := \text{APPLY}(\mathtt{F}, \mathtt{Act})$
6: **end while**
7: **return** PHASE2$(\mathtt{R}, \mathtt{Act})$

Figure 4.1: The SINGLELIMIT algorithm.

and other write actions. Each iteration of PHASE1 is done using the function APPLY. Let $Act \subseteq \Sigma_{rwc}$ be partitioned into unconditional write actions $Act_w = \{!a \mid !a \in Act\}$, and the rest $Act_r = Act \setminus Act_w$. In each iteration, if V is the set of currently reachable configurations, APPLY computes V' such that

$$V' \triangleq \{v \mid \exists u \in V,\ v \in (Act_r^{|u|} : u)\} \mid\mid (Act_w^* : \epsilon)\ .$$

Note that APPLY misses some reachable configurations. For example, let $Act = \{?a \rightarrow !c, ?b \rightarrow !d, !e\}$ and $I = ab$. Then, APPLY results in $\mathcal{L}(e^*ce^*de^*)$ and misses reachable configurations in $\mathcal{L}(be^*ce^*)$. This is fixed in the second phase, called PHASE2. Let W be a set of reachable configurations, the result of PHASE2 is a set W' such that

$$W' \triangleq \{w \mid \exists u, v, z,\ (v \cdot u \in W) \wedge (u \cdot z = w) \wedge (z \in \text{APPLY}(\{v\}, Act))\}\ .$$

These two phases are implemented using automata as described in the following sections.

4.2.1 PHASE1

As inputs PHASE1 takes an automaton $A = (\Sigma, Q, \delta, q^0, F)$, and a set of actions Act. Then, it iteratively computes a set of reachable configurations using function APPLY. Given automaton A and a set of actions Act, APPLY constructs an automaton $A' = (\Sigma, Q, \delta', q^0, F)$, where δ' consists of tuples of the form:

- (q, ϵ, q') if for some a it holds that $\delta(q, a, q')$ and $?a \in Act$, or
- (q, b, q') if for some a it holds that $\delta(q, a, q')$ and $?a \rightarrow !b \in Act$, or
- (q, c, q) if $!c \in Act$.

Intuitively, the first rule of δ' corresponds to unconditional reads, the second – to renaming the labels of the transitions according to the conditional actions, and the third – to unconditional writes.

For example, let $Act = \{?a \rightarrow !b, ?b \rightarrow !a, ?c, !a\}$ and $I = (ac)^*aba^*$. Figure 4.2(a) shows automaton A recognizing $\mathcal{L}(I)$. To construct $A' = \textsc{Apply}(A, Act)$, the transitions labeled by a are relabeled to b, transitions labeled by b are relabeled to a, and transitions labeled by c are replaced by ϵ-transitions. In addition, self-loop transitions labeled by a are added to every state. Figure 4.2(b) shows automaton A'. Similarly, we can construct automaton $A'' = \textsc{Apply}(A', Act)$ and $A''' = \textsc{Apply}(A'', Act)$ which are shown in Figure 4.2(c) and (d), respectively. As can be seen, applying function APPLY once more (to A''') results in automaton A'', thus, we have reached a fixpoint.

4.2.2 PHASE2

Let $A = (\Sigma, Q, q^0, \delta, F)$ be an automaton and s be a state in Q. We construct two automata: $A_1 = (\Sigma, Q, q^0, \delta, \{s\})$ and $A_2 = (\Sigma, Q, \{s\}, \delta, F)$. Let A'_1 be the automaton constructed by applying function APPLY to A_1, i.e., $A'_1 = \textsc{Apply}(A_1, Act)$. Then, the language of $A_2 \cdot A'_1$ contains a word $u \cdot z$ if and only if (i) there exists a word v such that $v \cdot u$ is accepted by A via a run passing through the state s, and (ii) $z \in \textsc{Apply}(\{v\}, Act)$. We call this operation $\textsc{Prefix}(A, s, Act)$. It is easy to see that

$$\textsc{Phase2}(A, Act) = \bigcup_{s \in Q} \textsc{Prefix}(A, s, Act)\,.$$

For our running example, Figure 4.3 shows how $\textsc{Prefix}(A, s, Act)$ is implemented using automata. The leftmost automaton in Figure 4.3 (automaton A) recognizes the

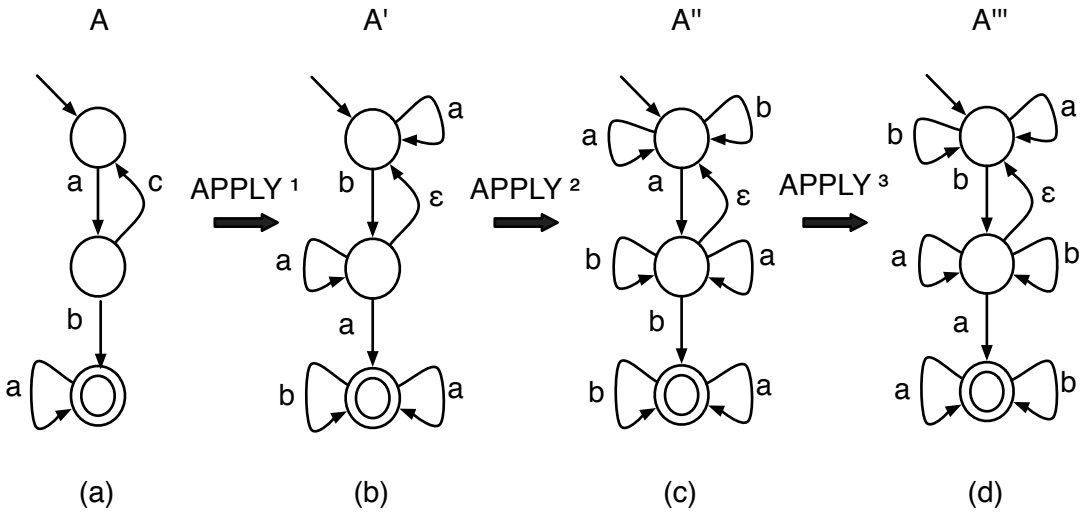

Figure 4.2: An example illustrating PHASE1 with automaton A and $Act = \{?a \rightarrow !b, ?b \rightarrow !a, ?c, !a\}$ as inputs.

language $I = (ac)^*aba^*$. To compute PREFIX(A, s, Act), we break A on state s, which results in two automata A_1 and A_2. We compute A_1' by applying function APPLY to A_1. Then, we concatenate A_2 and A_1'. The resulting automaton represents PREFIX(A, s, Act) and is shown on the rightmost of the Figure 4.3.

The algorithm in Figure 4.1 always terminates. Given an automaton A, APPLY produces an automaton with the same number of states as A. Thus, the set $\{\text{APPLY}^i(A, Act)\}_i$ is finite, and the algorithm always terminates.

Theorem 1 *Let A_I be an automaton representing a set of configurations, Act a set of actions, and A_L the automaton returned by* SINGLELIMIT(A_I, Act). *Then,* $\mathcal{L}(A_L) = (Act^* : \mathcal{L}(A_I))$.

Proof: According to the SINGLELIMIT algorithm shown in Figure 4.1,

$$\mathcal{L}(A_L) = \text{PHASE2}\left(\bigcup_{i \in \mathbb{N}} \text{APPLY}^i(A_I, Act), Act\right).$$

Note that since in each iteration APPLY produces an automaton with the same number of states as A_I, $\bigcup_i \text{APPLY}^i(A_I, Act)$ is a finite union.

Let $w \in (Act^* : \mathcal{L}(A_I))$ be a reachable channel content. Then, w is reached by reading the current channel content completely (and writing the results of conditional and other write actions) zero or more times, and then reading the resulting content partially. Let $\#$ – a fresh letter not in Σ, be a marker at the end of the initial channel content. The maker $\#$ is used only for establishing the proof and is eliminated later using $\text{ERASE}_{\#}$. Then,

$$w \in (Act^* : \mathcal{L}(A_I)) \Leftrightarrow \exists u, v,\ (u \cdot v) = w \wedge \\ \exists p, q,\ (u\#v) \in (((Act^*(?\#)(!\#))^p (Act)^q) : (\mathcal{L}(A_I) \cdot \#)) .$$

At the end of each iteration of PHASE1, $\#$ is read and then written again on the channel to mark the beginning of the new iteration.

The theorem follows from

$$(\text{APPLY}(\mathcal{L}(A), Act) \cdot \#) = (Act^*(?\#)(!\#)) : (\mathcal{L}(A) \cdot \#)$$

and

$$\text{PHASE2}(\mathcal{L}(A), Act) = \text{ERASE}_{\#}(Act^* : (\mathcal{L}(A) \cdot \#))$$

where $\text{ERASE}_{\#}$ projects out the letter $\#$. □

4.2.3 Complexity Analysis

Let $h = |A_I|$ denote the size of A_I – the automaton representing the set of initial configurations. As discussed above, $\text{APPLY}(A_I, Act)$ produces an automaton with the same number of states as A_I by relabeling the transitions of A_I. In the worst case, each transition can be updated at most $|\Sigma|$ times. Thus, the worst case complexity of the SINGLELIMIT algorithm is $|\Sigma|^h$.

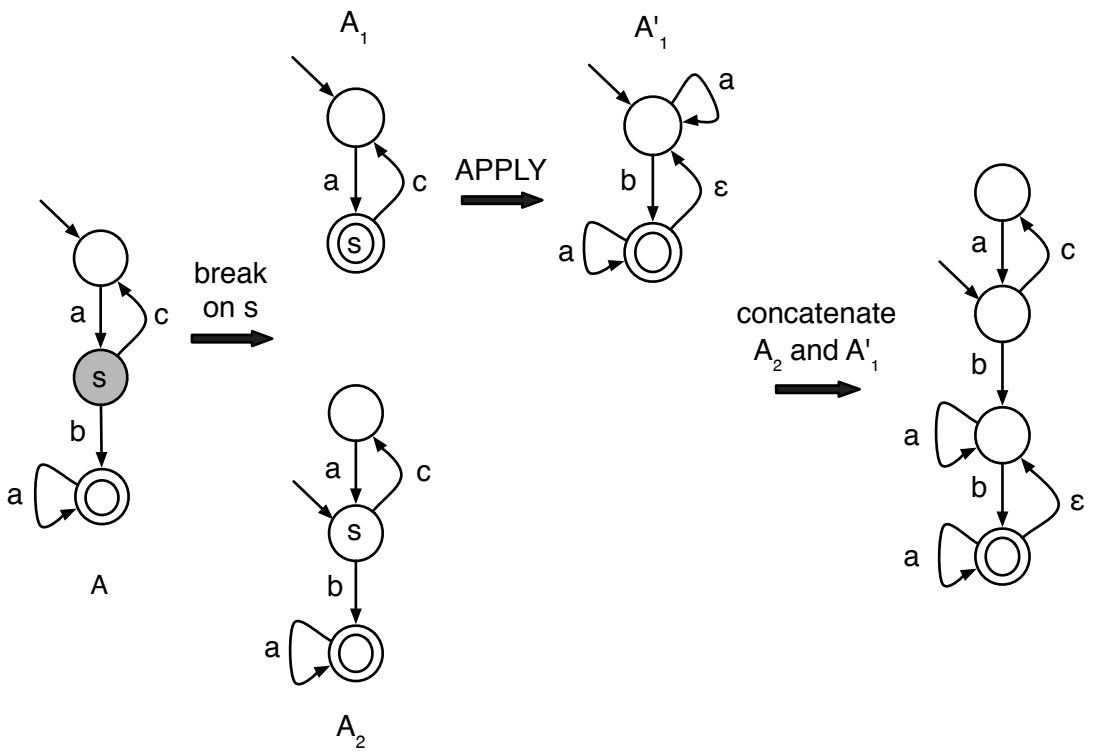

Figure 4.3: An example illustrating PREFIX operation with automaton A, state s, and $Act = \{?a \rightarrow !b, ?b \rightarrow !a, ?c, !a\}$ as inputs.

Theorem 2 *Let A_I be an automaton over a finite alphabet Σ representing a set of single-channel configurations, and $h = |A_I|$. Then, in the worst case, the running time of the* SINGLELIMIT *algorithm is $O(|\Sigma|^h)$.*

4.3 Multi-Channel Systems

In this section, we focus on the limit language problem for a set of actions, Act, on a k-channel system with an acyclic communication graph, $CG(Act)$. For ease of presentation, we develop the algorithm for the DAG topology incrementally by restricting $CG(Act)$ to *star*, *tree*, *inverted tree*, and eventually DAG topologies. We show correctness of each algorithm and discuss its complexity.

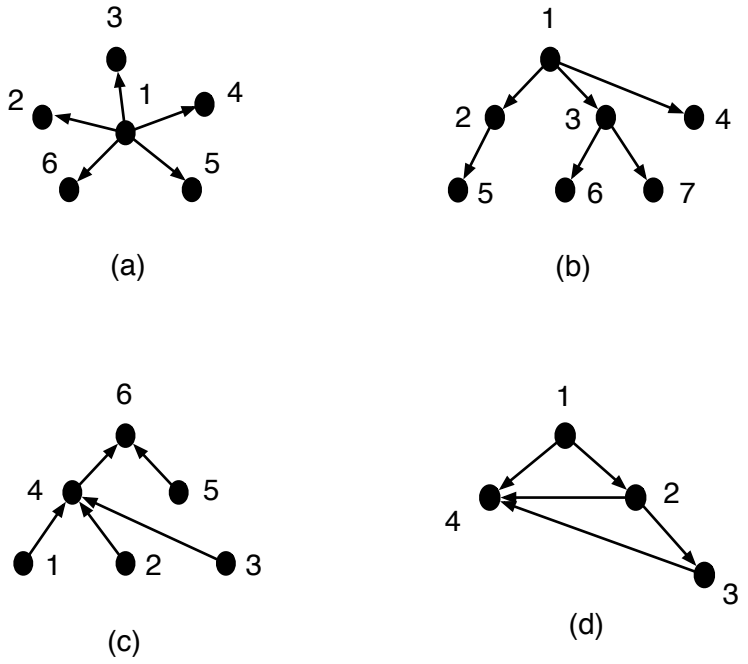

Figure 4.4: Communication topologies: (a) star, (b) tree, (c) inverted tree, (d) DAG.

Throughout, we assume that all actions are conditionals. This is not a significant limitation since: (i) unconditional reads can be modeled by conditionals that write to dummy channels, and (ii) unconditional writes can be handled easily, but are omitted for presentational convenience. The algorithms are based on automata, and operate on piecewise configurations (refer to Section 3.4). In this section, we assume a piecewise (k-channel) configuration $\mathbf{u}$ is represented by a tuple $\langle A_1, \ldots, A_k \rangle$, where each A_i is a PO-FSA over Σ. Thus, a piecewise configuration $\mathbf{u} = \langle A_1, \ldots, A_k \rangle$ represents a set of channel configurations $\mathcal{L}(A_1) \times \cdots \times \mathcal{L}(A_k)$, which is denoted by $\mathcal{L}(\mathbf{u})$. The *size* of $\mathbf{u}$ is the sum of the sizes of all of the automata in it. $\mathcal{L}$ is extended to finite sets of piecewise configurations in the usual way: $\mathcal{L}(\mathbf{U}) = \bigcup_{\mathbf{u} \in \mathbf{U}} \mathcal{L}(\mathbf{u})$.

For notational convenience, in the examples we use tuples of regular expressions instead of PO-FSA to represent piecewise configurations. For example, $\mathbf{u} = \langle a^*b, (c + d)^*e \rangle$ represents a piecewise configuration where $\mathbf{u}[1]$ is an automaton representing a^*b, and $\mathbf{u}[2]$ is an automaton representing $(c + d)^*e$. In pseudo-code, we use **Conf** for the type of piecewise configurations, and the notation "X **with** $[i] = y$" to mean $X[i \mapsto y]$.

4.3.1 Star Topology

A set of actions Act has a *star* topology if and only if there exists a unique channel o, the *origin*, such that for every action $i?a \rightarrow j!b$ in Act, $i = o$ and $j \neq o$, i.e., $CG(Act)$ is a star (see Figure 4.4(a)). In the sequel, we assume that channel 1 is the origin channel.

Let $\mathbf{u}$ be a piecewise channel configuration. The algorithm DOREAD, shown in Figure 4.5, computes the limit $(Act^* : \mathcal{L}(\mathbf{u}))$. The use of the argument `idx` is explained in Section 4.3.4. The `writeWL` and `readWL` are two global work lists used by the algorithm. DOREAD is driven by the automaton $\mathbf{u}[1]$ representing the content of channel 1. For example, if $\mathbf{u}[1] = M_1^* a_1 M_2^* a_2$, then the algorithm first computes all reachable configurations $\mathbf{w}$ whose channel 1 content, $\mathbf{w}[1]$, is in $\mathcal{L}(M_1^* a_1 M_2^* a_2)$, then all configurations with $\mathbf{w}[1]$ in $\mathcal{L}(M_2^* a_2)$, then all configurations with $\mathbf{w}[1]$ being ϵ. Each iteration of the algorithm is done using functions SATURATE and STEP. For our running example, in the first iteration, SATURATE computes all reachable configurations with $\mathbf{w}[1]$ in $\mathcal{L}(M_1^* a_1 M_2^* a_2)$ and STEP computes all configurations with $\mathbf{w}[1]$ in $\mathcal{L}(M_2^* a_2)$, etc.

SATURATE Let $\mathbf{u}$ be a piecewise configuration, where $\mathbf{u}[1] = M^* \cdot Z$ for some Z, i.e., $\mathbf{u}[1]$ is a PO-FSA with a single initial state q^0 and some self-loops on q^0. Note that, $\mathbf{u}$ represents a set of configurations with an arbitrary number of letters from M at the head of channel 1. The SATURATE phase computes a set of configurations that are reachable by reading an arbitrary number of these letters. Formally, $\text{SATURATE}(\mathbf{u}, 1)$ defines a piecewise configuration $\mathbf{u}'$ such that $\mathcal{L}(\mathbf{u}')$ is $\{\mathbf{w} \mid \mathbf{w} \in (\text{Act}^* : \mathcal{L}(\mathbf{u})) \wedge (\mathbf{w}[1] \in \mathcal{L}(\mathbf{u}[1]))\}$. It corresponds to a transformation $\text{SATURATE}(\mathbf{u}, 1) = \mathbf{u}'$ such that

$$\mathbf{u}'[i] \triangleq \begin{cases} \mathbf{u}[1] & \text{if } i = 1 \\ \mathbf{u}[i] \cdot (Act : M)^* & \text{otherwise.} \end{cases}$$

For example, given $Act = \{1?a \rightarrow 2!a, 1?a \rightarrow 3!a\}$ and $\mathbf{u} = \langle a^*(b+c), \epsilon, \epsilon \rangle$, $\text{SATURATE}(\mathbf{u}, 1)$ computes a piecewise configuration $\mathbf{u}' = \langle a^*(b+c), a^*, a^* \rangle$.

STEP Let $\mathbf{u}$ be a piecewise channel configuration, where $\mathbf{u}[1] = (a_0 + \cdots + a_n) \cdot Z$ for

```
1:  global List readWL
2:  global List writeWL
3:  List DoRead(Conf u, Channel ch)
4:    doReadRec(u, ch, 0, true)
5:    return writeWL
6:  proc doReadRec(Conf u, Channel ch, int idx, bool na)
7:    if u[ch] = (U_1 + ... + U_n) · W  for some W then
8:      for i ∈ [1..n] do doReadRec(u with [ch] := U_i, ch, idx, na)
9:    else if u[ch] = M* · W  for some W then
10:     u := Saturate(u, ch, idx)
11:     writeWL = writeWL ∪ {u}
12:     u := (u with [ch] := W)
13:     doReadRec(u, ch, idx + 1, false)
14:   else if na ∧ (u[ch] = ε) then
15:     writeWL := writeWL ∪ {u}
16:   else if u[ch] = a · W  for some W then
17:     if na then writeWL := writeWL ∪ {u}
18:     U := Step(u, ch, idx)
19:     forall u' ∈ U do doReadRec(u', ch, idx + 1, true)
20:   end if
```

Figure 4.5: DoRead algorithm for star topology and its supporting routines.

1: **Conf** SATURATE(**Conf** u, **Channel** ch, **int** idx)
2: **let** $(Q, q^0, \delta, F) = \mathtt{u}[\mathtt{ch}]$, $M = \{a \mid (q^0, a, q^0) \in \delta\}$
3: **forall** $i \in ([1..k] \setminus \{\mathtt{ch}\})$ **do**
4: **let** $M' = \{b \mid (\mathtt{ch}?a \to i!b) \in Act \wedge a \in M \wedge$
$(\forall j < \mathtt{ch},\ \text{idx}(j, b) = \text{idx}(j, a)) \wedge \text{idx}(\mathtt{ch}, b) = \mathtt{idx}\}$
5: $\mathtt{u}'[i] := (\mathtt{u}[i] \cdot (M')^*)$
6: **return** u′
7: **Set**⟨**Conf**⟩ STEP(**Conf** u, **Channel** ch, **int** idx)
8: $\mathtt{U}' := \emptyset$
9: **let** $(Q, q^0, \delta, F) = \mathtt{u}[\mathtt{ch}]$, $M = \{a \in \Sigma \mid \exists q',\ (q^0, a, q') \in \delta \wedge q' \neq q^0\}$
10: **forall** $a \in M \wedge i \in \{j \mid \exists b,\ (\mathtt{ch}?a \to j!b) \in Act\}$ **do**
11: $M' = \{b \mid (\mathtt{ch}?a \to i!b) \in Act \wedge$
$(\forall j < \mathtt{ch},\ \text{idx}(j, b) = \text{idx}(j, a)) \wedge \text{idx}(\mathtt{ch}, b) = \mathtt{idx}\}$
12: $\mathtt{u}'[\mathtt{ch}] := (Q, \delta(q^0, a) \setminus \{q^0\}, \delta, F)$
13: $\mathtt{u}'[i] := \mathtt{u}[i] \cdot M'$
14: $\mathtt{U}' := \mathtt{U}' \cup \{\mathtt{u}'\}$
15: **return** U′

Figure 4.6: STEP and SATURATE algorithms.

some Z, i.e, $\mathbf{u}[1]$ is a PO-FSA with a single initial state with no self-loops. Here, $\mathbf{u}$ represents a set of configurations whose channel 1 content starts with a letter in $\{a_0, \ldots, a_n\}$. The STEP phase computes all configurations that are reachable by reading exactly one letter from channel 1. Formally, $\text{STEP}(\mathbf{u}, 1)$ defines a set $\mathbf{U}'$ of piecewise configurations such that $\mathcal{L}(\mathbf{U}') \triangleq \{\mathbf{w} \mid \mathbf{w} \in (Act^* : \mathcal{L}(\mathbf{u})) \wedge (\mathbf{w}[1] \in Z)\}$. It corresponds to a transformation

$$\text{STEP}(\mathbf{u}, 1) \triangleq \bigcup_{\{1?a \to i!b \in Act \mid a \in \{a_0, \ldots, a_n\}\}} (1?a \to i!b) : \mathcal{L}(\mathbf{u})\,.$$

For example, given $Act = \{1?b \to 2!b, 1?c \to 3!c\}$ and $\mathbf{u} = \langle (b{+}c), \epsilon, \epsilon \rangle$, $\text{STEP}(\mathbf{u}, 1)$ computes two piecewise configurations $\mathbf{u}'_1 = \langle \epsilon, b, \epsilon \rangle$ and $\mathbf{u}'_2 = \langle \epsilon, \epsilon, c \rangle$.

Detailed implementations of SATURATE and STEP for a set of actions Act on k channels are shown in Figure 4.6.

Theorem 3 *Let* $\mathbf{u}$ *be a piecewise channel configuration,* Act *an action set with star topology and origin* o*, and* $\mathbf{U}'$ *the set returned by* DOREAD$(\mathbf{u}, o)$*. Then,* $\mathcal{L}(\mathbf{U}') = (Act^* : \mathcal{L}(\mathbf{u}))$.

Proof: The proof is straightforward by the induction on the structure of $\mathbf{u}[1]$. □

Complexity Analysis For a piecewise configuration $\mathbf{u}$, the depth of the recursion of DOREAD is bounded by $h = |\mathbf{u}[o]|$ for the origin o. Inside each call, SATURATE takes constant time and returns a single configuration; however, STEP may return a set of configurations. In a k-channel system, the size of this set is bounded by $k - 1$. Thus, the complexity of the DOREAD algorithm is bounded by the number of internal nodes of a $(k-1)$-ary tree of height h. There are h such nodes for $k = 2$, and $((k-1)^{(h+1)} - 1)/(k-2)$ for $k > 2$.

Theorem 4 *Let* $\mathbf{u}$ *be a piecewise channel configuration,* Act *a set of actions with star topology on* k *channels with origin* o*, and* $h = |\mathbf{u}[o]|$*. Then, in the worst case, the running time of* DOREAD$(\mathbf{u}, o)$ *is* $O(max(k^h, h))$.

4.3.2 Tree Topology

A set of actions Act has a *tree* (or, more generally, a *forest*) topology if and only if for all actions $i?a \rightarrow j!b$ and $i'?a' \rightarrow j'!b'$ in Act, $j = j' \Rightarrow i = i'$. That is, $CG(Act)$ is a tree (e.g. Figure 4.4(b)).

The DOREAD algorithm for the star topology is not applicable to the tree topology since it assumes that all reads come from a single channel. However, an action set with a tree topology can be partitioned such that each partition has a star topology. Formally, for a set of actions Act, let Act_i denote all the actions that read from channel i. Then, $\{Act_i\}$ partitions Act where each Act_i has a star topology with origin i. For example, consider the communication graph in Figure 4.4(b): here, $Act_1 = \{1? \rightarrow 2!, 1? \rightarrow 3!, 1? \rightarrow 4!\}$, $Act_2 = \{2? \rightarrow 5!\}$, and $Act_3 = \{3? \rightarrow 6!, 3? \rightarrow 7!\}$.

This way, DoRead can be used to compute $Act_i^* : \mathcal{L}(\mathbf{u})$ for any channel i and a piecewise configuration $\mathbf{u}$. Furthermore, it can be applied iteratively to compute sequential composition of the partitions of Act. For example, computation of $(Act_1^* \cdot Act_2^*) : \mathcal{L}(\mathbf{u})$ is done by using DoRead to first compute $\mathbf{U}'$ such that $\mathcal{L}(\mathbf{U}') = (Act_1^* : \mathcal{L}(\mathbf{u}))$, and using it again to compute $(Act_2^* : \mathcal{L}(\mathbf{U}'))$. In the following, we show how to extend this to the computation of the full limit language.

The graph $CG(Act)$ is acyclic and, therefore, induces a partial order $\preceq$ on channels (vertices of the graph). For channels i and j, $i \preceq j$ if and only if there exists a path from i to j in $CG(Act)$. Intuitively, channel i is less than channel j if the final content of j depends on the initial content of i. We say that channel j *depends* on channel i if $i \preceq j$, and that i and j are *interdependent* if either $i \preceq j$ or $j \preceq i$. Without loss of generality, we assume that the partial order $\preceq$ is extended to a total order and that the channels are numbered such that $i \preceq j$ if and only if $i \leq j$. For example, channel 3 may depend on channels 1 and 2, and 2 depends only on 1. The ordering and renaming of the channels can be done in time linear in the size of the CG.

If Act has a tree topology, every channel in $CG(Act)$ has at most one immediate predecessor. Thus, for every sequence $x \in Act^*$, there exists a sequence y such that: (i) y has the same actions as x, (ii) all reads of y are ordered, i.e., $y \in Act_1^* \cdot Act_2^* \cdot \cdots$, and (iii) if $(x : \mathbf{w}) \neq \emptyset$ for some $\mathbf{w}$, then $(y : \mathbf{w}) = (x : \mathbf{w})$. For example, for Act in Figure 4.4(b), and $x = 1? \rightarrow 2!\ 1? \rightarrow 3!\ 2? \rightarrow 5!\ 1? \rightarrow 4!\ 3? \rightarrow 6!$, an equivalent sequence y is:

$$y = \underbrace{1? \rightarrow 2!\ 1? \rightarrow 3!\ 1? \rightarrow 4!}_{\in Act_1^*}\ \underbrace{2? \rightarrow 5!}_{\in Act_2^*}\ \underbrace{3? \rightarrow 6!}_{\in Act_3^*}.$$

Theorem 5 *Let Act be an action set on k channels such that $CG(Act)$ is a tree, and $\mathbf{w}$ a channel configuration. Then,*

$$((Act^* : \mathbf{w}) = ((Act_1^* \cdot \cdots \cdot Act_k^*) : \mathbf{w}))\,.$$

Proof: We say that two action sequences x and y are enabled equivalent, written $x \equiv_e y$,

if x and y behave identically on the input sequences that enable all actions of x. Formally,

$$x \equiv_e y \triangleq (\forall \mathbf{w},\ (x : \mathbf{w} \neq \emptyset) \Rightarrow (x : \mathbf{w} = y : \mathbf{w}))\,.$$

To establish the proof, we use the following equivalences (rules):

$$\begin{array}{lllll}
(1) & i?a\ j?b & \equiv_e & j?b\ i?a & (i \neq j) \\
(2) & i!a\ j!b & \equiv_e & j!b\ i!a & (i \neq j) \\
(3) & i!a\ j?b & \equiv_e & j?b\ i!a & \\
(4) & i?a\ j!b & \equiv_e & j!b\ i?a &
\end{array}$$

Let $p?c \rightarrow q!d$ and $i?a \rightarrow j!b$ be two conditional actions such that $i \preceq p$. Note that the tree topology implies that $q \neq j$. We show that these two conditional actions can be commuted in any sequence of actions. The theorem follows by recursive application of this rule:

$$\begin{array}{lll}
 & p?c \rightarrow q!d\ i?a \rightarrow j!b & \text{notation} \\
= & p?c\ q!d\ i?a\ j!b & \text{using rule 3} \\
= & p?c\ i?a\ q!d\ j!b & \text{using rule 2} \\
= & p?c\ i?a\ j!b\ q!d & \text{using rule 1} \\
= & i?a\ p?c\ j!b\ q!d & \text{using rule 4} \\
= & i?a\ j!b\ p?c\ q!d & \text{notation} \\
= & i?a \rightarrow j!b\ p?c \rightarrow q!d &
\end{array}$$

Thus, given a sequence of actions $x \in Act^*$, there exists an enabled equivalent sequence of actions $y \in Act_1^* \cdot \cdots \cdot Act_k^*$ where the conditional actions of x are ordered based on the total order on the channels they are reading from. □

Theorem 5 leads to an obvious algorithm for computing the limit language in the tree topology: (i) establish a total order on channels based on the CG, and (ii) use this order to iteratively apply DoRead to each partition Act_i. We call this algorithm TreeLimit (see Figure 4.7). Since TreeLimit proceeds through a finite total order of channels, it always terminates.

```
1: List TREELIMIT (Conf u)
2:   readWL := u
3:   for ch = 0 to k do
4:     writeWL := ∅
5:     forall u ∈ readWL do doRead(u, ch)
6:     readWL := readWL ∪ writeWL
7:   return readWL
```

Figure 4.7: The TREELIMIT algorithm.

Theorem 6 *Let* $\mathbf{u}$ *be a piecewise configuration,* Act *an action set with tree topology, and* $\mathbf{U}'$ *the set of configurations returned by* TREELIMIT$(\mathbf{u})$. *Then,* $\mathcal{L}(\mathbf{U}') = (Act^* : \mathcal{L}(\mathbf{u}))$.

Complexity Analysis Without loss of generality, we assume that $CG(Act)$ is an N-ary tree with M internal nodes and that the initial content of all the channels except the root is empty. Let $\mathbf{u}$ be a piecewise configuration, and $h = |\mathbf{u}|$. By Theorem 4, computation of $Act_i^* : \mathcal{L}(\mathbf{u})$ produces at most $\max(N^h, h)$ piecewise configurations, each of size at most h. TREELIMIT applies computation $Act_i^* : \mathcal{L}(\mathbf{u})$, M times, which produces at most $\max(N^{h\times M}, h^M)$ configurations.

Theorem 7 *Let* $\mathbf{u}$ *be a piecewise configuration,* Act *a set of actions with a tree topology of degree* N *and* M *internal nodes, and* $h = |\mathbf{u}[1]|$. *In the worst case, the running time of* TREELIMIT$(\mathbf{u})$ *is* $O(\max(N^{h\times M}, h^M))$.

4.3.3 Inverted Tree Topology

A set of actions Act has an *inverted tree* topology if and only if for all conditional actions $i?a \rightarrow j!b$ and $i'?a' \rightarrow j'!b'$ in Act, $i = i' \Rightarrow j = j'$. That is, $CG(Act)$ is an inverted tree (e.g., see Figure 4.4(c)).

In the inverted tree topology, a channel may depend on several pairwise independent channels. Therefore, Theorem 5 is no longer applicable. For example, let $Act = \{1?a \rightarrow$

$3!a, 2?b \rightarrow 3!b\}$, and $\mathbf{w} = \langle aa, bb, \epsilon\rangle$ be a configuration. The partial order on the channels induced by $CG(Act)$ is $\{1 \preceq 3, 2 \preceq 3\}$, with two obvious linearizations. A configuration $\langle \epsilon, \epsilon, abab\rangle$ is reachable from $\mathbf{w}$, but does not belong to either $((1?a \rightarrow 3!a)^*(2?b \rightarrow 3!b)^*) : \mathbf{w}$, or $((2?b \rightarrow 3!b)^*(1?a \rightarrow 3!a)^*) : \mathbf{w}$, which contradicts the theorem.

For simplicity of presentation, we assume that there is a unique channel, referred to as l, that has multiple dependencies, like channel 3 in the above example. That means l is the only channel whose node in $CG(Act)$ has an in-degree greater than or equal to 2. In this case, it is possible to (i) replace channel l with new channels, called *shadows* of l, and turn Act into a tree topology, (ii) solve the new limit problem using TREELIMIT, and (iii) combine the contents of shadow channels together. This is further explained below.

We define a function ADDS that introduces shadow channels for channel l by redirecting each conditional action that reads from i and writes to l to write to a newly created shadow channel $\hat{l_i}$. Formally,

$$\text{ADDS}(i?a \rightarrow j!b, l) \triangleq \begin{cases} i?a \rightarrow \hat{l_i}!b & \text{if } j = l \\ i?a \rightarrow j!b & \text{otherwise.} \end{cases}$$

ADDS breaks dependencies between channels. Let $\widehat{Act} = \text{ADDS}(Act, l)$. If $CG(Act)$ is an inverted tree, then $CG(\widehat{Act})$ is a tree. For our running example, we introduce two shadow channels for channel 3; therefore, $\widehat{Act} = \{1?a \rightarrow \hat{3}_1!a, 2?b \rightarrow \hat{3}_2!b\}$. We use $\mathsf{S}(l)$ to denote the shadows of l.

Let $\mathbf{w}$ be a configuration, and $\hat{\mathbf{w}}$ be its extension to shadow channels. That is, $\hat{\mathbf{w}}[i] = \mathbf{w}[i]$ if $i \notin \mathsf{S}(l)$, and $\hat{\mathbf{w}}[i] = \epsilon$ otherwise. For example, if $\mathbf{w} = \langle aa, bb, b\rangle$, then by introducing two shadow channels for channel 3, $\hat{\mathbf{w}} = \langle aa, bb, b, \epsilon, \epsilon\rangle$.

The sets $(Act^* : \mathbf{w})$ and $((\widehat{Act})^* : \hat{\mathbf{w}})$ are closely related. Let $\mathbf{t} \in (x : \mathbf{w})$ be a configuration reachable from $\mathbf{w}$ by a sequence $x \in Act^*$, and $\hat{\mathbf{t}} \in (\text{ADDS}(x, l) : \hat{\mathbf{w}})$ be a configuration reachable from $\hat{\mathbf{w}}$, where ADDS is extended to sequences in an obvious way. ADDS only augments actions that write to l. Thus, $\mathbf{t}[i] = \hat{\mathbf{t}}[i]$ for any i that is different from l or its shadow channels $\mathsf{S}(l)$. By adding shadow channels for l, all the

writes on l are redirected to its shadows and $\hat{\mathbf{t}}[l]$ is the initial content of l, hence, it is a prefix of $\mathbf{t}[l]$. Each shadow channel $\hat{l}_i$ keeps track of what was read from channel i and written to l, hence, $\hat{\mathbf{t}}[\hat{l}_i]$ is a subsequence of $\mathbf{t}[l]$.

In our running example, $Act = \{1?a \rightarrow 3!a, 2?b \rightarrow 3!b\}$, $\widehat{Act} = \{1?a \rightarrow \hat{3}_1!a, 2?b \rightarrow \hat{3}_2!b\}$, $\mathbf{w} = \langle aa, bb, b\rangle$, and $\hat{\mathbf{w}} = \langle aa, bb, b, \epsilon, \epsilon\rangle$. Let x be a sequence in Act^* such as

$$x \;=\; 1?a \rightarrow 3!a \;\; 2?b \rightarrow 3!b \;\; 1?a \rightarrow 3!a \;\; 2?b \rightarrow 3!b\,.$$

Then,

$$\textsc{AddS}(x, 3) \;=\; 1?a \rightarrow \hat{3}_1!a \;\; 2?b \rightarrow \hat{3}_2!b \;\; 1?a \rightarrow \hat{3}_1!a \;\; 2?b \rightarrow \hat{3}_2!b\,.$$

In order to formalize the relation between $(Act^* : \mathbf{w})$ and $((\widehat{Act})^* : \hat{\mathbf{w}})$, we define a function MERGES. Given a configuration over shadow channels, MERGES produces all corresponding configurations without shadows. Formally,

$$\mathbf{t} \in \textsc{MergeS}(\hat{\mathbf{t}}, l) \Leftrightarrow (\forall i \neq l \wedge i \notin \mathsf{S}(l),\, \mathbf{t}[i] = \hat{\mathbf{t}}[i]) \wedge (\mathbf{t}[l] \in \mathcal{L}(\hat{\mathbf{t}}[l] \cdot ||_{j \in \mathsf{S}(l)} \{\hat{\mathbf{t}}[j]\})\,.$$

In the above example, let $\mathbf{t} = (x : \mathbf{w}) = \langle \epsilon, \epsilon, babab\rangle$ and $\hat{\mathbf{t}} = (\textsc{AddS}(x, 3) : \hat{\mathbf{w}}) = \langle \epsilon, \epsilon, b, aa, bb\rangle$. Then, $\textsc{MergeS}(\hat{\mathbf{t}}, l) = \{\langle \epsilon, \epsilon, b \cdot (aa \,||\, bb)\rangle\}$ that is equal to

$$\{\langle \epsilon, \epsilon, baabb\rangle, \langle \epsilon, \epsilon, bbbaa\rangle, \langle \epsilon, \epsilon, babab\rangle, \langle \epsilon, \epsilon, bbaba\rangle, \langle \epsilon, \epsilon, babba\rangle, \langle \epsilon, \epsilon, bbaab\rangle\}\,.$$

As can be seen, $\mathbf{t} \in \textsc{MergeS}(\hat{\mathbf{t}}, l)$.

Theorem 8 *Let Act, $\widehat{Act}$, $\mathbf{w}$, $\hat{\mathbf{w}}$, and l be as above. Then,*

$$\mathbf{t} \in (Act^* : \mathbf{w}) \textit{ if and only if } \exists \hat{\mathbf{t}} \in ((\widehat{Act})^* : \hat{\mathbf{w}}),\, \mathbf{t} \in \textsc{MergeS}(\hat{\mathbf{t}}, l)\,.$$

Proof: The proof follows directly from the definition of $\widehat{Act}$, $\hat{\mathbf{w}}$, and MERGES. By augmenting Act with shadow channels and extending $\mathbf{w}$ to $\hat{\mathbf{w}}$, all the writes to the channel with in-degree greater than or equal to 2 are forwarded to the corresponding shadow

```
1: Conf MERGES(Conf u, Channel ch);
2: Conf doWrite(Conf conf, Channel ch)
3:   return MERGES(u, ch)
4: List MULTILIMIT (Conf u)
5:   readWL := u
6:   for ch = 0 to k do
7:     writeWL := ∅
8:     forall u ∈ readWL do doRead(u, ch)
9:     if ch + 1 < k then
10:      forall u ∈ writeWL do
11:        readWL := readWL ∪ {doWrite(u, ch + 1)}
12:  return readWL
```

Figure 4.8: MULTILIMIT algorithm and its supporting routines.

channels. Then, MERGES computes all possible configurations reachable by different interleavings of actions by shuffling the contents of the shadow channels. □

Both Theorem 8 and MERGES are easily lifted to piecewise configurations such that if $\mathbf{u}$ is a piecewise configuration, then MERGES$(\mathbf{u}, l)$ defines a piecewise configuration as well. This follows from the fact that piecewise languages are closed under concatenation and shuffle (see Proposition 3).

The explained procedure can be extended to an arbitrary inverted tree. The correctness follows by induction on the number of channels. The final algorithm MULTILIMIT is shown in Figure 4.8. The algorithm assumes that shadow channels are introduced where they are needed. It traverses the channels according to the partial order induced by the CG, applying read and write phases. The read phase is the same as in the star and tree topologies (done by DOREAD). The write phase uses MERGES to merge the content of all the shadows of a channel before applying a read phase to it.

Theorem 9 *Let* $\mathbf{u}$ *be a piecewise configuration,* Act *an action set with inverted tree topology, and* $\mathbf{U}$ *the set of configurations returned by* MULTILIMIT$(\mathbf{u})$. *Then,* $\mathcal{L}(\mathbf{U}) = (Act^* : \mathcal{L}(\mathbf{u}))$.

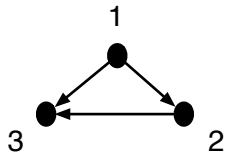

Figure 4.9: An example of a DAG communication topology.

4.3.4 DAG Topology

In this section, we present an algorithm for computing the limit language of a set of actions whose CG is an arbitrary directed acyclic graph (DAG) (e.g. see Figure 4.4(d) and Figure 4.9). This subsumes the algorithms from the previous sections for star, tree, and inverted tree topologies.

What makes the DAG topology different from the inverted tree is that the immediate predecessors (in the $\preceq$ partial order on the CG) of a channel may be interdependent. For example, consider $Act = \{1?a \rightarrow 3!a, 1?b \rightarrow 2!b, 2?b \rightarrow 3!b\}$ whose CG is shown in Figure 4.9. Channel 3 has channels 1 and 2 as its immediate predecessors, and channel 2 depends on channel 1. This extra layer of dependence precludes the possibility of breaking the topology by simply introducing shadow channels.

For our running example, consider the computation of reachable configurations starting from $\langle a^*b^*, \epsilon, \epsilon\rangle$. We can replace channel 3 with two shadow channels to obtain $\widehat{Act} = \{1?a \rightarrow \hat{3}_1!a, 1?b \rightarrow 2!b, 2?b \rightarrow \hat{3}_2!b\}$. By applying TREELIMIT to the resulting tree topology, we obtain two piecewise configurations $\{\langle a^*b^*, \epsilon, \epsilon, a^*, \epsilon\rangle, \langle b^*, b^*, \epsilon, a^*, b^*\rangle\}$. If we then proceed by merging the contents of the shadows of channel 3, as in the inverted tree topology, we obtain $\{\langle a^*b^*, \epsilon, a^*\rangle, \langle b^*, b^*, (a+b)^*\rangle\}$. The second piecewise configuration, i.e. $\langle b^*, b^*, (a+b)^*\rangle$, includes configurations in which the content of channel 3 is in b^+a^+. These configurations are infeasible since a came before b in channel 1 in any initial configuration and this order must be preserved when the content is copied to channel 3.

To solve this problem, we extend MULTILIMIT algorithm by modifying the shuffle used by MERGES (see Section 4.3.3) to respect the dependencies between the predeces-

sors of the channel whose shadows are merged. This requires (i) keeping track of the relative positions of each letter in a channel as it is copied between different channels, and (ii) restricting the shuffle based on the history of positions of each letter.

For a system with k channels, each letter is associated with a k-tuple of indices from IDX^k, where $\text{IDX} \triangleq [-1..\infty)$. Intuitively, the jth index of a letter a indicates the relative position of a when it was in channel j, with -1 meaning that a was never in that channel. Let $\text{idx}(i, a)$ be a function that extracts the ith index of a. For example, $\text{idx}(2, a) = 4$ means that a was at some point at position 4 in channel 2, and $\text{idx}(3, a) = -1$ means that a was never in channel 3. We use $\text{ch}(a)$ to denote the latest channel that a was in. Formally, $\text{ch}(a) \triangleq \max\{i \mid \text{idx}(i, a) \neq -1\}$.

To keep track of the indices, several parts of the MULTILIMIT are modified. The DOREAD algorithm (see Figure 4.5) is extended to accept as an argument the `ch`-index of a letter at the head of the current channel `ch`, and increment it at each recursive call (lines 13 and 19). SATURATE and STEP (see Figure 4.6) are extended to propagate and assign indices as well (lines 4 and 11).

The interdependence of the channels implies the following constraint on the content of every channel in every reachable configuration. Let w be a word describing the content of channel l. Let a and b be letters at positions p and q in w, respectively. Assume that i is the last channel a was in, and that i precedes the last channel that b was in, i.e., $i = ch(a) < ch(b)$. Furthermore, assume that a preceded b in channel i, i.e., $\text{idx}(i, a) < \text{idx}(i, b)$. Then a has to precede b in w, i.e., $p < q$, since a had to be read from channel i (and placed in w) before b could be read.

We denote the set of all words that satisfy the above condition by WO. Formally, it is the set of all words w in $(\Sigma \times \text{IDX}^k)^*$ that satisfy

$$\forall p, q,\ (a = w(p) \wedge b = w(q) \wedge i = \text{ch}(a) \wedge \\ \text{ch}(a) < \text{ch}(b) \wedge \text{idx}(i, a) < \text{idx}(i, b)) \Rightarrow p < q\,,$$

where $w(p)$ denotes the letter at position p of w.

For our running example, the word ba in channel 3 does not belong to WO: the last channel of a is 1 ($ch(a) = 1$) which precedes 2 – the last channel of b ($ch(b) = 2$). Thus, $ch(a) < ch(b)$. In the sequel, a preceded b in channel 1, i.e. $\text{idx}(1, a) < \text{idx}(1, b)$. Therefore, a must also precede b in channel 3.

The set WO defines a piecewise language, and is recognizable by a PO-FSA.

Theorem 10 *Let WO be an automaton recognizing all words that respect the dependencies between predecessors of channels whose shadows are merged. The language WO is piecewise.*

Proof: To prove the theorem, we construct an automaton A_{WO} that recognizes WO. Then, we show that this automaton is a PO-FSA.

Let k be the number of channels. The state space of A_{WO} is $\text{IDX}[1..k][1..k]$. An interpretation of a state is

$$q[i][j] = p\,,$$

if the automaton has seen a letter that its latest channel was i, and it was at some point at position p in channel j. The initial state q^0 is such that $\forall i, j,\ q^0[i][j] = -1$, and every state is accepting. The transition relation of A_{WO}, δ, is deterministic, and is defined as follows:

$$\begin{aligned}\delta(q, a, q') \Leftrightarrow (\forall \text{ch}(a) < i \leq k,\ q[i][\text{ch}(a)] \leq \text{idx}(\text{ch}(a), a)) \wedge \\ (\forall i, j,\ (i \neq \text{ch}(a) \Rightarrow q'[i][j] = q[i][j]) \wedge \\ (i = \text{ch}(a) \Rightarrow q'[i][j] = \max(q[i][j], \text{idx}(j, a))))\end{aligned}$$

The first conjunct of δ ensures that the WO condition is satisfied, and the second updates the state. Let I denote the latest channel letter a was in. Intuitively, the automaton accepts letter a if and only if it has not seen a letter from any channel greater than I which was behind of letter a in I. The state of the automaton is updated if in every channel letter a has a greater position than any other letter that the automaton has seen in that channel.

The automaton A_{WO} is a PO-FSA, where the partial order $\preceq$ on states is

$$q \preceq q' \Leftrightarrow \forall i, j,\ q[i][j] \leq q'[i][j]\,.$$

□

In order to restrict MERGES to only include words that satisfy WO, we replace it with a function MERGEDAGS defined as follows. Let $\hat{\mathbf{t}}$ be a configuration reachable from an initial configuration extended with shadow channels, and l a non-shadow channel. Then, $\text{MERGEDAGS}(\hat{\mathbf{t}}, l) \triangleq \text{MERGES}(\hat{\mathbf{t}}, l) \cap \text{WO}$. Since WO is piecewise (by Theorem 10) and piecewise languages are closed under intersection (by Proposition 3), MERGEDAGS defines a piecewise configuration.

With this change, MULTILIMIT algorithm computes the exact set of reachable configurations.

Theorem 11 *Let* $\mathbf{u}$ *be a piecewise configuration,* Act *a set of actions with DAG topology, and* $\mathbf{U}'$ *a set of configurations returned by* $\text{MULTILIMIT}(\mathbf{u})$ *algorithm, where* MERGES *is replaced by* MERGEDAGS. *Then* $\mathcal{L}(\mathbf{U}') = (Act^* : \mathcal{L}(\mathbf{u}))$.

4.4 Abridged Piecewise FIFO Systems

In this section, we show that important subclasses of multi-channel piecewise FIFO systems can be described by a finite-state, abridged structure representing an expressive abstraction of the system. We give a definition for the abridged model that contains a procedural definition of its transition relation. This procedural definition describes how to build an abridged structure in an automated way from the description of a piecewise FIFO system.

Recall that a piecewise FIFO system is composed of a set of partially-ordered automata that communicate over unbounded, perfect, FIFO channels. Throughout this section, we refer to a piecewise FIFO system as a Distributed State Machine (DSM) and its abridged model as an Abridged Distributed State Machine ($ADSM$).

4.4.1 Construction of ADSM

An abridged model of a piecewise FIFO system is closely related to its concrete model. The set of its control locations is the same as the set of control locations of the concrete model. However, the channel contents in the concrete model are replaced by simply piecewise expressions in the abridged model. Given a DSM, its abridged model $ADSM$ is defined as follows.

Definition 14 (ADSM) *For a given DSM, $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$, its abridged model is defined as $\mathcal{S}_A = (Q, T, q^0, \eta, \Phi)$, where Q is a finite set of control locations; T is a set of simply piecewise configurations over Σ; q^0 is the initial control location; $\eta \subseteq (Q \times T) \times (Q \times T)$ is the transition relation, which is given below; and Φ is a fairness constraint on the transitions.*

The fairness constraint requires that a transition that only reads from a channel should not be allowed to read an empty channel: if $i?a$ is enabled infinitely often then $i!a$ must also be enabled infinitely often. A *global state* of an $ADSM$ is a pair $(q, \mathbf{v})$ where q is a control location in Q and $\mathbf{v}$ is a simply piecewise configuration.

Below, we present a motivating example for the definition of the transition relation of $ADSM$, η.

Motivating Example Let $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$ be a DSM with 3 channels, $C = \{1, 2, 3\}$, and a single control location $Q = \{q\}$. Let $\delta = \{(q, 1?a \rightarrow 2!a, q), (q, 2?a \rightarrow 3!a, q), (q, 2?b \rightarrow 3!b, q)\}$ be a set of transition rules and $q^0 = (q, \mathbf{w})$ denote the initial global state where $\mathbf{w} \in \langle aa^*, bb^*, \epsilon \rangle$. Below, we explain how to construct an $ADSM$ corresponding to $\mathcal{S}$.

The initial state of the $ADSM$ is $(q, \langle aa^*, bb^*, \epsilon \rangle)$. In this state, two actions are enabled: $1?a \rightarrow 2!a$ and $2?b \rightarrow 3!b$. Therefore, either letter a is read from channel 1 and written on channel 2, or, letter b is read from channel 2 and written on channel 3. This results in two reachable global states $(q, \langle a^*, bb^*a, \epsilon \rangle)$ and $(q, \langle aa^*, b^*, b \rangle)$ (see Figure 4.10). In the figure we show only the channel configurations since the control location stays the same in all the reachable global states. Symbol ϵ is also shown by '–'.

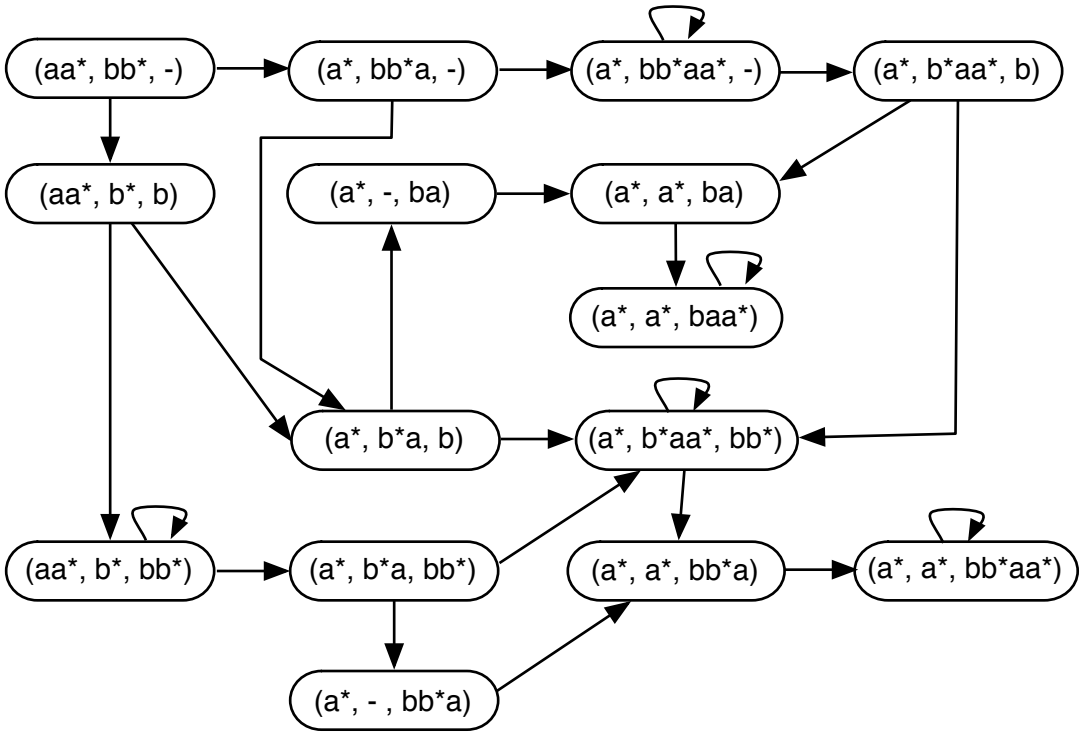

Figure 4.10: An example showing a representation of $ADSM$ global states constructed from a DSM with a single control location, 3 channels, and a set of transition rules $\delta = \{(q, 1?a \rightarrow 2!a, q), (q, 2?a \rightarrow 3!a, q), (q, 2?b \rightarrow 3!b, q)\}$.

The global state $(q, \langle a^*, bb^*a, \epsilon\rangle)$ represents a set of channel configurations in which there exist zero or more number of letter a on channel 1 and at least one letter b followed by a letter a on channel 2. Two actions are enabled on this state: $1?a \rightarrow 2!a$ and $2?b \rightarrow 3!b$. If there are more than one letter a on channel 1, then action $1?a \rightarrow 2!a$ will be enabled multiple times on this configuration. Thus, we group multiple applications of action $1?a \rightarrow 2!a$ to configuration $\langle a^*, bb^*a, \epsilon\rangle$. This results in the reachable global state $(q, \langle a^*, bb^*aa^*, \epsilon\rangle)$. Furthermore, applying action $1?b \rightarrow 3!b$ to $\langle a^*, bb^*a, \epsilon\rangle$ results in the reachable global state $(q, \langle a^*, b^*a, b\rangle)$.

The global state $(q, \langle aa^*, b^*, b\rangle)$ represents a set of channel configurations in which there exist at least one letter a on channel 1, zero or more letter b on channel 2, and a letter b on channel 3. Two actions are enabled on this state: $1?a \rightarrow 2!a$ and $2?b \rightarrow 3!b$.

By grouping multiple applications of action $2?b \rightarrow 3!b$ on channel 2, we compute the reachable global state $(q, \langle aa^*, b^*, bb^* \rangle)$. Furthermore, applying action $1?a \rightarrow 2!a$ results in the reachable global state $(q, \langle a^*, b^*a, b \rangle)$. Note that applying action $2?a \rightarrow 3!a$ to the global state $(q, \langle a^*, b^*a, b \rangle)$ results in the reachable global state $(q, \langle a^*, \epsilon, ba \rangle)$ since the global state $(q, \langle a^*, b^*a, b \rangle)$ includes configurations in which there exists only one letter a on channel 2. The rest of the reachability graph is constructed similarly as shown in Figure 4.10. $\square$

We introduce the following notation. For Z and Y arbitrary simply piecewise expressions and $M, N \subseteq \Sigma$, we define

$$Z \bullet M^* \triangleq \begin{cases} Z & \text{if } Z = Y \cdot N^* \text{ and } M \subseteq N \\ Y \cdot M^* & \text{if } Z = Y \cdot N^* \text{ and } N \subset M \\ Z \cdot M^* & \text{otherwise.} \end{cases}$$

In the definition of the transition relation of $ADSM$ we assume that the contents of the channels that are not explicitly mentioned stay the same during the transition. Without loss of generality, we assume that all actions are conditionals.

Let $M \subseteq \Sigma$, $a \in M$, and Z be an arbitrary simply piecewise expression. The transition relation of $ADSM$, η, is defined as follows: $(q, \mathbf{v}) \rightarrow (q', \mathbf{v}') \in \eta$ if and only if

- $q' \neq q$ and there exists a channel i such that
  - $\mathbf{v}[i] = a \cdot Z$ and there exists an action x in δ such that $x : q \overset{i?a \rightarrow j!b}{\longrightarrow} q'$, then $\mathbf{v}'[i] = Z$ and $\mathbf{v}'[j] = \mathbf{v}[j] \cdot b$, or
  - $\mathbf{v}[i] = M^* \cdot Z$ and
    - there exists an action x in δ such that $x : q \overset{i?a \rightarrow j!b}{\longrightarrow} q'$, then $\mathbf{v}'[i] = M^* \cdot Z$ and $\mathbf{v}'[j] = \mathbf{v}[j] \cdot b$, or
    - $(q, \mathbf{v}[i \mapsto Z]) \rightarrow (q', \mathbf{v}') \in \eta$

or

- $q' = q$ and there exists a channel i such that
  - $\mathbf{v}[i] = a \cdot Z$ and there exists an action x in δ such that $x : q \overset{i?a \to j!b}{\longrightarrow} q$, then $\mathbf{v}'[i] = Z$ and $\mathbf{v}'[j] = \mathbf{v}[j] \cdot b$, or
  - $\mathbf{v}[i] = M^* \cdot Z$ and
    - if

$$W_k = \{b \in \Sigma \mid \exists \text{ channel } j,\ \exists\, N \subseteq \Sigma,\ \exists\, a \in N,\ \exists\, Z,\ \mathbf{v}[j] = N^* \cdot Z \text{ and } q \overset{j?a \to k!b}{\longrightarrow} q \in \delta\}$$

      then, for all channel k, $\mathbf{v}'[k] = \mathbf{v}[k] \bullet W_k^*$, or

    - $(q, \mathbf{v}[i \mapsto Z]) \to (q', \mathbf{v}') \in \eta$.

We illustrate the construction of an $ADSM$ based on its transition relation definition through the following example.

ADSM Transition Relation Example Let $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$ be a DSM with 4 channels, $C = \{1, 2, 3, 4\}$, and two control locations $Q = \{q, q'\}$. Let

$$\begin{aligned} \delta \;=\; \{&(q, 1?a \to 2!a, q'), (q, 2?c \to 4!c, q'),\\ &(q', 1?a \to 2!a, q'), (q', 1?b \to 3!h, q'),\\ &(q', 1?b \to 3!d, q'), (q', 1?b \to 4!e, q'),\\ &(q', 2?c \to 4!f, q')\}, \end{aligned}$$

be a set of transition rules and $q^0 = (q, \mathbf{w})$ denote the initial global state where $\mathbf{w} \in \langle ab^*, c^*, \epsilon, \epsilon \rangle$. The initial state of the $ADSM$ corresponding to $\mathcal{S}$ is $(q, \langle ab^*, c^*, \epsilon, \epsilon \rangle)$. Two actions are enabled on this state: $1?a \to 2!a$ and $2?c \to 4!c$. According to δ, both of these actions cause a change in the control location from q to q'. Based on the first case in the definition of η (where $q' \neq q$), we compute two reachable global states: $(q', \langle b^*, c^*a, \epsilon, \epsilon \rangle)$ and $(q', \langle ab^*, c^*, \epsilon, c \rangle)$ (see Figure 4.11). In the global state

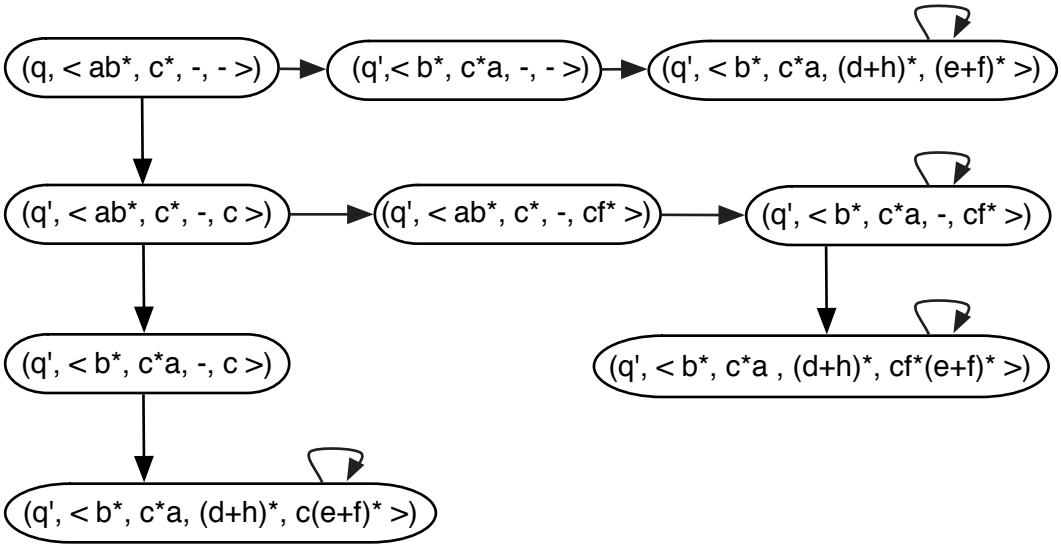

Figure 4.11: An example showing a representation of $ADSM$ global states constructed from a DSM with control locations $\{q, q'\}$, 4 channels, and a set of transition rules $\delta = \{(q, 1?a \rightarrow 2!a, q'), (q, 2?c \rightarrow 4!c, q'), (q', 1?a \rightarrow 2!a, q'), (q', 1?b \rightarrow 3!h, q'), (q', 1?b \rightarrow 3!d, q'), (q', 1?b \rightarrow 4!e, q'), (q', 2?c \rightarrow 4!f, q')\}$.

$(q', \langle b^*, c^*a, \epsilon, \epsilon\rangle)$, the following actions are enabled:

$$1?a \rightarrow 3!h, \quad 1?b \rightarrow 3!d, \quad 1?b \rightarrow 4!e, \quad 2?c \rightarrow 4!f$$

Note that none of these actions cause a change in the control location q'. Therefore, we can group these actions according to the second case in the definition of η (where $q' = q$) and compute the reachable global state $(q', \langle b^*, c^*a, (h + d)^*, (e + f)^*\rangle)$. The rest of the $ADSM$ reachability graph is constructed similarly as shown in Figure 4.11. □

The following lemma shows that the next-state-relation of the $ADSM$ is finite.

Lemma 4 *If $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$ and $\mathcal{S}_A = (Q, T, q^0, \eta, \Phi)$ is the abridged model of $\mathcal{S}$, given $(q, \mathbf{v}) \in Q \times T$, the set of $(q', \mathbf{v}') \in Q \times T$ such that $(q, \mathbf{v}) \rightarrow (q', \mathbf{v}') \in \eta$ is finite.*

Proof: The proof follows based upon the finiteness of the set of control locations Q, finiteness of Σ, and the definition of the transition relation η. □

Let $R = M_0^* a_0 M_1^* \cdots a_{n-1} M_n^*$ be a simply piecewise expression. We define the length of R, denoted by $|R|$, to be equal to the anchor length of R plus the number of the blocks with Kleen star closure. For example, if $R_1 = a(b+c)^* bd(b+a)^*$, then $|R| = 5$.

Given a DSM, its abridged model is constructed recursively: for each global state such as $(q, \mathbf{v}) \in Q \times T$ we calculate the set $\{(q', \mathbf{v}') \mid (q, \mathbf{v}) \rightarrow (q', \mathbf{v}') \in \eta\}$. Let q_i be a control location in Q and Act_i denote the set of self-loop actions on q_i in the DSM description. The following theorem shows that the procedure of constructing the $ADSM$ terminates if for all $q_i \in Q$, the communication graph of Act_i is acyclic.

Theorem 12 *Let $DSM = (\Sigma, C, Q, q^0, \delta)$, $q_i \in Q$, and $Act_i = \{act_i\}$ denote the set of actions where $q_i \xrightarrow{act_i} q_i \in \delta$. If for all $q_i \in Q$, $CG(Act_i)$ is acyclic, then the procedure of constructing the abridged model $ADSM = (Q, T, q^0, \eta, \Phi)$ terminates.*

Proof: In order to prove that the procedure of constructing $ADSM$ terminates, first, we show that the set of global states of $ADSM$ with the same control location is finite.

Consider the control location $q_i \in Q$ and the set of actions $Act_i = \{act_i\}$ where $q_i \xrightarrow{act_i} q_i \in \delta$. Since $CG(Act_i)$ is acyclic, there exists at least one node in the graph that does not have any incoming edge. In other words, there is at least one channel in $CG(Act_i)$ that is being read only and nothing is written on it. Let r denote one of these channels and l_r denote the length of the simply piecewise expression that represents its initial content. According to the transition relation η, in each iteration of the construction procedure, l_r either decreases or stays the same.

Consider the children of r in the $CG(Act_i)$. Let s denote one of these children and l_s denote the length of the simply piecewise expression that represents its content. Without loss of generality, assume that s has only one immediate predecessor that is r. In each iteration, l_s may decrease, due to the reads from s, may increase due to the writes resulting from reading from r, or it may stay the same. Therefore, according to the transition relation η, l_s is bounded by its initial value plus the initial value of l_r.

We can show by induction that for each channel in $CG(Act_i)$ the length of the simply piecewise expression that represents its content is bounded by the the length of its

initial content plus the length of the expressions that represent the initial contents of its ancestors. Thus, due to the finiteness of Σ and the definition of the transition relation η, the set of the global states of $ADSM$ with the same control location is finite.

Since the set of control locations of DSM comes with a partial-order that is also respected by the transition relation of $ADSM$, after finite number of iterations the construction procedure of $ADSM$ terminates. □

In the next section, we discuss the relationship between the computations of a DSM and its abridged model.

4.4.2 Automated Analysis

The main reason for constructing an abridged model of a DSM is to be able to reason about its infinite computations by analyzing the computations of the finite $ADSM$. In this section, we only consider piecewise FIFO systems with acyclic communication graphs; for which the procedure of computing the abridged model terminates (refer to Theorem 12). A computation path in a DSM is a finite or infinite sequence denoted $\psi = (q_0, \mathbf{w}_0) \rightarrow (q_1, \mathbf{w}_1) \rightarrow \ldots$, where $q_0 = q^0$ and for all i, $(q_i, \mathbf{w}_i) \rightarrow (q_{i+1}, \mathbf{w}_{i+1}) \in \Delta$. All the computation paths in DSM are accepting. Then, $\mathcal{L}^*(DSM)$ is a subset of $(Q \times \Sigma^*)^*$ and consists of all the finite computations in DSM and $\mathcal{L}^\omega(DSM)$ is a subset of $(Q \times \Sigma^*)^\omega$ and consists of all the infinite computations in DSM. The language of a DSM, denoted by $\mathcal{L}(DSM)$, is equal to $\mathcal{L}^*(DSM) \cup \mathcal{L}^\omega(DSM)$.

Similarly, a computation path in an $ADSM$ is a finite or infinite sequence denoted $\xi = (q_0, \mathbf{v}_0) \rightarrow (q_1, \mathbf{v}_1) \rightarrow \ldots$, where $q_0 = q^0$ and for all i, $(q_i, \mathbf{v}_i) \rightarrow (q_{i+1}, \mathbf{v}_{i+1}) \in \eta$. The $\mathcal{L}^*(ADSM)$ is a subset of $(Q \times T)^*$ and consists of all the finite computations in $ADSM$ and $\mathcal{L}^\omega(ADSM)$ is a subset of $(Q \times T)^\omega$ and consists of all the infinite and fair computations in $ADSM$. The language of an $ADSM$, denoted by $\mathcal{L}(ADSM)$, is equal to $\mathcal{L}^*(ADSM) \cup \mathcal{L}^\omega(ADSM)$.

Definition 15 *Let $\psi = (q_0, w_0) \rightarrow (q_1, w_1) \rightarrow (q_2, w_2) \rightarrow \ldots$, be a computation path in a DSM, and $\xi = (q_0, \mathbf{v}_0) \rightarrow (q_1, \mathbf{v}_1) \rightarrow (q_2, \mathbf{v}_2) \ldots$, be a computation path in its*

abridged model $ADSM$, where $q_0 = q^0$. ξ and ψ are two corresponding *computations if $\psi|_q$ and $\xi|_q$ are stuttering equivalent where $\psi|_q$ and $\xi|_q$ are projections of ψ and ξ on their control locations, respectively.*

Th following theorem shows the relationship between the computations of a DSM and its abridged model.

Theorem 13 *For every computation in a DSM, there exists a corresponding computation in its abridged model $ADSM$, and for every computation in an $ADSM$, there exists a set of corresponding computations in the concrete model DSM.*

As explained, the set of control locations of a DSM and its abridged model is the same. According to the construction procedure of an $ADSM$, all the appropriate channel contents in a DSM are represented by a set of simply piecewise expressions in its abridged model $ADSM$. The following lemma shows the relationship between the set of reachable global states of a DSM and its abridged model.

Lemma 5 *For every reachable state in a DSM, $(q, \mathbf{w})$, there exists a reachable state in its abridged model $ADSM$, $(q, \mathbf{v})$, where $\mathbf{w} \in \mathbf{v}$ and for every reachable state in an $ADSM$, $(q, \mathbf{v})$, there exists a reachable state in the concrete model DSM, $(q, \mathbf{w})$, where $\mathbf{w} \in \mathbf{v}$.*

Thus, we can perform reachability analysis on a DSM by an exhaustive search on the the state space of its abridged model $ADSM$.

The computations of an $ADSM$ satisfy property P if and only if there is no computation x of $ADSM$ such that $ADSM, x \models \neg P$.

We use a standard automata-theoretic technique to decide this problem [104]. This technique consists of creating an automaton, $\mathcal{B}_{\neg P}$, on infinite strings [27], which accepts only those strings that satisfy the property $\neg P$. We combine the $ADSM$ with $\mathcal{B}_{\neg P}$ to form the product automaton $ADSM \times \mathcal{B}_{\neg P}$. This is an automaton on infinite strings whose language is empty if and only if the computations of the $ADSM$ satisfy the property P.

The theory of finite automata on infinite strings was established by Büchi [27], Muller [91], and Rabin [97]. Given a finite alphabet Σ, Σ^ω denotes the set of ω-*words*, i.e. each word $\alpha \in \Sigma^\omega$ has length ω. An ω-word over Σ is written in the form of $\alpha = \alpha(0)\alpha(1)\ldots$. Thus, for an ω-word α, the 'infinity set' of α is

$$\text{Inf}(\alpha) \quad \triangleq \quad \{a \in \Sigma \mid \forall i,\, \exists j > i,\ \alpha(j) = a\}.$$

In other words, Inf(α) is the finite set of letters occurring infinitely often in α.

In the classical formal language theory, an input (a finite word) is accepted by an automaton if a run exists on the input that terminates in an accepting state. However, this notion of acceptance can not be used when the input is an ω-word. We can adapt the usual definitions of deterministic and nondeterministic automata to the case of automata that accepts ω-words by the introduction of new acceptance conditions.

Definition 16 (ω-automaton) *An ω-automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q^0, Acc)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition relation, $q^0 \in Q$ is the initial state, and Acc is the acceptance component.*

A run of $\mathcal{A}$ on an ω-word $\alpha(0)\alpha(1)\ldots$ from Σ^ω is a sequence $\sigma = \sigma(0)\sigma(1)\ldots$ such that $\sigma(0) = q^0$ and $(\sigma(i), \alpha(i), \sigma(i+1)) \in \delta$ for $i \geq 0$.

The Büchi acceptance condition is defined for nondeterministic ω-automata.

Definition 17 (Büchi Automaton) *An ω-automaton $\mathcal{B} = (Q, \Sigma, \delta, q^0, F)$ with acceptance component $F \subseteq Q$ is called* Büchi automaton *if it has the following acceptance condition: a word $\alpha \in \Sigma^\omega$ is accepted by $\mathcal{B}$ if and only if there exists a run σ of $\mathcal{B}$ on α satisfying the condition:*

$$\text{Inf}(\sigma) \cap F \quad \neq \quad \emptyset\,,$$

i.e. at least one of the states in F has to be visited infinitely often during the run. $\mathcal{L}(\mathcal{B}) = \{\alpha \in \Sigma^\omega \mid \mathcal{B}$ accepts $\alpha\}$ is the ω-language recognized by $\mathcal{B}$.

We consider system properties expressed by a restricted class of Büchi automata. Here, a Büchi automaton for a DSM has a fixed set of possible states, at most one for

each control location in the DSM. Since the set of computations in an $ADSM$ is a superset of the set of computations in DSM, we require that the language of each Büchi property automaton be stuttering-closed.

Lemma 6 *If $\mathcal{B}$ is a stuttering-closed Büchi property automaton for a DSM, for every computation in $\mathcal{L}^{\omega}(ADSM)$ and the language of the property automaton, $\mathcal{L}(\mathcal{B})$, there exists a corresponding computation in $\mathcal{L}^{\omega}(DSM)$ and $\mathcal{L}(\mathcal{B})$ and for every computation in $\mathcal{L}^{\omega}(DSM)$ and $\mathcal{L}(\mathcal{B})$ there exists a corresponding computation in $\mathcal{L}^{\omega}(ADSM)$ and $\mathcal{L}(\mathcal{B})$.*

4.5 Conclusion

We studied the reachability problem for piecewise FIFO systems. As it was shown in Chapter 3, this problem is reducible to computing the limit language of a regular language of actions. In this chapter, we concentrated on computing the limit language in piecewise FIFO systems.

We consider single-channel and multi-channel FIFO systems separately. For the single-channel case, we presented a new automata-theoretic algorithm for calculating the limit language starting with an arbitrary regular initial content. We showed that the worst case complexity of our algorithm is exponential in the size of the automaton representing the initial channel content. A prototype of the algorithm was implemented using the Automaton package [90].

For multi-channel systems, we presented an automata-theoretic algorithm for computing the limit language subject to the following conditions: (i) the initial language is piecewise, and (ii) the communication graph of actions is acyclic. For star and tree topologies we showed that the complexity of our algorithm is exponential in the size of the automaton representing the initial channel configuration. In the cases of inverted tree and DAG topologies the complexity of the algorithms remains an open problem.

We note that, in general, limit languages do not represent system computations but rather the reachable state set. In order to reason about the computations of piecewise

FIFO systems, we presented a procedure that given a piecewise FIFO systems, it constructs an abridged structure, representing an expressive abstraction of the system. We showed that the construction procedure of the abridged model terminates in piecewise FIFO systems with acyclic communication graphs. Furthermore, we showed that we can analyze the infinite computations of the more concrete model by analyzing the computations of the finite, abridged model.

Chapter 5

Reachability in Parameterized Systems

In this chapter, we describe parameterized systems and the reachability problem for them. We focus on the analysis of safety properties of parameterized systems, where each of the individual processes may be infinite-state. We present a framework that combines Abstract Interpretation style reasoning with a backward-reachability algorithm. Our key contribution is a new abstract domain for parameterized systems of infinite-state processes. Our abstract domain combines numeric domains with information about the state of multiple processes in a parameterized context. It is generic enough to be instantiated by different well-known numeric abstract domains such as octagons [89] and polyhedra [48].

5.1 Introduction

A parameterized system is a family of systems in which n processes execute the same program concurrently. The problem of parameterized verification is to prove whether for all values of n, the system with n processes is correct. Such systems arise naturally in many important applications ranging from communication protocols such as mutual-exclusion and leader election, to distributed systems such as web services, to cache coherence, resource sharing, transactional memory protocols, and others.

Parameterized system verification is undecidable in general. Apt and Kozen [7] showed that verification of parameterized systems of finite-state processes is undecidable. This negative result has naturally directed the research in parameterized systems analysis towards two directions: (i) studying decidability of restricted subclasses (e.g. [62, 55, 51]), and (ii) developing generally applicable but semi-automated proof principles that utilize induction and symmetry reduction (e.g. [38, 37, 98, 92]). In all of the cases above, it is assumed that each process is *finite-state*.

In this thesis, we focus on the analysis of parameterized systems of *infinite-state* processes. This setting is common in practice, for example, even in Lamport's bakery protocol [81], each process maintains an integer ticket, and therefore, has an infinite state space. We are interested in a sound, automated, and terminating procedure for verifying safety properties of such systems. Since this problem is undecidable, such a procedure is necessarily incomplete.

Incomplete but terminating algorithms are often used for reasoning about single-process infinite-state programs. Such algorithms are typically developed in the framework of Abstract Interpretation [46] (AI). In this thesis, we apply such a technique to parameterized systems of infinite-state processes. We present a framework that combines AI-style reasoning with a backward-reachability algorithm. Our key contribution is an abstract domain in which each element (a) represents the lower bound on the number of processes at a particular control location and (b) employs a numeric abstract domain to capture arithmetic relations among variables of the processes. Our abstract domain is sufficiently generic to be instantiated by different well-known numeric abstract domains such as octagons [89] and polyhedra [48].

We develop an algorithm to over-approximate backward-reachability in a parameterized system using our abstract domain. In its initial form, the algorithm is sound but it is not guaranteed to terminate. We show that there are two reasons for its divergence: one comes from the fact that the numeric domain is infinite, and the other is due to the existence of unbounded numbers of processes in a parameterized system. We show that it is possible to enforce sound termination of the algorithm by combining numeric widening

with a new approximation operator developed especially for our purpose. This results in an algorithm that is incomplete but is sound and terminating. That is, if the algorithm does not find an error state, then the system is correct. However, if the algorithm finds an error state, it is uncertain that the error actually is present in the system and is not an error introduced by the over-approximation. We illustrate an implementation of our algorithm on a variant of Lamport's bakery mutual-exclusion protocol (Algorithm 2 in [88]).

The rest of this chapter is organized as follows. Syntax and semantics of parameterized systems are defined in Section 5.2. The abstract domain for parameterized systems is introduced in Section 5.3, and is followed by the backward-reachability algorithm in Section 5.4. We discuss techniques to ensure termination of our algorithm and illustrate our algorithm on Lamport's bakery protocol in Section 5.5. We give an overview of related work in Section 5.6, and conclude with a summary of our contributions in this chapter in Section 5.7.

5.2 Parameterized Systems

We describe the system model used in the rest of this chapter.

Syntax A parameterized system $\mathcal{P}$ is a triple (Q, V, T), where Q is a finite set of control locations, V is a finite set of variables, and T is a finite set of guarded commands (or rules). Each $\tau \in T$ is of the form

$$\tau : q \xrightarrow{g} q' , \qquad \text{(guarded command)}$$

where $q, q' \in Q$, and g is a guard. We allow for three types of guards: local, universal global, and existential global that are defined below.

We write V' for the set $\{x' \mid x \in V\}$, and $\texttt{self}.V$ and $\texttt{other}.V$ for the set $\{\texttt{self}.x \mid x \in V\}$ and $\{\texttt{other}.x \mid x \in V\}$, respectively. A *local guard* is an expression on $\texttt{self}.(V \cup V')$ constraining current and next local states of a single process. The *universal*

and *existential global guards* can be written as

$$\forall \texttt{other} \neq \texttt{self},\ (\texttt{other}.\mathbf{pc} = q_o) \wedge \theta \quad \text{and} \quad \exists \texttt{other} \neq \texttt{self},\ (\texttt{other}.\mathbf{pc} = q_o) \wedge \theta\,,$$

respectively, where q_o is a control location in Q, $\texttt{other}.\mathbf{pc}$ is a special variable, and θ is an expression over $\texttt{self}.(V \cup V') \cup \texttt{other}.V$ variables. Intuitively, commands with local guards express how one process behaves independently of other processes in the system, commands with global guards allow a process to reference variables and control locations of the other processes in either universal or existential form. These three types of guarded commands are sufficient to express a wide variety of parameterized systems [3].

An example of a parameterized system where each process manipulates integer variables is shown in Figure 5.1. It consists of three commands: τ_1 with a local guard g_1, τ_2 with a universal guard g_2, and τ_3 with an existential guard g_3. Informally, a process executing τ_1 changes its control location from q_1 to q_2, increments local variable x, and does not change local variable y. Similarly, a process executing τ_2 goes from q_1 to q_3 but only if all other currently executing processes are in q_3 and the value of their copies of the variable x are positive. Furthermore, execution of τ_2 decrements the y variable of the current process by 2. Finally, a process executing τ_3 changes its control location from q_3 to q_1 but only if there exists another process that is at q_3 and whose variable x is greater than 1 and the difference between its y and x variables is greater than 2. During this transition, the value of the variables x and y of the executing process do not change.

We formalize the semantics of parameterized systems using transition systems.

Semantics A *process state* is a pair (q, v), where $q \in Q$ and v is a valuation assigning values to variables in V. We often treat a process state $u = (q, v)$ as a valuation of variables $V \cup \{\mathbf{pc}\}$ such that $u(\mathbf{pc}) = q$, and $u(y) = v(y)$ for all $y \in V$. An *n-process configuration* is a tuple $\langle u_1, \ldots, u_n \rangle$, where each u_i is a process state. We refer to the first (left-most) process in a configuration as P_1, to the second as P_2, etc, and refer to the number of the process as a process id (PID). So PID of P_1 is 1, PID of P_2 is 2, etc. For two configurations $c_1 = \langle u_1, \ldots, u_n \rangle$ and $c_2 = \langle w_1, \ldots, w_m \rangle$, we use $c_1 \cdot c_2$ to denote

$$\tau_1 : q_1 \xrightarrow{g_1} q_2,\ g_1 : (\texttt{self}.x' = \texttt{self}.x + 1) \wedge (\texttt{self}.y' = \texttt{self}.y)$$

$$\tau_2 : q_1 \xrightarrow{g_2} q_3,\ g_2 : \forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} = q_3) \wedge (\texttt{self}.x' = \texttt{self}.x) \wedge (\texttt{self}.y' = \texttt{self}.y - 2) \wedge (\texttt{other}.x > 0)$$

$$\tau_3 : q_3 \xrightarrow{g_3} q_1,\ g_3 : \exists \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} = q_3) \wedge (\texttt{self}.x' = \texttt{self}.x) \wedge (\texttt{self}.y' = \texttt{self}.y) \wedge (\texttt{other}.y - \texttt{other}.x > 2) \wedge (\texttt{other}.x > 1)$$

Figure 5.1: An example of a parameterized system, $\mathcal{P}_1$, with three control locations $\{q_1, q_2, q_3\}$, two integer variables $\{x, y\}$, and three guarded commands $\{\tau_1, \tau_2, \tau_3\}$.

their concatenation $\langle u_1, \ldots, u_n, w_1, \ldots, w_m \rangle$.

For an expression θ, we write $\theta[x \leftarrow y]$ for the result of substituting y for x in θ. A valuation σ is a model of an expression θ over V, written $\sigma \models \theta$, if θ is satisfied by σ, i.e., $\theta[x \leftarrow \sigma(x) \mid x \in X]$ is valid. For example, let $\sigma = \{x \mapsto 5, y \mapsto 10\}$, then $\sigma \models (x < y)$ since $5 < 10$ is valid, and $\sigma \not\models (x + y = 10)$ since $5 + 10 = 10$ is not valid. For a triple of valuations σ_c, σ_n, and σ_o over V, we write $(\sigma_c, \sigma_n, \sigma_o)$ for a valuation σ over $\texttt{self}.V \cup \texttt{self}.V' \cup \texttt{other}.V$ defined as $\sigma(\texttt{self}.y) \triangleq \sigma_c(y)$, $\sigma(\texttt{self}.y') \triangleq \sigma_n(y)$, and $\sigma(\texttt{other}.y) \triangleq \sigma_o(y)$. We write (σ_c, σ_n) for short when σ_o is irrelevant.

Let n be a natural number and $\mathcal{P} = (Q, V, T)$ a parameterized system. An n-process instance of $\mathcal{P}$ is a transition system $\mathcal{T}_n(\mathcal{P}) = (C_n, \Delta_n)$, where C_n is the set of all n-process configurations, and $\Delta_n \subseteq C_n \times C_n$ is a transition relation. Intuitively, a pair of configurations c and c' are in Δ_n if c' is reachable from c via an execution of a guarded command by a single process. For each $\tau \in T$ of the form $q \xrightarrow{g} q'$, let Δ_n^τ be defined such that $(c, c') \in \Delta_n^\tau$ if and only if $c = c_1 \cdot \langle u \rangle \cdot c_2$, $c' = c_1 \cdot \langle u' \rangle \cdot c_2$, and the following holds:

- g is a local guard and $(u, u') \models g$, or
- g is a universal global guard and $\forall u_o \in (c_1 \cdot c_2),\ (u, u', u_o) \models g$, or
- g is an existential global guard and $\exists u_o \in (c_1 \cdot c_2),\ (u, u', u_o) \models g$.

Then, $\Delta_n \triangleq \bigcup_{\tau \in T} \Delta_n^\tau$.

For example, consider the parameterized system $\mathcal{P}_1$ given in Figure 5.1. Let $c_1 = \langle (q_1, (x \mapsto 4, y \mapsto 6)) \rangle$ and $c_2 = \langle (q_2, (x \mapsto 5, y \mapsto 6)) \rangle$ be two 1-process configurations. Then, $(c_1, c_2) \in \Delta_1^{\tau_1}$ – the control location has changed from q_1 to q_2, the value of local variable x has incremented and the value of local variable y has not changed. Let $c_3 = \langle (q_3, (x \mapsto 4, y \mapsto 5)), (q_3, (x \mapsto 2, y \mapsto 7)) \rangle$ and $c_4 = \langle (q_1, (x \mapsto 4, y \mapsto 5)), (q_3, (x \mapsto 2, y \mapsto 7)) \rangle$ be two 2-process configurations. Then, $(c_3, c_4) \in \Delta_2^{\tau_3}$ where P_1 is the `self` process and P_2 is the `other`, process P_2 is in control location q_3 in configuration c_3 and the value of its variable x is greater than 1 and the difference between its y and x variables is greater than 2; on execution of τ_3, the control location of P_1 has changed from q_3 to q_1 and the value of its x and y variables have not changed during the transition.

In this thesis, we work with a single transition system instead of many instances. We use $\mathcal{T}(\mathcal{P}) \triangleq (C, \Delta)$, where $C \triangleq \bigcup_{n \in \mathbb{N}} C_n$, and $\Delta \triangleq \bigcup_{n \in \mathbb{N}} \Delta_n$. Note that $\mathcal{T}(\mathcal{P})$ contains all n-instantiations of $\mathcal{P}$ as sub-systems.

5.2.1 Reachability Problem

The reachability problem of parameterized systems is: given a set of *initial* configurations $I \subseteq C$, and a set of *error* configurations $E \subseteq C$, decide whether there exists two configurations $c_i \in I$ and $c_e \in E$ such that there is a path from c_i to c_e in $\mathcal{T}$. This formulation is equivalent to a more common one of deciding whether there exists an $n \in \mathbb{N}$, such that an error configuration is reachable from an initial configuration in $\mathcal{T}_n(\mathcal{P})$. It is well-known that the verification of any safety property can be reduced to a reachability problem.

A backward-reachability-based algorithm is: given a set of error configurations E, compute an over-approximation of the set of all configurations that can reach E, denoted by R. Then, decide whether the intersection of I and R is empty. In the rest of this chapter, we only focus on computing R. All of the computation of our algorithm is done

using a specialized abstract domain that we describe in the next section.

5.3 Abstract Domains for Parameterized Systems

We give a brief overview of numeric abstract domains and introduce our new domains for representing configurations of parameterized systems.

5.3.1 Numeric Abstract Domains

A well-known class of *numerical abstract domains* captures arithmetic (typically linear) relations between variables in a concrete domain. There exists many numerical abstract domains. The most used are the domain of *intervals* [44] (an example was given in Section 2.3), *octagons* [89], and *polyhedra* [48]. For a set of variables V, elements of the intervals abstract domain $\textsc{Int}(V)$ have the form of $[c_1; c_2]$, where c_1 and c_2 are constants; elements of the octagons abstract domain $\textsc{Oct}(V)$ are conjunctions of constraints of the form $(\pm x \pm y \leq c)$, where $x, y \in V$ and c is a constant; and the elements of the polyhedra abstract domain $\textsc{Poly}(V)$ are conjunctions of constraints of the form $(\alpha_1 x + \alpha_2 y \leq c)$, where $x, y \in V$ and α_1, α_2, and c are constants. Whereas the analysis using intervals abstract domain is very efficient—linear memory and time cost—but not precise, the analysis using polyhedra domain is much more precise but has a huge memory cost—in practice it is exponential in the number of variables. The octagons abstract domain is between the intervals and polyhedra abstract domains in terms of expressiveness and cost (see Figure 5.2).

We use the octagons abstract domain as an example of a numeric domain. The concretization $\gamma_{\textsc{Oct}}$ maps a conjunction of constraints to a set of valuations, e.g., $\gamma_{\textsc{Oct}}(x \leq 3) = \{\sigma \in V \to \mathbb{N} \mid \sigma(x) \leq 3\}$. Abstract ordering is implemented with the implication, e.g., $(x \leq 3) \sqsubseteq_{\textsc{Oct}} (x \leq 4)$ since $x \leq 3 \Rightarrow x \leq 4$. Join of two octagons is the smallest octagon containing their union. For example, $(x = 3) \sqcup_{\textsc{Oct}} (x = 5)$ is an octagon $3 \leq x \leq 5$ that can also be written as $-x \leq -3 \wedge x \leq 5$. We will use this domain for

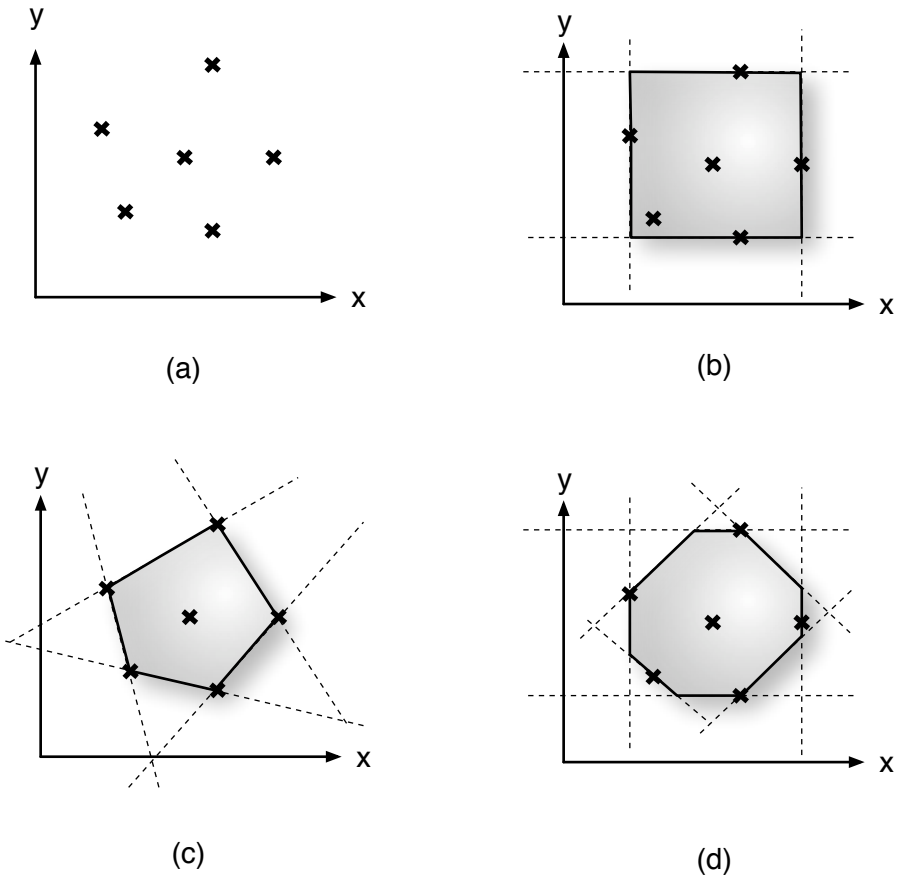

Figure 5.2: A set of points (a), and its best approximation in the intervals (b), polyhedra (c), and octagons (d) abstract domains.

all of the examples in this chapter. However, our results extend to other domains as well, such as polyhedra [48] and sets of octagons or polyhedra.

5.3.2 Abstract Domain PD

In this section, we define an abstract domain PD, called the *parametric domain*, that captures information about the control locations of configurations of a parameterized system. In the rest of this section, we fix a parameterized system $\mathcal{P}$, and use Q to denote its control locations. Elements of PD are called *abstract locations*. Each element $s \in \text{PD}$ is a map $Q \to \mathbf{2}^{\mathbb{N}}$ such that $s[q]$ is finite for all $q \in Q$ and for $q, q' \in Q$, if $q \neq q'$ then

$s[q] \cap s[q'] = \emptyset$. Intuitively, $s[q]$ represents some of the processes that are currently at control location q. For example, let

$$s_1 = (q_1 \mapsto \{1\}, q_2 \mapsto \{2, 3\})\,. \qquad (\star)$$

Intuitively, s_1 represents all concrete configurations in which there are *at least* three processes: one at q_1, and two at q_2. Note that the actual numeric PIDs are irrelevant and are only used for reference as we show below.

Let s be in PD. We write $\textsc{Proc}(s)$ for the set of all PIDs appearing in s. Formally, $\textsc{Proc}(s) \triangleq \bigcup_{q \in Q} s[q]$. We write $|s|$ for $|\textsc{Proc}(s)|$, and $\textsc{pc}(i, s)$ for the control location of process i, i.e., $\textsc{pc}(i, s) = q$ if and only if $i \in s[q]$. For example, for s_1 above, $\textsc{Proc}(s_1) = \{1, 2, 3\}$, $|s_1| = 3$, and $\textsc{pc}(1, s_1) = q_1$ and $\textsc{pc}(2, s_1) = q_2$. Without loss of generality, we assume whenever $|s| = m$, then $\textsc{Proc}(s) = \{1, .., m\}$.

In the rest of this section, we formalize the definitions of concretization γ, abstract ordering $\sqsubseteq$, and join $\sqcup$ for the parametric domain. Intuitively, $\gamma_{\text{PD}}(s)$ is the set of all configurations that have at least $|s[q]|$ processes at q, for all $q \in Q$. Let $c = \langle (q_1, v_1), \ldots, (q_n, v_n) \rangle$ be a configuration, $s \in$ PD such that $|s| = m \leq n$, and $h : \{1, .., m\} \to \{1, .., n\}$ be an injection. We say that c satisfies s under h, written $c \models_h s$ if and only if

$$\forall i \in \textsc{Proc}(s),\ \textsc{pc}(i, s) = q_{h(i)}$$

We define $\gamma_{\text{PD}}(s) \triangleq \{c \mid \exists h,\, c \models_h s\}$. It is easy to see that this definition captures our intuition. For example, let $c_1 = \langle (q_1, v_1), (q_2, v_2), (q_1, v_3), (q_2, v_4) \rangle$, where $\{v_i\}$ are arbitrary valuations, and $h = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 4\}$. Recall that, (q_1, v_1) is the process state of P_1, (q_2, v_2) is the process state of P_2, and so on. Then, $c_1 \models_h s_1$; thus $c_1 \in \gamma_{\text{PD}}(s_1)$.

For two abstract locations s and t, if for all $q \in Q$, $|t[q]| \leq |s[q]|$, then t approximates more concrete configurations. We define the ordering $\sqsubseteq_{\text{PD}}$ as

$$s \sqsubseteq_{\text{PD}} t \Leftrightarrow (\forall q \in Q,\ |t[q]| \leq |s[q]|)\,.$$

For example, let $s_2 = (q_1 \mapsto \{2\}, q_2 \mapsto \{1\})$ and $s_3 = (q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\})$. The abstract location s_2 represents all concrete configurations in which there are at least two processes: one at q_1, and one at q_2. The abstract location s_3 represents all concrete configurations in which there are at least three processes: two at q_1, and one at q_2. Thus, s_2 approximates more concrete configurations compared to s_3. Thus, $s_3 \sqsubseteq_{\text{PD}} s_2$, but $s_3 \not\sqsubseteq_{\text{PD}} s_1$ and $s_1 \not\sqsubseteq_{\text{PD}} s_3$.

Note that the abstract domain PD is not a lattice. Thus, the Galois connection framework of AI (Example 4.6 in [46]) is not applicable. Therefore, we follow a more general framework of Abstract Interpretation [46] (as explained in Section 2.3) that allows for an abstract domain to be a pre-order.

Let $\top_{\text{PD}}$ be defined as an element s such that for all $q \in Q, s[q] = \emptyset$. Then, $\top_{\text{PD}}$ is the $\sqsubseteq_{\text{PD}}$-largest element of PD. For $s, t \in$ PD, we define the join as $s \sqcup_{\text{PD}} t = t$ if $s \sqsubseteq_{\text{PD}} t$ and $\top_{\text{PD}}$ otherwise. At a first glance, our definition of join may look too imprecise. However, our analysis algorithm (see BACKREACH in Section 5.4) applies the join $s \sqcup_{\text{PD}} t$ only under the assumption that $s \sqsubseteq_{\text{PD}} t$.

Theorem 14 *The abstract ordering $\sqsubseteq_{\text{PD}}$ and the join $\sqcup_{\text{PD}}$ are sound.*

Proof: Let s_1 and s_2 be two elements of PD such that $s_1 \sqsubseteq_{\text{PD}} s_2$, and c be a concrete element in $\gamma_{\text{PD}}(s_1)$. We show that $c \in \gamma_{\text{PD}}(s_2)$. Let $|c| = n$, $|s_1| = m$, and $|s_2| = l$. By the definition of $\sqsubseteq_{\text{PD}}$, $l \leq m$, thus $l \leq m \leq n$. By the definition of γ_{PD}, there exists an injection $h_1 : \{1,..,m\} \rightarrow \{1,..,n\}$ such that $c \models_{h_1} s_1$. We can also deduce from the definition of $\sqsubseteq_{\text{PD}}$ that there exists an injection $h : \{1,..,l\} \rightarrow \{1,..,m\}$ such that, for every control location q, and all $i \in \text{PROC}(s_2)$, $i \in s_2[q] \Rightarrow h(i) \in s_1[q]$. Let $h_2 : \{1,..,l\} \rightarrow \{1,..,n\}$ be defined such that $h_2(x) = h_1(h(x))$. Then, $c \models_{h_2} s_2$. Hence, $c \in \gamma_{\text{PD}}(s_2)$.

The soundness of the join operator follows from the definition. □

In the next section, we will show how to extend the domain PD with a numeric (or even an arbitrary) abstract domain.

5.3.3 Abstract Domain PD(A)

We combine the parametric domain PD with an abstract domain A. The new domain is called PD(A). For clarity of presentation, we assume that A is a numerical abstract domain. We call elements of PD(A) *abstract global states* (AGS). An AGS is of the form (s, ψ), where $s \in$ PD and $\psi \in A$. Intuitively, s captures the control location information and ψ captures numerical constraints on process variables. For an AGS $r = (s, \psi)$, we write $\text{loc}(r)$ for the abstract location s.

In the rest of the section, we fix a parameterized system $\mathcal{P} = (Q, V, T)$. For $x \in V$, we write $P_i.x$ to refer to the variable x of process i. We require that for every element $(s, \psi) \in$ PD(A), ψ is an expression over variables in the set $\{P_i.x \mid x \in V, i \in \textsc{Proc}(s)\}$. For example, an AGS $(s_1, P_1.x < P_2.y)$, where s_1 is as defined in ($\star$), represents all concrete configurations that satisfy s_1 and, additionally, have a process i in state q_1 and a process j in state q_2 such that the value of variable x of process i is less than the value of the variable y of process j, i.e. $P_i.x < P_j.y$. Note that i and j are not necessarily 1 and 2, since the PIDs in the abstract global states are only used for reference and do not directly correspond to PIDs in concrete configurations.

We now proceed to define $\gamma_{\text{PD(A)}}$ formally. For a function $h : \mathbb{N} \to \mathbb{N}$ and an expression ψ, we write $h(\psi)$ for the result of permuting all process references in ψ according to h, i.e., $h(\psi) \triangleq \psi[P_i \leftarrow P_{h(i)} \mid i \in \mathbb{N}]$. Let $c = \langle u_1, \ldots, u_n \rangle$ be a concrete configuration. We write σ_c for a valuation corresponding to the configuration c, defined as follows: $\sigma_c(P_j.x) \triangleq u_j(x)$. Let (s, ψ) be an AGS, such that $|s| = m$ and $m \leq n$, and $h : \{1, .., m\} \to \{1, .., n\}$ be an injection. We say that c satisfies (s, ψ) under h, written, $c \models_h (s, \psi)$ if and only if $c \models_h s \wedge \sigma_c \models h(\psi)$. For example, let $c_1 = \langle (q_1, v_1), (q_2, v_2), (q_1, v_3), (q_2, v_4) \rangle$, where

$$
\begin{aligned}
v_1 &= (x \mapsto 4, y \mapsto 2) \\
v_2 &= (x \mapsto 5, y \mapsto 5) \\
v_3 &= (x \mapsto 1, y \mapsto 9) \\
v_4 &= (x \mapsto 3, y \mapsto 6)\,.
\end{aligned}
$$

Let $h = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 4\}$. We have $c_1 \models_h s_1$ and $\sigma_{c_1} \models (P_1.x < P_2.y)$; thus, $c_1 \models_h (s_1, (P_1.x < P_2.y))$. Finally, we define $\gamma_{\text{PD(A)}}((s, \psi)) \triangleq \{c \mid \exists h,\, c \models_h (s, \psi)\}$.

We now describe the ordering $\sqsubseteq_{\text{PD(A)}}$. Let s, t be in PD, such that $s \sqsubseteq_{\text{PD}} t$. We write, $\mathcal{U}(s, t)$ for the set of all functions h such that (a) h is an injection from $\{1, .., |t|\}$ to $\{1, .., |s|\}$, and (b) for all $i \in \text{PROC}(t)$, $\; i \in t[q] \Rightarrow h(i) \in s[q]$. That is, h maps each process of t to an equivalent process of s. For example, let $s_4 = (q_1 \mapsto \{1, 2\})$, and $s_5 = (q_1 \mapsto \{1\})$, then $\mathcal{U}(s_4, s_5) = \{h_1, h_2\}$, where $h_1 = \{1 \mapsto 1\}$ and $h_2 = \{1 \mapsto 2\}$. Note that if $s \sqsubseteq_{\text{PD}} t$, then $\mathcal{U}(s, t)$ is not empty. The ordering $\sqsubseteq_{\text{PD(A)}}$ is defined as

$$(s, \psi) \sqsubseteq_{\text{PD(A)}} (t, \varphi) \Leftrightarrow s \sqsubseteq_{\text{PD}} t \wedge \exists h \in \mathcal{U}(s, t),\, \psi \sqsubseteq_A h(\varphi)\,.$$

For example, let $\psi_1 = ((P_1.x > 0) \wedge (P_2.x > 4))$, and $\psi_2 = (P_1.x > 1)$, then $(s_4, \psi_1) \sqsubseteq_{\text{PD(A)}} (s_5, \psi_2)$, since $s_4 \sqsubseteq_{\text{PD}} s_5$ and ψ_1 implies $h_2(\psi_2) = (P_2.x > 1)$.

The $\sqsubseteq_{\text{PD(A)}}$-largest element is $(\top_{\text{PD}}, \top_A)$, where $\top_A$ is the $\sqsubseteq_A$-largest element of A. The join $\sqcup_{\text{PD(A)}}$ is defined as

$$(s, \psi) \sqcup_{\text{PD(A)}} (t, \varphi) \triangleq \begin{cases} (s, \psi \sqcup_A h(\varphi)) & \text{if } s \sqsubseteq_{\text{PD}} t \wedge t \sqsubseteq_{\text{PD}} s \\ \top_{\text{PD(A)}} & \text{otherwise,} \end{cases}$$

where h is any injection in $\mathcal{U}(s, t)$. Intuitively, we use the join $\sqcup_A$ of A to join the constraints of the variables, while aligning PIDs between s and t. Note that a different choice for h affects precision but it does not affect the soundness of the join. In practice, it is best to pick an h that leads to the $\sqsubseteq_{\text{PD(A)}}$-least result. As with abstract domain PD, it is possible to define join more precisely, but it was not needed for our algorithm.

Theorem 15 *The abstract ordering $\sqsubseteq_{\text{PD(A)}}$ and the join $\sqcup_{\text{PD(A)}}$ are sound.*

Proof: Let (s_1, ψ_1) and (s_2, ψ_2) be two elements of PD(A) such that $(s_1, \psi_1) \sqsubseteq_{\text{PD(A)}} (s_2, \psi_2)$, and c a concrete element in $\gamma_{\text{PD(A)}}((s_1, \psi_1))$. We show that $c \in \gamma_{\text{PD(A)}}((s_2, \psi_2))$. Let $|c| = n$, $|s_1| = m$, and $|s_2| = l$. Note that $l \leq m \leq n$. By the definition of $\gamma_{\text{PD(A)}}$, there exists an injection $h_1 : \{1, .., m\} \rightarrow \{1, .., n\}$ such that $c \models_{h_1} s_1$ and $\sigma_c \models h_1(\psi_1)$.

By the definition of $\sqsubseteq_{\text{PD(A)}}$, there exists an injection $h : \{1,..,l\} \rightarrow \{1,..,m\}$ such that $h \in \mathcal{U}(s_1, s_2)$ and $\psi_1 \sqsubseteq_A h(\psi_2)$. Note that $\sqsubseteq_A$ is preserved under renaming of variables, i.e. if $\psi_1 \sqsubseteq_A h(\psi_2)$, then $h_1(\psi_1) \sqsubseteq_A h_1(h(\psi_2))$. Let $h_2 : \{1,..,l\} \rightarrow \{1,..,n\}$ be defined such that $h_2(x) = h_1(h(x))$. Thus, $c \models_{h_2} s_2$ due to the soundness of $\sqsubseteq_{\text{PD}}$ and $\sigma_c \models h_2(\psi_2)$. Hence, $c \in \gamma_{\text{PD(A)}}((s_2, \psi_2))$.

Recall that $\sqcup_{\text{PD(A)}}$ is defined as

$$(s_1, \psi_1) \sqcup_{\text{PD(A)}} (s_2, \psi_2) \triangleq \begin{cases} (s_1, \psi_1 \sqcup_A h(\psi_2)) & \text{if } s_1 \sqsubseteq_{\text{PD}} s_2 \wedge s_2 \sqsubseteq_{\text{PD}} s_1 \\ \top_{\text{PD(A)}} & \text{otherwise,} \end{cases}$$

where h is an injection in $\mathcal{U}(s_1, s_2)$. The soundness follows from the fact that $\psi_1 \sqsubseteq_A (\psi_1 \sqcup_A h(\psi_2))$, and $h(\psi_2) \sqsubseteq_A (\psi_1 \sqcup_A h(\psi_2))$. □

Elements of PD(A) concisely represent (possibly infinite) sets of configurations of a concrete parameterized system. This domain is the basis of our backward-reachability algorithm that we present in the next section.

5.4 Backward Reachability Algorithm

We present our algorithm BACKREACH for over-approximating the backward-reachability in parameterized systems. We begin with an overview of the algorithm, then discuss its main operation, i.e. computation of the pre-image, and conclude with an example.

5.4.1 Overview

The algorithm BACKREACH is shown in Figure 5.3. As inputs, it takes a set `Trans` of guarded commands and an AGS `e`. The output is a set of AGSs that over-approximates all concrete configurations from which `e` is reachable.

The algorithm uses the list `RL` to keep track of all AGSs seen so far, and a work list `WL` to keep track of all AGSs to be explored. When `WL` becomes empty, the algorithm terminates. In each iteration, an AGS (s, ψ) is chosen from `WL` (lines 3–4), its predecessors

are computed (lines 6–7), and are added to RL and WL lists if needed (lines 8–19). The algorithm adds a computed AGS (t, φ) to the RL and WL lists if it is not subsumed by any another AGS in RL. The algorithm ensures that RL contains only one AGS for each abstract location by joining the AGSs with the same abstract locations (line 17). The computation of the predecessors is done using the function 'Pre', which is described in detail below.

5.4.2 Computing the Pre-Image

In the rest of this section, we describe the implementation of the pre-image computation (line 7 of BACKREACH algorithm). First, we describe the operation for the domain PD, and then extend it to PD(A).

Pre-Image for PD

Let s be an element of PD, $\tau : q \xrightarrow{g} q'$ a guarded command, and i a PID in $\textsc{Proc}(s)$. The result of pre-image operation $\text{Pre}_{\text{PD}}(s, \tau, i)$ is a set B of elements of PD that over-approximates all states from which a state in $\gamma(s)$ is reachable by process i executing τ. There are three cases, based on the type of guard, g.

Case 1 g is a local guard. If s is an abstract location obtained by process P_i executing τ, then, P_i is in state q' in s. Furthermore, P_i must have been in state q before executing τ. To formalize this, we define a helper function $\textsc{MoveProc}(s, i, q_1, q_2)$ that moves process i in s from location q_1 to location q_2: $\textsc{MoveProc}(s, i, q_1, q_2) \triangleq t$, where $t[q_1] = s[q_1] \setminus \{i\}$, $t[q_2] = s[q_2] \cup \{i\}$, and $t[q] = s[q]$ otherwise. Then,

$$\text{Pre}_{\text{PD}}(s, \tau, i) \triangleq \begin{cases} \{\textsc{MoveProc}(s, i, q', q)\} & \text{if } i \in s[q'] \\ \emptyset & \text{otherwise.} \end{cases}$$

For example, let $s_1 = (q_1 \mapsto \{1\}, q_2 \mapsto \{2, 3\})$, and $\tau = q_1 \xrightarrow{true} q_2$. Then, $\text{Pre}_{\text{PD}}(s_1, \tau, 1) = \emptyset$, since P_1 is not in q_2, and $\text{Pre}_{\text{PD}}(s_1, \tau, 2) = (q_1 \mapsto \{1, 2\}, q_2 \mapsto$

1: **Set of AGS** BACKREACH (**Set** $\texttt{Trans}$, **AGS** $\texttt{e}$)
2: $\quad \texttt{WL} \leftarrow \{\texttt{e}\},\ \texttt{RL} \leftarrow \{\texttt{e}\}$
3: $\quad$ **forall** $(s, \psi) \in \texttt{WL}$ **do**
4: $\quad\quad \texttt{WL} \leftarrow \texttt{WL} \setminus \{(s, \psi)\}$
5: $\quad\quad P \leftarrow \emptyset$
6: $\quad\quad$ **forall** $\{\tau \in \texttt{Trans}, i \in \text{PROC}(s) \mid \tau = (q \xrightarrow{g} q') \text{ and } i \in s[q']\}$ **do**
7: $\quad\quad\quad P \leftarrow P \cup \text{Pre}((s, \psi), \tau, i)$
8: $\quad\quad$ **forall** $r \in P$ **do**
9: $\quad\quad\quad \texttt{skip} \leftarrow \text{false}, \texttt{saved} \leftarrow \text{null}$
10: $\quad\quad\quad$ **forall** $u \in \texttt{RL}$ **do**
11: $\quad\quad\quad\quad$ **if** $r \sqsubseteq_{\text{PD(A)}} u$ **then**
12: $\quad\quad\quad\quad\quad \texttt{skip} \leftarrow \text{true}$, **break**
13: $\quad\quad\quad\quad$ **if** $\text{loc}(r) = \text{loc}(u)$ **then**
14: $\quad\quad\quad\quad\quad \texttt{saved} \leftarrow u$
15: $\quad\quad\quad$ **if** $\texttt{skip} = \text{false}$ **then**
16: $\quad\quad\quad\quad$ **if** $\texttt{saved} \neq \text{null}$ **then**
17: $\quad\quad\quad\quad\quad \texttt{RL} \leftarrow (\texttt{RL} \setminus \{\texttt{saved}\}) \cup \{\texttt{saved} \sqcup_{\text{PD(A)}} r\}, \texttt{WL} \leftarrow \texttt{WL} \cup \{\texttt{saved} \sqcup_{\text{PD(A)}} r\}$
18: $\quad\quad\quad\quad$ **else**
19: $\quad\quad\quad\quad\quad \texttt{RL} \leftarrow \texttt{RL} \cup \{r\}, \texttt{WL} \leftarrow \texttt{WL} \cup \{r\}$
20: $\quad$ **return** $\texttt{RL}$

Figure 5.3: The BACKREACH algorithm.

$\{3\}$).

Case 2 g is a universal global guard: $\forall \texttt{other} \neq \texttt{self},\ (\texttt{other.}\mathbf{pc} = q_o) \wedge \theta$. Then, the pre-image computation is similar to Case 1 except that all processes other than i must be in control location q_o in s. Thus, $\text{Pre}_{\text{PD}}(s, \tau, i) \triangleq \{\text{MOVEPROC}(s, i, q', q)\}$, if $i \in s[q']$ and $\forall j \in \text{PROC}(s) \setminus \{i\},\ \text{PC}(s, j) = q_o$, and $\emptyset$ otherwise.

Case 3 g is an existential global guard: $\exists \texttt{other} \neq \texttt{self},\ (\texttt{other.}\mathbf{pc} = q_o) \wedge \theta$. Then, τ can only be executed from an abstract location that has a process different from i at location q_o. The computation of Pre_{PD} is partitioned based on the choice of that `other` process. The `other` process can be either a process in $\text{PROC}(s)$, or a new process with PID = $(|s| + 1)$. Let

$$\text{Pre}_{\text{PD}}(s, \tau, i) \triangleq \bigcup_{j \in s[q_o] \setminus \{i\}} \text{OPre}_{\text{PD}}(s, \tau, i, j) \cup \text{OPre}_{\text{PD}}(s, \tau, i, |s| + 1)\,,$$

where $\text{OPre}_{\text{PD}}(s, \tau, i, j)$ is the pre-image under the assumption that P_j is the `other` process. We define another helper function called $\text{MOVEADDPROC}(s, i, q_1, q_2, j, q_3)$ that in addition to moving process i from q_1 to q_2 adds a new process j to q_3: $\text{MOVEADDPROC}\ (s, i, q_1, q_2, j, q_3) \triangleq t$, where $t[q_1] = s[q_1] \setminus \{i\}$, $t[q_2] = s[q_2] \cup \{i\}$, $t[q_3] = s[q_3] \cup \{j\}$, and $t[q] = s[q]$ otherwise. Then,

$$\text{OPre}_{\text{PD}}(s, \tau, i, j) \triangleq \begin{cases} \{\text{MOVEPROC}(s, i, q', q)\} & \text{if } j \in s[q_o] \setminus \{i\} \\ \{\text{MOVEADDPROC}(s, i, q', q, j, q_o)\} & \text{if } j = |s| + 1 \\ \emptyset & \text{otherwise.} \end{cases}$$

For example, let $s_1 = (q_1 \mapsto \{1\}, q_2 \mapsto \{2, 3\})$ and $\tau_1 = q_1 \xrightarrow{g_1} q_2$ and $g_1 : \exists \texttt{other} \neq \texttt{self},\ (\texttt{other.}\mathbf{pc} = q_2) \wedge \theta$. Then, $\text{Pre}_{\text{PD}}(s_1, \tau_1, 2)$ is the union of $\text{OPre}_{\text{PD}}(s, \tau, 2, 3)$ and $\text{OPre}_{\text{PD}}(s, \tau, 2, 4)$. In the former, P_3 is considered as the `other` process, and in the latter, a newly added process P_4 is the `other` process.

Then, $\text{OPre}_{\text{PD}}(s, \tau, 2, 3) = (q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3\})$ and $\text{OPre}_{\text{PD}}(s, \tau, 2, 4) = (q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3, 4\})$.

Theorem 16 *The pre-image operation of* PD *is sound.*

Proof: Let s be an element of PD, $c \in \gamma_{\text{PD}}(s)$, $\tau : q' \xrightarrow{g} q$ a guarded command, and i a PID. Let $|c| = n$ and $|s| = m$. By the definition of γ_{PD}, there exists an injection $h : \{1, .., m\} \rightarrow \{1, .., n\}$ such that $c \models_h s$. Let $s' = \text{Pre}_{\text{PD}}(s, \tau, i)$ and $k = h(i)$. We show that any c' that can reach c by process k executing τ, i.e. $c' \xrightarrow{\tau_k} c$, is in $\gamma_{\text{PD}}(s')$.

Case 1 g is a local guard. If $i \notin s[q]$, then the set of predecessors is empty and thus the pre-image operation is sound. If $i \in s[q]$, then process k is in state q in c. Thus, c' is derived from c by moving process k from state q to q'. Hence, $c' \models_h s'$ and $c' \in \gamma_{\text{PD}}(s')$.

Case 2 If g is a universal global guard, then the soundness of the pre-image operation is established similar to Case 1.

Case 3 g is an existential global guard. The soundness of the pre-image operation follows from the soundness of $\text{OPre}_{\text{PD}}(s, \tau, i, j)$ and $\text{OPre}_{\text{PD}}(s, \tau, i, |s| + 1)$. Let $c \in \gamma_{\text{PD}}(s)$. Thus, there exists an injection h such that $c \models_h s$. Let $h(i) = k$ and $h(j) = l$. Similar to Case 1, we can show that any c' that can reach c by process k executing τ while l is the `other` process, i.e. $c' \xrightarrow{\tau_{k,l}} c$, is in $\gamma_{\text{PD}}(s')$.

□

Pre-Image for PD(A)

We assume that the numeric abstract domain A has a pre-image operation $\text{Pre}_A(\psi, R)$ that takes an element of the domain $\psi \in A$, and a relation R described by an expression over primed and unprimed variables. It returns an abstract element that over-approximates the

pre-image of $\gamma_A(\psi)$ over R. Many numeric abstract domains satisfy this assumption. For example, in OCT, $\text{Pre}_{\text{OCT}}(x \geq 1, x' = x + 1)$ is $x \geq 0$.

Let (s, ψ) be an element of PD(A), $\tau : q \xrightarrow{g} q'$ a guarded command, and i a PID in $\text{PROC}(s)$. The pre-image operation in PD(A) is defined using the following templates. If g is either a local or a universal global guard, then

$$\text{Pre}_{\text{PD(A)}}((s, \psi), \tau, i) \triangleq \text{Pre}_{\text{PD}}(s, \tau, i) \times \text{Pre}_A(\psi, R_i)$$

and if g is an existential global guard, then $\text{Pre}_{\text{PD(A)}}((s, \psi), \tau, i)$ is defined similar to Pre_{PD} where

$$\text{OPre}_{\text{PD(A)}}((s, \psi), \tau, i, j) \triangleq \text{OPre}_{\text{PD}}(s, \tau, i, j) \times \text{Pre}_A(\psi, R_{i,j})\,.$$

Here, j is a PID in $\text{PROC}(s)$, and R_i, $R_{i,j}$ are relations defined based on g as described below.

Case 1 g is a local guard. Assume $g = \theta$, where θ is an expression over $\texttt{self}.(V \cup V')$.

Let Θ_i and Γ_i be defined as

$$\Theta_i \triangleq \theta[\texttt{self} \leftarrow P_i] \qquad \Gamma_i \triangleq \bigwedge_{j \in (\text{PROC}(s) \setminus \{i\})} \bigwedge_{x \in V} P_j.x' = P_j.x$$

Then, $R_i \triangleq \Theta_i \wedge \Gamma_i$. Intuitively, Θ_i instantiates the guard to process i, and Γ_i ensures that the variables of processes other than i are not affected. For example, let (s_1, ψ_1) be an AGS where s_1 is as defined in ($\star$) and $\psi_1 = ((P_1.x > 0) \wedge (P_2.x > 1) \wedge (P_3.x > 2))$. Let $\tau_1 = q_1 \xrightarrow{g_1} q_2$ and $g_1 : \texttt{self}.x' = \texttt{self}.x + 1$. Then, $\text{Pre}_{\text{PD(A)}}((s_1, \psi_1), \tau, 2) = \text{Pre}_{\text{PD}}(s_1, \tau, 2) \times \text{Pre}_{\text{OCT}}(\psi_1, R_2)$. We have $\text{Pre}_{\text{PD}}(s_1, \tau, 2) = (q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3\})$, and $R_2 = ((P_2.x' = P_2.x + 1) \wedge (P_1.x' = P_1.x) \wedge (P_3.x' = P_3.x))$. Thus, $\text{Pre}_{\text{OCT}}(\psi_1, R_2) = ((P_1.x > 0) \wedge (P_2.x > 0) \wedge (P_3.x > 2))$. Therefore, $\text{Pre}_{\text{PD(A)}}((s_1, \psi_1), \tau, 2) = ((q_1 \mapsto \{1, 2\}, q_2 \mapsto \{3\}), ((P_1.x > 0) \wedge (P_2.x > 0) \wedge (P_3.x > 2)))$.

Case 2 g is a universal global guard: $\forall \texttt{other} \neq \texttt{self},\ (\texttt{other}.\mathbf{pc} = q_o) \wedge \theta$, where θ is an expression over $\texttt{self}.(V \cup V') \cup \texttt{other}.V$ variables. We need to instantiate θ with two PIDs: one for `self`, and one for `other`. Let $\Theta_{i,j}$ be defined as

$$\Theta_{i,j} \triangleq \theta[\texttt{self} \leftarrow P_i, \texttt{other} \leftarrow P_j]\,.$$

Then, $R_i \triangleq \bigwedge_{j \in (\text{PROC}(s) \setminus \{i\})} \Theta_{i,j} \wedge \Gamma_i$. Intuitively, R_i ensures that all processes other than i satisfy the universal global guard but only values of process i are affected during the transition.

Case 3 g is an existential global guard: $\exists \texttt{other} \neq \texttt{self},\ (\texttt{other}.\mathbf{pc} = q_o) \wedge \theta$, where θ is again an expression over $\texttt{self}.(V \cup V') \cup \texttt{other}.V$ variables. However, in this case, the pre-image operator provides a PID j to instantiate the `other` process. Thus, $R_{i,j}$ is defined as $R_{i,j} \triangleq \Theta_{i,j} \wedge \Gamma_i$.

Theorem 17 *The pre-image operation of* PD(A) *is sound.*

Proof: Let (s, ψ) be an element of PD(A), $\tau : q \xrightarrow{g} q'$ a guarded command, and i a PID in $\text{PROC}(s)$. In order to prove the soundness of $\text{Pre}_{\text{PD(A)}}((s, \psi), \tau, i)$, we decompose the guarded commands as follows:

Case 1 g is a local guard. Let $\tau_i : q \xrightarrow{g[i]} q'$ be an instantiation of the guarded command to process i (for `self`). Then, $q \xrightarrow{g[i]} q'$ is equivalent to the atomic execution of $q \rightarrow q'$ followed by $q' \xrightarrow{R_i} q'$ where R_i is defined in Case 1 of definition $\text{Pre}_{\text{PD(A)}}$.

Case 2 g is a universal global guard. Let $\tau_i : q \xrightarrow{g[i]} q'$ be an instantiation of the guarded command to process i (for `self`). Let g_c denote $(\forall j \neq i,\ \texttt{other}.\mathbf{pc} = q_o)$. Then, $q \xrightarrow{g[i]} q'$ is equivalent to the atomic execution of $q \xrightarrow{g_c} q'$ followed by $q' \xrightarrow{R_i} q'$ where R_i is defined in Case 2 of definition $\text{Pre}_{\text{PD(A)}}$.

Case 3 g is an existential global guard. Let $\tau_{i,j} : q \xrightarrow{g[i,j]} q'$ be an instantiation of the guarded command to process i and j (for `self` and `other`). Then, $q \xrightarrow{g[i,j]} q'$ is

Name	Abstract Location	Constraints
1	$(q_2 \mapsto \{1\})$	$(P_1.x > 1) \wedge (P_1.y > 3)$
2	$(q_1 \mapsto \{1\})$	$(P_1.x > 0) \wedge (P_1.y > 3)$
3	$(q_3 \mapsto \{1, 2\})$	$(P_1.x > 0) \wedge (P_1.y > 3) \wedge (P_2.y - P_2.x > 2) \wedge (P_2.x > 1)$
4	$(q_1 \mapsto \{1\}, q_3 \mapsto \{2\})$	$(P_1.x > 0) \wedge (P_1.y > 5) \wedge (P_2.y - P_2.x > 2) \wedge (P_2.x > 1)$
5	$(q_1 \mapsto \{2\}, q_3 \mapsto \{1\})$	$(P_1.x > 0) \wedge (P_1.y > 3) \wedge (P_2.y - P_2.x > 4) \wedge (P_2.x > 1)$

Table 5.1: An example of a computation of BACKREACH.

equivalent to the atomic execution of $q \xrightarrow{j.\mathbf{pc}=q_o} q'$ followed by $q' \xrightarrow{R_{i,j}} q'$ where $R_{i,j}$ is defined in Case 3 of definition $\text{Pre}_{\text{PD(A)}}$.

The soundness follows based upon the soundness of Pre_{PD} and Pre_A and distributivity of pre-image operation over sequential composition.

□

5.4.3 Example

In this section, we illustrate a run of our BACKREACH algorithm on an example using abstract domain PD(OCT). We use the parameterized system shown in Figure 5.1, and let `e` be $((q_2 \mapsto \{1\}), ((P_1.x > 1) \wedge (P_1.y > 3)))$.

We present the AGSs computed by the algorithm in Table 5.1. Each row in the table represents a single AGS (s, ψ) where the first column is a numeric reference, the second is the abstract location l, and the third is the octagon constraint ψ. Row 1 of the table corresponds to `e` defined above. We refer to the rows of Table 5.1 by numeric references.

In the first iteration, the algorithm computes $\text{Pre}(e, \tau_1, 1)$ that results in the AGS (s_2, ψ_2) shown in row 2. In the second iteration, the algorithm computes $(s_3, \psi_3) = \text{Pre}((s_2, \psi_2), \tau_3, 1)$ shown in row 3. In this iteration, a new process P_2 is added at control location q_3 to serve as the `other` process due to the existential global guard on τ_3. In the third iteration, τ_2 is enabled twice: once for process P_1, and once for process P_2. Row 4 shows (s_4, ψ_4), the result of pre-image of τ_2 with respect to process P_1, i.e.,

Pre$((s_3, \psi_3), \tau_2, 1)$. This state is subsumed by (s_2, ψ_2) since $s_4 \sqsubseteq_{\text{PD}} s_2$ and $\psi_4 \Rightarrow \psi_2$. Thus, it is not added to the list RL. Row 5 shows (s_5, ψ_5), the result of pre-image of τ_2 with respect to process P_2, i.e., Pre$((s_3, \psi_3), \tau_2, 2)$. This state is subsumed by (s_2, ψ_2) as well. The reason is slightly more complicated. First, $s_5 \sqsubseteq_{\text{PD}} s_2$. Second, the process P_2 of s_5 corresponds to the process P_1 of s_2. Then, $\psi_2[P_1 \leftarrow P_2] = ((P_2.x > 1) \wedge (P_2.y > 3))$ and $\psi_5 \Rightarrow ((P_2.x > 1) \wedge (P_2.y > 3))$. Thus, this AGS is not added to the list RL.

At this point, the work list WL becomes empty and the algorithm terminates. Thus, the RL contains only the AGSs shown in the first three rows of Table 5.1.

BACKREACH is sound: if it terminates, it always computes the correct result.

Theorem 18 *Let $\mathcal{P} = (Q, V, T)$ be a parameterized system and* e *an abstract global state. If* BACKREACH$(\mathtt{T}, \mathtt{e})$ *terminates, it returns an over-approximation of the set of backward-reachable states from $\gamma_{\text{PD(A)}}(\mathtt{e})$.*

BACKREACH is incomplete and may run forever. In the next section, we show how sound termination can be enforced.

5.5 Enforcing Convergence

The BACKREACH algorithm is not guaranteed to terminate. There are two reasons for a possible divergence of BACKREACH. First, the numeric abstract domain A may be infinite (like octagons or polyhedra), thus BACKREACH may get stuck in an infinite numeric computation. Second, successive applications of pre-image to a transition with an existential global guard may introduce unbounded numbers of processes. Here, we illustrate divergence of the BACKREACH algorithm through a set of examples and show how we can enforce termination.

5.5.1 Numeric Divergence

We begin with an example that illustrates numeric divergence in the abstract domain PD(OCT). Let $\mathcal{P} = (Q, V, T)$, where $Q = \{q\}$, $V = \{x\}$, and $T = \{\tau\}$, where τ is

$q \xrightarrow{g} q$, $g : (\texttt{self}.x \geq 0) \Rightarrow (\texttt{self}.x' = \texttt{self}.x - 1)$. Let $\texttt{e}$ be $((q \mapsto \{1\}), (P_1.x = 5))$. Consider the execution of BACKREACH$(\texttt{T}, \texttt{e})$. In the first iteration, the algorithm computes the state $((q \mapsto \{1\}), (P_1.x = 6))$. It is joined to $\texttt{e}$ at line 17 of the algorithm in Figure 5.3, resulting in

$$((q \mapsto \{1\}), ((P_1.x = 5) \sqcup_{\text{OCT}} (P_1.x = 6))) = ((q \mapsto \{1\}), (5 \leq P_1.x \leq 6)) .$$

Similarly, the result of the second iteration is $((q \mapsto \{1\}), (5 \leq P_1.x \leq 7))$, etc. Thus, the BACKREACH$(\texttt{T}, \texttt{e})$ diverges.

In AI, a common approach to force sound convergence is to use *widening* (refer to Definition 7) instead of join to combine the reachable states. Recall, a widening operator for abstract domain A, denoted by $\bigtriangledown_A$, is an operator that over-approximates join, and for any increasing chain $x_0 \sqsubseteq_A x_1 \sqsubseteq_A \ldots \sqsubseteq_A x_n \ldots$ in A, the increasing chain $y_0 = x_0, \ldots, y_{n+1} = y_n \bigtriangledown_A x_{n+1}, \ldots$ stabilizes after a finite number of terms. Thus, replacing join with widening forces convergence of any least-fixpoint computation.

We extend the widening operator of A to PD(A) in the following way. Given two abstract global states (s, ψ) and (t, φ), then

$$(s, \psi) \bigtriangledown_{\text{PD(A)}} (t, \varphi) \triangleq \begin{cases} (s, \psi \bigtriangledown_A h(\varphi)) & \text{if } s \sqsubseteq_{\text{PD}} t \wedge t \sqsubseteq_{\text{PD}} s \\ \top_{\text{PD(A)}} & \text{otherwise.} \end{cases}$$

Theorem 19 *The operator* $\bigtriangledown_{\text{PD(A)}}$ *is a widening on* PD(A).

Proof: Let $(s_0, \psi_0) \sqsubseteq_{\text{PD(A)}} (s_1, \psi_1) \sqsubseteq_{\text{PD(A)}} \ldots \sqsubseteq_{\text{PD(A)}} (s_n, \psi_n) \sqsubseteq_{\text{PD(A)}} \ldots$ be an increasing chain in PD(A). Consider the chain $y_0 = (s_0, \psi_0), \ldots, y_{n+1} = y_n \bigtriangledown_{\text{PD(A)}} (s_{n+1}, \psi_{n+1})$. We show that this chain stabilizes after finite terms. There are two cases: either there exists an i such that $s_i \neq s_{i+1}$, or such i does not exist. In the first case, $y_i \bigtriangledown_{\text{PD(A)}} (s_{i+1}, \psi_{i+1}) = \top_{\text{PD(A)}}$. For any element z in PD(A), $\top_{\text{PD(A)}} \bigtriangledown_{\text{PD(A)}} z = \top_{\text{PD(A)}}$. Thus, the chain stabilizes after finite terms. In the second case, $s_i = s_{i+1} = s$ for all $i \in \mathbb{N}$ and the proof follows from the fact that $\bigtriangledown_A$ is a widening on A. □

In order to use this widening operator in our algorithm, we replace $\texttt{saved} \sqcup_{\text{PD(A)}} r$ with $\texttt{saved} \bigtriangledown_{\text{PD(A)}} (\texttt{saved} \sqcup_{\text{PD(A)}} r)$ at line 17 of the algorithm in Figure 5.3. We refer to the resulting algorithm as BACKREACH with widening.

Consider the previous example. With widening, the result of the first iteration is computed as follows:

$$\begin{aligned} & ((q \mapsto \{1\}), (P_1.x = 5)) \bigtriangledown_{\text{PD(OCT)}} ((q \mapsto \{1\}), (5 \leq P_1.x \leq 6)) \\ = \; & ((q \mapsto \{1\}), (P_1.x = 5) \bigtriangledown_{\text{OCT}} (5 \leq P_1.x \leq 6)) \\ = \; & ((q \mapsto \{1\}), (5 \leq P_1.x)) \end{aligned}$$

The algorithm converges after a single iteration. In this case, the result happens to be the exact set of all reachable states.

Successive applications of pre-image to transitions with only local or universal global guards do not increase the number of processes in the reachable abstract global states. Therefore, systems with no existential global guards may only experience numerical divergence. In such systems, adding widening is sufficient to enforce convergence.

Theorem 20 *Let $\mathcal{P} = (Q, V, T)$ be a parameterized system with no existential transition and $\texttt{e} \in$ PD(A). The* BACKREACH$(\texttt{T}, \texttt{e})$ *with widening terminates and returns an over-approximation of the set of backward-reachable configurations from $\gamma_{\text{PD(A)}}(\texttt{e})$.*

5.5.2 Parametric Divergence

Consider the following example. Assume the abstract domain is PD(OCT). Let $\mathcal{P} = (Q, V, T)$, where $Q = \{q_1, q_2\}$, $V = \{x\}$, and $T = \{\tau\}$, where

$$\tau \;:\; q_1 \xrightarrow{g} q_2 \quad , g \;:\; \exists \texttt{other} \neq \texttt{self},\, (\texttt{other}.\mathbf{pc} = q_2) \wedge (\texttt{other}.x = \texttt{self}.x - 3)\,.$$

Let $\texttt{e} = ((q_2 \mapsto \{1\}), (2 \leq P_1.x \leq 5))$ as shown in row 1 of Table 5.2. The first iteration of BACKREACH$(\texttt{T}, \texttt{e})$ computes an AGS shown in row 2 of Table 5.2. In this iteration, a new process P_2 is added due to the existential guard. In the second iteration,

Name	Abstract Location	Constraints
1	$(q_2 \mapsto \{1\})$	$(2 \leq P_1.x \leq 5)$
2	$(q_1 \mapsto \{1\}, q_2 \mapsto \{2\})$	$(2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8)$
3	$(q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\})$	$(2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (8 \leq P_3.x \leq 11)$
4	$(q_1 \mapsto \{1,2,3\}, q_2 \mapsto \{4\})$	$(2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (8 \leq P_3.x \leq 11) \wedge$ $(11 \leq P_4.x \leq 14)$

Table 5.2: An example of a divergent computation of BACKREACH.

the algorithm computes the AGS shown in row 3 of Table 5.2, etc. The algorithm does not terminate—each iteration adds a new AGS with more processes than in any AGS seen so far.

To mitigate this, we introduce an approximation operator called k-*compact*, denoted by $\rhd_k$, where $k \in \mathbb{N}$. Given an AGS (s, ψ) where $|s| > k$, $\rhd_k$ computes an AGS (t, φ) such that $(s, \psi) \sqsubseteq_{\text{PD(A)}} (t, \varphi)$ and $|t| = k$. The operator k-compact, $\rhd_k((s, \psi))$, is implemented by: (a) choosing a process, say i, in s, (b) removing i from s, and (c) existentially projecting away all variables of the form $P_i.x$ from ψ.

Note that the choice of which process to drop affects only the precision and not the soundness of k-compact operator.

Theorem 21 *The approximation operator k-compact is sound.*

Proof: Let (s, ψ) be an element of PD(A) and $\rhd_k((s, \psi)) = (t, \varphi)$. Let $c \in \gamma_{\text{PD(A)}}((s, \psi))$. We show $c \in \gamma_{\text{PD(A)}}((t, \varphi))$. By definition of $\rhd_k$, we construct t by removing processes from s so that $|t| = k$. Thus, $s \sqsubseteq_{\text{PD}} t$. Moreover, $\rhd_k$ constructs φ by existentially projecting away all variables of the form $P_i.x$ for all processes i that were removed from s. Thus, $\psi \sqsubseteq_A \varphi$. Therefore, $(s, \psi) \sqsubseteq_{\text{PD(A)}} (t, \varphi)$. By the soundness of $\sqsubseteq_{\text{PD(A)}}$ we have $c \in \gamma_{\text{PD(A)}}((t, \varphi))$. □

To incorporate $\rhd_k$ in the BACKREACH algorithm, we apply it after the pre-image computation at line 7 of the algorithm in Figure 5.3. This ensures that the number of processes in each AGS never becomes larger than k.

Consider the previous example. Assume $k = 3$. Let ϕ denote the AGS computed in the third iteration (row 4 of Table 5.2). Assume $\rhd_3$ drops process P_3, then $\rhd_3(\phi)$ is

$$((q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\}), ((2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (11 \leq P_3.x \leq 14))) .$$

The algorithm joins this AGS with the AGS computed in the second iteration (row 3 of Table 5.2) using widening and obtains

$$((q_1 \mapsto \{1,2\}, q_2 \mapsto \{3\}), ((2 \leq P_1.x \leq 5) \wedge (5 \leq P_2.x \leq 8) \wedge (8 \leq P_3.x))) .$$

The algorithm terminates with an over-approximation of the set of reachable states.

Theorem 22 *Let $\mathcal{P} = (Q, V, T)$ be a parameterized system and* $\mathtt{e} \in \text{PD(A)}$. *The* BACKREACH$(\mathtt{T}, \mathtt{e})$ *algorithm with widening and k-compact operator always terminates and returns an over-approximation of the set of backward-reachable configurations from* $\gamma_{\text{PD(A)}}(\mathtt{e})$.

5.5.3 Lamport's Bakery Mutual-Exclusion Protocol

Figure 5.4 shows a variant of Lamport's bakery mutual-exclusion protocol (Algorithm 2 in [88]). The algorithm maintains two shared counters: *next* and *serv*, where *next* is the value of the next available ticket, and *serv* is the value of the ticket of the next process to be served. The shared variables belong to neither `self` nor `other`. We extend our framework to accommodate shared variables.

To enter the critical section, a process (i) obtains a ticket by incrementing *next* (as shown in τ_1), and storing its value in a local variable named *tick* (τ_2), (ii) picks a delay (τ_3) and spins for d steps (τ_4) and (τ_5), and (iii) enters its critical section when its ticket is being served (τ_6), i.e. its ticket value is equal to *serv*. When a process leaves the critical section, it goes back to the *idle* state and increments *serv* (τ_7).

The guards on τ_1 and τ_2 ensure that no other process changes *next* while a process is acquiring a ticket. A delay between consecutive reads of the *serv* is added to reduce

$$
\begin{array}{lcllcl}
\tau_1 & : & idle \xrightarrow{g_1} choose & , g_1 & : & \forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} \neq choose) \wedge (next' = next + 1) \\
\tau_2 & : & choose \xrightarrow{g_2} wait & , g_2 & : & \forall \texttt{other} \neq \texttt{self} : (\texttt{other}.\mathbf{pc} \neq choose) \wedge (\texttt{self}.tick' = next) \\
\tau_3 & : & wait \xrightarrow{g_3} pause & , g_3 & : & (\texttt{self}.d' = \texttt{self}.tick - serv) \\
\tau_4 & : & pause \xrightarrow{g_4} pause & , g_4 & : & ((\texttt{self}.d > 0) \Rightarrow (\texttt{self}.d' = \texttt{self}.d - 1)) \\
\tau_5 & : & pause \xrightarrow{g_5} wait & , g_5 & : & (\texttt{self}.d \leq 0) \wedge (\texttt{self}.tick > serv) \\
\tau_6 & : & pause \xrightarrow{g_6} use & , g_6 & : & (serv = \texttt{self}.tick) \wedge (\texttt{self}.d \leq 0) \\
\tau_7 & : & use \xrightarrow{g_7} idle & , g_7 & : & (next \geq serv + 1) \wedge (serv' = serv + 1)
\end{array}
$$

Figure 5.4: Lamport's bakery mutual-exclusion protocol with proportional back-off.

network contention due to the polling of the common shared variable *serv*. In [88], the authors suggest that a 'reasonable' delay is the number of processes already waiting to enter their critical section. The protocol ensures FIFO service by serving the processes in the same order in which they first requested it. The protocol is fair in a strong sense that it eliminates the possibility of starvation.

We have implemented the BACKREACH algorithm in JAVA using the APRON library for octagons abstract domain[1]. We have used this implementation to validate that the state $(idle \mapsto \{1, 2\})$ is not reachable from $(use \mapsto \{1, 2\})$. That means a state in which there are at least two processes in critical section is not reachable. The experiments were performed on a P4 3.2 GHz machine running Linux SUSE 10.3. The computation with widening converges after 56 iterations and takes 3.475 seconds. The widening is crucial for handling τ_4 that is similar to the example in the beginning of this section.

5.6 Related Work

The problem of verification of parameterized systems is, in general, undecidable [7, 103]. The undecidability results are achieved by a reduction from the non-halting problem for Turing machines. These negative results have led naturally to two approaches of analysis. One approach is to find a restricted but practically useful family of parameterized

[1]Available at `http://www.swen.uwaterloo.ca/~nghafari/AIPMCTool`

systems for which verification is decidable. To this end, many authors heavily restrict both the systems and the correctness properties to achieve decidability; [51, 55, 62] are representative examples. Another approach is to develop sound but incomplete analysis techniques. Examples of such techniques are network invariants [84] and abstraction and approximation of network invariants [39]. The success of these methods depends on the heuristic used in the generation of candidate invariant.

Some approaches target particular applications. For example, [10] presents techniques for finite-state abstraction for the verification of systems specified in WS1S and [80] provides heuristics to discover *indexed predicates* and applies them to German's protocol as well as to the bakery algorithm.

Let P^n represent the parameterized system $P_0||P_1||\ldots||P_{n-1}$, produced by asynchronous composition of n isomorphic copies of a process P. A fundamental concept is that the behavior of arbitrary instances of such a system can be summarized by a finite-state process. The *closure* process, introduced in [38, 37], is precisely such a summary. While the approach in [38] requires the manual construction of the bisimulation relations between global state graphs of systems of different sizes, [37] attempts to avoid the problem of having to explicitly construct the bisimulation relations. However, it still needs some creativity to construct the closure process.

Among the representatives of the first approach, we can count the work of German and Sistla [62] which considers a parameterized system, where processes communicate synchronously in a complete network. They use automata-theoretic methods to construct process closures and use them to establish single-index properties. Multi-index properties can be indirectly catered for, but the complexity becomes multi-exponential.

The *process invariant* method, introduced in [79, 106], requires that the summarizing process, I, is also an invariant, in the sense that the behavior of both P and $P||I$ is simulated by I. This method is not fully automated because an invariant process must be provided. Often, an invariant process may be obtained by composing a small number of the system processes, then making minor modifications.

The *cutoff* method, implicit in [79, 62] and made explicit in [55], requires that the

process invariant be a union of instances up to a (typically small) cutoff bound K, i.e. $I = P^1 + P^2 + \cdots + P^K$. The cutoff method formalizes the verifier's intuition that all interesting patterns of behavior are already present in instances of a small size. Emerson and Namjoshi [55] provide a decision procedure for proving a class of properties in the parameterized ring networks which was later extended to fully connected networks [41]. Emerson and Kahlon [51, 52, 53, 54] consider a general parameterized system allowing several classes of processes in which the transitions are labeled by conjunctive or disjunctive guards.

Among the sound but incomplete approaches, methods using *regular model-checking* [75, 76] deserve a special mention. In such a framework, the states of a system are modeled as words in a regular language and the transition relations are described by finite-state transducers. Although regular model-checking can model parameterized systems, successful analysis requires special acceleration techniques. Such acceleration often demands user intervention and ingenuity. In contrast, our approach uses widening to enforce convergence—it is automated and does not require any user guidance.

The sound but incomplete methods also include the work of [99, 40, 39]. Shtadler and Grumberg [99] use a network grammar to specify a communication topology. They show that if certain sufficient conditions are satisfied, then, every network generated by the grammar satisfies specifications written in the linear temporal logic. The sufficiency check, however, may require a time exponential in the size of an individual process. The work in [99] has been extended to global safety properties in [40, 39], which present a construction of an invariant process whose states are regular expressions denoting global states of arbitrarily large instances. These approaches are fully automated but do not appear to have a clearly defined class of systems in which they are guaranteed to succeed.

Abdulla and Jonsson [6] consider the case of 1-clock timed systems. They show that the verification of a class of safety properties is decidable under some restrictions on the constraints used. Inspired by [6], Bozzano and Delzanno [23] present a safety verification technique for parameterized systems with unbounded *local* data variables. Their approach is based on assertions that combine multiset rewriting over first order

formulas and constraints. Decidability is achieved by restricting constraints to a constant-free subclass of *difference constraints* (itself a subclass of linear arithmetic). Similar techniques to [23] have been applied in [49] to verify client-server protocols, like *home-based consistency protocols*, i.e. protocols designed for systems with a central monitor that serializes the accesses to shared data. For this class of protocols, [49] considers abstract models built via counters and global Boolean variables used as guards in the corresponding transitions. In order to symbolically handle global states for this kind of models, [49] used a combination of Presburger and Boolean constraints incorporated in a backward reachability algorithm [30]. In [3], the method of [23] is extended to GAP constraints. GAP constraints are linear constraints of the form: $x = y$, $x \leq y$, or $x + k < y$, where x and y are variables and k is a *positive* constant.

The method of [23] is generalized into an analysis framework in [20, 17] by using a constrained (multiset) rewriting system on words over an infinite alphabet. In this framework, each configuration is composed from a label over a finite set of symbols and a vector of data in a potentially infinite domain. The constraints are expressed in a logic that is an extension of a monadic first order theory of the natural ordering on positive integers (corresponding to positions on the word). This logic is also parameterized with a first order theory on the considered data domain such as Presburger arithmetics. In [20, 17] the authors present decidability results for satisfiability of a particular fragment of this logic. They also prove that this fragment is closed under the computations of post and pre images. This result together with the decidability of the satisfiability problem can be used for deciding whether a given assertion is an inductive invariant of a system.

In this thesis, we propose an alternative framework to multiset rewriting framework of [20, 17]. In our framework, we deligate the reasoning about constraints to abstract interpretation. The advantage is two-fold. First, our technique can use any constraints for which there are efficient abstract domains available. Second, to a large extend, the termination of the analyis is guaranteed by the widenning operator of the abstract domain.

Many of the techniques above are based on *counter abstraction* (e.g. [86, 95, 42]). The main idea is to keep track of the (upper bound of the) number of processes that

satisfy a certain property. For example, this could be the number of processes in the critical section. To ensure that the abstract system is finite-state, the work of [95] restricts the value of counters to either 0, 1, or infinity. In [42], counter abstraction and predicate abstraction are combined together to achieve more flexibility. However, the system model is more restrictive than ours. Our abstract domain PD can be seen as a *variant* of counter abstraction that maintains the *lower* bound on the number of processes satisfying a certain condition.

In contrast to symbolic methods for finite collections of processes with local integer variables [31], our abstract domains are defined over an unbounded collection of variables. The number of variables during the backward-search is not bounded a priori. This allows us to reason about systems with global conditions over any number of processes.

5.7 Conclusion

In this chapter, we presented a framework for the analysis of safety properties of parameterized systems where each of the individual processes may be infinite-state. We introduced a new abstract domain for the parameterized systems of infinite-state processes. Each element of our abstract domain (a) represents the lower bound on the number of processes at a particular control location and (b) employs a numeric abstract domain to capture arithmetic relations among variables of the processes. We described an algorithm that over-approximates backward-reachability in parameterized systems. We combine numeric widening with an extrapolation operator developed for our abstract domain to enforce sound termination of our algorithm. We illustrated our technique by automatically establishing the mutual-exclusion property in a variant of Lamport's bakery protocol.

Chapter 6

Conclusion

In this chapter, we summarize the contributions made in this thesis and outline directions for future research.

6.1 Summary of the Thesis

The main topic of this thesis is development of algorithmic verification techniques for analysis of software systems. We have presented techniques for algorithmic analysis of two important classes of infinite-state systems: FIFO systems and parameterized systems. FIFO systems, consisting of a set of finite-state machines that communicate via unbounded, perfect, FIFO channels, arise naturally in the analysis of distributed protocols such as communication and routing protocols. Parameterized systems, on the other hand, are a common model of computation for concurrent systems consisting of an arbitrary number of homogenous processes, for example cache coherence and resource sharing protocols.

In Chapter 3, we showed that piecewise languages play an important role in the study of FIFO systems. We have studied the reachability problem for a class of FIFO systems composed of piecewise components. We showed that this problem is reducible to computing all possible channel contents that may arise from an initial state, i.e. the limit

language. For multi-channel piecewise systems, we show that the limit language is not regular, in general. However, we are able to establish the regularity of the limit language by excluding the conditional actions from the actions allowed by piecewise components. We further showed that in the presence of conditional actions, the limit language is piecewise if the initial channel language is piecewise. However, the construction of the limit language may not always be effective.

In Chapter 4, we presented algorithms for computing the limit language in piecewise FIFO systems. We considered single-channel and multi-channel systems separately. For the single-channel case, we presented a new automata-theoretic algorithm for calculating the limit language starting with an arbitrary regular initial content. We showed that the worst case complexity of our algorithm is exponential in the size of the automaton representing the initial channel content. For multi-channel systems, we presented an automata-theoretic algorithm for computing the limit language subject to the following conditions: (i) the initial language is piecewise, and (ii) the communication graph of actions is acyclic. For star and tree topologies we showed that the complexity of our algorithm is exponential in the size of the automaton representing the initial channel configuration. In addition, we demonstrated that important subclasses of multi-channel piecewise FIFO systems, can be described by a finite-state, abridged structure, representing an expressive abstraction of the system. We presented a procedure for building the abridged model and proved that this procedure terminates. Furthermore, we showed that we can analyze the infinite computations of the more concrete model by analyzing the computations of the finite, abridged model.

In Chapter 5, we studied the reachability problem in parameterized systems of infinite-state processes. We presented a framework that combines Abstract Interpretation with a backward-reachability algorithm. Our main contribution is an abstract domain in which each element (a) represents the lower bound on the number of processes at a control location and (b) employs a numeric abstract domain to capture arithmetic relations among variables of the processes. We described an algorithm that over-approximates backward-reachability. We provided an extrapolation operator for our abstract domain to guar-

antee sound termination of the backward-reachability algorithm. Our abstract domain is generic enough to be instantiated by different well-known numeric abstract domains such as octagons and polyhedra. This makes the framework applicable to a wide range of parameterized systems. We illustrated an implementation of our algorithm on a variant of Lamport's bakery mutual-exclusion protocol.

6.2 Future Work

In this section, we outline some of the future directions for the work presented in this thesis.

6.2.1 Automated Analysis of FIFO Systems

In this thesis, we showed that piecewise FIFO systems are analyzable. In particular, in Section 3.4, we demonstrated that the limit language of the multi-channel piecewise FIFO systems is piecewise if the initial language is piecewise. However, the proof of Proposition 13 (refer to Section 3.4) does not provide us with any algorithm for calculating the limit language in a general case. Therefore, computation of the limit language in multi-channel piecewise FIFO systems with cyclic communication graphs is still an open problem.

Analysis of single-channel systems provides us with some insights on how to address the issues in multi-channel piecewise systems with cyclic communication graphs. In the single-channel systems (as shown in Section 4.2), we were able to compute the limit language with the presence of conditional actions that read and write on the same channel. We believe that by using similar automata-theroetic techniques, we can compute the limit languages in those multi-channel systems that have cyclic communication graphs in which the only loops that are allowed are the self-loops on the vertices of the graph. This will allow us to algorithmically analyze more general subclasses of multi-channel FIFO systems.

Another research direction that would be interesting to explore is the applicability of our analysis techniques to other distributed protocols. Recall that in Chapter 3, we showed that piecewise languages can be used to express descriptions of IP-telephony features in BoxOS, the next generation telecommunication service over IP developed at AT&T Research [74]. We believe that piecewise languages are amenable to describing a broader range of distributed protocols, such as session-oriented software, composite web services, and routing protocols. Currently, we are applying our analysis techniques to the verification of composite web services specified in Business Process Execution Language (BPEL) [72]. Interestingly, the behavior of peers in such services can be expressed by piecewise languages.

Bultan and Fu in [28] investigated the realizability of *collaboration* diagrams. A collaboration diagram specifies the set of allowable conversations among the peers participating in a composite web service. The authors show that if a collaboration diagram is realizable, they can check properties about its conversations by generating a conversation protocol (over unbounded, perfect, FIFO channels) from the collaboration diagram. However, they are not able to check properties that refer to the states of the peers. We believe that we can use our algorithms for checking reachability and computation path properties of collaboration diagrams. In particular, our result may allow one to check the state and conversation properties of any realizable collaboration diagrams.

6.2.2 Parameterized Systems Verification

In Chapter 5, we developed a framework that combines Abstract Interpretation with a backward-reachability algorithm for verifying safety properties of parameterized systems of infinite-state processes. In Section 5.5, we provided an extrapolation operator to guarantee the sound termination of our algorithm. It is interesting to consider other possible operators like k-compact that increase the precision of approximation by choosing the process to drop based upon heuristics derived from the features of the analyzed system. With a more precise extrapolation operator, we will be able to get more conclusive results from our analysis.

In the future, we intend to develop a tool based on the implementation of our backward-reachability algorithm that can employ implementations of different numeric abstract domains. It will also be interesting to investigate a range of resource sharing and cache coherence protocols to which our analysis framework is applicable.

In this thesis, we considered parameterized systems in which each process manipulates numeric variables. We are interested in extending our analysis framework to parameterized systems where the processes manipulate unbounded data structures. We believe this continues to be a very interesting research direction since it broadens the applicability of our framework to a wide range of concurrent systems such as software written based on Software Transactional Memory (STM) principles [82]. STM is a programming abstraction intended to simplify the synchronization of conflicting memory accesses in concurrent software without the headaches associated with locks. At a sufficient level of abstraction, STM can be modeled as a parameterized system where each process manipulates unbounded data structures [43]. Most of the research on STM has been focused on performance issues and little work has been done on the issue of reliability and verification. It has been argued that STM should facilitate the task of ensuring reliability of the concurrent software, but little evidence has been provided for this claim.

References

[1] P. A. Abdulla, A. Annichini, and A. Bouajjani. "Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol". In *Proc. of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *LNCS*, pages 208–222, 1999. 5, 29, 30, 49

[2] P. A. Abdulla, N. Bertrand, A. Rabinovich, and Ph. Schnoebelen. "Verification of Probabilistic Systems with Faulty Communication". *Information and Computation*, 202(2):141–165, 2005. 49

[3] P. A. Abdulla, G. Delzanno, and A. Rezine. "Parameterized Verification of Infinite-State Processes with Global Conditions". In *Proc. of the19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 145–157, 2007. 6, 90, 115

[4] P. A. Abdulla and B. Jonsson. "Verifying Programs with Unreliable Channels". In *Proc. of the 8th IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 160–170, 1993. 30, 49

[5] P. A. Abdulla and B. Jonsson. "Undecidable Verification Problems for Programs with Unreliable Channels". In *Proc. of the 21st International Colloquium on Automata, Languages and Programming (ICALP'94)*, volume 820 of *LNCS*, pages 316–327, 1994. 49

[6] P. A. Abdulla and B. Jonsson. "Verifying Networks of Timed Processes (Extended Abstract)". In *TACAS'98*, volume 1384 of *LNCS*, pages 298–312, 1998. 114

[7] K. R. Apt and D. C. Kozen. "Limits for Automatic Verification of Finite-State Concurrent Systems". *Information Processing Letters*, 22(6):307–309, 1986. 6, 8, 88, 112

[8] C. Baier and B. Engelen. "Establishing Qualitative Properties for Probabilistic Lossy Channel Systems: An Algorithmic Approach". In *Proc. of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 34–52, 1999. 49

[9] T. Ball and S. K. Rajamani. "The SLAM Toolkit". In *Proc. of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 260–264, 2001. 3, 4

[10] K. Baukus, Y. Lakhnech, and K. Stahl. "Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness". In *Proc. of the 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'02*, pages 317–330, 2002. 113

[11] I. Beer, S. Ben-David, C Eisner andD. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. "RuleBase: Model Checking at IBM". In *Proc. of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 480–483, 1997. 3

[12] B. Boigelot. *"Symbolic Method for Exploring Infinite States Spaces"*. PhD thesis, Université de Liège, 1998. 30, 50, 51

[13] B. Boigelot and P. Godefroid. "Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs". *Formal Methods in System Design*, 14(3):237–255, 1999. 30, 48, 50, 51

[14] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. "The Power of QDDs". In *Proc. of the 4th International Symposium on Static Analysis (SAS'97)*, volume 1302 of *LNCS*, pages 172 – 186, 1997. 5, 29, 30, 48, 50, 51

[15] B. Boigelot and P. Wolper. "Symbolic Verification with Periodic Sets". In *Proc. of the 6th International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 55–67, 1994. 50

[16] G. W. Bond, F. Ivančić, N. Klarlund, and R. Trefler. "ECLIPSE Feature Logic Analysis". In *Proc. of the Second IP Telephony Workshop*, 2001. 5, 29, 31, 32

[17] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. "Rewriting Systems with Data". In *FCT'07*, volume 4639 of *LNCS*, pages 1–22, 2007. 115

[18] A. Bouajjani, P. Habermehl, and T. Vojnar. "Verification of Parametric Concurrent Systems with Prioritized FIFO Resource Management". In *Proc. of the 14th International Conference on Concurrency Theory (CONCUR'03)*, volume 2761 of *LNCS*, pages 172–187, 2003. 6

[19] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. "Regular Model Checking". In *Proc. of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 403–418, 2000. 30

[20] A. Bouajjani, Y. Jurski, and M. Sighireanu. "A Generic Framework for Reasoning About Dynamic Networks of Infinite-State Processes". In *TACAS'07*, volume 4424 of *LNCS*, pages 690–705, 2007. 115

[21] A. Bouajjani, A. Muscholl, and T. Touili. "Permutation Rewriting and Algorithmic Verification". In *Proc. of the 16th IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 399 – 408, 2001. 13, 14, 15, 30

[22] R. S. Boyer and J. S. Moore. *"A Computational Logic Handbook"*. Academic Press Professional, 1988. 2

[23] M. Bozzano and G. Delzanno. "Beyond Parameterized Verification". In *TACAS'02*, volume 2280 of *LNCS*, pages 221–235, 2002. 114, 115

[24] D. Brand and P. Zafiropulo. "On Communicating Finite-State Machines". *Journal of the ACM*, 30(2):323–342, 1983. 5, 29, 30, 48

[25] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers*, 35(8):677– 691, 1986. 4

[26] A. Brzozowski and I. Simon. "Characterization of Locally Testable Events". *Discrete Mathematics*, 4:243–271, 1973. 12

[27] J. R. Buchi. "On a Decision Method in Restricted Second Order Arithmetic". In *Proc. of International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11, 1960. 83, 84

[28] T. Bultan and X. Fu. "Specification of Realizable Service Conversations Using Collaboration Diagrams". *Service Oriented Computing and Applications*, 2(1):27–39, 2008. 120

[29] T. Bultan, X. Fu, R. Hull, and J. Su. "Conversation Specification: A New Approach to Design and Analysis of e-Service Composition". In *Proc. of the 12th international conference on World Wide Web (WWW'03)*, pages 403–410, 2003. 5, 29

[30] T. Bultan, R. Gerber, and C. League. "Composite Model-Checking: Verification with Type-Specific Symbolic Representations". *ACM Transaction on Software Engineering Methodology*, 9(1):3–50, 2000. 115

[31] T. Bultan, R. Gerber, and W. Pugh. "Model-Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations, and Experimental Results". *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999. 116

[32] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". *Information and Computation*, 98(2):142 – 170, 1992. 3, 4

[33] G. Cece, A. Finkel, and S. P. Iyer. "Unreliable Channels are Easier to Verify than Perfect Channels". *Information and Computation*, 124(1):20–31, 1996. 5, 7, 29, 30, 49, 50

[34] E. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 168–176, 2004. 4

[35] E. M. Clarke and E.A. Emerson. "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic". In *Logic of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71, 1981. 2, 3, 4

[36] E. M. Clarke, E.A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244 – 263, 1986. 3

[37] E. M. Clarke and O. Grumberg. "Avoiding the State Explosion Problem in Temporal Logic Model Checking". In *Proc. of the 6th ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 294–303, 1987. 88, 113

[38] E. M. Clarke, O. Grumberg, and M. C. Browne. "Reasoning about Networks with Many Identical Finite-State Processes". In *Proc. of the 5th ACM Symposium on Principles of Distributed Computing (PODC'86)*, pages 240–248, 1986. 88, 113

[39] E. M. Clarke, O. Grumberg, and S. Jha. "Verifying Parameterized Networks using Abstraction and Regular Languages". In *Proc. of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 395–407, 1995. 113, 114

[40] E. M. Clarke and S. Jha. Symmetry and induction in model checking. In *Computer Science Today*, volume 1000 of *LNCS*, pages 455–470. 1995. 114

[41] E. M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *Proc. of the 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 276–291, 2004. 114

[42] E. M. Clarke, M. Talupur, and H. Veith. "Environment Abstraction for Parameterized Verification". In *Proc. of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 126–141, 2006. 115, 116

[43] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. "Verifying Correctness of Transactional Memories". In *Proc. of the Formal Methods in Computer Aided Design (FMCAD'07)*, pages 37–44, 2007. 6, 121

[44] P. Cousot and R. Cousot. "Static Determination of Dynamic Properties of Programs". In *Proc. of the 2nd International Symposium on Programming*, pages 106–130, 1976. 93

[45] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proc. of the 4th Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977. 20, 25

[46] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". *Journal of Logic and Computation*, 2(4):511–547, 1992. 10, 20, 25, 88, 96

[47] P. Cousot and R. Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation". In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 269–295, 1992. 28

[48] P. Cousot and N. Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In *Proc. of the 5th ACM Symposium on Principles of Programming Languages (POPL’78*, pages 84–97, 1978. 9, 24, 87, 88, 93, 94

[49] G. Delzanno and T. Bultan. “Constraint-Based Verification of Client-Server Protocols”. In *Proc. of the 7th International Conference on Principles and Practice of Constraint Programming (CP’01)*, volume 2239 of *LNCS*, pages 286–301, 2001. 6, 115

[50] M. Dowson. “The Ariane 5 Software Failure”. *Software Engineering Notes*, 22(2):84, 1997. 1

[51] E. A. Emerson and V. Kahlon. “Reducing Model Checking of the Many to the Few”. In *Proc. of the 17th International Conference on Automated Deduction (CADE’00)*, volume 1831 of *LNCS*, pages 236–254, 2000. 6, 88, 113, 114

[52] E. A. Emerson and V. Kahlon. Model checking large-scale and parameterized resource allocation systems. In *Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’02)*, volume 2280 of *LNCS*, pages 251–265, 2002. 114

[53] E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *Proc. of the 18th IEEE Symposium on Logic in Computer Science (LICS’03)*, pages 361–370, 2003. 6, 114

[54] E. A. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’03)*, volume 2619 of *LNCS*, pages 144–159, 2003. 6, 114

[55] E. A. Emerson and K. S. Namjoshi. “On Reasoning about Rings”. In *Proc. of the 22nd ACM Symposium on Principles of Programming Languages (POPL’95)*, pages 85–94, 1995. 6, 88, 113, 114

[56] E. A. Emerson and K. S. Namjoshi. “Verification of Parameterized Bus Arbitration Protoco”. In *Proc. of the 10th International Conference on Computer Aided Verification (CAV’98)*, volume 1427 of *LNCS*, pages 452–463, 1998. 6

[57] A. Finkel. “Decidability of the Termination Problem for Completely Specified Protocols”. *Distributed Computing*, 7(3):129–135, 1994. 49

[58] A. Finkel, S. P. Iyer, and G. Sutre. “Well-abstracted transition systems: application to FIFO automata”. *Information and Computation*, 181(1):1–31, 2003. 50

[59] A. Finkel and L. Rosier. “A Survey on the Decidability Questions for Classes of FIFO Nets”. In *Advances in Petri Nets 1988*, volume 340 of *LNCS*, pages 106–132. 1988. 48, 50

[60] X. Fu, T. Bultan, and J. Su. “Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services”. In *Proc. of the 8th International Conference on Implementation and Application of Automata (CIAA’03)*, volume 2759 of *LNCS*, pages 188–200, 2003. 5, 29

[61] M. K. Ganai and A. Gupta. “Tunneling and Slicing: Towards Scalable BMC”. In *Proc. of the 45th Design Automation Conference (DAC’08)*, pages 137–142, 2008. 4

[62] S. M. German and A. P. Sistla. “Reasoning about Systems with Many Processes”. *Journal of the ACM*, 39(3):675–735, 1992. 88, 113

[63] N. Ghafari, A. Gurfinkel, N. Klarlund, and R. J. Trefler. “Algorithmic Analysis of Piecewise FIFO Systems”. In *Proc. of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD’07)*, pages 45–52, 2007. 7

[64] N. Ghafari, A. Gurfinkel, N. Klarlund, and R. J. Trefler. “Algorithmic Analysis of Piecewise FIFO systems”. *submitted to Formal Methods in System Design*, 2008. 7

[65] N. Ghafari, A. Gurfinkel, and R. J. Trefler. “Verification of Parameterized Systems with Combinations of Abstract Domains”. 2008 (submitted for publication). 9

[66] N. Ghafari and R. J. Trefler. “Piecewise FIFO Channels Are Analyzable”. In *Proc. of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’06)*, volume 3855 of *LNCS*, pages 252–266, 2006. 6, 8, 31

[67] N. Halbwachs. “On the Design of Widening Operators”. In *Proc. of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’06)*, volume 3855 of *LNCS*, 2006. 28

[68] K. Havelund and N. Shankar. “Experiments in Theorem Proving and Model Checking for Protocol Verification”. In *Proc. of the 3rd International Symposium of Formal Methods (FME’96)*, volume 1051 of *LNCS*, pages 662–681, 1996. 2

[69] G. Higman. “Ordering by divisibility in abstract algebras”. *Proceedings of the London Mathematical Society*, 2(7):326–336, 1952. 42

[70] G.J. Holzmann. “The Model Checker SPIN”. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. 3

[71] J. E. Hopcroft and J. D. Ullman. *“Introduction to Automata Theory, Languages and Computation”*. Addison-Wesley, 1979. 10, 11

[72] IBM. *“Business Process Execution Language for Web Services (BPEL) Version 1.1”*, 2007. Available from `http://www-128.ibm.com/developerworks/library/specification/ws-bpel`. 31, 120

[73] S. P. Iyer and M. Narasimha. “Probabilistic Lossy Channel Systems”. In *Proc. of the 7th International Joint Conference on Theory and Practice of Software Development (TAPSOFT’97)*, volume 1214 of *LNCS*, pages 667–681, 1997. 49

[74] M. Jackson and P. Zave. "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services". *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998. 31, 120

[75] B. Jonsson and M. Nilsson. "Transitive Closures of Regular Relations for Verifying Infinite-state Systems". In *Proc. of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 220–234, 2000. 114

[76] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. "Symbolic Model Checking with Rich Assertional Languages". In *Proc. of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 424–435, 1997. 114

[77] N. Klarlund and R. J. Trefler. "Regularity Results for FIFO Channels". *Electronic Notes in Theoretical Computer Science*, 128(6):21–36, 2005. 6, 7, 13, 48

[78] J. B. Kruskal. "The Theory of Well-Quasi-Ordering: A Frequently Discovered Concept". *Journal of Combinatorial Theory, Series A*, 13(3):297–305, 1972. 41

[79] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proc. of the 8th Annual ACM Symposium on Principles of Distributed Computing (PODC'89)*, pages 239–247, 1989. 113

[80] S. K. Lahiri and R. E. Bryant. "Indexed Predicate Discovery for Unbounded System Verification". In *Proc. of the 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 135–147, 2004. 113

[81] L. Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". *Communication of ACM*, 17(8):453–455, 1974. 9, 88

[82] J. R. Larus and R. Rajwar. *"Transactional Memory"*. Morgan & Claypool, 2006. 121

[83] T. Legall, B. Jeannet, and T. Jron. "Verification of Communication Protocols Using Abstract Interpretation of FIFO Queues". In *Proc. of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST'06)*, volume 4019 of *LNCS*, 2006. 48

[84] D. Lesens, N. Halbwachs, and P. Raymond. "Automatic Verification of Parameterized Linear Networks of Processes". In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 346–357, 1997. 113

[85] N. G. Leveson and C. S. Turner. "Investigation of the Therac-25 Accidents". *IEEE Computer*, 26(7):18–41, 1993. 1

[86] B. D. Lubachevsky. "An Approach to Automating the Verification of Compact Parallel Coordination Programs". *Acta Informatica*, 21:125–169, 1984. 115

[87] K. L. McMillan. *"Symbolic Model Checking"*. Kluwer Academic Publishers, 1993. 3, 4

[88] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems*, 9(1):21–65, 1991. 9, 89, 111, 112

[89] A. Miné. "The Octagon Abstract Domain". *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. 9, 24, 87, 88, 93

[90] A. Møller. "An Automaton/Regular Expression Library for Java". Available at `http://www.brics.dk/automaton`, 2007. 85

[91] D. E. Muller. "Infinite sequences and finite machines". In *Proc. of the 4th Ann. IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 3 – 16, 1963. 84

[92] K. S. Namjoshi. "Symmetry and Completeness in the Analysis of Parameterized Systems". In *Proc. of the 8th International Conference on Verification, Model*

Checking, and Abstract Interpretation (VMCAI'07), volume 4349 of *LNCS*, pages 299 – 313, 2007. 88

[93] J. K. Pachl. "Protocol Description and Analysis Based on a State Transition Model with Channel Expressions". In *Proc. of the 7th International Conference on Protocol Specification, Testing and Verification*, pages 207–219, 1987. 5, 29, 30, 48, 49

[94] A. Pnueli. "The Temporal Logic of Programs". In *Proc. of the 18th IEEE Symposium on Foundation of Computer Science*, pages 46–67, 1977. 3

[95] A. Pnueli, J. Xu, and L. D. Zuck. "Liveness with (0, 1, ∞)-Counter Abstraction". In *Proc. of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 107–122, 2002. 115, 116

[96] J. P. Queille and J. Sifakis. "Specification and Verification of Concurrent Programs in CESAR". In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, pages 195–220, 1982. 2, 3

[97] M. O. Rabbin. "Decidability of Second-Order Theories and Automata on Infinite Trees". *Transactions of the American Mathematical Society*, 141:1–35, 1969. 84

[98] A. Roychoudhury and I. V. Ramakrishnan. "Automated Inductive Verification of Parameterized Protocols". In *Proc. of the 13th International Conference on Computer Aided Verification (CAV'01)*, pages 25–37, 2001. 88

[99] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Proc. of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 151–165, 1990. 114

[100] A. P. Sistla. *"Theoretical Issues in The Design and Verification of Distributed Systems"*. PhD thesis, Harvard University, 1983. 3

[101] A. P. Sistla and E. M. Clarke. "The complexity of propositional linear temporal logics". *Journal of the ACM*, 32(3):733–749, 1985. 3

[102] A. P. Sistla and L. D. Zuck. "Automatic Temporal Verification of Buffer Systems". In *Proc. of the 3rd International Workshop on Computer Aided Verification (CAV'91)*, pages 59–69, 1991. 48

[103] I. Suzuki. Proving properties of a ring of finite-state machines. *Information Processing Letters*, 28(4):213–214, 1988. 112

[104] M. Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". In *Proc. of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332–344, 1986. 83

[105] P. Wodey, G. Camarroque, F. Baray, R. Hersemeule, and J.-P. Cousin. "LOTOS Code Generation for Model Checking of STBus Based SoC: the STBus interconnect". In *Proc. of the 1st ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2003)*, 2003. 5, 29

[106] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proc. of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1990. 113

[107] S. Yu. "Regular Languages". In Rozenberg and Salomaa, editors, *Handbook of Language Theory, Vol. I*. Springer Verlag, 1997. 13