# Abstraction for Verification and Refutation in Model Checking

by

Ou Wei

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

# Abstract

Abstraction for Verification and Refutation

in Model Checking

Ou Wei

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2009

Model checking is an automated technique for deciding whether a computer program satisfies a temporal property. Abstraction is the key to scaling model checking to industrial-sized problems, which approximates a large (or infinite) program by a smaller abstract model and lifts the model checking result over the abstract model back to the original program. In this thesis, we study abstraction in model checking based on *exact-approximation*, which allows for verification and refutation of temporal properties within the same abstraction framework. Our work in this thesis is driven by problems from both practical and theoretical aspects of exact-approximation.

We first address challenges of effectively applying symmetry reduction to *virtually* symmetric programs. Symmetry reduction can be seen as a *strong* exact-approximation technique, where a property holds on the original program if and only if it holds on the abstract model. In this thesis, we develop an efficient procedure for identifying virtual symmetry in programs. We also explore techniques for combining virtual symmetry with symbolic model checking.

Our second study investigates model checking of *recursive* programs. Previously, we have developed a software model checker for non-recursive programs based on exact-approximating predicate abstraction. In this thesis, we extend it to reachability and non-termination analysis of recursive programs. We propose a new program semantics that effectively removes call stacks while preserving reachability and non-termination. By doing this, we reduce recursive analysis

to non-recursive one, which allows us to reuse existing abstract analysis in our software model checker to handle recursive programs.

A variety of *partial* transition systems have been proposed for construction of abstract models in exact-approximation. Our third study conducts a systematic analysis of them from both semantic and logical points of view. We analyze the connection between semantic and logical consistency of partial transition systems, compare the expressive power of different families of these formalisms, and discuss the precision of model checking over them.

Abstraction based on exact-approximation uses a uniform framework to prove correctness and detect errors of computer programs. Our results in this thesis provide better understanding of this approach and extend its applicability in practice.

*To my parents.*

# Acknowledgements

I am heartily grateful to my supervisor, Professor Marsha Chechik, for her guidance during my Ph.D. study. Marsha always encouraged me to explore research topics that I am interested in, and provided me with many valuable suggestions. Without her continued encouragement and support, it would have been impossible for me to finish this thesis.

I would like to express my gratitude to Dr. Arie Gurfinkel for collaboration. He helped me a lot on my research, and also shared with me his knowledge of many interesting things other than model checking, e.g., snow skiing (although I am still a beginner level skier).

I am deeply grateful to Professors Eric Hehner, Steve Easterbrook, Richard Trefler, and Azadeh Farzan for being the members of my Ph.D. committee. Their advice is very helpful for my work.

It is an honor to have Professor Michael Huth as my external reviewer. He read this thesis very carefully, pointed out related work, and provided useful comments.

I am grateful to the System Analysis and Verification Group at NEC Laboratories for the valuable internship experience in static program analysis.

Thanks to the students of the Formal Methods Group, past and present, who made the graduate study fun.

Many thanks to my friends at the University of Toronto. Their friendship has accompanied me through all those years.

Finally, my thanks to my mother, my father, and Yun for their love and support.

# Contents


**1 Introduction** **1**

1.1 Model Checking and Abstraction . . . . . 1

1.1.1 Model Checking . . . . . 1

1.1.2 Fighting State-Explosion Problem . . . . . 2

1.1.3 Abstraction . . . . . 3

1.2 Scope of This Thesis . . . . . 8

1.2.1 Strong Exact-Approximation . . . . . 9

1.2.2 Weak Exact-Approximation . . . . . 11

1.3 Contributions of This Thesis . . . . . 14

1.3.1 Full Virtual Symmetry Reduction . . . . . 15

1.3.2 Reachability and Non-Termination Analysis of Recursive Programs . . 17

1.3.3 Analysis of Partial Modeling Formalisms . . . . . 18

1.4 Organization . . . . . 19

**2 Background** **20**

2.1 Truth Domains . . . . . 20

2.2 Model Checking . . . . . 21

2.2.1 Temporal Logics . . . . . 21

2.2.2 Models of Computation . . . . . 24

2.3 Abstraction Framework . . . . . 27

2.3.1 Strong Exact-Approximation . . . . . 29
2.3.2 Weak Exact-Approximation . . . . . 31
2.4 Summary . . . . . 38

**3 Full Virtual Symmetry Reduction 39**

3.1 Introduction . . . . . 39
3.2 Preliminaries . . . . . 41
3.3 Abstraction and Virtual Symmetry . . . . . 43
3.4 Specification Language . . . . . 45
3.4.1 Specifying Symmetric Programs . . . . . 45
3.4.2 Specifying Asymmetric Programs . . . . . 47
3.5 Identification of Full Virtual Symmetry . . . . . 48
3.5.1 Full Virtual Symmetry in Asynchronous Transition Systems . . . . . 49
3.5.2 Constraint-Based Identification of Full Virtual Symmetry . . . . . 53
3.6 Counter Abstraction for Full Virtual Symmetry . . . . . 55
3.7 Experiments . . . . . 60
3.8 Related Work . . . . . 62
3.9 Conclusion . . . . . 63

**4 Reachability and Non-Termination Analysis of Recursive Programs 66**

4.1 Introduction . . . . . 66
4.2 Preliminaries . . . . . 70
4.3 Programming Language and Semantics . . . . . 71
4.4 Mixed Semantics . . . . . 76
4.5 On-the-Fly Reachability and Non-Termination . . . . . 84
4.5.1 On-the-Fly Reachability . . . . . 84
4.5.2 On-the-Fly Non-Termination . . . . . 89
4.6 Abstract Analysis . . . . . 94

4.6.1 Abstract Domains and Operations . . . . . 94
4.6.2 Predicate Abstraction . . . . . 95
4.7 Experiments . . . . . 98
4.8 Related Work . . . . . 101
4.9 Conclusion . . . . . 103

**5 Analysis of Partial Modeling Formalisms 105**

5.1 Introduction . . . . . 105
5.2 Preliminaries . . . . . 108
5.3 Monotone Partial Transition Systems . . . . . 111
5.4 Consistency . . . . . 115
5.4.1 Logical and Semantic Consistency for Consistent Statespaces . . . . . 115
5.4.2 Logical and Semantic Consistency for Arbitrary Statespaces . . . . . 121
5.5 Expressiveness . . . . . 123
5.5.1 GTOM: Translation from GKMTSs to MixTSs . . . . . 123
5.5.2 MTOK: Translation from MixTSs to KMTSs . . . . . 126
5.6 Reduced Inductive Semantics . . . . . 131
5.6.1 Example . . . . . 132
5.6.2 Reduced Inductive Semantics for Partial Models . . . . . 133
5.6.3 Reduced Inductive Semantics for Monotone Models . . . . . 138
5.7 Symbolic Model Checking of RIS using BDDs . . . . . 140
5.8 Experiments . . . . . 145
5.9 Related Work . . . . . 149
5.10 Conclusion . . . . . 153

**6 Conclusion 155**

6.1 Summary of The Thesis . . . . . 155
6.2 Limitations and Future Work . . . . . 157

6.2.1 Extended Symmetry Reduction . . . . . 157
6.2.2 Termination and Non-Termination Analysis . . . . . 158
6.2.3 Partial Modeling Formalisms . . . . . 160
6.2.4 Combination of Abstraction and Testing . . . . . 160

**Bibliography** **162**

# List of Tables


3.1 A mapping between abstraction and symmetry reduction. . . . . 44

3.2 Basic guard elements for ensuring full symmetry. . . . . 47

3.3 Invariant for the three-process R&W. . . . . 54

3.4 Experimental results for generalized R&W and asymmetric sharing of resources. 65

4.1 Relational operations. . . . . 71

4.2 The rules of operational and mixed semanics. $U$ is the set of local variables in the scope of the function call; $[\![f]\!]$ is natural semantics, $\mathbf{p}_f$ are the formals, and $\mathbf{r}_f$ are the returns of $f$. . . . . 74

4.3 Experimental results: overall analysis time in seconds. . . . . 99

5.1 Experimental results for SIS and RIS over $\texttt{Prog}_1$. . . . . 147

5.2 Experimental results for SIS and RIS over $\texttt{Prog}_2$. . . . . 148

# List of Figures


1.1 Overview of abstraction in model checking, where $\mathcal{M}$ denotes the concrete model of a program, MC($\mathcal{M}, \varphi$) – the concrete model checking, $\mathcal{M}^\alpha$ – the abstract model, and MC($\mathcal{M}^\alpha, \varphi$) – the abstract model checking. . . . . 4

1.2 Illustration of the three types of abstraction. . . . . 9

2.1 (a)-(c) Truth domains: (a) $\mathbf{2}$-valued boolean logic, (b) $\mathbf{3}$-valued Kleene logic, and (c) $\mathbf{4}$-valued Belnap logic. . . . . 21

2.2 An example of boolean transition system $B_1$. . . . . 25

2.3 (a)-(b) Examples of partial transition systems (dotted lines represent *may* transitions and solid – *must*): (a) a MixTS $M_1$ and (b) a GKMTS $M_2$, where the dashed ellipse denotes the set of states that are the destination of the *must* hyper-transition from $s_1$ ; and (c) a $\mathbf{4}$-valued transition system $M_3$. . . . . 32

3.1 (a) Synchronization Skeleton for MUTEX. (b) GSST for three-process R&W. . 46

4.1 (a) A program $\texttt{EX}_0$, (b) its over-approximation $\mathcal{O}(\texttt{EX}_0)$ using predicate $p : x > 0$. . . . . 67

4.2 (a) A program $\texttt{EX}_1$ and (b) its ICFG. . . . . 73

4.3 Illustration of proof, where $\Gamma_{k+1} = \Gamma_n = s_k$. . . . . 80

4.4 (a) A $\mathbf{4}$-valued transition relation $r$, and (b) a mixed transition relation $r'$ transformed from $r$, where solid and dotted lines represent *must* and *may* transitions, respectively. . . . . 96

4.5 (a) The template for experiments. (b) `<stmt>` for template **T1(**$n$**)**. (c) `<stmt>` for **T2(**$n$**)**. . . . . 98

4.6 Non-terminating programs: (a) `Ack`; (b) `Shift`; (c) Buggy `Quicksort`. . . 101

5.1 Four MixTSs: $M_1$, $M_2$, $M_3$, and $M_4$, where $M_1$ and $M_4$ are monotone. Solid and dashed lines represent *must* and *may* transitions, respectively. . . . . 112

5.2 Two GKMTSs: $G_1$, $G_2$, and two MixTSs: $M_5$, $M_6$, where $G_1$ and $G_2$ are semantically equivalent to $M_5$ and $M_6$, respectively. . . . . 124

5.3 One MixTSs: $M_7$, and two KMTSs: $K_1$, $K_2$, where $M_7$ and $K_4$ are semantically equivalent. . . . . 127

5.4 The RIS algorithm and its supporting functions. . . . . 143

5.5 Code examples for experiments. `nondet` denotes non-deterministic choice. . . 146

# Chapter 1

# Introduction

## 1.1 Model Checking and Abstraction

### 1.1.1 Model Checking

Computer systems are being used widely in our daily lives. We often rely on hardware and software programs to control safety critical, mission critical, or economically vital tasks. Ensuring the correctness of these programs thus has become a major challenge in the computer-dependent society. Formal methods provide us with techniques that support strict reasoning about program correctness. In these techniques, a computer program is modeled by a mathematical object, and the correctness of the program is formulated using mathematical specification. A formal reasoning approach is then provided to determine whether the program satisfies its specification. Thanks to the underlying rigorous mathematical foundations, formal methods are considered as a promising approach to increase our confidence about computer programs.

Model checking [CE81, QS82] is a formal analysis technique that checks behavioral properties of computer programs based on state-exploration. In this method, a hardware or software program is modeled using a finite state transition system, where the computational behaviors of the program are given as paths in the transition system. The desired program properties are expressed using temporal logic formulas. Temporal logic [Pnu77, CE81] provides tempo-

ral operators such as “always” and “eventually” and quantifiers over paths that capture a wide range of behavioral properties of programs. A model checker then uses a statespace search algorithm to determine whether the behavior described by a temporal property holds on the model of the program. If the answer is true, the program satisfies the property, i.e., the property is verified; a false answer means that the program violates the property, i.e., the property is refuted. When an error is detected, the model checker usually reports a counterexample for debugging. An advantage of model checking is that the search algorithm can be executed completely automatically. Furthermore, a model checker conducts an exhaustive exploration of all possible program behaviors. Therefore, subtle errors of a program that often elude simulation and testing approaches can be found in this way.

### 1.1.2 Fighting State-Explosion Problem

Despite the great success in checking hardware and software programs [BDEGW03, IYG+05, Kur08], scalability is still the primary challenge of applying model checking in the real world. Since model checking is a state-exploration method, it is restricted to analyzing programs that can be modeled by finite state transition system of small size. However, in practice, the statespace of a program can be extremely large, which is known as the *state-explosion problem*. For example, in the design of communication protocols, the size of the statespace of a program often grows exponentially with the number of components. Model checking such programs would require a significant amount of memory and CPU time. For software programs, the existence of infinite data domain, e.g., integers, and complex control structures, e.g., recursion, results in an infinite statespace, which makes direct model checking impossible. Therefore, a central issue in the research on model checking is dealing with the state-explosion problem.

Several approaches have been proposed to reduce the state-explosion problem based on symbolic representation, partial order reduction, compositional reasoning, induction, and abstraction. We introduce them below.

*Symbolic model checking* [McM93] uses binary decision diagrams (BDDs) [Bry92] to rep-

resent program models, which allows for compact representation of the states and transition relations of a program. *Partial order reduction* [GW94, CGMP99] reduces the size of the statespace that needs to be searched by model checking algorithms. It takes advantage of the commutativity of concurrent events, avoiding exploration of redundant interleaving behaviors in asynchronous programs. *Compositional reasoning* [Pnu84, CLM89, GL91] exploits modular structures of a program to reduce the complexity of the model checking task. It typically follows the assume-guarantee paradigm where the correctness of each component under an assumption of its running environment implies the correctness of the whole program. For parameterized protocols that define an infinite family of programs, the *induction* [WL89, KM95] approach analyzes properties based on an invariant structure of the protocol. The results in [EN95, EK00] show that such invariants can be defined as a union of program instances up to a finite cutoff bound. *Abstraction* [Kur89, CGL94, ID96, CEJS98, KP00] is arguably the most fundamental approach to scale model checking to realistic programs, which is the theme of this thesis and is discussed below.

### 1.1.3 Abstraction

Abstraction in model checking can be seen as an instance of *abstract interpretation* [CC92] on analyzing temporal properties over state transition systems. In this case, when the model of a program is too large to be directly handled by a model checker, we build a smaller abstract model of the program and analyze properties over it. The hypothesis of abstraction techniques is that for particular properties that we are interested in, many aspects of the original program, e.g., specific values of program variables and process identities, are not necessary for the analysis. Therefore, we can abstract away these details of the program and use the simplified model to check properties.

Abstraction in model checking (Figure 1.1) starts from a finite state abstraction that collapses sets of concrete program states into abstract ones. A finite abstract model $\mathcal{M}^{\alpha}$, which is smaller than the concrete model $\mathcal{M}$ of the program, is built over the abstract states to de-

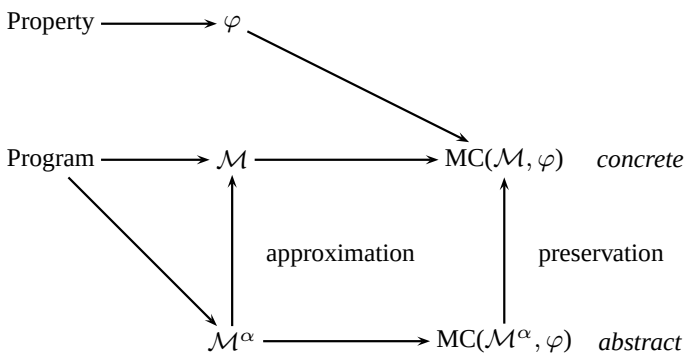


Figure 1.1: Overview of abstraction in model checking, where $\mathcal{M}$ denotes the concrete model of a program, MC($\mathcal{M}, \varphi$) – the concrete model checking, $\mathcal{M}^\alpha$ – the abstract model, and MC($\mathcal{M}^\alpha, \varphi$) – the abstract model checking.

scribe the program behaviors in an abstract way. The construction of the abstract model $\mathcal{M}^\alpha$ can be viewed as an interpretation of the program using non-standard semantics defined over the abstract statespace. $\mathcal{M}$ and $\mathcal{M}^\alpha$ are related by an approximation relation that determines how program behaviors are approximated. This results in a property preservation relation that defines how to lift abstract model checking results back to the concrete level. If the result is conclusive, i.e., it allows us to determine whether $\varphi$ is satisfied over $\mathcal{M}$, the abstract model checking process is finished. Otherwise, additional refinement steps are performed to produce more precise abstract models for analysis.

In the following, we review the techniques for abstract model checking using the following steps: (a) abstraction of statespace, (b) construction of abstract model, and (c) refinement.

*Abstraction of statespace*. Abstract statespace is the basis for abstraction in model checking. It determines the information that an abstract model can represent and properties that can be effectively analyzed over it. A simple abstraction of the statespace is called *localization reduction* [Kur94, CGP99], where abstract states are defined over a subset of program variables relevant to analysis of particular properties. Since localization reduction does not reduce the

domain of variable values, it cannot produce a finite model if program variables are defined over an infinite data domain. *Data abstraction*, traditionally used in program analysis to compute abstract program invariants [CC92, NNH05, Sch98], is applied to abstract model checking in [CGL94, CDH+00, PDV01]. An abstract statespace in data abstraction is defined based on simple data facts about program variables, e.g., parity, sign, or range. A limitation of data abstraction is that it cannot capture relations between program variables, because the abstract states only express independent attributes of them. This limitation is avoided by using *predicate abstraction* [GS97]. First proposed by Graf and Saidi, predicate abstraction has become a popular technique used in abstract model checking [BPR01, CCG+04, HJMS02, GC06]. In this case, an abstract statespace is defined based on a set of predicates. Concrete data values are mapped to boolean variables represented by these predicates at the abstract level, while the original data variables are eliminated.

In practice, an appropriate choice of abstract statespace depends on the class of programs and properties to be analyzed. For example, to analyze parameterized programs, one can choose a variant of predicate abstraction called *environment abstraction* [CTV06], where predicates contain not only the information about individual processes, but also the relationship between them. For shape analysis, predicate abstraction represents points-to relations in shape graphs of heap storage [SRW99, DN03]. For programs composed of identical processes, abstract statespaces can be defined as symmetric classes of program states [ES96, CJEF96].

*Construction of Abstract Model*. Given an abstract statespace, we need to build an abstract model over it for model checking properties. Research on *optimality* [LGS+95, CIY95, DGG97, Sch04, GWC06a] defines structural conditions that characterize the best abstract model, which represents the most precise approximation of concrete program behaviors. However, the optimal conditions are usually defined with respect to concrete transitions in original programs. Directly using them in the construction of abstract models would require first building the concrete model, which is often infeasible due to the large or infinite statespace. Therefore, in practice, we look for approaches that compute abstract models directly from the program text,

where transitions between abstract states are constructed based on evaluation of an abstract semantics of programs. The resulting abstract model is usually less precise than the optimal one, but can be constructed more efficiently.

Building abstract models from program text is studied in [CGL94], where concrete states are abstracted using data abstraction, and relational semantics of a program is represented by first-order formulas derived from programs. An abstract model is constructed by approximating the concrete semantics of the program. Automatically computing abstract models based on predicate abstraction is suggested in [GS97], and is now a standard technique in abstract model checking of software programs [BMMR01, BPR01, HJMS02, CCG+04, GC06]. Typically, this approach first translates a program into a boolean program over the variables defined by predicates. Each statement in the original program is approximated by one or more statements on the predicates in the boolean program, which describe how the values of the predicates are changed when the concrete statement is executed. This step is conducted based on strengthening the weakest precondition for each predicate with the aid of a theorem prover. An abstract model is then constructed based on an approximating semantics of the boolean program. In [CKSY05], an abstract model over predicates is computed with a SAT solver using the similar approach as [CGL94].

*Refinement*. If an abstract model, e.g., built by symmetry reduction [ES96, CJEF96], has behaviors equivalent to the original program, model checking results over it are conclusive. Otherwise, loss of information introduced in the abstraction process may result in inconclusive analysis, and a refinement step is necessary. Automation of the refinement process is called *abstraction refinement* [Kur94, BPR02, Dam03], which starts with a coarse initial abstract model of the program, iteratively refining it until the abstract model contains sufficient details. Completeness results of abstract model checking [KP00, DN04, DN05] show that for every program and temporal property, there exists a finite abstract model such that satisfiability of the property over it implies its satisfiability over the original program. However, in general finding an appropriate abstraction for an infinite state program is not computable. Otherwise,

the program verification problem would be decidable. In practice, heuristic methods are often used to guide the refinement process.

In [NK00, PPV05, YBS06], refinement is based on computing weakest precondition, which generates new predicates for more precise abstraction. This approach is guided by precise updates of existing predicates with respect to statements in a program. Recently, a counterexample-guided refinement approach has been used widely in abstract model checking [CGJ+03, Bal04, HJMS02]. Typically, a *counterexample* is an abstract program execution produced in a previous model checking run, which is used to demonstrate the violation of a property on the abstract model. Because of imprecision introduced by abstraction, the counterexample may be infeasible. That is, the counterexample does not correspond to a concrete execution of the original program. A refinement process then tries to remove this spurious counterexample by refining the abstract statespace. This leads to a more precise abstract model for analysis. The new facts for refinement, e.g., new predicates, can be discovered using different ways. The approach in [CGJ+03] uses symbolic algorithms to simulate a counterexample on the concrete program. If the counterexample is spurious, a shortest feasible prefix of the counterexample is identified, and the last abstract state in the prefix, called a failure state, is split into more precise abstract states to eliminate the spurious counterexample. The feasibility of a counterexample can also be checked using a theorem prover [BR01, HJMM04] or SAT solver [CCK+02]. In this case, a formula is generated such that it is satisfiable if and only if the counterexample is feasible. If the formula is not satisfiable, more predicates are picked for refinement based on analysis of the unsatisfiability proof of the formula. In [GC03, GWC06b], a counterexample is represented as a proof of the property being analyzed. Steps in the proof correspond to transitions between abstract states. If a real counterexample exists, a complete proof is produced. Otherwise, only a part of the proof can be generated. In this case, by analyzing the proof steps that can not be completed, new information is derived for refinement.

Abstraction is an essential approach for fighting the state-explosion problem in model checking. We have presented an overview of the approach. In the next section, we introduce

the class of abstraction methods and the problems investigated in this thesis.

## 1.2 Scope of This Thesis

In this thesis, we focus on abstraction methods that support both verification and refutation of program properties in the same framework, which we refer to as *exact-approximation*[1].

In general, abstraction in model checking falls into three types (illustrated in Figure 1.2), depending on the approximation relations between concrete and abstract models and property preservation relations for temporal properties. In the *over-approximation* abstraction framework [CGL94, LGS+95, KP00, BPR01], an abstract model contains more behaviors than the original program. Such abstraction is sound for proving universal temporal properties that hold along all executions of the program. For example, if we can prove absence of error on an abstract model, it implies that no error exists in the original program either. This framework is the one for verification, since traditionally correctness of programs is often expressed using universal properties. The dual of this framework is *under-approximation* [PPV05, BKY05, GC06, BK07], which is also known as the framework for refutation. In this case, an abstract model contains less behaviors than the concrete one, which enables us to prove properties that hold along some executions of the program, e.g., presence of a bug.

An obvious limitation of over- and under-approximations is that they only preserve soundness for fragments of temporal properties that are not closed under negation, i.e., the negation of a universal (resp. existential) property is an existential (resp. universal) property. Therefore, if a property fails to hold on the abstract model, we know nothing about the property on the original program (depending on the property being checked and the framework). For example, if proving absence of error fails in an over-approximating abstract analysis, we do not know whether an error exists in the original program. This limitation can be avoided by

[1]Different terminology has been used to describe abstraction for both verification and refutation, e.g., *exact-abstraction* [CGL94, PPV05, Eme08], *3-valued abstraction* [BG99, HJS01, dAGJ04, SG06], and *Belnap abstraction* [GWC06a, GWC06b].

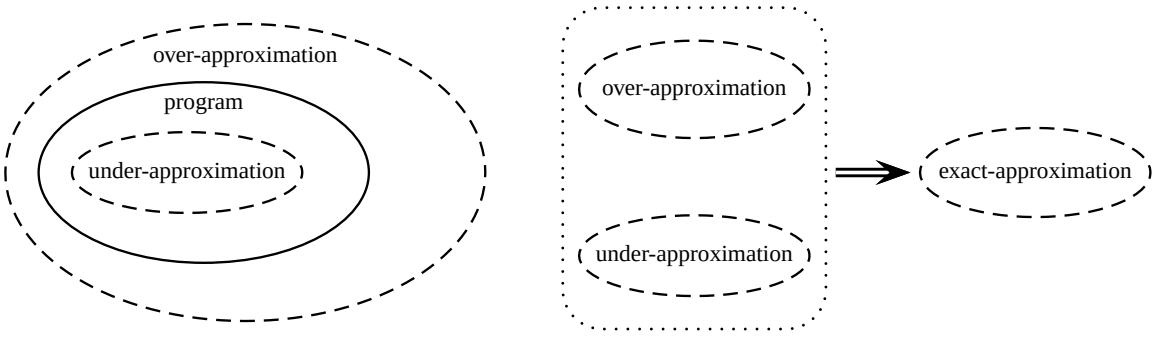


Figure 1.2: Illustration of the three types of abstraction.

using the *exact-approximation* framework [Kur89, BBLS92, CIY95, DGG97, BG99, HJS01]. An abstract model in this framework can be seen as a combination of both over- and under-approximations. The abstract analysis based on such abstract models is sound for the full set of temporal properties, that is, both true and false results are preserved to the concrete level. Therefore, exact-approximation supports both verification and refutation of temporal properties with the same effectiveness.

Exact-approximation can be defined as *strong* and *weak*, according to the degree of preservation of temporal properties. We investigate both of them in this thesis. In the following, we define the problems addressed by this thesis in the context of strong and weak exact-approximations, respectively.

### 1.2.1 Strong Exact-Approximation

In strong exact-approximation [Kur89, BBLS92, DGG97], model checking results are not only preserved from the abstract level to the concrete level, but also in the reverse direction. Therefore, a property holds on an abstract model if and only if it holds on the concrete one. Abstract models used for strong exact-approximation are built over classical boolean transition systems. The relation between abstract and concrete models is captured by bisimulation equivalence [Mil80, Par81], where both models can simulate each other's behaviors in a stepwise

manner and thus satisfy the same set of properties. In this case, an abstract model can be seen as a combination of the same over- and under-approximations of the original program. The advantage of strong exact-approximation is that we can always get conclusive model checking results from abstract analysis.

In this thesis, we study a special strong exact-approximation technique, called *symmetry reduction* [CJEF96, ES96, ID96], which explores symmetric structures of a program for abstraction. Many concurrent programs or communication protocols, e.g., mutual exclusion protocol, consist of the coordination of several identical processes. The program behaviors are unchanged under permutation of process identities. Such high level symmetry is reflected in the program statespace, which can be used to avoid exploring the states that are symmetric to the ones that have been explored before. Based on this observation, symmetry reduction uses equivalence classes of symmetric states as abstract states, and builds a quotient structure over them that is guaranteed to be bisimilar to the concrete program model. If the symmetry set is large, the quotient structure is substantially smaller than the concrete model.

In practice, however, there exist many programs which are not genuinely symmetric: they are composed of many similar, but not identical processes, e.g., readers-and-writers (a variant of the mutual exclusion protocol). Although symmetry is exhibited in a large part of such programs, their global behaviors are not symmetric. To extend the scope of symmetry reduction to such "almost" symmetric programs, Emerson et al. have proposed weaker notions of symmetry, including *near* or *rough* symmetry [ET99] and *virtual* symmetry [EHT00]. In particular, virtual symmetry is the most general notion under which the program model is bisimilar to its reduced quotient structure. While virtual symmetry increases a potential domain of problems that can be symmetry reduced, its practical application depends on successful solutions to the following questions:

*Question 1: How to identify virtually symmetric programs?* The first step of using symmetry reduction is the identification of symmetry in a program. To avoid construction of the concrete model of the program, which is usually not feasible, we need an efficient approach to

detect this symmetry from the program description. Genuine symmetry often exhibits certain patterns in the design level, and can be identified from the specification of the program. However, for virtually symmetric programs, asymmetric behaviors may arise for different reasons. Lack of regularity in such programs makes it difficult to identify virtual symmetry syntactically.

*Question 2: How to combine virtual symmetry and symbolic model checking effectively?* Symbolic model checking uses compact data structures to represent program models. Symmetry reduction reduces the size of program model based on state abstraction. Since the two approaches exploit different features of programs for fighting the state-explosion problem, it is desirable to combine them to obtain better model checking performance. This idea has been investigated in the context of genuine symmetry [ET99, BG02, EW03]. In order to analyze a wider class of programs, it is interesting to investigate how to extend these results to virtual symmetry as well.

Symmetry reduction is a strong exact-approximation technique that uses a symmetry-reduced structure for abstract model checking. A limitation of strong exact-approximation is that bisimulation-based abstraction is too restrictive. In practice, it is not always possible to find an abstract model such as the symmetry-reduced one that has behaviors equivalent to the original program and enables significant statespace reduction at the same time. The reason is that abstraction often introduces some loss of information, and thus, some program behaviors become unknown. Therefore, enforcing bisimulation restricts the choice of abstract models, which is not helpful for reduction of model size. This problem can be avoided using *weak* exact-approximation [CIY95, DGG97, BG99, GJ03, DN04] that allows for more general abstraction by accommodating unknown information. We discuss it below.

### 1.2.2 Weak Exact-Approximation

Unlike strong exact-approximation, over- and under-approximating behaviors represented by abstract models in weak exact-approximation may not be the same. The gap between them represents the unknown behaviors caused by abstraction. We refer to such abstract models as

*partial*, since they capture an incomplete view of the original program behaviors. The relation between partial and concrete models is generalized from bisimulation, describing approximation relations for over- and under-approximating behaviors, respectively. If a partial model has enough information for proving or disproving a property, it gives conclusive results, i.e., true or false, respectively, which are preserved to the concrete level. Otherwise, an unknown result is reported, and a refinement step is necessary.

We study weak exact-approximation from the following two aspects: (1) model checking recursive programs and (2) analysis of partial modeling formalisms.

(1) Software model checking directly checks a program by combining automated predicate abstraction [GS97, BMMR01] and counterexample-guided abstraction refinement [CGJ$^+$03]. Using a list of predicates over program variables, a software model checker constructs a finite abstract model of a program for property analysis. If the result is inconclusive, counterexamples are generated to find additional predicates for refinement. The process continues until either the property is successfully proved or disproved, or resources are exhausted. Traditional software model checkers [BMMR01, HJMS02, CKSY05] build an over-approximating abstraction of the programs, and typically bias their analysis towards verification of properties. In our previous work, we have developed a software model checker YASM [GWC06b], which constructs an abstract model based on exact-approximating semantics of predicate abstraction [GWC06a, GC06]. Therefore, it supports both verification and refutation. In this thesis, we extend such analysis ability of YASM by addressing the following problem:

*Question 3: How to model check recursive program with exact-approximation?* Software programs often involve significant use of recursion. A limitation of YASM is that it cannot handle programs with recursive functions. A state in a recursive program is an unbounded call stack of activation records, which introduces another source of infinity on the statespace other than data domain. Since YASM only uses predicate abstraction to abstract data aspects of program states, it does not support abstract analysis with call stacks. A naive solution to this problem requires the development of new abstract models that combine call stack and predicate

abstraction, and subsequently, new algorithms for analyzing these models. In this thesis, we investigate how to avoid this problem and look for a simple approach that allows us to reuse existing abstract analysis in YASM.

(2) Partial models play a fundamental role in weak exact-approximation. They support a combination of over- and under-approximations without requiring them to be the same. A variety of modeling formalisms have been proposed for this in literature. In this thesis, we conduct a systematic analysis of a set of partial modeling formalisms that are used widely for exact-approximation [GHJ01, HJS01, DGG97, dAGJ04, CDEG03, SG04]. In general, these modeling formalisms consist of two kinds of transition relations, one corresponding to over-approximation, called *may* transitions, and the other — under-approximation, called *must* transitions. We call these formalisms *partial transition systems*, which can be classified into three families, represented by *Kripke Modal Transition Systems* (KMTSs) [HJS01] with the requirement of every *must* transition is also a *may* transition, *Mixed Transition Systems* (MixTSs) [DGG97] with independent *may* and *must* transitions, and *Generalized Kripke Modal Transition Systems* (GKMTSs) [SG04] with *must* hyper-transitions. In this thesis, we study these formalisms from two points of view: a semantic one, using partial transition systems for abstracting concrete programs, and a logical one, using partial transition systems for temporal logic model checking. Specifically, we address the following questions:

*Question 4: What are the connections between semantic and logical consistency of partial transition systems?* Semantic and logical consistency correspond to the semantic and logical view of partial transition systems, respectively: semantic consistency ensures that a partial transition system does approximate some concrete program, and logical consistency ensures that a partial transition system gives consistent interpretation to all temporal formulas, that is, no formula can be interpreted both true and false at the same time. Both notions of consistency are required for meaningful abstract model checking. We are interested in analyzing the equivalence relation between semantic and logical consistency. We also want to find out if there is a structural condition that can capture them.

*Question 5: Does the structural difference affect the expressive power of partial transition systems?* Expressive power measures the abstraction ability of partial transition systems. The three families of partial transition systems represented by KMTSs, MixTSs, and GKMTSs have similar but different structures, which have been introduced for various reasons, e.g., to obtain optimal abstraction and better refinement. It is interesting to see whether these structurally different formalisms have the same expressiveness. Dams and Namjoshi have shown that all of these formalisms are subsumed by tree automata [DN05]. However, a comparison of expressive power between them is still missing.

*Question 6: How to use partial transition systems effectively in practical model checking?* In practice, model checking is often conducted based on a tractable inductive semantics of temporal logic. Based on this semantics, a GKMTS allows for more precise analysis results than the semantically equivalent MixTS or KMTS, i.e., proving or disproving more properties. However, while both MixTSs and KMTSs have been used in the development of practical symbolic model checkers (e.g., [GC06, CDEG03]), the direct use of GKMTSs has been hampered by the difficulty of encoding hyper-transitions symbolically. To obtain more precise symbolic model checking using partial transition systems, it is interesting to investigate approaches that allow us to combine both a symbolic encoding of MixTSs and a better model checking precision of GKMTSs.

## 1.3 Contributions of This Thesis

The main theme of this thesis is the study of abstraction in model checking based on exact-approximation. In general, this thesis presents several theoretical and practical contributions in the following aspects:

- We study symmetry reduction on full virtual symmetry [WGC05]. We formalize the connection between symmetry reduction and abstraction. Based on that, we address Question 1 by providing an efficient procedure for identifying full virtual symmetry,

and provide a solution to Question 2 by extending counter abstraction to fully virtually symmetric programs.

- We propose a novel approach for analyzing reachability and non-termination properties of recursive programs with exact-approximation [GWC08], which provides a solution to Question 3. We accomplish this by using a mixed program semantics to remove call stacks, which leads to a natural combination of analysis of recursive program with exact predicate abstraction. We also develop on-the-fly algorithms to improve analysis performance.

- We provide answers to Questions 4, 5 and 6 about partial modeling formalisms [WGC09a, WGC09b]. We prove equivalence between semantic and logical consistency over a class of partial transition systems, and provide a necessary and sufficient structural condition to characterize them. We show that the three families of modeling formalisms, KMTSs, MixTSs, and GKMTSs, have the same expressive power. We also propose a new inductive semantics of temporal logic, which results in more precise symbolic model checking using partial transition systems.

We give a more detailed overview of these contributions below.

### 1.3.1 Full Virtual Symmetry Reduction

Our study of symmetry reduction in this thesis focuses on the problems of identification and symbolic model checking of fully virtually symmetric programs, i.e., programs that are virtually symmetric up to exchanging the roles of processes. This form of virtual symmetry is interesting because symmetry reduction over these programs often allows for exponential reduction in the model size.

Our solutions are based on a view of symmetry reduction from the perspective of abstraction. Symmetry reduction is usually defined based on the permutations of processes in programs. While this provides a natural way to understand symmetry reduction from the design

perspective, its connection with the general framework of abstraction is missing. We complete this by formalizing symmetry reduction using the notions in abstraction. We define the mapping between the components of symmetry reduction and abstraction, which gives us an alternative characterization of symmetry reduction.

Based on this characterization, we provide an efficient approach to identify full virtual symmetry from program specifications. The problem of identifying full symmetry (the genuine counterpart of full virtual symmetry) has been avoided by using specification languages with special restrictions on syntax [ET99, EW03]. We show that lack of regularity in asymmetric programs makes it difficult to capture restrictions that ensure full virtual symmetry syntactically. We then provide an algorithmic approach to the problem. We show that identification of full virtual symmetry can be reduced to satisfiability of a quantifier-free Presburger formula. This formula is built directly from the specification of a program, and can be checked automatically using existing decision procedures [BB04, Pug92]. .

Our solution to symbolic symmetry reduction of full virtual symmetry is based on counter abstraction. The bottleneck of symbolic symmetry reduction is that the BDDs of orbit relations, which define the equivalence between symmetric states, often have exponential size [CJEF96]. In [ET99, EW03], Emerson et al. showed that for full symmetry, the problem of building orbit relation can be avoided via *counter abstraction* [PXZ02]. The idea of this technique is based on the observation that equivalence classes of symmetric states under full symmetry can be generically represented using counters of local process states. Then a fully symmetric program can be abstracted into another one that operates on counter variables. The model of the translated program is isomorphic to the symmetry-reduced structure of the original program, and can be symbolically analyzed directly. In the thesis, we show that this technique can be extended to handle full virtual symmetry as well. For the translation step, we identify several cases of asymmetric transitions and provide procedures to translate them to the ones defined over counter variables. We report on experiments to illustrate the feasibility of our approach.

### 1.3.2 Reachability and Non-Termination Analysis of Recursive Programs

We investigate abstract analysis of recursive programs with respect to reachability and non-termination. These two properties are often used in practice for analyzing software programs. Our approach is based on a mixed program semantics that allows us to combine analysis of recursive programs with exact predicate abstraction without explicitly dealing with call stacks.

We notice that reachability and non-termination properties depend only on top activation records of call stacks. To analyze these properties, we develop a stack-free program semantics where each state is a *single* activation record. The stack-free semantics combines both operational and natural semantics [NN92] that correspond to the executions of single statements and functions, respectively. It uses non-determinism at call sites to simulate the executions within and outside of function bodies, which effectively eliminates the call-stack while preserving stack-independent properties. Based on the stack-free semantics, we develop algorithms for analyzing reachability and non-termination properties of recursive programs. Since there are no call stacks involved in the stack-free semantics, we can reuse the existing exact predicate abstraction in our software model checker YASM [GWC06b] to obtain abstract analysis of these properties. To improve the performance of the analysis, we also develop on-the-fly versions of the algorithms, where computation of natural semantics of functions, i.e., the summary of the behaviors of functions, is driven by the analysis of particular properties.

Our algorithms share many insights with techniques in other tools for analyzing recursive programs (e.g., [BR00, ACEM05, BCP06, PSW05]), i.e., they are functional [SP81] in terms of interprocedural analysis, and apply only to stack-independent properties. However, all those tools use *over-approximation* to analyze infinite programs. It is not clear how to combine them with *exact-approximation*. The novelty of our approach is that it separates the analysis of recursive programs from abstraction of data domains. Therefore, combining the analysis algorithm with different abstractions is trivial in our work. Moreover, over-approximation makes it impossible to use the existing tools for detecting non-termination since over-approximation may introduce spurious non-terminating computations, whereas this is not a problem in our case.

We have implemented our approach in the software model-checker YASM. Our approach allows us to reuse existing abstract analysis in YASM to handle recursive programs. We experimented on reachability and non-termination analysis of several non-trivial C programs.

### 1.3.3 Analysis of Partial Modeling Formalisms

We investigate the three families of partial transition systems represented by *Kripke Modal Transition Systems* (KMTSs) [HJS01], *Mixed Transition Systems* (MixTSs) [DGG97] and *Generalized Kripke Modal Transition Systems* (GKMTSs) [SG04], providing answers to Questions 4, 5 and 6.

For the relation between semantic and logical consistency of partial transition systems, we show that while in general they are not equivalent, there is a class of partial transition systems for which semantic and logical consistency coincide. We call this class *monotone* because of the monotonicity condition imposed on the transition relations. The class of monotone transition systems is as expressive as the class of all partial transition systems. That is, for every partial transition system, there is an equivalent monotone one. We also provide a structural condition to capture both notions of consistency. We show that a previous requirement of "every *must* transition is also a *may* transition" [HJS01, dAGJ04] is sufficient for logical consistency, but not necessary. For semantic consistency, the requirement is neither necessary nor sufficient. Over the class of monotone transition systems where semantic and logical consistency coincide, we define an alternative structural condition and show that it is both necessary and sufficient to guarantee consistency.

We compare the expressive power of the three families of formalisms, KMTSs, MixTSs, and GKMTSs, and show that they are equally expressive, i.e., for any partial transition system expressed in one formalism, there exists another one in the other such that the two transition systems approximate the same set of concrete programs. That is, neither hyper-transitions nor restrictions on *may* and *must* transitions affect expressiveness. They do, however, affect the size of the models: GKMTSs and KMTSs can be converted to semantically equivalent MixTSs

of smaller or equal size. Dams and Namjoshi have shown that the three families of formalisms are less expressive than tree automata [DN05]. Our results complete the picture by showing the expressive equivalence between those formalisms.

While analysis of properties over the GKMTS using the standard inductive semantics is more precise than that over a corresponding MixTS (or KMTS) obtained by semantics-preserving translations, the direct use of GKMTSs in symbolic model checking has been hampered by the difficulty of symbolic encoding of hyper-transitions. To address this problem, we develop a new semantics, called *reduced*, that is still inductive (and tractable) but more precise than the standard one. We show that GKMTSs and MixTSs are equivalent with respect to the reduced inductive semantics, and give a symbolic approach for computing the semantics. The outcome is an algorithm that combines the benefits of the symbolic encoding of MixTSs with the better precision of GKMTSs. We implement our algorithm and evaluate it empirically over MixTSs constructed using predicate abstraction.

## 1.4 Organization

The rest of this thesis is organized as follows. In Chapter 2, we set our notation and describe temporal logic, model checking, and the abstraction framework of exact-approximation. In Chapter 3, we describe our work on symmetry reduction with full virtual symmetry, reported in [WGC05]. In Chapter 4, we describe our approach for analyzing reachability and non-termination properties of recursive programs, reported in [GWC08]. In Chapter 5, we describe our results of analysis of partial modeling formalisms, reported in [WGC09a] and [WGC09b]. Finally, we conclude in Chapter 6 with a summary of this thesis and a discussion of limitations of our work and future research directions.

# Chapter 2

# Background

The work presented in this thesis is concerned with abstraction in model checking. This chapter introduces the concepts and fixes the notation used in later chapters. In Section 2.1, we present truth domains that are associated with concrete and abstract model checking results. In Section 2.2, we introduce model checking, including temporal logics and models of computation. In Section 2.3, we define an abstraction framework for model checking, and introduce strong and weak exact-approximations.

## 2.1 Truth Domains

A *truth-domain* $\mathcal{D}$ is a collection of elements $D$, referred to as *truth values*, together with a truth ordering $\sqsubseteq$ and a negation operator $\neg : D \rightarrow D$, such that $\mathcal{D} = (D, \sqsubseteq, \neg)$ is a De Morgan algebra. The truth ordering orders the elements based on their truth content; thus, $a \sqsubseteq b$ stands for "$a$ is less true than $b$". The meet ($\wedge$) and join ($\vee$) of the truth ordering are called *conjunction* and *disjunction*, respectively.

The most known truth domain is the classical boolean logic $\mathbf{2}$ (Figure 2.1(a)) with values $\mathsf{t}$ (*true*) and $\mathsf{f}$ (*false*) such that $\mathsf{f} \sqsubseteq \mathsf{t}$. Kleene logic [Kle52] $\mathbf{3}$ (Figure 2.1(b)) extends $\mathbf{2}$ with an additional element $\mathsf{m}$, representing "unknown" information. The truth ordering of the logic is extended as $\mathsf{f} \sqsubseteq \mathsf{m}$ and $\mathsf{m} \sqsubseteq \mathsf{t}$, and negation as $\neg\mathsf{m} = \mathsf{m}$. We define an additional ordering

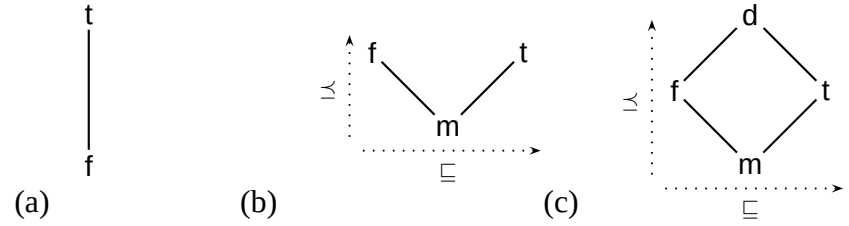


Figure 2.1: (a)-(c) Truth domains: (a) **2**-valued boolean logic, (b) **3**-valued Kleene logic, and (c) **4**-valued Belnap logic.

$\preceq$, that relates values based on the amount of *information*; thus $\mathsf{m} \preceq \mathsf{t}$ and $\mathsf{m} \preceq \mathsf{f}$, so that $\mathsf{m}$ represents the least amount of information. Belnap logic [Bel77] **4** (Figure 2.1(c)) extends **3** with an additional element $\mathsf{d}$ representing "inconsistent" information. The truth ordering is extended so that $\mathsf{f} \sqsubseteq \mathsf{d}$ and $\mathsf{d} \sqsubseteq \mathsf{t}$, and negation as $\neg\mathsf{d} = \mathsf{d}$, i.e., $\mathsf{d}$ is equivalent to $\mathsf{m}$ with respect to this ordering. Finally, the information ordering is extended by making $\mathsf{d}$ be the largest element, i.e., $\mathsf{f} \preceq \mathsf{d}$ and $\mathsf{t} \preceq \mathsf{d}$.

## 2.2 Model Checking

Given a hardware or software program, model checking automatically determines whether the model of the program satisfies a temporal property. In this section, we first introduce two temporal logics, the modal $\mu$-calculus [Koz83] and the Computation Tree Logic (CTL) [CES83]. We then describe the classical models of programs based on boolean transition systems, and the semantics of temporal logics over them.

### 2.2.1 Temporal Logics

We begin by introducing the modal $\mu$-calculus.

**Definition 2.1** (Modal $\mu$-Calculus)**.** *Let* Var *be a set of variables, and* $AP$ *be a set of atomic*

*propositions. The logic $L_\mu(AP)$ is the set of all formulas satisfying the grammar*

$$\varphi \quad ::= \quad p \mid Z \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Diamond\varphi \mid \mu Z \cdot \varphi(Z),$$

*where $p$ is an atomic proposition in $AP$, and $Z$ is a fixpoint variable from in* Var.

An occurrence of a variable $Z$ in a formula $\varphi$ is *bound* if it appears in the scope of a $\mu$ quantifier and is *free* otherwise. For example, $Z$ is free in $p \vee \Diamond Z$, and is bound in $\mu Z \cdot p \vee \Diamond Z$. A formula $\varphi$ is *closed* if it does not contain any free variables.

We define the following syntactic abbreviations:

$$\begin{array}{lcl} \varphi \vee \psi & \triangleq & \neg(\neg\varphi \wedge \neg\psi) \\ \varphi \Rightarrow \psi & \triangleq & \neg\varphi \vee \psi \\ \Box\varphi & \triangleq & \neg\Diamond\neg\varphi \\ \nu Z \cdot \varphi(Z) & \triangleq & \neg\mu Z \cdot \neg\varphi(\neg Z/Z) \end{array}$$

where $\varphi(\psi/Z)$ denotes the syntactical substitution of $\psi$ for free occurrences of $Z$ in $\varphi$.

A $\mu$-calculus formula is *syntactically monotone* if and only if for every formula of the form $\mu Z \cdot \varphi(Z)$, all occurrences of the fixpoint variable $Z$ in $\varphi$ fall under an even number of negations in $\varphi$. From this point onwards, we consider syntactically monotone $\mu$-calculus formulas only.

The modal operator $\Diamond$ is typically interpreted as "an existence of an immediate future". For example, "$p$" means that $p$ holds now, "$\Diamond p$" means that there exists an immediate future where $p$ holds, and "$\Box p$" means that $p$ holds in all immediate futures. The quantifiers $\mu$ and $\nu$ stand for least and greatest fixpoint, respectively.

We often write $L_\mu$ to denote the set of $\mu$-calculus formulas over some unspecified set of atomic propositions. Every $L_\mu$ formula can be transformed to an $L_\mu$ formula $\texttt{NNF}(\varphi)$, called the *negation normal form* of $\varphi$, where negation is restricted to the level of atomic propositions [DGG97]. We say an $L_\mu$ formula is *universal* (resp. *existential*) if the only allowed modality in its negation normal form is $\Box$ (resp. $\Diamond$). We use $\Box L_\mu$ and $\Diamond L_\mu$ to denote the universal and existential fragments of $L_\mu$, respectively.

*Computation Tree Logic* (CTL) is a restricted subset of $L_\mu$, which is often used in the specification and analysis of temporal properties.

**Definition 2.2** (CTL)**.** *Let $AP$ be a set of atomic propositions. The temporal logic $CTL$ over $AP$ is is the set of all formulas satisfying the grammar*

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid EX\varphi \mid E[\varphi\, U\, \varphi] \mid EG\varphi$$

*where $p$ is an atomic proposition in $AP$.*

Additionally, we define the following syntactic abbreviations:

$$\begin{aligned}
AX\varphi \quad &\triangleq \quad \neg EX\neg\varphi \\
A[\varphi\, U\, \psi] \quad &\triangleq \quad \neg E[\neg\psi\, U\, \neg\varphi \wedge \neg\psi] \wedge \neg EG\neg\varphi \\
AG\varphi \quad &\triangleq \quad \neg E[\mathtt{true}\, U\, \neg\varphi] \\
EF\varphi \quad &\triangleq \quad \neg AG\neg\varphi \\
AF\varphi \quad &\triangleq \quad \neg EG\neg\varphi
\end{aligned}$$

The meaning of the operator $X$ is that a property holds in next time; $U$ specifies that the first property holds until the second property becomes true sometime in the future; The operators $F$ and $G$ requires a property to hold eventually and globally respectively. The universal and existential path quantifiers $A$ and $E$ specify that some property holds for all computational paths and for some path, respectively.

CTL has a fixpoint characterization. Every CTL formula can be translated to an $L_\mu$ formula according to the following definition[1].

$$\begin{aligned}
EX\varphi \quad &\triangleq \quad \Diamond\varphi \\
E[\varphi\, U\, \psi] \quad &\triangleq \quad \mu Z \cdot \psi \vee (\varphi \wedge \Diamond Z) \\
EG\varphi \quad &\triangleq \quad \nu Z \cdot \varphi \wedge \Diamond Z
\end{aligned}$$

[1]This translation is based on the assumption that the transition relations of CTL models are total. A translation without this assumption can be found in [Bra91].

The universal and existential fragments $ACTL$ and $ECTL$ [GL91] are the intersections of $CTL$ with $\Box L_\mu$ and $\Diamond L_\mu$, where only universal and existential path quantifiers are allowed, respectively.

In the following, we define classical models of computation and the semantics of temporal logics over them.

### 2.2.2 Models of Computation

A model of a program is built based on a state transition system. A *transition system* of a program describes the transition relations between program states. A *model* extends a transition system with a state labeling function, which is used to interpret atomic propositions in temporal logic formulas. A transition system can be associated with different labeling functions for model checking of different temporal properties. We refer to a class of transition systems as *modeling formalism*.

We first describe classical models of computation, *Kripke structures*, that are built over *boolean transition systems*.

**Definition 2.3** (Boolean Transition Systems)**.** *A* boolean transition system *(BTS) is a tuple* $B = \langle S, R \rangle$*, where*

- $S$ *is a set of states and*
- $R \subseteq S \times S$ *is a transition relation.*

For example, the picture in Figure 2.2 represents a BTS $B_1$ with

- $S_1 = \{s_1, s_2, s_3\}$,
- $R_1 = \{(s_1, s_2), (s_2, s_3), (s_2, s_2), (s_3, s_3), (s_3, s_1)\}$.

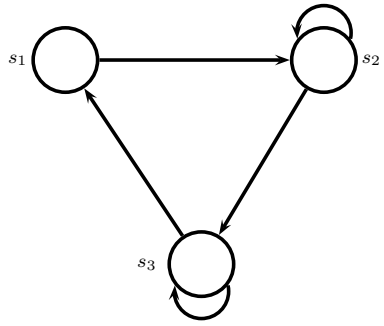


Figure 2.2: An example of boolean transition system $B_1$.

For a pair of states $s$ and $t$ related by the transition relation, i.e., $R(s,t)$ holds, we say that $t$ is a successor of $s$, and $s$ is a predecessor of $t$, and use $R(s)$ to denote the set of all successors of $s$. For a transition relation $R$, we define the pre-image of $R$, *pre*$[R] : \mathbf{2}^S \rightarrow \mathbf{2}^S$, as

$$\textit{pre}[R](Q) \triangleq \{s \in S \mid R(s) \cap Q \neq \emptyset\}$$

*pre*$[R](Q)$ is a set of states that have $R$-successors in $Q$. For example, in $B_1$, *pre*$[R_1](\{s_1\}) = \{s_3\}$, and *pre*$[R_1](\{s_2\}) = \{s_1, s_2\}$.

Let $B = \langle S, R\rangle$ be a BTS, and $AP$ be a set of atomic propositions. A *labeling function* $L : AP \rightarrow \mathbf{2}^S$ is an interpretation of atomic propositions over a set of states $S$ such that $s \in L(p)$ iff the atomic proposition $p$ is true at the state $s$. The pair $\mathcal{B} \triangleq \langle B, L\rangle$ defines a classical computational model, called a *Kripke Structure*.

We now define the semantics of temporal logics over Kripke structures. A semantics of $L_\mu$ is called *inductive* if it is defined inductively on the syntax of the logic. The inductive semantics of $L_\mu$ over Kripke structures is defined as follows.

**Definition 2.4** ($L_\mu$ semantics over Kripke Structures)**.** *Let $\mathcal{B} = \langle B, L\rangle$ be a Kripke structure, where $B = \langle S, R\rangle$ is a BTS. Let* Var *be a set of fixpoint variables, and $\varepsilon$ : Var $\rightarrow \mathbf{2}^S$ be an object assignment for free variables. The* inductive semantics *(or* interpretation*) of an $L_\mu$*

*formula $\varphi$ over $\mathcal{B}$, denoted $||\varphi||^{\mathcal{B}}_{\varepsilon}$, is defined as follows:*

$$
\begin{array}{lcllcl}
||p||^{\mathcal{B}}_{\varepsilon} & \triangleq & L(p) & ||Z||^{\mathcal{B}}_{\varepsilon} & \triangleq & \varepsilon(Z) \\
||\varphi \wedge \psi||^{\mathcal{B}}_{\varepsilon} & \triangleq & ||\varphi||^{\mathcal{B}}_{\varepsilon} \cap ||\psi||^{\mathcal{B}}_{\varepsilon} & ||\neg\varphi||^{\mathcal{B}}_{\varepsilon} & \triangleq & S \setminus ||\varphi||^{\mathcal{B}}_{\varepsilon} \\
||\mu Z \cdot \varphi||^{\mathcal{B}}_{\varepsilon} & \triangleq & \mathit{lfp}^{\subseteq}\left(\lambda Q \cdot ||\varphi||^{\mathcal{B}}_{\varepsilon[Z \mapsto Q]}\right) & ||\Diamond\varphi||^{\mathcal{B}}_{\varepsilon} & \triangleq & \mathit{pre}[R](||\varphi||^{\mathcal{B}}_{\varepsilon})
\end{array}
$$

*where $Z \in Var$ is a fixpoint variable, $\varepsilon[Z \mapsto Q]$ denotes the object assignment that is the same as $\varepsilon$ except that $\varepsilon[Z \mapsto Q](Z) = Q$, and $\mathit{lfp}^{\subseteq} f$ is the $\subseteq$-least fixpoint of $f$.*

For a closed $L_\mu$ formula $\varphi$, $||\varphi||^{\mathcal{B}}_{\varepsilon} = ||\varphi||^{\mathcal{B}}_{\varepsilon'}$ for any $\varepsilon$ and $\varepsilon'$. Thus, we write $||\varphi||^{\mathcal{B}}$ for that value. When it is clear from context, we simply call the inductive semantics as the semantics of $L_\mu$ and omit $\mathcal{B}$.

The semantics $||\varphi||^{\mathcal{B}}$ defines a mapping from the states $S$ to a $\mathbf{2}$-valued truth domain such that for each state $s \in S$,

$$
||\varphi||^{\mathcal{B}}(s) = \begin{cases} \mathsf{t} & \text{if } s \in ||\varphi||^{\mathcal{B}} \\ \mathsf{f} & \text{if } s \notin ||\varphi||^{\mathcal{B}} \end{cases}
$$

If $||\varphi||^{\mathcal{B}}(s) = \mathsf{t}$, that means $\varphi$ is satisfied at $s$, i.e., $\mathcal{B}, s \models \varphi$; otherwise, $\varphi$ is refuted at $s$, i.e., $\mathcal{B}, s \models \neg\varphi$.

Given a model $\mathcal{B}$ and a temporal property $\varphi$, model checking automatically determines whether $\varphi$ is satisfied or refuted at the states in $\mathcal{B}$ according to the semantic of $\varphi$ . For the BTS $B_1$ shown in Figure 2.2, let $AP = \{p, q\}$ be a set of atomic propositions, and a labeling function $L_1$ be defined as $L_1(p) = \{s_1, s_2\}$, and $L_1(q) = \{s_2\}$. We present several example properties over the Kripke structure $\mathcal{B}_1 = \langle B_1, L_1 \rangle$.

- *Propositional property* $(\varphi \triangleq p \wedge q)$). This property requires that both $p$ and $q$ are true at the same time. According to the labeling function $L_1$, the property $\varphi$ is satisfied at the state $s_2$.

- *Modal property* ($\varphi \triangleq EX(p \wedge q)$). This property requires that a state has at least one successor where both $p$ and $q$ are true. Since $p \wedge q$ holds at $s_2$ and $s_2$ is a successor of $s_1$ and $s_2$, the property $\varphi$ is satisfied at the states $s_1$ and $s_2$.

- *Reachability property* ($\varphi \triangleq EF(p \wedge q)$). This property requires that a state can reach some state satisfying $p \wedge q$ in zero or more steps. Since $p \wedge q$ is satisfied at $s_2$, the property $\varphi$ is satisfied at the states $s_2$, $s_1$, and $s_3$, which can reach $s_2$ in 0, 1 and 2 steps, respectively.

- *Non-termination property* ($\varphi \triangleq EGp$)). This property requires that there is an infinite path from a state such that $p$ always holds on the path. Note that $p$ is true at $s_1$ and $s_2$ in $\mathcal{B}_1$. Since there is a self-loop at $s_2$, and $s_1$ is a predecessor of $s_2$, the property $\varphi$ is satisfied at $s_1$ and $s_2$.

## 2.3 Abstraction Framework

Abstraction in model checking builds a smaller abstract model to approximate a large or infinite program, and uses the abstract model checking result to derive the one at concrete level. An abstraction framework [CC92] formalizes the connection between concrete and abstract model checking, which defines a class of modeling formalisms for abstraction, the approximation relation between concrete and abstract models, and the preservation relation of temporal properties.

Abstract model checking starts with an abstraction of statespace. Let $\mathcal{B} = \langle B, L_B \rangle$ be a Kripke structure representing the concrete model of a program, where $B = \langle C, R \rangle$ is a BTS. The set of states $C$ in $B$ is called a *concrete* statespace. An *abstract* statespace approximating $C$ is a finite set of states $A$ together with a *soundness* relation $\rho \subseteq C \times A$, where $(c, a) \in \rho$ means that $a$ *$\rho$-approximates* $c$. $\rho$ induces a *concretization* function $\gamma(a) \triangleq \{c \mid (c, a) \in \rho\}$. That is, $\gamma(a)$ is the set of all concrete states approximated by $a$. For a set $Q \subseteq A$, we define $\gamma(Q) \triangleq \cup_{a \in Q} \gamma(a)$.

The abstract states describe concrete states in an abstract way. For the purpose of model checking, other components of $\mathcal{B}$ are lifted into the abstract world as well. An abstract model $\mathcal{M} = \langle M, L_M \rangle$ describes abstract program behaviors, consisting of a transition system $M$ over the abstract statespace $A$ and a labeling function $L_M$ for $A$. Let $a$ and $c$ be an abstract and concrete states, respectively, and $a$ approximates $c$. We distinguish three kinds of abstraction frameworks based on approximation of program behaviors and preservation of temporal properties.

1. *over-approximation*: Intuitively, an over-approximating abstract model $\mathcal{M}$ consists of "more" behaviors than the concrete one. Therefore, if an *universal* property holds on the abstract model, it also holds on the concrete one. Since abstract model checking starts from a notion of abstraction of states, the preservation of temporal properties is often defined on the level of individual states. That is,

$$\forall \varphi \in \Box L_\mu \cdot (\mathcal{M}, a \models \varphi) \Rightarrow (\mathcal{B}, c \models \varphi)$$

   Note that over-approximation only preserves soundness for the universal fragment of temporal logic. Therefore, if an universal property $\varphi$ is *refuted* at $a$, it does not imply that $\varphi$ is refuted at $c$ as well.

2. *under-approximation*: An under-approximating abstract model $\mathcal{M}$ consists of "less" behaviors than the concrete program. In this case, if an *existential* property holds on the abstract model, it also holds on the concrete one. That is,

$$\forall \varphi \in \Diamond L_\mu \cdot (\mathcal{M}, a \models \varphi) \Rightarrow (\mathcal{B}, c \models \varphi)$$

   Similar to over-approximation, under-approximation only preserves soundness for a fragment of temporal properties.

3. *exact-approximation*: An exact-approximating abstract model $\mathcal{M}$ combines both over- and under-approximating abstract program behaviors, which allows for verification and

refutation of *arbitrary* $L_\mu$ formulas in the framework. In this case,

$$\forall \varphi \in L_\mu \cdot (\mathcal{M}, a \models \varphi) \Rightarrow (\mathcal{B}, c \models \varphi)$$

If the reverse direction also holds, that is, the values of $\varphi$ over $\mathcal{M}$ are either *true* or *false*, then $\varphi$ is satisfied (resp. refuted) over the concrete model if and only if it is satisfied (resp. refuted) over the abstract model. We refer to such exact-approximation as being *strong*. More general exact-approximation frameworks, called *weak*, allow for unknown values of formulas over the abstract model. In this case, the preservation of *true* and *false* results only hold from the abstract to the concrete level.

In the rest of this section, we describe modeling formalisms and approximation relations between concrete and abstract models in strong and weak exact-approximations, respectively.

### 2.3.1 Strong Exact-Approximation

In strong exact-approximation frameworks, we use boolean transitions system as the modeling formalism, and represent abstract models using Kripke structures. The approximation relation between abstract and concrete models is based on *bisimulation*.

**Definition 2.5** (Bisimulation between BTSs)**.** *[Mil89] Let $B_1 = \langle S_1, R_1 \rangle$ and $B_2 = \langle S_2, R_2 \rangle$ be two BTSs. $H \subseteq S_1 \times S_2$ is a* bisimulation *between $B_1$ and $B_2$ if for any $(s_1, s_2) \in H$, the following two conditions hold:*

$$(a)\ \forall t_1 \in S_1 \cdot (s_1, t_1) \in R_1 \Rightarrow \exists t_2 \in S_2 \cdot (s_2, t_2) \in R_2 \wedge (t_1, t_2) \in H$$

$$(b)\ \forall t_2 \in S_2 \cdot (s_2, t_2) \in R_2 \Rightarrow \exists t_1 \in S_1 \cdot (s_1, t_1) \in R_1 \wedge (t_1, t_2) \in H$$

*In this case, we say $B_1$ and $B_2$ are* H-bisimilar*, written $B_1 \equiv_H B_2$.*

**Definition 2.6** (Equivalence between Labeling Functions)**.** *Let $AP$ be a set of atomic propositions. Let $L_1$ and $L_2$ be a labeling function for the statespaces $S_1$ and $S_2$, respectively. Let*

*$H \subseteq S_1 \times S_2$ be a relation. $L_1$ and $L_2$ are* H-equivalent*, denoted $L_1 \equiv_H L_2$, if the following condition hold:*

$$\forall (s_1, s_2) \in H \cdot \forall p \in AP \cdot s_1 \in L_1(p) \Leftrightarrow s_2 \in L_2(p)$$

Note that Definition 2.5 and Definition 2.6 are defined over a specific relation $H$. The reason for this is that in abstract model checking, we often consider the relationship between concrete and abstract models with respect to the soundness relation between concrete and abstract statespaces, instead of an arbitrary one. A concrete and an abstract Kripke structures are *bisimilar* if the underlying transition systems and labeling functions respectively are bisimilar and equivalent with respect to the soundness relation.

**Definition 2.7** (Bisimulation between Kripke Structures)**.** *[CGP99] Let $\rho$ be a soundness relation between a concrete statespace $C$ and an abstract statespace $A$. Let $AP$ be a set of atomic propositions. Let $\mathcal{B}_1 = \langle B_1, L_1 \rangle$ be an abstract Kripke structure over $A$, and $\mathcal{B}_2 = \langle B_2, L_2 \rangle$ be a concrete Kripke structure over $C$, where $B_1 = \langle A, R_1 \rangle$ and $B_2 = \langle C, R_2 \rangle$ are the abstract and the concrete BTS, respectively. $\mathcal{B}_1$ is* bisimilar *to $\mathcal{B}_2$ iff $B_1 \equiv_\rho B_2$ and $L_1 \equiv_\rho L_2$, i.e., for any $(c_1, a_1) \in \rho$, the following conditions hold:*

$$(a)\ \forall a_2 \in A \cdot (a_1, a_2) \in R_1 \Rightarrow \exists c_2 \in C \cdot (c_1, c_2) \in R_2 \wedge (c_2, a_2) \in \rho$$

$$(b)\ \forall c_2 \in C \cdot (c_1, c_2) \in R_2 \Rightarrow \exists a_2 \in A \cdot (a_1, a_2) \in R_1 \wedge (c_2, a_2) \in \rho$$

$$(c)\ \forall p \in AP \cdot a \in L_1(p) \Leftrightarrow c \in L_2(p)$$

The following theorem shows that if $\mathcal{B}_1$ and $\mathcal{B}_2$ are bisimilar, then for any abstract $a$ and a concrete state $c$ approximated by $a$, an $L_\mu$ formula $\varphi$ is satisfied (resp. refuted) at $a$ over $\mathcal{B}_1$ iff it is satisfied (resp. refuted) at $c$ over $\mathcal{B}_2$, i.e., $\forall \varphi \in L_\mu \cdot (\mathcal{B}_1, a \models \varphi) \Leftrightarrow (\mathcal{B}_2, c \models \varphi)$.

**Theorem 2.8.** *[CGP99] Let $\mathcal{B}_1 = \langle B_1, L_1 \rangle$ be an abstract Kripke Structure that is bisimilar to a concrete Kripke Structure $\mathcal{B}_2 = \langle B_2, L_2 \rangle$, and $\varphi \in L_\mu$. Then, $\gamma(||\varphi||^{\mathcal{B}_1}) = ||\varphi||^{\mathcal{B}_2}$.*

### 2.3.2 Weak Exact-Approximation

Weak exact-approximation uses more expressive modeling formalisms than boolean transition systems to support combination of over- and under-approximations. These formalisms typically have two types of transition relations, *may* and *must*, corresponding to over- and under-approximating behaviors, respectively. We refer to these formalisms as *partial transition systems* since they allow us to describe undefined behaviors of programs. Partial transition systems are equivalent to *multi-valued transition systems* where transition relations are defined w.r.t. a multi-valued domain. We introduce both of them below.

*Partial transition systems*. We start by introducing exact-approximation based on partial transition systems.

A *partial transition system* allows us to describe possible and necessary behaviors of programs. In the following, we first we first define several transitions systems. Each of them can be referred to as a partial transition system.

**Definition 2.9** (GKMTS, MixTS, and KMTS)**.** *[DGG97, BG99, HJS01, SG04] A* Generalized Kripke Modal Transition System *(GKMTS) is a tuple* $M = \langle S, R^{\text{may}}, R^{\text{must}} \rangle$*, where* $S$ *is the statespace, and* $R^{\text{may}} \subseteq S \times S$*,* $R^{\text{must}} \subseteq S \times \mathbf{2}^S$ *are the* may *and* must *transition relations, respectively. A* Mixed Transition System *(MixTS) is a GKMTS s.t.* $R^{\text{must}} \subseteq S \times S$*. A* Kripke Modal Transition System *(KMTS) is a MixTS s.t.* $R^{\text{must}} \subseteq R^{\text{may}}$.

Intuitively, *may* and *must* transitions represent possible and necessary behaviors, respectively. Examples of partial transition systems are shown in Figure 2.3(a)-(b). In this thesis, we often write $s \xrightarrow{\text{may}} t$ for $(s,t) \in R^{\text{may}}$, $s \xrightarrow{\text{must}} t$, and $s \xrightarrow{\text{must}} Q$ for $(s,t) \in R^{\text{must}}$ and $(s,Q) \in R^{\text{must}}$, respectively. Note that partial transition systems subsume BTSs: a BTS can be seen as a special KMTS where $R^{\text{may}} = R^{\text{must}}$.

The pre-image function in a partial transition system is defined over a pair of sets $\langle Q_1, Q_2 \rangle \in$

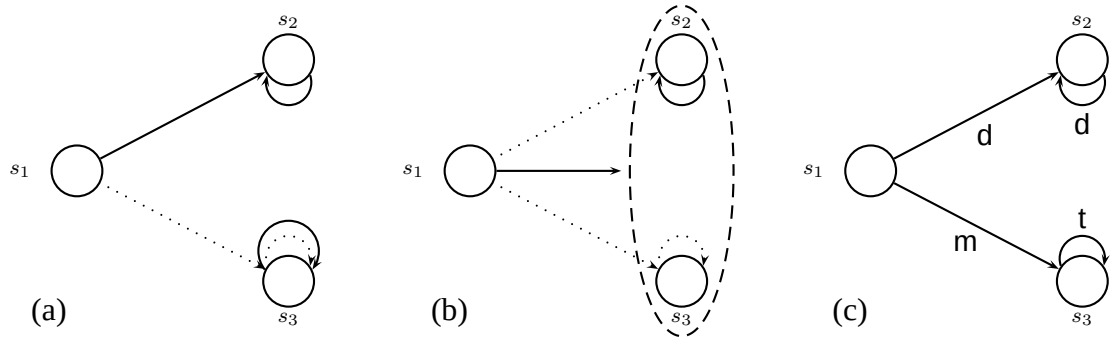


Figure 2.3: (a)-(b) Examples of partial transition systems (dotted lines represent *may* transitions and solid – *must*): (a) a MixTS $M_1$ and (b) a GKMTS $M_2$, where the dashed ellipse denotes the set of states that are the destination of the *must* hyper-transition from $s_1$ ; and (c) a $\mathbf{4}$-valued transition system $M_3$.

$\mathbf{2}^S \times \mathbf{2}^S$, and takes into account the two types of transition relations:

$$pre[\langle R^{\text{must}}, R^{\text{may}}\rangle](\langle Q_1, Q_2\rangle) \triangleq \langle pre_{\mathsf{U}}(Q_1), pre_{\mathsf{O}}(Q_2)\rangle$$

where

$$pre_{\mathsf{U}}(Q) \triangleq \begin{cases} \{s \mid \exists t \in Q \cdot s \xrightarrow{\text{must}} t\} & \text{if } M \text{ is a MixTS} \\ \{s \mid \exists U \subseteq Q \cdot s \xrightarrow{\text{must}} U\} & \text{if } M \text{ is a GKMTS} \end{cases}$$

$$pre_{\mathsf{O}}(Q) \triangleq \{s \mid \exists t \in Q \cdot s \xrightarrow{\text{may}} t\}$$

Let $AP$ be a set of atomic propositions, and $Lit(AP)$ be a set of literals of $AP$. A *partial* labeling function $L : S \to 2^{Lit(AP)}$ assigns to each state $s$ a set of literals that are true at $s$. That is, $p$ is true at $s$ if $p \in L(s)$, and false if $\neg p \in L(s)$; inconsistent if $p, \neg p \in L(s)$; otherwise, the value of $p$ at $s$ is unknown. A pair $\mathcal{M} = \langle M, L\rangle$ of a partial transition system $M$ and a labeling $L$ is called a *partial* model. The semantics of an $L_\mu$ formula $\varphi$ over $\mathcal{M}$ is given by a pair $e = \langle U, O\rangle$, where $U, O \subseteq S$. Intuitively, $U$ is the set of states that must satisfy $\varphi$, and $O$ is the set of states that do not refute (may satisfy) $\varphi$. The inductive semantics of $\varphi$ over $\mathcal{M}$, denoted $||\varphi||^{\mathcal{M}}$, is defined as follows.

Let $e$ be a pair $\langle U, O\rangle$. Let $\overline{U}$ and $\overline{O}$ denote the sets $S \setminus U$ and $S \setminus O$, respectively. We write $\mathsf{U}(e)$ and $\mathsf{O}(e)$ to denote $U$ and $O$, respectively. We define the operators $\sim$ and $\sqcap$ as follows:

$$\sim\langle U, O\rangle \triangleq \langle \overline{O}, \overline{U}\rangle$$
$$\langle U_1, O_1\rangle \sqcap \langle U_2, O_2\rangle \triangleq \langle U_1 \cap U_2, O_1 \cap O_2\rangle\,.$$

**Definition 2.10** (Inductive Semantics of $L_\mu$ over Partial Models)**.** *[DGG97, BG99, HJS01, SG04]. Let $\mathcal{M} = \langle M, L\rangle$ be a partial model, $M = \langle S, R^{\text{may}}, R^{\text{must}}\rangle$, Var a set of fixpoint variables, and $\varepsilon$ : Var $\to \mathbf{2}^S \times \mathbf{2}^S$. The inductive semantics of $\varphi \in L_\mu$ is:*

$$\begin{aligned}
||p||^{\mathcal{M}}_{\varepsilon} &\triangleq \langle \{s \mid p \in L(s)\}, \{s \mid \neg p \notin L(s)\}\rangle \\
||\neg\varphi||^{\mathcal{M}}_{\varepsilon} &\triangleq \sim||\varphi||^{\mathcal{M}}_{\varepsilon} \\
||\varphi \wedge \psi||^{\mathcal{M}}_{\varepsilon} &\triangleq ||\varphi||^{\mathcal{M}}_{\varepsilon} \sqcap ||\psi||^{\mathcal{M}}_{\varepsilon} \\
||\Diamond\varphi||^{\mathcal{M}}_{\varepsilon} &\triangleq \langle pre_{\mathsf{U}}(\mathsf{U}(||\varphi||^{\mathcal{M}}_{\varepsilon})), pre_{\mathsf{O}}(\mathsf{O}(||\varphi||^{\mathcal{M}}_{\varepsilon}))\rangle \\
||Z||^{\mathcal{M}}_{\varepsilon} &\triangleq \varepsilon(Z) \\
||\mu Z \cdot \varphi||^{\mathcal{M}}_{\varepsilon} &\triangleq \langle \mathit{lfp}^{\subseteq}\left(\lambda Q \cdot \mathsf{U}(||\varphi||^{\mathcal{M}}_{\varepsilon[Z\mapsto Q]})\right), \mathit{lfp}^{\subseteq}\left(\lambda Q \cdot \mathsf{O}(||\varphi||^{\mathcal{M}}_{\varepsilon[Z\mapsto Q]})\right)\rangle
\end{aligned}$$

*where $Z \in$ Var and lfp is the least fixpoint.*

The semantics $||\varphi||^{\mathcal{M}}$ defines a mapping from the states $S$ to a **4**-valued truth domain such that for each state $s \in S$,

$$||\varphi||^{\mathcal{M}}(s) = \begin{cases} \mathsf{t} & \text{if } s \in \mathsf{U}(||\varphi||^{\mathcal{M}}) \cap \mathsf{O}(||\varphi||^{\mathcal{M}}) \\ \mathsf{f} & \text{if } s \notin \mathsf{U}(||\varphi||^{\mathcal{M}}) \cup \mathsf{O}(||\varphi||^{\mathcal{M}}) \\ \mathsf{m} & \text{if } s \in \mathsf{O}(||\varphi||^{\mathcal{M}}) \setminus \mathsf{U}(||\varphi||^{\mathcal{M}}) \\ \mathsf{d} & \text{if } s \in \mathsf{U}(||\varphi||^{\mathcal{M}}) \setminus \mathsf{O}(||\varphi||^{\mathcal{M}}) \end{cases}$$

If $||\varphi||^{\mathcal{M}}(s)$ is $\mathsf{t}$ or $\mathsf{f}$, it means that $\varphi$ is satisfied or refuted at $s$, respectively; $\mathsf{m}$ means that the value of $\varphi$ is unknown at $s$, and $\mathsf{d}$ denotes an inconsistent result since in this case $s$ must satisfy $\varphi$ ($s \in \mathsf{U}(||\varphi||^{\mathcal{M}})$) and must refute $\varphi$ ($s \notin \mathsf{O}(||\varphi||^{\mathcal{M}})$) at the same time. For example, consider

the MixTS $M_1$ shown in Figure 2.3(a). Let $AP = \{p, q\}$ be the set of atomic propositions and the labeling function $L_1$ be defined as $L_1(s_1) = \{p, q\}$, $L_1(s_2) = \{p\}$, and $L_1(s_3) = \{q\}$. Over the partial model $\mathcal{M}_1 = \langle M_1, L_1 \rangle$, we have $||p||^{\mathcal{M}_1} = \langle \{s_1, s_2\}, \{s_1, s_2, s_3\} \rangle$; that is, $p$ is true at $s_1$ and $s_2$, and unknown at $s_3$; For the modal property $\varphi \triangleq \Diamond p$, we have $||\varphi||^{\mathcal{M}_1} = \langle \{s_1, s_2\}, \{s_1, s_3\} \rangle$, that is, the property $\varphi$ is true at $s_1$, unknown at $s_3$, and inconsistent at $s_2$.

The approximation relation between a partial model and a concrete model (Kripke structure) is defined based on *mixed simulation*.

**Definition 2.11** (Mixed Simulation between MixTSs)**.** *[DGG97] Let $M_1 = \langle S_1, R_1^{may}, R_1^{must} \rangle$ and $M_2 = \langle S_2, R_2^{may}, R_2^{must} \rangle$ be two MixTSs. $H \subseteq S_1 \times S_2$ is a* mixed simulation *between $M_1$ and $M_2$ if for any $(s_1, s_2) \in H$, the following two conditions hold:*

$$(a)\ \forall t_2 \in S_2 \cdot (s_2, t_2) \in R_2^{\text{must}} \Rightarrow \exists t_1 \in S_1 \cdot (s_1, t_1) \in R_1^{\text{must}} \wedge (t_1, t_2) \in H$$

$$(b)\ \forall t_1 \in S_1 \cdot (s_1, t_1) \in R_1^{\text{may}} \Rightarrow \exists t_2 \in S_2 \cdot (s_2, t_2) \in R_2^{\text{may}} \wedge (t_1, t_2) \in H$$

*In this case, we say $M_2$ $H$-*simulates *$M_1$, written $M_2 \preceq_H M_1$.*

Intuitively, $M_2$ simulates $M_1$ whenever $M_2$ is less precise about its behaviour than $M_1$. This definition generalizes to GKMTSs [SG04].

**Definition 2.12** (Approximation between Labeling Functions)**.** *Let $AP$ be a set of atomic propositions. Let $L_1$ and $L_2$ be a partial labeling function for the statespace $S_1$ and $S_2$, respectively. Let $H \subseteq S_1 \times S_2$ be a relation. $L_1$* H-approximates *$L_2$, denoted $L_1 \preceq_H L_2$, if the following condition hold:*

$$\forall (s_1, s_2) \in H \cdot L_1(s_1) \subseteq L_2(s_2)$$

Let $C$ and $A$ be a concrete and an abstract statespace, respectively, and $\rho$ be the soundness relation, and $\gamma$ be the concretization function. A partial transition system $M$ over $A$ *approximates* a concrete BTS $B$ over $C$ (or, equivalently $B$ *refines* $M$) iff $M$ $\rho$-simulates $B$, i.e.,

$M \preceq_\rho B$. The set of all BTSs that refine $\mathcal{M}$ is denoted by $\mathbb{C}[M]$. Let $L_M$ and $L_B$ be the state labelings for $A$ and $C$, respectively. $L_M$ *approximates* $L_B$ iff $L_M$ *$\rho$-approximates* $L_B$, i.e., $L_M \preceq_\rho L_B$. A partial model $\mathcal{M} = \langle M, L_M \rangle$ *approximates* a concrete model $\mathcal{B} = \langle B, L_B \rangle$ (or, equivalently, $\mathcal{B}$ *refines* $\mathcal{M}$) iff $M$ approximates $B$ and $L_M$ approximates $L_B$.

**Definition 2.13** (Approximation between Partial Models)**.** *[DGG97] Let $\rho$ be a soundness relation between a concrete statespace $C$ and an abstract statespace $A$. Let $\mathcal{M} = \langle M, L_M \rangle$ be a partial model over $A$ where $M = \langle A, R_M^{\text{may}}, R_M^{\text{must}} \rangle$ is a MixTS. Let $\mathcal{B} = \langle B, L_B \rangle$ be a concrete model over $C$ where $B = \langle C, R_B^{\text{may}}, R_B^{\text{must}} \rangle$ is a BTS, i.e., $R_B^{\text{may}} = R_B^{\text{must}}$.* $\mathcal{M}$ approximates $\mathcal{B}$ *iff $M \preceq_\rho B$ and $L_M \preceq_\rho L_B$. That is, for any $(c_1, a_1) \in \rho$, the following conditions hold:*

$$(a)\ \forall a_2 \in A \cdot (a_1, a_2) \in R_M^{\text{must}} \Rightarrow \exists c_2 \in C \cdot (c_1, c_2) \in R_B^{\text{must}} \wedge (c_2, a_2) \in \rho$$

$$(b)\ \forall c_2 \in C \cdot (c_1, c_2) \in R_B^{\text{may}} \Rightarrow \exists a_2 \in A \cdot (a_1, a_2) \in R_M^{\text{may}} \wedge (c_2, a_2) \in \rho$$

$$(c)\ L_M(a_1) \subseteq L_B(c_1)$$

This definition generalizes to partial models over GKMTSs [SG04]. We denote the set of all concrete refinements of $\mathcal{M}$ by $\mathbb{C}[\mathcal{M}]$.

The following theorem shows that if $\mathcal{M}$ approximates $\mathcal{B}$, then for any abstract $a$ and a concrete state $c$ approximated by $a$, if an $L_\mu$ formula $\varphi$ is satisfied (resp. refuted) at $a$ over $\mathcal{M}$, then it is satisfied (resp. refuted) at $c$ over $\mathcal{B}$, i.e., $\forall \varphi \in L_\mu \cdot (\mathcal{M}, a \models \varphi) \Rightarrow (\mathcal{B}, c \models \varphi)$.

**Theorem 2.14.** *[DGG97, SG04] Let $\mathcal{M} = \langle M, L_M \rangle$ be a partial model that approximates a concrete model $\mathcal{B} = \langle B, L_B \rangle$, and $\varphi \in L_\mu$. Then, $\gamma(\mathsf{U}(|||\varphi|||^{\mathcal{M}})) \subseteq \mathsf{U}(\|\varphi\|^{\mathcal{B}})$, and $\gamma(\overline{\mathsf{O}(\|\varphi\|^{\mathcal{M}})})$ $\subseteq \overline{\mathsf{O}(\|\varphi\|^{\mathcal{B}})}$.*

*Multi-valued Transition Systems.* We now introduce exact-approximation using multi-valued transition systems, which is based on extension of sets and transitions to multi-valued truth domains.

We first introduce *multi-valued* sets. Given a collection of elements $C$ and a truth domain $\mathcal{D}$ , a *$\mathcal{D}$-valued* set $X$ over $C$ is a total function $S \to \mathcal{D}$. For example, a $\mathbf{2}$-valued set is simply a

boolean or a classical set, and a $\mathbf{4}$-valued set is a function from $C$ to Belnap logic $\mathbf{4}$. Given a set over $C$ and an element $c$ from $C$, the value $X(c)$ represents the degree to which $c$ belongs to $C$. For example, $X(s) = \mathsf{t}$ means that $c$ is contained in $X$, $\mathsf{f}$ means that $c$ is not contained in $C$, $\mathsf{m}$ means that $c$ may be contained in $X$, and $\mathsf{d}$ indicates an inconsistent case. A fuzzy set [Zad87] is also a multi-valued set defined over fuzzy logic, where the truth values are formed by the set of all real numbers in the closed interval $[0, 1]$ such that $0$ stands for false, $1$ for true, and the remaining values stand for degrees of truth. We use $\mathcal{D}^C$ to denote all the $\mathcal{D}$-valued sets over $C$.

Set ordering and operations are defined by pointwise extensions. Let $S_1, S_2 \in \mathcal{D}^C$ be two $\mathcal{D}$-valued sets. Then

$$\begin{aligned} S_1 \subseteq S_2 &\triangleq \forall x \cdot S_1(x) \sqsubseteq S_2(x) \\ S_1 \cup S_2 &\triangleq \lambda x \cdot S_1(x) \vee S_2(x) \\ \overline{S_1} &\triangleq \lambda x \cdot \neg S_1(x) \\ S_1 \cap S_2 &\triangleq \lambda x \cdot S_1(x) \wedge S_2(x) \end{aligned}$$

Note that the classical set theory is a special case where the truth domain is $\mathbf{2}$.

**Definition 2.15** (Multil-Valued Transition Systems)**.** *A $\mathcal{D}$-valued transition system is a tuple $M = \langle S, R, \mathcal{D} \rangle$, where*

- *$S$ is a set of states,*
- *$\mathcal{D}$ is a truth domain,*
- *$R : S \times S \to \mathcal{D}$ is a transition relation,*

In the rest of this thesis, we only consider multi-valued transition systems with truth domains from the set of $\{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$, which are equivalent to the partial transition systems we discussed previously. An example of $\mathbf{4}$-valued transition system is shown in Figure 2.3(c), and a BTS is simply a $\mathbf{2}$-valued transition system. For a relation $R : S \times S \to \mathcal{D}$, we define the *preimage* of a set $Q \in \mathcal{D}^S$ w.r.t. $R$ as

$$\mathit{pre}[R](Q) \triangleq \lambda s \in S \cdot \bigvee_{t \in S} R(s, t) \wedge Q(t)$$

Let $AP$ be a set of atomic propositions. A labeling function associated with a multi-valued transition system is a function $L : AP \to \mathbf{4}^S$. A pair $\mathcal{M} = \langle M, L \rangle$ of a multi-valued transition system $M$ and a state labeling $L$ is called a *multi-valued Kripke structure*. The *inductive semantics* of an $L_\mu$ formulas $\varphi$ over $\mathcal{M}$, denoted $||\varphi||^{\mathcal{M}}$, is a $\mathbf{4}$-valued set over $S$, which is defined in the same way as that over classical Kripke structures (Definition 2.4), with the only difference that set operations are defined based on multi-valued truth domains.

Multi-valued Kripke structures are isomorphic to partial models w.r.t. $L_\mu$ formulas [GJ03, GWC06a]. For example, a labeling function $L : S \to 2^{Lit(AP)}$ can be transformed to a function $L' : AP \to \mathbf{4}^S$ such that $L'(p)(s)$ is t if $p \in L(s)$ and $\neg p \notin L(s)$, f – if $\neg p \in L(s)$ and $p \notin L(s)$, m – if $p, \neg p \notin L(s)$, and d – if $p, \neg p \in L(s)$. A MixTS can be easily transformed to a $\mathbf{4}$-valued transition system by assigning *may* and *must* transitions the value t, *may* and not *must* – m, *must* and not *may* – d, and empty transitions – f, e.g., the MixTS in Figure 2.3(a) can be transformed to the $\mathbf{4}$-valued transition system in Figure 2.3(c). Similarly, a KMTS and a GKMTS can be respectively transformed to a $\mathbf{3}$-valued transition system and a $\mathbf{4}$-valued transition system with extension of hyper-transitions.

The isomorphism lifts approximation relation from partial models to multi-valued Kripke structures.

**Definition 2.16** (Approximation between Multi-Valued Models)**.** *[GJ03, GWC06a] Let $\mathcal{B}$ be a concrete model over a statespace $C$, and $\mathcal{M}$ be a multi-valued Kripke structure over an abstract statespace $A$.* $\mathcal{M}$ approximates *$\mathcal{B}$ iff the partial model $\mathcal{M}'$ that is isomorphic to $\mathcal{M}$ approximates $\mathcal{B}$.*

In this case, the preservation of temporal properties can be characterized using the information ordering on truth domains.

**Theorem 2.17.** *[GJ03, GWC06a] Let $\mathcal{M}$ be a multi-valued Kripke structure that approximates a concrete model $\mathcal{B}$, and $\varphi \in L_\mu$. Then, for any abstract state $a$ and a corresponding concrete state $c$, $||\varphi||^{\mathcal{M}}(a) \preceq ||\varphi||^{\mathcal{B}}(c)$.*

That is, if $||\varphi||^{\mathcal{M}}(a)$ is *true* or *false*, $||\varphi||^{\mathcal{B}}(c)$ is also *true* or *false*, respectively.

## 2.4 Summary

In this chapter, we have introduced temporal logics and computational models used for model checking, as well as the abstraction framework for strong and weak exact-approximations.

We have described several modeling formalisms for abstract model checking, which are used throughout in this thesis. Specifically, in Chapter 3, we study a strong exact-approximation technique – symmetry reduction, where symmetry-reduced structures are represented using boolean transition systems. In Chapter 4, we study software model checking of recursive programs, and discuss abstract analysis based on multi-valued transition systems, where multi-valued logic provides a convenient way to define operations used for abstract analysis. In Chapter 5, we study modeling formalisms for weak exact-approximation based on partial transition systems, where the *may* and *must* transitions allow for a natural way to understand abstract behaviors.

# Chapter 3

# Full Virtual Symmetry Reduction

In this chapter, we investigate symmetry reduction, which is a special technique for strong exact-approximation. Based on our characterization of symmetry reduction from the perspective of abstraction, we provide the solutions to identification and symbolic symmetry reduction of fully virtually symmetric programs.

## 3.1 Introduction

Symmetry is naturally exhibited in concurrent programs or protocols that consist of synchronization and coordination of several identical processes. Such symmetry can be seen as a form of redundancy, and model checking can then be performed on the symmetry-reduced quotient structure that is bisimilar to, and often substantially smaller than, the original model of the program [CJEF96, ES96]. To extend symmetry reduction to "almost" symmetric programs, Emerson et al. [EHT00] defined *virtual symmetry* as the most general condition under which the transition system of a program is bisimilar to its symmetry-reduced quotient structure, and thus symmetry reduction can be applied. Although virtual symmetry increases a potential domain of problems that can be symmetry reduced, its practical application depends on successful solutions to the following questions:

(1) How does one identify virtual symmetry without building the transition system of the program (which is typically infeasible)?

(2) How does one symbolically model check a virtually symmetric program?

In this chapter, we answer these questions for *fully* virtually symmetric programs, i.e., programs that are virtually symmetric up to exchanging the roles of processes. This form of symmetry typically arises in programs composed of similar, but not identical, processes. An example of such a program is Readers-and-Writers (R&W): a variant of a well-known mutual exclusion protocol (MUTEX), where writer processes are given a higher priority than reader processes for entering the critical section [EHT00]. Like full symmetry, full virtual symmetry often leads to an exponential reduction on the statespace of the system, which is the focus of our study in this chapter.

Specifically, this chapter includes the following technical contributions:

1. We provide a characterization of symmetry reduction from the perspective of abstraction. We consider symmetry reduction as a special technique for strong exact-approximation, where existential and universal abstractions over symmetric equivalence classes coincide.

2. We provide an algorithmic way to identify full virtual symmetry. We first show that virtual symmetry of a program is equivalent to virtual symmetry of local transitions of processes, which reduces the problem of checking virtual symmetry, a global property of a program, to a local property of each transition. Then, based on our characterization of symmetry reduction, we further reduce identification of full virtual symmetry of local transitions to satisfiability of a quantifier-free Presburger formula built directly from the description of the program.

3. We extend the counter abstraction technique to full virtual symmetry, which avoids the bottleneck problem of symbolic symmetry reduction. We achieve this by translating the

description of a fully virtual symmetric program to the description of another program over counter variables of local process states. The resulting program defines a transition system isomorphic to the symmetry-reduced structure of the original program, and can be directly analyzed symbolically.

4. We evaluate our techniques of identification and symbolic symmetry reduction of full virtual symmetry over two families of programs used in practice, where processes have different proprieties and asymmetric permissions for accessing resources, respectively.

This chapter is organized as follows. Section 3.2 reviews the basics of symmetry reduction and fixes the notation used in this chapter. Section 3.3 formalizes the connection between symmetry reduction and abstraction. Section 3.4 introduces our specification language for asymmetric programs. Section 3.5 provides our approach for identifying full virtual symmetry. Section 3.6 describes the extension of counter abstraction to handle fully virtually symmetric programs. Section 3.7 reports our experimental results. Section 3.8 discusses related work, and Section 3.9 concludes this chapter.

## 3.2 Preliminaries

*Symmetry Reduction.* Let $B = (S, R)$ be a transition system[1]. A permutation $\sigma$ on $S$ is a bijection $\sigma : S \to S$. Let $G$ be a permutation group on $S$. The group $G$ induces an equivalence partition on $S$. The equivalence class of a state $s$ is called the *orbit* of $s$ under $G$, defined by $\theta_G(s) \triangleq \{s' \in S \mid \exists \sigma \in G \cdot \sigma(s) = s'\}$. We use $\theta(s)$ to denote the orbit of $s$ when $G$ is clear from the context. The extension of $\theta$ to a set of states $Q \subseteq S$ is defined by $\theta(Q) \triangleq \bigcup_{s \in Q} \theta(s)$.

The *quotient structure* of $B$ induced by $G$ is a transition system $B^G = (S^G, R^G)$ where $S^G \triangleq \{\theta(s) \mid s \in S\}$, and $\forall s, t \in S \cdot (\theta(s), \theta(t)) \in R^G \Leftrightarrow \exists s' \in \theta(s) \cdot \exists t' \in \theta(t) \cdot (s', t') \in R$. A permutation group $G$ is an *automorphism group* for $B$ if it preserves the transition relation

[1]Abstract models in symmetry reduction are defined over boolean transition systems. Therefor, we only consider boolean transition systems in this chapter. For simplicity, we call them transition systems.

$R$, i.e., $\forall s,t \in S \cdot (s,t) \in R \Rightarrow \forall \sigma \in G \cdot (\sigma(s),\sigma(t)) \in R$. A transition system $B$ is called *symmetric* with respect to a permutation group $G$, if $G$ is an automorphism group for it. In this case, $B$ is bisimilar to its symmetry-reduced quotient structure w.r.t. the relation $\rho_G \triangleq \{(s,\theta(s))|s \in S\}$.

**Theorem 3.1.** [CJEF96, ES96] *Let $B = (S,R)$ be a transition system, $G$ be a permutation group acting on $S$. Then, $B \equiv_{\rho_G} B^G$ if $G$ is an automorphism group for $B$.*

Therefore, model checking an $L_\mu$ formula $\varphi$ on $B$ can be reduced to model checking $\varphi$ on $B^G$, provided that the state labeling associated with atomic propositions of $\varphi$ are preserved by $\rho_G$.

*Compositional Transition Systems*. Symmetry reduction is often applied to a parallel composition of similar processes. Such a composition is modeled by a transition system whose statespace is assignments of local states to each process.

Let $I = [1..n]$ be the index set of $n$ processes which have the same set of local states $\mathcal{L}$. The composition of the processes is modeled by a *compositional transition system* $B = (S,R)$, where $S = \mathcal{L}^n$. Then a global state $s$ in $S$ is an $n$-tuple $(l_1,\ldots,l_n) \in \mathcal{L}^n$. For each $i \in I$, we use $s(i)$ to denote the value of $l_i$, i.e., the current local state of the $i$th process, $P_i$, at $s$. Let $K \subseteq I$ be a set of processes. The *group counter* of a local state $L$ with respect to $K$ is a function $\#L[K] : \mathcal{L}^n \rightarrow [0..n]$ such that for any global state $s$, $\#L[K](s) = |\{i \in K \mid s(i) = L\}|$. That is, $\#L[K](s)$ is the number of processes in $K$ whose current state at $s$ is $L$. In particular, if $K = I$, we use $\#L$ to denote $\#L[I]$, and call $\#L$ the *total counter* of $L$.

The *full symmetry group* of $I$, i.e., the group of all permutations acting on $I$, is denoted by $Sym(I)$. A permutation $\sigma \in Sym(I)$ is extended to act on a state $s$ of a compositional transition system $B$ as follows: $\forall i,j \in I \cdot \sigma(s)(i) = s(j) \Leftrightarrow \sigma(i) = j$. In the rest of the chapter, we do not distinguish between a permutation group on $S$ or $I$. A transition system $B$ is called *fully* symmetric if $B$ is symmetric with respect to $Sym(I)$.

## 3.3 Abstraction and Virtual Symmetry

In this section, we formalize the connection between symmetry reduction and abstraction. We then show how this connection can be used to establish a necessary and sufficient condition for the application of symmetry reduction. This condition, referred to by Emerson et al. as *virtual symmetry* [EHT00], generalizes the notion of automorphism-based symmetry [CJEF96, ES96] (see Theorem 3.1) and increases the applicability of symmetry reduction.

In this chapter, we only consider partition-based abstract statespaces where the concretization of abstract states partitions the concrete statespace. In this case, the soundness relation $\rho$ associates each concrete state with exactly one abstract state, which defines an abstraction function $\alpha$ such that for any concrete state $s$, $\alpha(s)$ is the unique $a$ such that $(s, a) \in \rho$.

Given a transition system $B = (S, R)$, let $S_\alpha$ be a statespace that abstracts the concrete statespace $S$. Let $\rho \subseteq S \times S_\alpha$, $\alpha : S \to S_\alpha$, and $\gamma : S_\alpha \to \mathbf{2}^S$ be the soundness relation, abstraction, and concretization functions, respectively. Following [DGG97], we define two transition systems over $S_\alpha$ as follows. A relation $R_\alpha^{\exists\exists} \subseteq S_\alpha \times S_\alpha$ is an *existential abstraction* of $R$ where $(a, b) \in R_\alpha^{\exists\exists}$ if and only if $R$ has a transition between *some* concretizations of $a$ and $b$; $R_\alpha^{\forall\exists}$ is a *universal abstraction* where $(a, b) \in R_\alpha^{\forall\exists}$ if and only if $R$ has a transition from *every* concretization of $a$ to *some* concretization of $b$:

$$R_\alpha^{\exists\exists} \triangleq \{(a, b) \mid \exists s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot R(s, t)\} \qquad \text{(existential abstraction)}$$

$$R_\alpha^{\forall\exists} \triangleq \{(a, b) \mid \forall s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot R(s, t)\} \qquad \text{(universal abstraction)}$$

Accordingly, we define $B_\alpha^{\exists\exists} = (S_\alpha, R_\alpha^{\exists\exists})$ and $B_\alpha^{\forall\exists} = (S_\alpha, R_\alpha^{\forall\exists})$ to be the existential and the universal abstractions of $B$, respectively.

**Theorem 3.2.** *$B$ is $\rho$-bisimilar to $B_\alpha^{\exists\exists}$ if and only if $B_\alpha^{\exists\exists}$ is isomorphic to $B_\alpha^{\forall\exists}$:* $B_\alpha^{\exists\exists} \equiv_\rho B \Leftrightarrow B_\alpha^{\exists\exists} = B_\alpha^{\forall\exists}$.

**Proof:**

The proof of this theorem follows from the definitions of $R_\alpha^{\forall\exists}$, $R_\alpha^{\exists\exists}$, and bisimulation. Note that the isomorphism between $B_\alpha^{\forall\exists}$ and $B_\alpha^{\exists\exists}$ can be defined by the identity function $id : S_\alpha \to S_\alpha$. □

| **Abstraction** | **Symmetry Reduction** |
|---|---|
| abstract statespace : $S_\alpha$ | orbits induced by $G$: $S^G$ |
| soundness relation : $\rho$ | mapping from states to orbits: $\rho_G$ |
| abstraction function : $\alpha$ | orbit function $\theta_G$: $\alpha_G(s) \triangleq \theta_G(s)$ |
| concretization function : $\gamma$ | identity function: $\gamma_G(\theta_G(s)) \triangleq \theta_G(s)$ |
| existential abstraction of $R$ : $R_\alpha^{\exists\exists}$ | quotient of $R$ with respect to $G$: $R^G$ |
| abstract equivalence: $\alpha(s) = \alpha(t)$ | orbit equivalence: $\theta(s) = \theta(t) \Leftrightarrow \exists\sigma \in G \cdot s = \sigma(t)$ |

Table 3.1: A mapping between abstraction and symmetry reduction.

Symmetry reduction of a transition system $B = (S, R)$ with respect to a permutation group $G$ can be seen as a form of abstraction. Formally, let $S^G$, the set of orbits of $S$, be the abstract statespace, and $\rho_G$ be the soundness relation. Under this interpretation, the quotient $B^G$ of $B$ is equivalent to the existential abstraction of $B$. A mapping between key concepts in abstraction and symmetry reduction is summarized in Table 3.1.

Using this connection between symmetry and abstraction, we reinterpret Theorem 3.2 as a necessary and sufficient condition for bisimilarity between $B$ and its quotient $B^G$. Note that

$$R_\alpha^{\exists\exists} = R_\alpha^{\forall\exists} \quad \text{if and only if} \quad (s,t) \in R \;\Rightarrow\; \forall s' \in \gamma(\alpha(s)) \cdot \exists t' \in \gamma(\alpha(t)) \cdot (s',t') \in R$$

In the context of symmetry reduction, $\gamma(\alpha(s))$, the abstract equivalence class of $s$, is simply its orbit $\theta(s)$. Furthermore, $s$ and $s'$ share an orbit, i.e., $s' \in \theta(s)$ if and only if there exists a permutation $\sigma \in G$ such that $s' = \sigma(s)$. By combining the above, we obtain the following theorem.

**Theorem 3.3.** *Let $B = (S, R)$ be a transition system, $G$ be a permutation group acting on $S$, and $\rho_G \triangleq \{(s, \theta(s)) \mid s \in S\}$. Then, $B \equiv_{\rho_G} B^G$ if and only if*

$$\forall s,t \in S \cdot (s,t) \in R \Rightarrow \forall \sigma \in G \cdot \exists \sigma' \in G \cdot (\sigma(s), \sigma'(t)) \in R \tag{3.1}$$

**Proof:**

The proof follows from the reasoning in the previous paragraph. □

Note that Theorem 3.3 is a generalization of Theorem 3.1 since $G$ is no longer required to be an automorphism group for $B$, and thus $B$ is not necessarily symmetric with respect to $G$.

**Definition 3.4.** *A transition system $B$ is* virtually *symmetric with respect to a permutation group $G$ if and only if $B \equiv_{\rho_G} B^G$.*

The problem of establishing a necessary and sufficient condition for a quotient $B^G$ to be bisimilar to $B$ has also been addressed by Emerson et al. [EHT00]. Unlike us, they do not use abstraction, but proceed directly to show that $B$ is virtually symmetric with respect to $G$ if and only if it can be "completed" to a transition system $B'$ such that $B'$ is both symmetric with respect to $G$ and bisimilar to $B$. Thus, Theorem 3.3 provides an alternative (and, in our opinion, simpler) characterization of virtual symmetry. In the rest of the chapter, we show how this new characterization leads to an efficient identification of full virtual symmetry and combination with symbolic model checking.

## 3.4 Specification Language

In this section, we define our specification language for concurrent programs. We begin by reviewing existing approaches for specifying fully symmetric programs in Section 3.4.1 and then extend them to asymmetric programs in Section 3.4.2.

### 3.4.1 Specifying Symmetric Programs

Consider an asynchronous composition of $n$ processes $\{P_1, \ldots, P_n\}$ executing a common concurrent program. Each process is specified using a finite directed graph, called a *synchronization skeleton* [CE81]. Nodes in the graph represent states of the process, and edges, labeled with boolean expressions called *guards*, represent guarded transitions. For example, a synchronization skeleton of a process participating in MUTEX is shown in Figure 3.1(a). A MUTEX

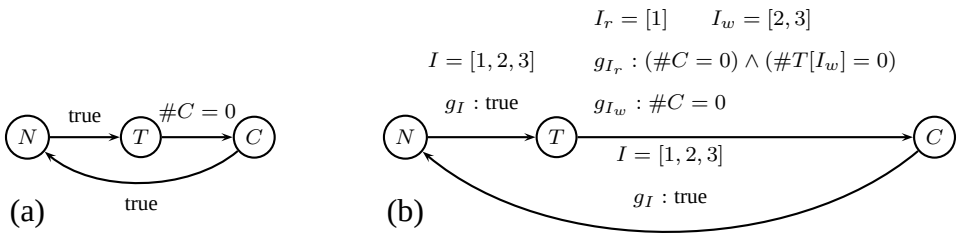


Figure 3.1: (a) Synchronization Skeleton for MUTEX. (b) GSST for three-process R&W.

process has 3 states: Non-critical ($N$), Trying ($T$), and Critical ($C$); it can enter states $N$ and $T$ freely, but can only enter the state $C$ if no other process is currently in state $C$.

When all processes have identical synchronization skeletons, their asynchronous composition can be specified using a single skeleton $P$. This skeleton can be seen as a template from which skeletons of each individual process are instantiated. Thus, Figure 3.1(a) is also a synchronization skeleton *template* for MUTEX.

A synchronization skeleton template $P$ defines a compositional transition system $B(P)$ in which a (global) transition results from a local transition of some process. For example, in the three-process MUTEX, $B(P)$ has a transition from $(N, N, T)$ to $(N, N, C)$ because the third process, $P_3$, can move from $T$ to $C$.

Note that when each transition guard in $P$ is invariant under any permutation of process indices, the transition system $B(P)$ is unchanged by any permutation of process indices; that is, it is fully symmetric [ET99]. For example, the three-process MUTEX is fully symmetric since if the guard $(\#C = 0)$ is true in a state $s$, it is also true in a state $\sigma(s)$ for any permutation $\sigma \in Sym([1, 2, 3])$. Symmetry reduction of a fully symmetric program can often yield an exponential reduction in the number of states. In practice, full symmetry of a synchronization skeleton is ensured by restricting basic elements of the guards to the ones shown in the left column of Table 3.2, where $l_i = L$ is true in a state $s$ if the $i$th process is in a state $L$, i.e., $s(i) = L$. The basic elements can be equivalently expressed using total counters, as shown in the right column of Table 3.2 [ET99].

| Basic Elements | Predicates on Total Counters |
|---|---|
| $\forall i \cdot l_i = L, \forall i \cdot l_i \neq L$ | $\#L = n, \#L = 0$ |
| $\exists i \cdot l_i = L, \exists i \cdot l_i \neq L$ | $\#L \geq 1, \#L \leq n-1$ |
| $\exists i \neq j \cdot l_i = L \wedge l_j = L$ | $\#L \geq 2$ |

Table 3.2: Basic guard elements for ensuring full symmetry.

### 3.4.2 Specifying Asymmetric Programs

In this chapter, we are interested in applying symmetry reduction to asymmetric programs composed of many similar, but not identical processes, such as R&W. In this case, since the condition for entering the critical section is different between the two groups of processes (writers have a higher priority than readers), the program cannot be specified by a single synchronization skeleton. Thus, for such asymmetric programs, we need both a more general specification formalism, and an approach to identify whether the program is fully virtually symmetric. To address the first problem, we define a *generalized synchronization skeleton template*.

**Definition 3.5.** *A* generalized synchronization skeleton template *(GSST) for an asynchronous program with $n$ processes is a tuple $P = (\mathcal{L}, \mathcal{R}, I, \tau)$, where $\mathcal{L}$ is a finite set of (local) states, $\mathcal{R} \subseteq \mathcal{L} \times \mathcal{L}$ is a (local) transition relation, $I = [1..n]$ is the index set, and $\tau : \mathcal{R} \rightarrow [I \rightarrow G]$ is a labeling function that labels each transition with a guard for each process. Here, $G : \mathcal{L}^n \rightarrow \{true, false\}$ is a set of transition guards.*

We assume that for any local transition $u \rightarrow v \in \mathcal{R}$, $u \neq v$, i.e., no self-loops are allowed in a GSST.

**Definition 3.6.** *A GSST $P = (\mathcal{L}, \mathcal{R}, I, \tau)$ defines an asynchronous transition system $B(P) = (S, R)$, where $S = \mathcal{L}^{|I|}$ is the global statespace, and $R \subseteq S \times S$ is the global transition relation defined as follows:*

(a) *for any local transition $u \rightarrow v \in \mathcal{R}$,*

$$R_{u \rightarrow v}(s,t) \triangleq \exists i \in I \cdot (s(i) = u \wedge t(i) = v \wedge (s \models \tau(u \rightarrow v)(i)) \wedge \forall j \neq i \cdot s(j) = t(j))$$

*where* $s \models \tau(u \to v)(i)$ *means that* $s$ *satisfies the guard for the* $i$*th process at the transition* $u \to v$*, and*

(b) $R \triangleq \bigcup_{r \in \mathcal{R}} R_r$.

Intuitively, $R_{u \to v}$ is the set of all global transitions resulting from some process changing its state from $u$ to $v$. We say that $s \to t \in R$ is a result of firing a local transition $u \to v$ if $s \to t$ is in $R_{u \to v}$.

For a local transition $r \in \mathcal{R}$, the labeling function $\tau : \mathcal{R} \to [I \to G]$ can be seen as: (a) a partition $\Pi_r = \{I_1, \ldots, I_d\}$ of processes into process groups, (b) an index mapping function $\pi : I \to \Pi_r$, and (c) a function $\eta : \Pi_r \to G$ assigning a guard to each process group, i.e., for any $i \in I$, $\tau(r)(i) = \eta(\pi(i))$. For example, in the GSST for the three-process R&W shown in Figure 3.1(b), the guards for the local transition $T \to C$ are described by partitioning the processes into two groups: $I_r = \{P_1\}$ (readers) and $I_w = \{P_2, P_3\}$ (writers). Readers have the guard $g_{I_r} : (\#C = 0) \wedge (\#T[I_w] = 0)$, and writers $g_{I_w} : \#C = 0$. Note that this allows us to specify not only the static process partitioning, i.e., $\forall r, r' \in \mathcal{R} \cdot \Pi_r = \Pi_{r'}$, but a dynamic one as well, that is, processes can be divided into different groups at different local transitions.

Motivated by R&W, we restrict our attention to a counter-based syntax of guards. Formally, a guard for a transition $u \to v$ is a boolean combination of *group counter constraints* on the local state $u$, i.e., $\#u[I_k] \bowtie b$, or *total counter constraints* on any local states, i.e., $(\sum_i \#L_i) \bowtie b$, where $b$ is a positive integer, and $\bowtie$ is one of $\{\leq, \geq, =\}$. For example, in Figure 3.1(b), $\#C = 0$ means no process is currently in the local state $C$, whereas $\#T[I_w] = 0$ means that no *writer* process is currently in $T$.

## 3.5 Identification of Full Virtual Symmetry

In this section, we address the problem of identifying full virtual symmetry. Notice that we cannot simply use Condition (3.1) of Theorem 3.3 since it requires building the transition relation of the program, which may not be feasible. In Section 3.5.1, we discuss conditions that

ensure that the specified program is fully virtually symmetric, and show how to decide these conditions using constraints derived directly from the program description in Section 3.5.2.

### 3.5.1 Full Virtual Symmetry in Asynchronous Transition Systems

Let $P$ be a GSST and $r$ be a transition in $P$. If all processes at $r$ belong to the same group, i.e., $|\Pi_r| = 1$, then the transition guard is defined on total counters and is independent of any permutation of process indices. Furthermore, if this is the case for all transitions in $P$, then $P$ is just a synchronization skeleton, and the underlying transition system $B(P)$ is fully symmetric (see Section 3.4.1). In general, when $P$ contains a transition $r$ with $|\Pi_r| > 1$, even restricting guards to just total counter constraints is not sufficient to ensure that $B(P)$ is fully virtually symmetric. For example, consider the GSST shown in Figure 3.1(b) and assume that we change the guard $g_{I_r}$ of the transition $T \rightarrow C$ to $(\#C = 0) \wedge (\#T = 2)$. In this case, $B(P)$ contains a global transition from $s = (N, N, T)$ to $t = (N, N, C)$ corresponding to the process $P_3$ entering state $C$. Let $\sigma \in Sym(I)$ be a permutation that switches process indices 1 and 3. Then, the only two states reachable from $\sigma(s) = (T, N, N)$ are $t_1 = (T, T, N)$ and $t_2 = (T, N, T)$. Since neither $t_1$ nor $t_2$ can be obtained by applying a permutation $\sigma' \in Sym(I)$ to $t$, transitions of the form $\sigma(s) \rightarrow \sigma'(t)$ are *not* in $B(P)$ for any permutation $\sigma'$; hence, $B(P)$ is not fully virtually symmetric.

As illustrated by the example above, it is difficult to capture the restrictions that ensure full virtual symmetry syntactically. The difficulty comes from lack of regularity in asymmetric systems. Therefore, we seek an algorithmic way to identify symmetry. As mentioned before, we cannot simply use Condition (3.1) of Theorem 3.3 since it requires building the transition relation of $B(P)$.

Notice that in our example, full virtual symmetry is broken at a global transition resulting from firing a local transition where the processes are partitioned into several groups. We generalize from this example and show that virtual symmetry of a transition system is equivalent to virtual symmetry of each transition relation subset defined by a local transition. This allows us

to decompose the problem of identifying virtual symmetry of a system along *local* transitions. Formally, we establish the following theorem.

**Theorem 3.7.** *Given a GSST $P = (\mathcal{L}, \mathcal{R}, I, \tau)$ and a permutation group $G \subseteq Sym(I)$, the transition system $B(P) = (S, R)$, where $R \triangleq \bigcup_{r \in \mathcal{R}} R_r$, is virtually symmetric with respect to $G$ if and only if each transition relation subset $R_r$ is virtually symmetric with respect to $G$, i.e.,* $R^{\exists\exists}_{\alpha_G} = R^{\forall\exists}_{\alpha_G} \Leftrightarrow \forall r \in \mathcal{R} \cdot (R_r)^{\exists\exists}_{\alpha_G} = (R_r)^{\forall\exists}_{\alpha_G}$.

Before giving the proof of this theorem, we provide the following lemma. Let $B = (S, R)$ be a transition system, and $\alpha : S \rightarrow S_\alpha$ be an abstraction function. We define a restriction of $R$ to a pair of abstract states $(a, b)$ as

$$R_{|(a,b)} \triangleq \{(s,t) \in R \mid s \in \gamma(a) \wedge t \in \gamma(b)\}$$

Note that $R = \bigcup_{a,b \in S_\alpha} R_{|(a,b)}$, and the universal and the existential abstractions of $R$ coincide if and only if they coincide for each $R_{|(a,b)}$. The following lemma generalizes this observation.

**Lemma 3.8.** *Let $B = (S, R)$ be a transition system, $\alpha : S \rightarrow S_\alpha$ be an abstraction function, and $R = \bigcup_{i \in [1...k]} R_i$ such that $\forall i \in [1...k] \cdot \exists D \subseteq S \times S \cdot R_i = \bigcup_{(s,t) \in D} R_{|(\alpha(s),\alpha(t))}$. Then,* $R^{\forall\exists}_\alpha = R^{\exists\exists}_\alpha \Leftrightarrow \forall i \in [1...k] \cdot (R_i)^{\forall\exists}_\alpha = (R_i)^{\exists\exists}_\alpha$.

**Proof:**

($\Leftarrow$) Since $R^{\forall\exists}_\alpha \subseteq R^{\exists\exists}_\alpha$ always holds, we only need to show that $R^{\forall\exists}_\alpha \supseteq R^{\exists\exists}_\alpha$. For any $a, b \in S$,

we have that

$$
\begin{aligned}
& (a,b) \in R_\alpha^{\exists\exists} \\
\Rightarrow\ & \text{(by the definition of } R_\alpha^{\exists\exists}\text{)} \\
& \exists s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R \\
\Rightarrow\ & \text{(since } R = \textstyle\bigcup_{i\in[1...k]} R_i\text{)} \\
& \exists i \in [1...k] \cdot \exists s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R_i \\
\Rightarrow\ & \text{(by the definition of } (R_i)_\alpha^{\exists\exists}\text{)} \\
& \exists i \in [1...k] \cdot (a,b) \in (R_i)_\alpha^{\exists\exists} \\
\Rightarrow\ & \text{(since } (R_i)_\alpha^{\exists\exists} = (R_i)_\alpha^{\forall\exists}\text{)} \\
& \exists i \in [1...k] \cdot (a,b) \in (R_i)_\alpha^{\forall\exists} \\
\Rightarrow\ & \text{(by the definition of } (R_i)_\alpha^{\forall\exists}\text{)} \\
& \exists i \in [1...k] \cdot \forall s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R_i \\
\Rightarrow\ & \text{(since } R_i \subseteq R\text{)} \\
& \forall s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R \\
\Rightarrow\ & \text{(by the definition of } R_\alpha^{\forall\exists}\text{)} \\
& (a,b) \in R_\alpha^{\forall\exists}
\end{aligned}
$$

(⇒) Since $(R_i)_\alpha^{\forall\exists} \subseteq (R_i)_\alpha^{\exists\exists}$ always holds, we only need to show that $(R_i)_\alpha^{\forall\exists} \supseteq (R_i)_\alpha^{\exists\exists}$ for any $i \in [1...k]$. For any $(a,b) \in (R_i)_\alpha^{\exists\exists}$, we have that

$$
\begin{aligned}
& (a,b) \in (R_i)_\alpha^{\exists\exists} \\
\Rightarrow\ & \text{(by the definition of } (R_i)_\alpha^{\exists\exists}\text{)} \\
& \exists s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R_i \\
\Rightarrow\ & \text{(by the assumption of } R_i\text{)} \\
& \exists s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R_i \wedge \exists s', t' \in S \cdot (s,t) \in R_{|(\alpha(s'),\alpha(t'))} \\
\Rightarrow\ & \text{(since } S_\alpha \text{ is a partition-based abstract statespace,} \\
& \alpha(s) = \alpha(s') = a \text{ and } \alpha(t) = \alpha(t') = b\text{)} \\
& \exists s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R_i \wedge (s,t) \in R_{|(a,b)} \\
\Rightarrow\ & \text{(by the assumption of } R_i\text{)} \\
& R_{|(a,b)} \subseteq R_i \qquad (\star)
\end{aligned}
$$

Furthermore,

$$
\begin{aligned}
& (a,b) \in (R_i)^{\exists\exists}_{\alpha} \\
\Rightarrow\ & \text{(since } R_i \subseteq R\text{)} \\
& (a,b) \in R^{\exists\exists}_{\alpha} \\
\Rightarrow\ & \text{(since } R^{\exists\exists}_{\alpha} = R^{\forall\exists}_{\alpha}\text{)} \\
& (a,b) \in R^{\forall\exists}_{\alpha} \\
\Rightarrow\ & \text{(by the definition of } R^{\forall\exists}_{\alpha}\text{)} \\
& \forall s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R \\
\Rightarrow\ & \text{(by the definition of } R_{|(a,b)}\text{)} \\
& \forall s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R_{|(a,b)} \\
\Rightarrow\ & \text{(by } (\star)\text{: } R_{|(a,b)} \subseteq R_i\text{)} \\
& \forall s \in \gamma(a) \cdot \exists t \in \gamma(b) \cdot (s,t) \in R_i \\
\Rightarrow\ & \text{(by the definition of } (R_i)^{\forall\exists}_{\alpha}\text{)} \\
& (a,b) \in (R_i)^{\forall\exists}_{\alpha}
\end{aligned}
$$

□

Based on Lemma 3.8, we now give the proof of Theorem 3.7.

**Proof:**

We prove this theorem by showing that each $R_r$ satisfies the precondition of Lemma 3.8. Recall that in the context of symmetry reduction, $\alpha_G(s)$ is equivalent to $\theta(s)$ (see Table 3.1). We only need to show that $R_r = \bigcup_{(s,t)\in R_r} R_{|(\theta(s),\theta(t))}$. That is, we need to show that if a transition $s \to t$ is a result of firing a local transition $r$, then for any permutations $\sigma, \sigma' \in G$, a transition $\sigma(s) \to \sigma'(t)$ is a result of firing $r$ as well. This follows from the following facts:

(1) two states $s_1$ and $s_2$ share an orbit only if they agree on total counters, and

(2) a global transition $s \to t$ is a result of firing a local transition $u \to v$ if and only if $\#u$ at $s$ is one more than that at $t$, $\#v$ at $s$ is one less than that at $t$, and the total counters of other local states at $s$ and $t$ are the same.

□

When $G$ is the full symmetry group $Sym(I)$, Theorem 3.7 can be simplified further since here two states share an orbit *if and only if* they agree on total counters. Note that if $R_r$ is fully

virtually symmetric, i.e., $(R_r)^{\forall\exists}_{\alpha_G} = (R_r)^{\exists\exists}_{\alpha_G}$, then $Dom(R_r)$ contains its orbit $\theta(Dom(R_r))$, which follows from the definitions of existential and universal abstractions. On the other hand, if $Dom(R_r)$ contains $\theta(Dom(R_r))$, then for any pair of states $s$ and $s'$ in the same orbit, if $s \rightarrow t$ is in $R_r$ for some state $t$, then there exists a state $t'$ such that $s' \rightarrow t'$ is in $R_r$. Furthermore, $t$ and $t'$ agree on total counters, and thus belong to the same orbit. Hence, by Theorem 3.3, $R_r$ is fully virtually symmetric. Since $\theta(Dom(R_r))$ always contains $Dom(R_r)$, we obtain the following theorem.

**Theorem 3.9.** *Given a GSST $P = (\mathcal{L}, \mathcal{R}, I, \tau)$, the transition system $B(P) = (S, R)$ is fully virtually symmetric if and only if $\forall r \in \mathcal{R} \cdot \theta(Dom(R_r)) = Dom(R_r)$.*

**Proof:**

The proof follows from the reasoning in the previous paragraph. □

Thus, we have reduced the problem of checking virtual symmetry of $R$, a global property of the entire system, to a local property of each transition subset $R_r$.

### 3.5.2 Constraint-Based Identification of Full Virtual Symmetry

In this section, we present a technique for identifying full virtual symmetry based on Theorem 3.9. Specifically, we construct Presburger formulas representing sets of states directly from the description of the GSST.

By Theorem 3.7, checking whether a transition system $B(P)$ is fully virtually symmetric is equivalent to checking whether $R_r$ is fully virtually symmetric for each local transition $r$ of the GSST $P$. Note that if all processes belong to the same group at a local transition $r$, i.e., $|\Pi_r| = 1$, then $R_r$ is fully symmetric and no check is required. Otherwise, when $|\Pi_r| > 1$, by Theorem 3.9, we need to check whether the domain of $R_r$, $Dom(R_r)$, is equal to its orbit, $\theta(Dom(R_r))$. In this section, we show that both $Dom(R_r)$ and $\theta(Dom(R_r))$ can be represented by Presburger formulas and their equivalence can be reduced to checking satisfiability of a Quantifier Free Presburger (QFP) formula.

| Constraints | Meaning |
|---|---|
| $0 \leq \#N[I_r] \quad 0 \leq \#T[I_r] \quad 0 \leq \#C[I_r]$<br>$0 \leq \#N[I_w] \quad 0 \leq \#T[I_w] \quad 0 \leq \#C[I_w]$ | each group counter<br>is a positive integer |
| $\#N[I_r] + \#T[I_r] + \#C[I_r] = 1$<br>$\#N[I_w] + \#T[I_w] + \#C[I_w] = 2$ | there is one reader process<br>and two writer processes |

Table 3.3: Invariant for the three-process R&W.

We illustrate the procedure on the $T \rightarrow C$ transition of the R&W whose GSST is shown in Figure 3.1(b). The counter-based syntax of the guards provides a compact representation of a set of states in the transition system $B(P)$ using Presburger formulas on group counters. The formula $\varphi_{T \rightarrow C}$ representing $Dom(R_{T \rightarrow C})$ is constructed based on the transition guards in the GSST as follows. According to the interleaving semantics, a state $s$ is in $Dom(R_{T \rightarrow C})$ if and only if either a reader or a writer process can move from $T$ to $C$ at $s$. In the first case, $s$ must satisfy the guard $g_{I_r}$, and since the current local state of the reader process is $T$, $s$ satisfies $g_{I_r} \wedge \#T[I_r] \geq 1$; similarly, in the second case, $s$ satisfies $g_{I_w} \wedge \#T[I_w] \geq 1$. Therefore, $Dom(R_{T \rightarrow C})$ can be represented by the formula $\varphi_{T \rightarrow C} = \varphi_{T \rightarrow C, I_r} \vee \varphi_{T \rightarrow C, I_w}$, where

$$\begin{aligned} \varphi_{T \rightarrow C, I_r} &\triangleq g_{I_r} \wedge \#T[I_r] \geq 1 \wedge inv_{T \rightarrow C} \\ \varphi_{T \rightarrow C, I_w} &\triangleq g_{I_w} \wedge \#T[I_w] \geq 1 \wedge inv_{T \rightarrow C} \end{aligned}$$

and the invariant $inv_{T \rightarrow C}$, defined as the conjunction of the constraints in the left column of Table 3.3, represents the statespace of the system. Note that $\varphi_{T \rightarrow C}$ is still defined only on group counters since $\#C$ is equivalent to $\#C[I_r] + \#C[I_w]$. In general, for a local transition $r$, the formula $\varphi_r$ representing $Dom(R_r)$ is a disjunction of formulas representing subsets of $Dom(R_r)$ with respect to each process group.

We now show how to derive a formula $\tilde{\varphi}_r$ representing $\theta(Dom(R_r))$ from $\varphi_r$. For simplicity, assume that $P$ contains only two local states, $X$ and $Y$, and the processes are partitioned into two groups. Let $Dom(R_r)$ and the invariant of the statespace be represented by $\varphi_r(X_1, X_2, Y_1, Y_2)$ and $inv_r(X_1, X_2, Y_1, Y_2)$, respectively. Then $\tilde{\varphi}_r$ representing $\theta(Dom(R_{T \rightarrow C}))$

is defined as

$$\tilde{\varphi}_r(X_1, X_2, Y_1, Y_2) \triangleq \exists X_1', X_2', Y_1', Y_2' \cdot \quad (inv_r(X_1, X_2, Y_1, Y_2) \wedge \varphi_r(X_1', X_2', Y_1', Y_2') \wedge (X_1 + X_2 = X_1' + X_2') \wedge (Y_1 + Y_2 = Y_1' + Y_2'))$$

That is, a state $s$ satisfies $\tilde{\varphi}_r$ if and only if there exists a state $s'$ satisfying $\varphi_r$ ($s' \in Dom(R_r)$) and $s$ and $s'$ agree on total counters, i.e., they are in the same orbit. Since $Dom(R_r)$ is a subset of $\theta(Dom(R_r))$, $Dom(R_r) = \theta(Dom(R_r))$ if and only if the sentence

$$\psi \triangleq \exists X_1, X_2, Y_1, Y_2 \cdot (\tilde{\varphi}_r \wedge \neg \varphi_r)$$

is unsatisfiable. Since $\psi$ contains only existential quantifiers, this is equivalent to unsatisfiability of a QFP formula obtained from $\psi$ by removing all quantifiers, which can be checked using any existing decision procedure for QFP [BB04, Pug92, WB95].

Note that while the satisfiability problem of a Presburger formula has a worst-case super-exponential complexity, satisfiability of a QFP formula is NP-complete [Pap81]. Furthermore, the number of local transitions in a GSST that need to be checked is expected to be small, since we are interested in asynchronous systems in which processes are relatively similar to one another. Indeed, if the processes differ significantly, it does not seem appropriate to consider full virtual symmetry at all. In practice, the structure of the guards often leads to further optimizations of the decision procedure. As illustrated by experiments in Section 3.7, full virtual symmetry can be identified efficiently when the guards are defined on a small number of local states.

## 3.6 Counter Abstraction for Full Virtual Symmetry

The naive way of constructing a symmetry-reduced quotient structure requires a representative function for choosing a state as the unique representative from each orbit [CJEF96, ES96]. The abstract transition relation is then defined on the set of representatives. For symbolic model checking, computation of the representative function requires building an orbit relation which, for many groups, including the full symmetry group, has a BDD representation that is

exponential in the minimum of the number of processes and the number of local states in each process [CJEF96], decreasing the effectiveness of symbolic model checking.

An alternative is to use a *counter abstraction* (or generic representatives) technique proposed by Emerson at el. [ET99, EW03], which avoids building the orbit relation. As we have seen before, under the full symmetry group, states in the same orbit agree on all total counters. Thus, each orbit can be uniquely represented by values of these counters. For example, in the three-process MUTEX, the orbit $\{(N,T,T),\ (T,N,T),\ (T,T,N)\}$ is represented by a tuple $(1,2,0)$ which corresponds to the counters of states $N$, $T$ and $C$. In this section, we extend the counter-based abstraction technique to handle a fully virtually symmetric structure specified by a GSST. The key idea is that instead of using the orbit relation, a structure isomorphic to the quotient structure is constructed on the statespace of total counters directly from the GSST.

For the rest of this section, let $P = (\mathcal{L}, \mathcal{R}, I, \tau)$ be a GSST of a fully virtually symmetric program with local states $\mathcal{L} = \{L_1, \ldots, L_m\}$ and process indices $I = [1..n]$. A *counter abstraction* $\alpha : S \to S_\alpha$ on the structure $B(P) = (S, R)$ is constructed using a set of assignments to a vector $\mathbf{x} = (x_1, \ldots, x_m)$ of $m$ counter variables ranging over $[0..n]$. Each variable $x_i$ corresponds to a total counter $\#L_i$ of a local state $L_i$. Since there are $n$ processes, the sum of the values of $\mathbf{x}$ must always equal $n$. Therefore,

$$S_\alpha \triangleq \{(c_1, \ldots, c_m) \in [0..n]^m \mid \sum_{i=1}^{m} c_i = n\}$$

The abstraction function $\alpha : S \to S_\alpha$ maps a state $s \in S$ to an abstract state $a \in S_\alpha$ if and only if for each $i \in I$, $a(i)$ equals $\#L_i(s)$. The concretization function $\gamma : S_\alpha \to 2^S$ maps an abstract state $a$ to an orbit $\theta$ where states in $\theta$ agree with $a$ on total counters. In what follows, let $R_\alpha$ denote the existential abstraction of $R$ with respect to $\alpha$.

**Theorem 3.10.** *Given a GSST $P$ and a counter abstraction $\alpha$, the abstract structure $B(P)_\alpha = (S_\alpha\ ,\ R_\alpha)$ is isomorphic to the quotient structure $B(P)^{Sym(I)} = (S^{Sym(I)},\ R^{Sym(I)})$ via a bijection $h : S_\alpha \to S^{Sym(I)}$, where $\forall s \in S \cdot h(\alpha(s)) \triangleq \theta(s)$.*

**Proof:**

The proof follows from the fact that the orbits and transitions in a fully symmetric program can be characterized by total counters. □

The above definition of $B(P)_\alpha$ guarantees that the abstract transition relation $R_\alpha$ can be constructed directly from $P$ for a fully virtually symmetric program. Since existential abstraction distributes over union, and $R = \bigcup_{r\in\mathcal{R}} R_r$ by Definition 3.6, it follows that $R_\alpha = \bigcup_{r\in\mathcal{R}}(R_r)_\alpha$. Therefore, we only need to show how to construct $(R_r)_\alpha$ for a local transition $r$.

We start by illustrating the construction in the case of an unguarded local transition $r$. If $r$ is of the form $L_i \rightarrow L_j$, then $r$ can be fired from a global state $s$ if and only if $s$ contains a process whose current state is $L_i$; in other words, $Dom(R_r)$ is $\#L_i \geq 1$. Furthermore, if $s \rightarrow t$ is in $R_r$, then the counters $\#L_i$ and $\#L_j$ at $t$ are one less and one more than those at $s$, respectively. From the definition of existential abstraction, for any abstract states $a$ and $b$, a transition $a \rightarrow b$ is in $(R_r)_\alpha$ if and only if $s \rightarrow t \in R_r$ for some $s \in \gamma(a)$ and $t \in \gamma(b)$. Therefore,

$$(R_r)_\alpha \;\equiv\; x_i \geq 1 \wedge (x_i := x_i - 1;\; x_j := x_j + 1)$$

which is a formula over counter variables. Generalizing from this example, we obtain that for every local transition $r$ of the form $L_i \rightarrow L_j$,

$$(R_r)_\alpha \;\equiv\; g_r \wedge (x_i := x_i - 1;\; x_j := x_j + 1)$$

where $g_r$ is a formula defined over counter variables **x** representing the "existential" abstraction of $Dom(R_r)$. Specifically,

$$a \models g_r \;\Leftrightarrow\; \exists s \in \gamma(a) \cdot s \in Dom(R_r)$$

Since $B(P)_\alpha$ is isomorphic to the quotient structure, the above construction allows us to combine symmetry reduction and symbolic model checking without building the orbit relation. The only remaining problem is the construction of the formula $g_r$ for an arbitrary local transition $r$. In the rest of this section, we show how to do this for cases where $r$ is guarded by (a) a single guard on total counters, (b) multiple guards on total counters, and (c) multiple guards on group counters of the source state of $r$ and arbitrary total counters.

*Case (a).* Let $r$ be a local transition $L_i \rightarrow L_j$. Suppose $r$ is guarded by a single guard $g$, i.e., $|\Pi_r| = 1$. Then $Dom(R_r)$ can be represented by $\psi_r = (\#L_i \geq 1 \wedge g)$, i.e., $s \in Dom(R_r)$ if there is at least one process at $s$ in local state $L_i$ and $s$ satisfies $g$. Let $sub(\psi_r)$ denote a formula obtained from $\psi_r$ by replacing each occurrence of a total counter with its corresponding counter variable. For example, $sub(\#L_i \geq 0) = (x_i \geq 0)$ and $sub(\#L_i \geq 1 \wedge \#L_j \leq 3) = (x_i \geq 1 \wedge x_j \leq 3)$. Since $g$ contains only total counter constraints, we define $g_r \triangleq sub(\#L_i \geq 1 \wedge g)$. Note that this procedure constructs a counter abstraction for a fully symmetric synchronization skeleton, and is effectively equivalent to the *generic representatives* approach of Emerson and Trefler [ET99].

*Case (b).* Suppose that $r$ is guarded by multiple guards, i.e., $|\Pi_r| = d > 1$, but each guard is expressed using only total counters. In this case, $Dom(R_r)$ is represented by $\psi_r = \bigvee_{k\in[1..d]}(\#L_i[I_k] \geq 1 \wedge g_{I_k})$, where $g_{I_k}$ is the guard for the process group $I_k$. Since $\psi_r$ depends on group counters, we cannot simply define $g_r$ to be $sub(\psi_r)$. However, $R_r$ is fully virtually symmetric, so $Dom(R_r) = \theta(Dom(R_r))$ by Theorem 3.9, and $\theta(Dom(R_r))$ is representable by $\tilde{\psi}_r = (\#L_i \geq 1 \wedge (\bigvee_{k\in[1..d]} g_{I_k}))$. Thus, we define $g_r \triangleq sub(\tilde{\psi}_r)$.

*Case (c).* Finally, we look at the case where the guards of $r$ depend on group counters. In this case, $\tilde{\psi}_r$ defined above still contains group counters. However, this problem can be solved for cases where group counters in guards for a transition $r : L_i \rightarrow L_j$ are defined only over $L_i$.

First, let $Q \subseteq S$ be some non-empty set of states given by some formula $\psi$ defined only on group counters of $L_i$. That is,

$$\psi \;=\; \bigwedge_{k\in[1..d]}(min_k \leq \#L_i[I_k] \leq max_k)$$

where $\{min_k\}$ and $\{max_k\}$ are positive integers. Then, the orbit $\theta(Q)$ under $Sym(I)$ is given by the formula

$$\tilde{\psi} = (min \leq \#L_i \leq max)$$

where

$$min \;\triangleq\; \sum_{k\in[1..d]} min_k \qquad max \;\triangleq\; \sum_{k\in[1..d]} max_k$$

For example, suppose there are only two local states, $L_1$ and $L_2$, $d = 2$, and $Q$ is given by

$$\psi = (1 \leq \#L_1[I_1] \leq 4) \wedge (1 \leq \#L_1[I_2] \leq 4)$$

Then $\theta(Q)$ is $\tilde{\psi} = (2 \leq \#L_1 \leq 8)$ since for any state $s$ in $S$ satisfying $\tilde{\psi}$ there exists a state $s'$ in $S$ satisfying $\psi$ such that $s$ and $s'$ agree on total counters of $L_1$ and $L_2$, i.e., they are in the same orbit. Furthermore, if $Q$ is encoded by a conjunction $\psi^t \wedge \psi^g$, where $\psi^t$ and $\psi^g$ are defined only on total and group counters, respectively, then the orbit of $Q$ is given by $\psi^t \wedge \tilde{\psi}^g$.

Second, suppose a guard $g_{I_k}$ contains group counter constraints. Let $Dom(R_r)_{I_k}$ denote the subset of $Dom(R_r)$ containing states in which the local transition $r$ of some process in the group $I_k$ can be fired. If the formula $\psi_{r,I_k}$ representing $Dom(R_r)_{I_k}$ can be decomposed as $\psi_{r,I_k} = \psi^t_{r,I_k} \wedge \psi^g_{r,I_k}$, then a total counter formula representing $\theta(Dom(R_r)_{I_k})$ is computed as described above. Otherwise, $\psi_{r,I_k}$ can be converted to a DNF, and formulas corresponding to the orbit of each clause are computed as above. Since $Dom(R_r) = \bigcup_{k\in[1..d]} Dom(R_r)_{I_k}$, and $\theta$ distributes over union, i.e., $\theta(Q_1 \cup Q_2) = \theta(Q_1) \cup \theta(Q_2)$, we can define $\tilde{\psi}_r$ representing $\theta(Dom(R_r))$ as a disjunction of the clause formulas. Finally, $\tilde{\psi}_r$ depends only on total counters; thus, we define $g_r$ to be $sub(\tilde{\psi}_r)$.

For example, the domain of the transition $T \rightarrow C$ of the R&W shown in Figure 3.1(b), is the union of the domain for the readers and that of the writers. For readers,

$$\begin{aligned} Dom(R_{T\rightarrow C})_{I_r} &\equiv \#T[I_r] >= 1 \wedge \#T[I_w] = 0 \wedge \#C = 0 \\ &\equiv \#T[I_r] = 1 \wedge \#T[I_w] = 0 \wedge \#C = 0 \end{aligned}$$

since there is only one reader. Using only total counters, the orbit $\theta(Dom(R_{T\rightarrow C})_{I_r})$ is represented by $\tilde{\psi}_r = (\#T = 1 \wedge \#C = 0)$. Similarly, for the writers,

$$Dom(R_{T\rightarrow C})_{I_w} \equiv \#T[I_w] \geq 1 \wedge \#C = 0$$

and the orbit $\theta(Dom(R_{T\rightarrow C})_{I_w})$ is represented by $\tilde{\psi}_w = (\#T \geq 1 \wedge \#C = 0)$. Finally, $g_{T\rightarrow C}$ is defined by $sub(\tilde{\psi}_r \vee \tilde{\psi}_w) = (\#T \geq 1 \wedge \#C = 0)$.

## 3.7 Experiments

In this section, we report on experiments of identifying full virtual symmetry and performing counter abstraction-based symbolic model-checking. We used the Omega library [Pug92] as the QFP solver to check for full virtual symmetry as described in Section 3.5, and used NuSMV [CCGR99] as the model-checker for both the direct and the counter abstraction-based analysis : we constructed NuSMV programs to represent the original and the counter abstracted programs and then run NuSMV to check properties.

The examples we used for experiments are two typical asymmetric programs in practice where processes share resources based on different priorities or permissions. The first example is a generalized R&W (GR&W) [ET99], where we assumed that each process has $m$ local states $\{L_1, \ldots, L_m\}$, where $L_m$ represents the critical section. Each process can be in one of the local states, and must go through $L_1$ to $L_{m-1}$ before accessing the critical section. The process can return from $L_m$ to $L_1$ freely. The processes are partitioned into $d$ groups, each of size $q$, based on their priorities: a process cannot access the critical section if another process with higher priority is waiting for it. For this example, we verified the standard safety property $AG(\#L_m \leq 1)$, which ensures that no two processes can access the critical section at the same time. The second example is an asymmetric sharing of resources (ASR) [EHT00]. In this example, there are a set of $r$ resources shared by $n$ processes. There is one non-critical section ($N$) for all the resources, and for each resource $i$, there is a trying ($T_i$) and a critical ($C_i$) section associated with it. The processes have different permissions to access the resources, and the number of processes that can be waiting for each resource and using it is bounded. This example is motivated by the drinking philosophers problem [CM84], where a set of bottles are shared by a set of philosophers, and a philosopher can only drink from the bottles that are assigned to him. For this example, we check the maximum sharing of the resources; that is, we checked that whether it is possible for all the resources to be used at the same time, i.e., $EF\big(\bigwedge_{i\in[1..r]}(\#C_i > 0)\big)$.

To experiment the scalability of our approach, we created instances of the examples with

varying values of the numbers of processes or shared resources. The programs of these instances are automatically generated using template codes. The experiments were performed on a Sun Fire V440 server (4@1.3GHz, USPARC3i, 16384M). The results of the direct (*NuSMV*) and the counter abstraction-based (*Symmetry Reduction with Counter Abstraction*) analysis are summarized in Table 3.4 (Page 65), where dashes indicate that verification did not complete due to either memory or time limits. Where appropriate, we separate the checking time into identifying symmetry (*CkSym*) and checking the resulting reduced model (*ModelCk*). For ASR, we also reported the results of computing the set of reachable states first, before evaluating the property (the `-f` option of NuSMV).

Since counter abstraction is based on full symmetry groups, the reduction of the statespace can be exponential. It has been shown that counter abstraction enables significant reduction of memory and CPU usage on model checking fully symmetric programs [EW03]. From our experiment results, we obtained the same observation of applying counter abstraction to fully virtually symmetric programs. Moreover, let $p$ be the number of processes in a program, and $l$ be the number of local states of each process. As shown in [ET99], counter abstraction reduces a problem of worst case size $l^p$ that is exponential in $p$, to one of worst case size $p^l$ that is polynomial for a fixed number of local states. Therefore, it is assumed that counter abstraction is most useful in the case where $l$ is a fixed constant and $p$ is a parameter. From our experiment results, we see that memory usage grows slowly with the number of processes, which shows that the method is applicable for programs comprised of a large number of processes.

In these examples, the time it took to identify full virtual symmetry was relatively small. One reason is that the guards depend only on a small number of process groups and local states. Otherwise, more specialized solvers may be useful. For example, identifying symmetry of GR&W with $d$ = 100 and $q$ = 20 took us many hours with the Omega library and only 17 seconds with the pseudo-Boolean solver (PBS) [ARMS02].

## 3.8 Related Work

Concurrent programs are often composed of identical processes. The global transition relations of such programs exhibit a great deal of symmetry, which can be used for statespace reduction. The ideas of exploiting symmetry reduction in model checking were introduced in [CJEF96, ES96, ID96]. Symmetry reduction techniques have been implemented in model checking tools, such as SMC [SGE00], Murphi [ID96], SymmSpin [BDH02], and Verisoft [God97].

To extend symmetry reduction to asymmetric programs, Emerson and Trefler first proposed "looser" notions of *near* symmetry where asymmetric behaviors initiate only from highly symmetric states, and *rough* symmetry [ET99] where asymmetry arises from static process priorities. They generalized these notations by *virtual* symmetry in [EHT00], which applies to a broader class of asymmetric programs. In this chapter, we viewed symmetry reduction as strong exact-approximation, and give an alternative characterization of virtual symmetry directly from the perspective of abstraction. Dams et al. [DGG97] have used existential and universal abstractions to define over- and under-approximations of program behaviors, respectively. We showed that symmetry reduction can be seen as an abstraction over symmetric equivalence classes where existential and universal abstractions coincide.

Identification of symmetry is a necessary step for applying symmetry reduction. In practice, the problem of identifying genuine symmetry has been avoided by imposing restrictions on the specification languages [ID96, SGE00, ET99, EW03]. For example, the input program in SMC [SGE00] is divided into modules and each module specifies a set of processes that are identical up to renaming. In particular, the counter-based synchronization skeletons used in [ET99, EW03] guarantees that a program is fully symmetric. However, as we showed in this chapter, lack of regularity in asymmetric programs makes it difficult to capture the restrictions that ensure full virtual symmetry syntactically. Emerson et al. proposed a combinatorial condition for checking virtual symmetry based on counting the missing transitions [EHT00], which seems to require the construction of the transition system of a program. Based on our characterization of symmetry reduction, we avoided this problem by checking satisfiability of

a QFP formula built from a program description.

Symbolic symmetry reduction was studied in [CJEF96], where the authors showed that construction of the orbit relations is a bottleneck, because the BDDs of orbit relations for many symmetry groups are exponential. To address this problem, they proposed to perform symmetry reduction in a coarse way by choosing multiple elements from each orbit. Emerson et al. [ET99] showed that using a *generic representatives* technique (also called counter abstraction [PXZ02]), symmetry-reduced structures can be directly constructed from a program description translated from the original one. Therefore, the problem of computing orbit relations is avoided. This approach was later applied to fully symmetric programs on processes communicating via shared variables [EW03], and the experiments show that it is superior to that of the unique and multiple representatives. Our work extended the applicability of this technique to fully virtually symmetric programs.

There are other approaches that combine symmetry reduction and symbolic model checking with different flavors. The dynamic symmetry reduction proposed by Emerson [EW05] avoids building the symbolic representation of the symmetry-reduced structure. Instead, they provided a symbolic abstract transfer function that computes transition images with respect to the underlying symmetry groups. This function is embedded into the model checking process, used to compute fixpoints for temporal properties. Barner and Grumberg proposed on-the-fly symbolic model checking with symmetry reduction [BG02] that incrementally explores the reachable states. Symmetry is used there to avoid including states that are symmetric to the ones explored before. This approach in general discovers a part of the reachable states, and therefore, is mainly for refutation of universal properties.

## 3.9 Conclusion

In this chapter, we studied a strong exact-approximation technique – symmetry reduction in the context of full virtual symmetry. We formalized its connection with abstraction, and provided

an alternative characterization of symmetry reduction. Based on this, we developed techniques to address challenges of applying symmetry reduction in practice. We first developed an efficient approach to identify full virtual symmetry based on satisfiability of formulas built from the program description. We then extended counter abstraction for symbolic model checking of fully virtually symmetric programs, which avoids the problem of computing orbit relations. We reported on experiments that illustrate the feasibility of our approach.

We believe that our techniques have a potential to increase the scope of symmetry reduction, and would like to investigate this in the future. Note that our work assumed that group counters occurring in a guard are defined only on the source state (see Section 3.6). While this did not pose a problem for examples we have tried, we do not know what the consequences of this restriction are, and would like to explore this further. Since counter abstraction abstracts away process identities, it does not allow us to analyze the behavior properties of an individual process, e.g., the property stating that it is always true that if a process tries to access a shared resource, it will be granted in future. In [PXZ02], counter abstraction is extended to handle properties of an individual process by abstracting all the processes in a program except for a generic one. Since the generic process is left intact during the abstraction, it can be used for checking properties of an individual process. We would like to investigate how to extend this technique to handle full virtual symmetry in the future.

| | Parameter | NuSMV | | | Symmetry Reduction with Counter Abstraction | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | BDD Nodes Allocated | Mem. (MB) | Time (sec.) | BDD Nodes Allocated | Mem. (MB) | Time (sec.) CkSym | ModelCk | Total |
| **Generalized R&W** | d (q=20, m=10) | | | | | | | | |
| | 5 | 51,778,281 | 931 | 241 | 25,146 | 7 | 0.07 | 0.27 | 0.34 |
| | 10 | - | - | - | 31,772 | 8 | 0.83 | 0.53 | 1.36 |
| | 15 | - | - | - | 38,927 | 8 | 5.09 | 1.26 | 6.35 |
| | m (d=5, q=20) | | | | | | | | |
| | 10 | 51,778,281 | 931 | 241 | 25,146 | 7 | 0.07 | 0.27 | 0.34 |
| | 20 | 121,392,365 | 2,041 | 837 | 130,891 | 10 | 0.07 | 0.59 | 0.66 |
| | 30 | - | - | - | 379,336 | 14 | 0.07 | 1.35 | 1.42 |
| | q (d=10, m=20) | | | | | | | | |
| | 10 | 121,408,515 | 2,040 | 742 | 131,010 | 10 | 0.80 | 0.58 | 1.38 |
| | 30 | - | - | - | 187,469 | 12 | 0.81 | 24.14 | 24.95 |
| | 50 | - | - | - | 195,653 | 13 | 0.75 | 67.21 | 67.96 |
| **Asymmetric Sharing of Resources** | n (r=2) | | | | | | | | |
| | 40 | 8,151,508 | 151 | 30.74 | 427,075 | 14 | 0.10 | 4.35 | 4.45 |
| | 80 | 57,163,279 | 1,001 | 2928.81 | 289,566 | 18 | 0.10 | 36.83 | 36.93 |
| | n (r=3) | | | | | | | | |
| | 40 | 44,877,253 | 782 | 43108.92 | 390,715 | 17 | 0.15 | 9.68 | 9.83 |
| | 80 | - | - | - | 420,347 | 20 | 0.15 | 80.61 | 80.76 |
| | n (r=5) | | | | | | | | |
| | 40 | - | - | - | 67,060 | 19 | 0.30 | 28.31 | 28.61 |
| | 80 | - | - | - | 342,060 | 39 | 0.30 | 279.89 | 280.19 |
| | n (r=10) | | | | | | | | |
| | 40 | - | - | - | 484,260 | 48 | 3.00 | 251.87 | 254.87 |
| | 80 | - | - | - | 671,318 | 153 | 3.00 | 1409.53 | 1412.53 |
| **Asym. Sharing of Resources (reachable states)** | n (r=2) | | | | | | | | |
| | 40 | 8,543,329 | 159 | 34.47 | 10,165 | 7 | 0.10 | 0.15 | 0.25 |
| | 80 | 57,375,594 | 1,006 | 528.25 | 18,611 | 7.2 | 0.10 | 0.25 | 0.35 |
| | n (r=3) | | | | | | | | |
| | 40 | 42,633,638 | 805 | 1614.32 | 21,647 | 7.3 | 0.15 | 0.21 | 0.36 |
| | 80 | - | - | - | 38,913 | 7.7 | 0.15 | 0.39 | 0.54 |
| | n (r=5) | | | | | | | | |
| | 40 | - | - | - | 71,925 | 8.2 | 0.30 | 0.49 | 0.79 |
| | 80 | - | - | - | 133,034 | 9.5 | 0.30 | 1.03 | 1.33 |
| | n (r=10) | | | | | | | | |
| | 40 | - | - | - | 394,722 | 14 | 3.00 | 2.55 | 5.55 |
| | 80 | - | - | - | 404,477 | 18 | 3.00 | 6.13 | 9.13 |

Table 3.4: Experimental results for generalized R&W and asymmetric sharing of resources.

# Chapter 4

# Reachability and Non-Termination Analysis of Recursive Programs

In this chapter, we propose an approach for analyzing reachability and non-termination properties of recursive programs. We first define a mixed program semantics that reduces recursive program analysis to non-recursive one by removing call stacks. Based on this semantics, we develop a simple approach for reachability and non-termination analysis of recursive programs, which can be combined with exact-approximating predicate abstraction that has been implemented in our software model checker YASM [GWC06b].

## 4.1 Introduction

Software model checking is one of the prominent analysis techniques that enables checking of program code. It combines automated construction of a finite abstract model with automated analysis by model checking and iterative abstraction refinement. Traditional software model checking, e.g., SLAM [BPR03], relies on an over-approximating abstraction of the program and thus is biased towards establishing correctness of safety properties. To exploit the bug detection ability of model checkers and to extend the scope of abstract model

(a)

```
1. x=read(); y=read();
2. if(x>0){
3.   while(x>0) {
4.     x=x+1;
5.     if(x<=0) ERROR;}
6. } else
7.     while(y>0) y=y-1;
8. END;
```

(b)

```
1. p = *;
2. if(p){
3.   while(p) {
4.   p = p?true:*;
5.     if(!p) ERROR;}
6. } else
7.     while(*) p = p;
8. END;
```

Figure 4.1: (a) A program $\texttt{EX}_0$, (b) its over-approximation $\mathcal{O}(\texttt{EX}_0)$ using predicate $p : x > 0$.

checkers to richer properties, recent research has proposed abstract analysis based on exact-approximation [BG99, GHJ01, SG03, SG04, BKY05, GWC06a, GC06]. It combines both over- and under-approximations, and therefore can be used to prove and disprove properties with the same effectiveness. Although such an abstraction has been shown to be effective in practice [GC06, GWC06b], until now this line of research has focused exclusively on analyzing non-recursive programs. In this chapter, we propose a novel approach to extend exact-approximating analyses to *recursive* programs. We illustrate our approach on non-termination and reachability analysis of several C programs, including the benchmarks from BEBOP [BR00], VERA [ACEM05], and MOPED [ES01, BEM97], the `Ack` program from [CPR06a] and a buggy version of `Quicksort` from [ES01]. To our knowledge, this is the first time that *non-termination* of such C programs was established completely automatically.

As a motivation, we review an over-approximation-based approach for model checking of non-recursive programs and its limitations. Assume we want to check whether the `ERROR` label is reachable in the C program $\texttt{EX}_0$ shown in Figure 4.1(a). This safety property is expressed in CTL as $\varphi : AG\ (pc \neq \texttt{ERROR})$. An over-approximating abstraction $\mathcal{O}(\texttt{EX}_0)$ of $\texttt{EX}_0$ using the predicate $p : x > 0$ is shown in Figure 4.1(b), where '`*`' is interpreted as a non-deterministic

choice, `!p` means that $p$ is false, and the assignment `p = p?true:*` means that if the current value of $p$ is true, then $p$ is true after execution of this statement; otherwise, the value of $p$ is assigned non-deterministically. $\mathcal{O}(\texttt{EX}_0)$ is a finite *boolean* model which over-approximates the original program: it contains all feasible and some infeasible (or spurious) executions. For example, $\mathcal{O}(\texttt{EX}_0)$ has an execution which gets stuck in the `while(*)` loop on line 7, but $\texttt{EX}_0$ does not have the corresponding execution. Thus, if a universal temporal property, i.e., in the one expressed in ACTL, holds in $\mathcal{O}(\texttt{EX}_0)$, it also holds in $\texttt{EX}_0$. For example, our property $\varphi$ is satisfied by $\mathcal{O}(\texttt{EX}_0)$, which means `ERROR` is unreachable in $\texttt{EX}_0$. However, when a property is falsified by $\mathcal{O}(\texttt{EX}_0)$, the result cannot be trusted since it may be caused by a spurious behavior. For example, consider checking whether $\texttt{EX}_0$ always terminates, i.e., whether it satisfies $\psi$ : $AF\ (pc = \texttt{END})$. $\psi$ is falsified on our abstraction, but this result cannot be trusted due to the infeasible non-terminating execution around the `while(*)` loop on line 7.

The falsification (or refutation) ability of predicate abstraction can be dramatically improved by using an *under*-approximating abstraction, where each abstract behavior is simulated by some concrete one. In this case, if a bug (or an execution) is present in the abstract model, it *must* exist in the concrete program. For example, the predicate $p$ *must* always be *true* in the `while(p)` loop at line 3 (assuming `int` is interpreted as mathematical integers). Thus, an under-approximation based on predicate $p$ is sufficient to establish that $\texttt{EX}_0$ is non-terminating.

In our previous work, we have developed a software model checker YASM [GWC06b] for checking non-recursive programs based on exact-approximating predicate abstraction [GC06] that combines both over- and under-approximations. Our goal in this chapter is to extend its analysis ability to recursive programs. One way to do this is extending pushdown systems to support exact-approximation and developing analysis algorithms for this new modeling formalism. In this chapter, we propose an alternative solution to this problem. The key to our approach is to separate the analysis of recursion from abstraction of the data domain based on a new semantics of recursive programs. By doing this, our approach does not require the development of new specialized types of pushdown systems, nor new specialized analysis algo-

rithms, which allows us to reuse the existing abstract analysis in YASM for analyzing recursive programs. Specifically, this chapter makes the following contributions:

1. We define a stack-free semantics of recursive programs that combines operational and natural semantics (commonly referred to as *function summaries*). We call this semantics *mixed*, which effectively removes call stacks while preserving *stack-independent* properties such as reachability and non-termination properties.

2. Based on mixed semantics, we describe algorithms for checking reachability and non-termination of recursive programs over finite data domains. We also show how to compute only the needed part of natural semantics, resulting in on-the-fly algorithms.

3. We show how to construct abstract versions of our reachability and non-termination analysis algorithms based on exact-approximating predicate abstraction. Its basis consists of defining the abstract domains and sound abstract versions of operations needed for deriving the on-the-fly algorithms.

4. We implement these algorithms in YASM, and report on their performance on a collection of reachability and non-termination benchmarks from BEBOP, VERA, and MOPED, for cases where the property needs to be validated or refuted.

The rest of this chapter is organized as follows. We present preliminaries and fix our notation in Section 4.2. We present a simple programming language PL and its natural, and operational semantics in Section 4.3. We introduce mixed semantics in Section 4.4, and derive on-the-fly algorithms for reachability and non-termination analysis of finite recursive programs in Section 4.5. In Section 4.6, we describe abstract versions of the algorithms for handling programs with infinite data domain. Experiments are reported in Section 4.7. We discuss related work in Section 4.8 and conclude this chapter in Section 4.9.

## 4.2 Preliminaries

A *valuation* $\sigma$ on a set of typed variables $V$ is a function that maps each variable $x$ in $V$ to a value $\sigma(x)$ in its domain. We assume that valuations extend to expressions in the obvious way. The domain of $\sigma$ is called a *valuation type* and is denoted by $\tau(\sigma)$. For example, if $\sigma = \{x \mapsto 5, y \mapsto 10\}$ then $\tau(\sigma) = \{x, y\}$. The projection of $\sigma$ on a subset $U \subseteq V$ is denoted by $\sigma|_U$.

The set of all valuations over $V$ is denoted by $\Sigma_V \triangleq \{\sigma \mid \tau(\sigma) = V\}$. Note that $\Sigma_\emptyset$ is well-defined and consists of the unique empty valuation. A relation $r$ on two sets of variables $U$ and $V$ is a subset of $\Sigma_U \times \Sigma_V$. The *relational type* of $r$ is $U \to V$, denoted by $\tau(r)$. For example, the type of an assignment $x' = y$, where $x'$ refers to the value of $x$ at the next state, is from $y$ to $x$, that is, $\tau(x' = y) = \{y\} \to \{x\}$. In this chapter, we use several simple relations: $true$ is the **true** relation, $id$ is the identity relation (e.g., $id(x) \triangleq x' = x$), $decl$ is a relation for variable declaration, and *kill* — for variable elimination. Formally, they are defined as follows, with the format *name* '$\triangleq$' *expression* ':' *type*:

$$
\begin{array}{lcl}
\mathit{true}(U \to V) & \triangleq & \Sigma_U \times \Sigma_V : U \to V \\
\mathit{decl}(V) & \triangleq & true(\emptyset \to V) : \emptyset \to V \\
\mathit{kill}(V) & \triangleq & true(V \to \emptyset) : V \to \emptyset \\
\mathit{id}(V) & \triangleq & \{(\sigma, \sigma') \in \Sigma_V \times \Sigma_V \mid \sigma = \sigma'\} : V \to V
\end{array}
$$

Operations on relations are defined in Table 4.1, where $\vee$, $\circ$ and $\times$ are *asynchronous*, *sequential* and *parallel* composition, respectively, *assume* is a restriction of the identity relation to a set $Q$ of valuations, $[\cdot]$ is *variable introduction*, and $(\cdot \to \cdot)$ is *scope extension*. Note that $\times$ combines the outputs of two relations, and $[\cdot]$ extends the source of a relation with new variables. Together these operators allow constructing complex relations from simple ones. For example, $[\{x, y\}](x' = y) \times [\{x, y\}](y' = x)$ is the relation $(x' = y) \wedge (y' = x)$ with the type $\{x, y\} \to \{x, y\}$. Directly composing $x' = y$ and $y' = x$ without variable introduction, i.e., $(x' = y) \times (y' = x)$, is invalid because $\tau(x' = y) = \{y\} \to \{x\}$ and

| Operation | Assumption | Definition | Type |
|---|---|---|---|
| $r_1 \vee r_2$ | $\tau(r_1) = \tau(r_2)$ | $\lambda a, a' \cdot r_1(a, a') \vee r_2(a, a')$ | $\tau(r_1)$ |
| $r_1 \circ r_2$ | $\tau(r_1) = U \to V$ $\wedge \quad \tau(r_2) = V \to W$ | $\lambda a, a' \cdot \vee_{a''} (r_1(a, a'') \wedge r_2(a'', a'))$ | $U \to W$ |
| $r_1 \times r_2$ | $\tau(r_1) = U \to V_1$ $\wedge \quad \tau(r_2) = U \to V_2$ $\wedge \quad V_1 \cap V_2 = \emptyset$ | $\lambda a, a' \cdot r_1(a, a'\|_{V_1}) \wedge r_2(a, a'\|_{V_2})$ | $U \to (V_1 \cup V_2)$ |
| *assume*$(Q)$ | | $\lambda a, a' \cdot Q(a) \wedge$ *id*$(\tau(Q))(a, a')$ | $\tau(Q) \to \tau(Q)$ |
| $[W]r$ | $\tau(r) = U \to V$ | $\lambda a, a' \cdot r(a\|_U, a')$ | $(U \cup W) \to V$ |
| $(W \to Z)r$ | $\tau(r) = U \to V \wedge U \subseteq W \wedge (Z \setminus V) \subseteq W$ | $([W]r) \times ([W](id(Z \setminus V)))$ | $W \to Z$ |

Table 4.1: Relational operations.

$\tau(y' = x) = \{x\} \to \{y\}$ have different source types. Scope extension extends a relation by combining it with the identity on new variables. For example, $(\{x, y\} \to \{x, y\})(x' = x + 1)$ is $(x' = x + 1) \wedge (y' = y)$. The assumptions for scope extension ensure that any new variables introduced in the destination of $r$ must also be available in the source. For example, the extension $(\{x, y\} \to \{x, z\})(x' = x + 1)$ is not allowed since $z$ is not available in the source of the relation.

## 4.3 Programming Language and Semantics

We use a simple imperative programming language PL which allows non-determinism and recursive function calls. We assume that

(a) functions have a set of call-by-value formal parameters and a set of return variables;

(b) each variable has a unique name and explicit scope;

(c) there are no global variables (they can be simulated by local variables); and

(d) a type expression is associated with each statement and explicitly defines the pre- and post-variables of the statement.

**Syntax.** Let *var* denote variables, *func* function identifiers, *e* expressions, and *T* valuation types. The syntax of PL is defined as follows:

$$
\begin{array}{lcl}
Atomic & ::= & \textbf{skip} \mid var^+ := e^+ \mid \textbf{assume}(e) \mid \textbf{var}\ var^+ \mid \textbf{kill}\ var^+ \mid (T \to T)Atomic \\
Stmt & ::= & Atomic \mid Stmt\,;Stmt \mid Stmt \,[\!]\, Stmt \mid \textbf{if}(e)\ \textbf{then}\ Stmt\ \textbf{else}\ Stmt \\
 & & \mid \textbf{while}(e)\ Stmt \mid var^+ := func(var^+) \mid (T \to T)Stmt \\
Fdef & ::= & func(var^+) : var^+\ Stmt \\
Prog & ::= & Fdef^+
\end{array}
$$

We use bold lower case letters to represent vectors, e.g., a statement $\mathbf{x}{:=}\mathbf{e}$ means an assignment $x_1, \cdots, x_n := e_1, \cdots, e_n$. For a function $f$ with declaration $f(p_1, \cdots, p_n) : r_1, \cdots, r_k$, we write $\mathbf{p}_f$ and $\mathbf{r}_f$ to denote the formal parameters and the return variables of $f$, respectively. *var*$(e)$ denotes the variables of $e$, and we assume that each program has a "main" function $f_1$, not called by other functions.

**Base Semantics.** Let $\Sigma$ denote the set of all valuations in a PL program. With each *atomic* statement $S$, we associate *base semantics* that interprets the statement as a relation $[\![S]\!] \subseteq \Sigma \times \Sigma$ on valuations of program variables:

$$
\begin{array}{lcl}
[\![\textbf{skip}]\!] & \triangleq & id(\emptyset) \\
[\![\textbf{var}\ \mathbf{x}]\!] & \triangleq & decl[\mathbf{x}] \\
[\![\textbf{kill}\ \mathbf{x}]\!] & \triangleq & kill[\mathbf{x}] \\
[\![(U \to V)(S)]\!] & \triangleq & (U \to V)[\![S]\!] \\
[\![\mathbf{x} := \mathbf{e}]\!] & \triangleq & \{(\sigma, \sigma') \mid \tau(\sigma) = \mathit{var}(\mathbf{e}) \wedge \sigma' = [\mathbf{x}_i \mapsto \sigma(\mathbf{e}_i)]\} \\
[\![\textbf{assume}(e)]\!] & \triangleq & \{(\sigma, \sigma') \mid (\sigma, \sigma') \in id(\mathit{var}(e)) \wedge \sigma \models e\}
\end{array}
$$

Note that for the type cast statement $(U \to V)S$, we only consider those cases where the assumptions for the scope extension are satisfied.

(a)

```
1:  f1() {
2:  en1: skip;
3:  var x,y;
4:  x,y := 3,0;
5:  x := f2(x);
6:  skip;
7:  while (x==2 && y<=0) {
8:    y := f2(y);
9:  }
10: kill x,y;
11: ex1: }


12: f2(z):z {
13: en2: skip;
14: while (z < 0) {
15:   z := z+1;
16: }
17: z := z-1;
18: ex2: }
```

(b)

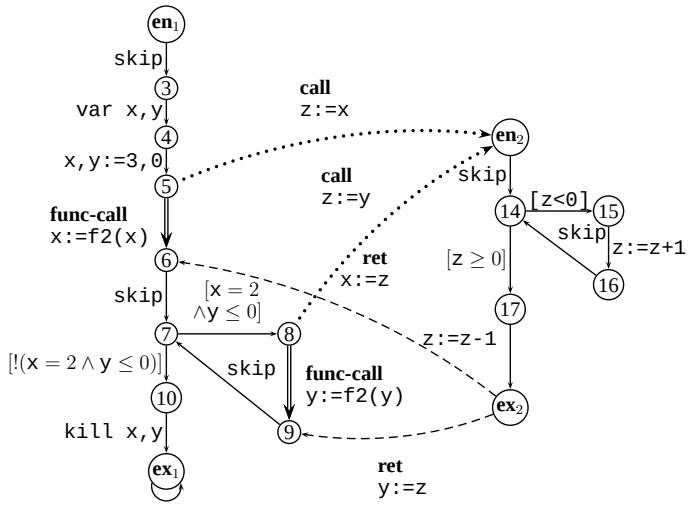


Figure 4.2: (a) A program $\texttt{EX}_1$ and (b) its ICFG.

**Interprocedural Control Flow Graph.** A PL program is represented by an *Interprocedural Control Flow Graph* (ICFG) [SP81]. An ICFG is a labeled graph $G = \langle Loc, Edge, \pi \rangle$, where $Loc$ is a finite set of locations, $Edge \subseteq Loc \times Loc$ is a set of edges, and $\pi$ labels each edge with a program statement. For example, the ICFG for the program $\texttt{EX}_1$ (see Fig. 4.2(a)) is shown in Fig. 4.2(b). In ICFGs, (a) each function has a unique *entry* (**en**) and *exit* (**ex**); (b) there is a self-loop at **ex** of $f_1$ to ensure existence of an infinite execution; (c) each function call (**func-call**) is: a **call** edge, where the values of actual parameters of the callee function are assigned to the formal parameters, a function body, and a **ret** edge, where the return values are assigned to the variables of the caller.

We assume that **call** and **ret** edges are uniquely determined by each other. For a **call** edge $(k, \mathbf{en})$ and the corresponding **ret** edge $(\mathbf{ex}, l)$, $k$ is the call location, $\mathit{call}(l) \triangleq k$, and $l$ is the return location, $\mathit{ret}(k) \triangleq l$.

| **Statement** $\pi(\langle k,l\rangle)$ | **Operational Semantics** $r_{\langle k,l\rangle}$ | **Mixed Semantics** $r^{\mathrm{m}}_{\langle k,l\rangle}$ |
|---|---|---|
| **func-call** edge $(U \to U)\ \mathbf{x} := f(\mathbf{a})$ | $\emptyset$ | $(U \to U) \quad ([\![\mathbf{p}_f := \mathbf{a}]\!] \circ [\![f]\!] \circ [\![\mathbf{x} := \mathbf{r}_f]\!])$ |
| **call** edge $S \equiv (U \to \mathbf{x})\ \mathbf{x} := \mathbf{e}$ | $\Gamma_t = s \wedge (\sigma_k, \sigma_l) \in [\![S]\!]$ | $[\![S]\!]$ |
| **ret** edge $(U \to V)\ \mathbf{x} := \mathbf{r}$ | let $\quad (c, \sigma_c).\Gamma_c = \Gamma_s$ in<br>$\Gamma_t = \Gamma_c \wedge l = ret(c)$<br>$\wedge \quad \sigma_l = \sigma_c[\{\mathbf{x}_i \mapsto \sigma_k(\mathbf{r}_i)\}]$ | $\emptyset$ |
| **Intraprocedural**: $S$ | $\Gamma_t = \Gamma_s \wedge (\sigma_k, \sigma_l) \in [\![S]\!]$ | $[\![S]\!]$ |

Table 4.2: The rules of operational and mixed semanics. $U$ is the set of local variables in the scope of the function call; $[\![f]\!]$ is natural semantics, $\mathbf{p}_f$ are the formals, and $\mathbf{r}_f$ are the returns of $f$.

**Operational Semantics** of a program $P = \langle Loc, Edge, \pi\rangle$ is a boolean transition system $B = \langle \mathcal{S}, \mathcal{R}\rangle$. Each state in $\mathcal{S}$ is a stack of activation records where each record is of the form $\langle pc, \sigma\rangle$, where $pc \in Loc$ is a program counter, corresponding to a particular control location in $P$, and $\sigma \in \Sigma_{V(pc)}$ is the valuation for variables in the scope of $pc$ (denoted by $V(pc)$). For a state $s = (k, \sigma_k).\Gamma$, the record $(k, \sigma_k)$ is the *top* element of $s$, denoted by $top(s)$. We use $|s|$ and$|\Gamma|$ to denote the length of $s$ and $\Gamma$, respectively. For a pair of states $s = (k, \sigma_k).\Gamma_s$ and $t = (l, \sigma_l).\Gamma_t$, the transition relation $\mathcal{R}$ is defined as $\mathcal{R}(s,t) \triangleq \langle k, l\rangle \in Edge \wedge r_{\langle k,l\rangle}(s,t)$, where $r_{\langle k,l\rangle}$ is a deterministic (but not necessarily total) relation on $\mathcal{S}$ at the edge $\langle k, l\rangle$, as defined in the 2nd column of Table 4.2. An intraprocedural statement only modifies the top activation record, and a statement on a **call** or a **ret** edge pushes a new record or pops one, respectively. The transition relations on **func-call** edges are empty, i.e., these edges are removed.

**Natural Semantics** [NN92] (a.k.a. big-step) of a block of code $S$ is a relation $[\![S]\!] \subseteq \Sigma \times \Sigma$ between the input and output of $S$: i.e., $(\sigma, \sigma') \in [\![S]\!]$ iff the execution of $S$ on $\sigma$ terminates and results in $\sigma'$. Natural semantics of a program $P \equiv f_1, \cdots, f_n$ is a set of relations, one per function, i.e., $[\![P]\!] = \langle [\![f_1]\!], \cdots, [\![f_n]\!]\rangle$.

The semantic rules for PL are defined compositionally on the syntax using the function $[\![\cdot]\!]_\varepsilon$, where $\varepsilon$ is an environment mapping free fixpoint variables (used for loops and functions) to relations with an appropriate type. Natural semantics for atomic statements is the same as base semantics; the other cases are:

$$
\begin{array}{lcl}
[\![S_1; S_2]\!]_\varepsilon & \triangleq & [\![S_1]\!]_\varepsilon \circ [\![S_2]\!]_\varepsilon \\
[\![\mu X \cdot \mathcal{S}(X)]\!]_\varepsilon & \triangleq & \text{lfp}\left(\lambda Z \cdot [\![\mathcal{S}(X)]\!]_{\varepsilon\{X \mapsto Z\}}\right) \\
[\![S_1 \parallel S_2]\!]_\varepsilon & \triangleq & [\![S_1]\!]_\varepsilon \vee [\![S_2]\!]_\varepsilon \\
[\![\mathbf{x} := f(\mathbf{a})]\!]_\varepsilon & \triangleq & [\![\mathbf{p}_f := \mathbf{a}; X_f; \mathbf{x} := \mathbf{r}_f]\!]_\varepsilon \\
[\![X]\!]_\varepsilon & \triangleq & \varepsilon(X) \\
[\![\textbf{while}(e)\ S]\!]_\varepsilon & \triangleq & [\![\mu X_w \cdot \textbf{if}(e)\ \textbf{then}\ (S; X_w)]\!]_\varepsilon \\
[\![\textbf{if}(e)\ \textbf{then}\ S_1\ \textbf{else}\ S_2]\!]_\varepsilon & \triangleq & [\![(\textbf{assume}(e);\ S_1) \parallel (\textbf{assume}(\neg e);\ S_2)]\!]_\varepsilon
\end{array}
$$

where **lfp** denotes for least fixpoint, $\tau(\varepsilon(X_f)) = \mathbf{p}_f \rightarrow \mathbf{r}_f$ and $\tau(\varepsilon(X_w)) = \tau([\![S]\!]_\varepsilon)$. A program $P \equiv f_1, \cdots, f_n$ induces the system of equations

$$\varepsilon(X_{f_i}) = [\![S_{f_i}]\!]_\varepsilon \quad (1 \leq i \leq n) \tag{nat}$$

Natural semantics of $P$ is the least fixpoint solution to this system.

For example, for the function $f_2$ in the program $\texttt{EX}_1$, we use an additional fixpoint variable $X_w$ to model the loop by tail recursion, and the instance of equation (nat) is:

$$\varepsilon(X_{f_2}) = \varepsilon(X_w) \circ [\![\mathtt{z} := \mathtt{z} - 1]\!]_\varepsilon$$
$$\varepsilon(X_w) = [\![\textbf{if}(\mathtt{z} < 0)\ \textbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w)]\!]_\varepsilon$$

The least solution to this system is computed as follows:

1. Following the base semantics of $\mathtt{z} := \mathtt{z} - 1$, we have that $[\![\mathtt{z} := \mathtt{z} - 1]\!]_\varepsilon = z' = z - 1$.

2. By induction, the least fixpoint for $\varepsilon(X_w)$ is $\varepsilon(X_w) = (z > 0 \wedge z' = z) \vee (z \leq 0 \wedge z' = 0)$.

3. Finally,

$$\begin{aligned}
\varepsilon(X_{f_2}) &= \big((z > 0 \wedge z' = z) \vee (z \leq 0 \wedge z' = 0)\big) \circ \big(z' = z - 1\big) \\
&= (z > 0 \wedge z' = z - 1) \vee (z \leq 0 \wedge z' = -1)
\end{aligned}$$

Therefore, natural semantics of $f_2$ is $(z > 0 \wedge z' = z - 1) \vee (z \leq 0 \wedge z' = -1)$.

Since natural semantics of a function captures relation between input and output of terminating executions of the function, every function call over these executions returns.

**Theorem 4.1.** *[NN92] Let $P \equiv f_1, \cdots, f_n$ be a program and $B = \langle \mathcal{S}, \mathcal{R} \rangle$ be its operational semantics. For a pair of activation records $\langle k, \sigma_k \rangle$ and $\langle l, \sigma_l \rangle$, $(\sigma_k, \sigma_l)$ is in $[\![f_i]\!]$ iff there exists a path $s_0, \cdots, s_m$ in $B$ such that $s_0 = \langle k, \sigma_k \rangle.\Gamma_0$ and $s_m = \langle l, \sigma_l \rangle.\Gamma_m$, such that $\Gamma_0 = \Gamma_m$, $k$ and $l$ are **en** and **ex** of $f_i$, respectively, and for all other $s_j = \langle p, \sigma_p \rangle.\Gamma_j$ either $\Gamma_j \neq \Gamma_0$ or $p$ is not **ex** of $f_i$.*

## 4.4 Mixed Semantics

We focus on reachability and non-termination properties of recursive programs in this chapter because of their practical interest. In this section, we define a stack-free operational semantics, called *mixed* semantics, for PL programs, which removes call stacks but preserves reachability and non-termination properties with respect to operational semantics.

**Reachability and Non-termination.** Given a boolean transition system $B = \langle \mathcal{S}, \mathcal{R} \rangle$, where $\mathcal{S}$ is a set of states and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation. Let $p$ be an atomic proposition, and $\mathcal{S}_p \triangleq \{s \in \mathcal{S} \mid s \models p\}$ be the set of states satisfying $p$. Recall that a *reachability* property (*EF* $p$) is true at a state $s$ if there exists a path from $s$ to a state in $\mathcal{S}_p$, and a *non-termination* property (*EG* $p$) is true in a state $s$ if there exists an infinite path starting at $s$ and contained in $\mathcal{S}_p$. The set $RS$ of all states satisfying *EF* $p$ is the least solution to equation (reach), and the set $NT$ of all states satisfying *EG* $p$ is the greatest solution to equation (non-term):

$$RS = \mathcal{S}_p \cup \mathit{pre}[\mathcal{R}](RS) \qquad \text{(reach)}$$

$$NT = \mathit{pre}[\mathcal{R} \cap \mathcal{S}_p](NT) \qquad \text{(non-term)}$$

Reachability and non-termination of a recursive program can be reduced to finding the fixpoint solutions to the equation (reach) and (non-term), respectively, w.r.t. a transition system

of operational semantics of a program. However, since operational semantics explicitly exposes a potentially unbounded call stack at each state, these equations must be solved over an infinite transition system (even when all program variables range over finite domains). Thus, the exact fixpoint solution may not be computable.

However, many program properties depend only on the top of the call stack: i.e., they are *stack-independent*. Analysis of such properties can be done using *stack-free* operational semantics in which everything except for the *top* activation record is abstracted away. We apply this idea to the analysis of reachability and non-termination. In this chapter, a reachability property is expressed as *EF* $p$ where $p$ is a proposition that depends only on the top activation record. Without loss of generality, we further assume that $p$ only depends on program locations, i.e., it is of the form $pc = x$. We assume there is a *main* function in a program that is not called by other functions. The *exit* location of the *main* function is denoted by `END`. The non-termination property is expressed as *EG* $(pc \neq \texttt{END})$.

**Mixed Semantics.** We now define a stack-free operational semantics, called mixed semantics, for PL programs which removes the call stack but preserves reachability and non-termination properties w.r.t. operational semantics. Intuitively, *mixed* semantics is a combination of operational and natural semantics, in which a program is executed as follows: an atomic statement is executed as usual; a function call $\texttt{x} := \texttt{foo}(\texttt{y})$ is executed as a *non-deterministic* choice between (a) executing `foo`, i.e., updating the top activation record according to natural semantics of `foo`, and (b) entering the body of `foo`, *and* forgetting all but the top activation record. Upon reaching the end of the main function, the execution enters a self-loop indicating the end of the program, and blocks at all other exit locations since it does not remember the origin of the call. For example, consider mixed execution of the program $\texttt{EX}_1$ starting from line 5 with $x = 3$ and $y = 0$. At this point, the execution can either (a) move to line 6 and decrease $x$ by one according to natural semantics of $f_2$, or (b) move to `en2` (line 13), assign 3 to $z$, and forget about $x$ and $y$. Within $f_2$, the execution continues until it blocks at `ex2` (line 18) with $z = 2$.

Formally, mixed semantics of a program $P = \langle Loc, Edge, \pi \rangle$ is a boolean transition system

$B^{\mathfrak{m}} = \langle \mathcal{S}^{\mathfrak{m}}, \mathcal{R}^{\mathfrak{m}} \rangle$, where each state is a *single* activation record $\langle pc, \sigma \rangle$. For a pair of states $s = \langle k, \sigma_k \rangle$ and $t = \langle l, \sigma_l \rangle$, the transition relation is

$$\mathcal{R}^{\mathfrak{m}}(s,t) \triangleq (\langle k,l \rangle \in Edge) \wedge r^{\mathfrak{m}}_{\langle k,l \rangle}(\sigma_k, \sigma_l)$$

where $r^{\mathfrak{m}}_{\langle k,l \rangle}$ is a relation on valuations, as defined in the 3rd column of Table 4.2. Note that $r^{\mathfrak{m}}_e$ for $\mathbf{ret}$ edges is empty, which is equivalent to removing those edges from the ICFG.

Mixed semantics preserves reachability and non-termination properties w.r.t. operational semantics. If an execution of a function $f$ reaches a state $s$ under the latter, then either $s$ is a location within $f$, or it is inside some other function that $f$ calls (directly or indirectly). The non-deterministic treatment of function calls in the former ensures that both of these cases are covered. Similarly, if there exists an infinite execution starting inside $f$, then either this execution lies within $f$, or $f$ calls a function that does not return the control back to $f$. Again, both cases are captured by mixed semantics.

In the remainder of this section, let $B$ and $B^{\mathfrak{m}}$ be the operational and mixed semantics of a given program, respectively.

**Theorem 4.2.** *Let $p$ be a propositional formula on control locations, and $s$ be a single activation record. Then, the followings hold*

*(1) $s$ satisfies EF $p$ on $B$ if and only if it satisfies EF $p$ on $B^{\mathfrak{m}}$;*

*(2) $s$ satisfies EG $(pc \neq \mathsf{END})$ on $B$ if and only if it satisfies EG $(pc \neq \mathsf{END})$ on $B^{\mathfrak{m}}$.*

We first provide the following lemmas that are used for the proof of this theorem.

**Lemma 4.3.** *Let $s$ be a single activation record. Then if there exists a path from $s$ to a state $s'$ in $B$, there also exists a path from $s$ to $top(s')$ in $B^{\mathfrak{m}}$.*

**Proof:**

We consider two cases of $s'$

(a) $|s'| = 1$, i.e., $s'$ is a single activation record.

(b) $|s'| > 1$

**Case (a):** Let $\tau : s = s_0, s_1, ..., s_n = s'$ be the path from $s$ to $s'$ in $B$. Since $s'$ is a single activation record, we have $n \geq 0$. We prove the result by induction on the length of $\tau$, denoted by $|\tau|$.

**Base case:** $|\tau| = 0$, is trivial.

**Inductive case:** Suppose the result holds for $|\tau| \leq n$. We show that the result also holds for $|\tau| = n + 1$, where $\tau : s = s_0, s_1, ..., s_n, s_{n+1} = t$. Consider the following two cases of $s_n$:

(i) $|s_n| = 1$, i.e., $s_n$ is a single activation record:

Let $s_n = (l_n, \sigma_n)$ and $s_{n+1} = s' = (l_{n+1}, \sigma_{n+1})$.

Since $(s_n, s_{n+1})$ is in $B$, and both $s_n$ and $s_{n+1}$ are single activation records, by the operational semantics, the edge $\langle l_n, l_{n+1} \rangle$ is labeled with an intraprocedural statement.

By the mixed semantics of intraprocedural statements, $(s_n, s_{n+1})$ is in $B^m$.

Furthermore, by inductive hypothesis, there exists a path from $s$ to $s_n$ in $B^{\mathfrak{m}}$. Therefore, there exists a path from $s$ to $s_{n+1} = s'$ in $B^{\mathfrak{m}}$.

(ii) $|s_n| > 1$, i.e., $s_n$ is not a single activation record:

Since $s_{n+1}$ is a single activation record, by the operational semantics, the transition $(s_n, s_{n+1})$ corresponds to the execution along a **ret** edge (illustrated in Figure 4.3).

That is, we have $s_n = (l_n, \sigma_n).\Gamma_n$, and $s_{n+1} = (l_{n+1}, \sigma_{n+1})$ such that $l_n$ is the exit (**ex**) location of a function $f$, and the edge $\langle l_n, l_{n+1} \rangle$ is a **ret** edge from the **ex** of $f$.

By the operational semantics, there exist $s_k$ and $s_{k+1}$ (where $0 \leq k \leq n-1$) on $\tau$ such that the transition $(s_k, s_{k+1})$ corresponds to the execution along a **call** edge and matches the transition $(s_n, s_{n+1})$.

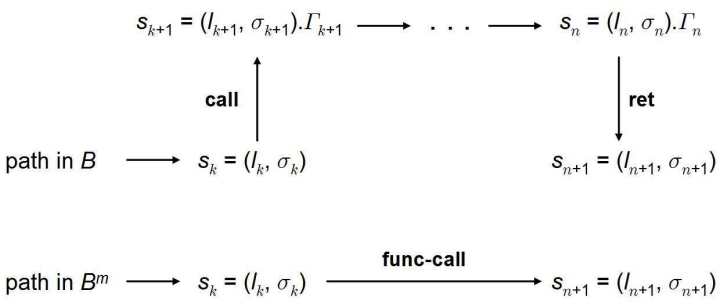


Figure 4.3: Illustration of proof, where $\Gamma_{k+1} = \Gamma_n = s_k$.

That is, we have $s_k = (l_k, \sigma_k)$ and $s_{k+1} = (l_{k+1}, \sigma_{k+1}).\Gamma_{k+1}$ such that $l_{k+1}$ is the entry (**en**) location of the function $f$, $\Gamma_{k+1} = \Gamma_n = s_k$, and $\langle l_k, l_{k+1} \rangle$ is a **call** edge to the **en** of $f$.

By Theorem 4.1, $(\sigma_{k+1}, \sigma_n)$ is in $||f||$.

Then, by the mixed semantics, $(s_k, s_{n+1})$ is in $B^m$, which corresponds to the execution along a **func-call** edge of $f$.

Furthermore, since $s_k$ is a single activation record and $k < n$, by inductive hypothesis, there exists a path from $s$ to $s_k$ in $B^{\mathfrak{m}}$. Therefore, there exists a path from $s$ to $s_{n+1}$ in $B^{\mathfrak{m}}$.

**Case (b):** In this case, $s'$ is not a single activation record. Let $s' = t.\Gamma$, where $t = top(s')$. Let $\tau : s = s_0, s_1, ..., s_n = s'$ be the path from $s$ to $s'$ in $B$. We show that there exists a path from $s$ to $t$ in $B^m$. We prove the result by induction on $|\tau|$. Since $s'$ is not a single activation record, $|\tau| \geq 1$.

**Base case:** $|\tau| = 1$.

In this case, since $s$ is a single activation record, but $s'$ is not, the transition $(s, s')$ corresponds to the execution along a **call** edge to a function $f$.

That is, we have $s = (l_0, \sigma_0)$ and $t = top(s') = (l_1, \sigma_1)$ such that $l_1$ is the entry (**en**) location of the function $f$, and $\langle l_0, l_1 \rangle$ is a **call** edge to the **en** of $f$.

By the mixed semantics, $(s,t)$ is in $B^m$, which corresponds to the execution along the **call** edge.

**Inductive case:** Suppose the result holds for $|\tau| \leq n$. We show that the result also holds for $|\tau| = n+1$, where $\tau : s = s_0, s_1, ..., s_n, s_{n+1} = s' = t.\Gamma$.

Let $t = (l_{n+1}, \sigma_{n+1})$. Let $s_n = t_n.\Gamma_n$ and $t_n = (l_n, \sigma_n)$. Consider the following cases of the edge $\langle l_n, l_{n+1}\rangle$ that corresponds to the transition $(s_n, s_{n+1})$ on $\tau$.

(i) $\langle l_n, l_{n+1}\rangle$ is labeled with an intraprocedural statement, or is a **call** edge:

In this case, by the mixed semantics, $(t_n, t)$ is in $B^m$.

Furthermore, by induction hypothesis, there is a path from $s$ to $t_n$ in $B^{\mathfrak{m}}$. Therefore, there is a path from $s$ to $t$ in $B^{\mathfrak{m}}$.

(ii) $\langle l_n, l_{n+1}\rangle$ is a **ret** edge from a function $f$:

In this case, by the operational semantics, there exist $s_k$ and $s_{k+1}$ (where $0 \leq k \leq n-1$) on $\tau$ such that the transition $(s_k, s_{k+1})$ corresponds to the execution along a **call** edge to the function $f$ and matches the transition $(s_n, s_{n+1})$.

Similar to the inductive case (2) in Case (a), let $t_k = top(s_k)$, we have $(t_k, t)$ in $B^m$ that corresponds to the execution along a **func-call** edge of $f$.

Furthermore, since $k < n$, by induction hypothesis, there is a path from $s$ to $t_k$ in $B^{\mathfrak{m}}$. Therefore, there is a path $s$ to $t$ in $B^{\mathfrak{m}}$.

□

**Lemma 4.4.** *Let $s$ and $t$ be two single activation records. If there exists a path from $s$ to $t$ in $B^{\mathfrak{m}}$, then there exists a path from $s$ to a state $s' = t.\Gamma$ in $B$.*

**Proof:**

Let $\tau : s = s_0, s_1, ..., s_n = t$ be a path in $B^{\mathfrak{m}}$. We prove the result by induction on $|\tau|$.

**Base case:** $|\tau| = 0$, is trivial.

**Inductive case:** Suppose the result holds for $|\tau| \leq n$. We show that it holds for $|\tau| = n+1$, where $\tau : s = s_0, s_1, ..., s_n, s_{n+1} = t$.

By induction hypothesis, there exists a path from $s$ to $s'_n = s_n.\Gamma_n$ in $B$. Let $s_n = (l_n, \sigma_n)$ and $t = (l_{n+1}, \sigma_{n+1})$. We show that there exists a path from $s'_n$ to $s' = t.\Gamma$ in $B$. We consider the following cases of the edge $\langle l_n, l_{n+1}\rangle$ that corresponds to the transition $(s_n, t)$,

(i) $\langle l_n, l_{n+1}\rangle$ is labeled with an intraprocedural statement:

Let $\Gamma = \Gamma_n$, i.e., $s' = t.\Gamma_n$. By the operational semantics of intraprocedural statements, $(s'_n, s')$ is in $B$.

(ii) $\langle l_n, l_{n+1}\rangle$ is a **call** edge:

Let $\Gamma = s'_n$, i.e., $s' = t.s'_n$. By the operational semantics for **call** edges, $(s'_n, s')$ is in $B$.

(iii) $\langle l_n, l_{n+1}\rangle$ is a **func-call** edge of a function $f$:

Let $\Gamma = \Gamma_n$, i.e., $s' = t.\Gamma_n$. By the mixed semantics of **func-call** edges and Theorem 4.1, there exists a path from $s'_n$ to $s'$ in $B$ that corresponds to the execution along a **call** edge to $f$, the function $f$, and a **ret** edge from $f$.

□

We now give the proof of Theorem 4.2

**Proof:**

(1) The proof that $s$ satisfies *EF* $p$ on $B$ if and only if it satisfies *EF* $p$ on $B^{\mathfrak{m}}$ trivially follows from Lemma 4.3 and Lemma 4.4.

(2) We show that $s$ satisfies *EG* $(pc \neq \texttt{END})$ on $B$ if and only if it satisfies *EG* $(pc \neq \texttt{END})$ on $B^{\mathfrak{m}}$.

**(⇒)** Let $\tau : s = s_0, s_1, ...$ be an infinite path in $B$ such that $top(s_i) \models (pc \neq \texttt{END})$ for any $i \geq 0$.

We first define an infinite sequence $\tau'$ over the states in $\tau$ such that each pair in $\tau'$ corresponds to one or more transitions on $\tau$. $\tau'$ is defined as follows:

– The first element of $\tau'$ is $s_0$, i.e., $\tau'[0] = s_0$.

– For each $i \geq 0$, suppose $s_k$ is the state in $\tau$ such that $s_k = \tau'[i]$. Consider the following cases of the transition $(s_k, s_{k+1})$ on $\tau$:

(i) if $(s_k, s_{k+1})$ corresponds to the execution of an intraprocedural statement, then $\tau'[i+1] = s_{k+1}$.

(ii) if $(s_k, s_{k+1})$ corresponds to the execution along a **call** edge and is not matched by any transition along a **ret** edge on $\tau$, i.e., $\forall j \geq k+1 \cdot |s_j| > |s_k|$, then $\tau'[i+1] = s_{k+1}$.

(iii) if $(s_k, s_{k+1})$ corresponds to the execution along a **call** edge and is matched by a transition $(s_m, s_{m+1})$ ($m > k+1$) along a **ret** edge on $\tau$, then $\tau'[i+1] = s_{m+1}$.

Based on $\tau'$, we define an infinite sequence $\tau^m$ of single activation records such that $\tau^m[i] = top(\tau'[i])$ for any $i \geq 0$.

By the definition of mixed semantics, every pair $(\tau^m[i], \tau^m[i+1])$ is in $B^m$, which corresponds to the execution of an intraprocedural statement (for case (i)), along a **call** edge (for case (ii)), or along a **func-call** edge (for case (iii)).

Since $s = s_0 = \tau^m[0]$, and $\tau^m[i] \models (pc \neq \texttt{END})$ for any $i \geq 0$, $s$ satisfies *EG* $(pc \neq \texttt{END})$ on $B^{\mathfrak{m}}$.

**(⇐)** The proof is symmetric to the one above. For each infinite path in $B^{\mathfrak{m}}$ demonstrating that $s$ satisfies *EG* $(pc \neq \texttt{END})$, by Lemma 4.4, we can construct an infinite path to show the satisfaction of the property at $s$ on $B$ as well.

□

When all variables of a given program $P$ range over finite domains, mixed semantics of $P$ is a finite boolean transition system. Theorem 4.2 implies the following analysis algorithm:

Step 1: compute natural semantics of $P$ by solving equation (nat);

Step 2: construct the structure $B^{\mathfrak{m}}$ following the rules of mixed semantics;

Step 3: solve equations (reach) or (non-term) on $B^{\mathfrak{m}}$ for reachability or non-termination,

respectively.

While sound and complete, this algorithm is not efficient, since it relies on the (potentially unnecessary) computation of "full" natural semantics of all functions (for Step 2) and the construction of "full" mixed semantics before the analysis of the property can even begin. As a trivial example, consider checking *EF*$(pc = 5)$ on the program $\texttt{EX}_1$. Since reachability of line 6 is irrelevant for this analysis, there is no need to construct the transition relation corresponding to **func-call** edge $\langle 5, 6\rangle$ and thus no need to compute natural semantics of $f_2$. Following this observation, in the next section, we show that the three steps of the above algorithm can be combined into an *on-the-fly* algorithm that only computes the necessary parts of mixed and natural semantics.

## 4.5 On-the-Fly Reachability and Non-Termination

We formalize the on-the-fly analysis of reachability and non-termination as equation systems in Section 4.5.1 and Section 4.5.2, respectively.

### 4.5.1 On-the-Fly Reachability

Intuitively, the analysis of *EF* $p$ properties only needs the part of mixed semantics that is used for solving equation (reach), and that, in turn, drives the computation of the necessary parts of natural semantics. To illustrate, consider checking *EF*$(pc = 8)$ on $\texttt{EX}_1$. Natural semantics of $f_2$ is $[\![f_2]\!] \equiv (z > 0 \wedge z' = z - 1) \vee (z \leq 0 \wedge z' = -1)$. After a few iterations, the reachability algorithm computes a pre-condition $Q \equiv x = 2 \wedge y \leq 0$ for reaching line 8 from line 6. To determine a pre-condition for $Q$ w.r.t. a function call `x:=f2(x)` at line 5, it needs to compute *pre*$[r^{\mathtt{m}}_{\langle 5,6\rangle}](Q) = (x = 3 \wedge y \leq 0)$, where

$$r^{\mathtt{m}}_{\langle 5,6\rangle} \equiv (y' = y) \wedge ((x > 0 \wedge x' = x - 1) \vee (x \leq 0 \wedge x' = -1))$$

is the instantiation of $[\![f_2]\!]$ to the call site. However, instead of using the "full" version of $[\![f_2]\!]$, it is sufficient to compute a pre-condition that *assumes* $Q$ as a post-condition, i.e., to restrict $r^{\mathfrak{m}}$ to $x' = 2$ (the relevant part of $Q$) yielding

$$\hat{r}^{\mathfrak{m}} \equiv y' = y \wedge x = 3 \wedge x' = 2$$

$\hat{r}^{\mathfrak{m}}$ is an instantiation of $z = 3 \wedge z' = 2$ in the context of the call, obtained by (a) converting $Q$ to a postcondition of $f_2$ by taking its pre-image over the **ret** edge (which eliminates $y$ and renames $x$ to $z$), and (b) restricting $[\![f_2]\!]$ to this post-condition:

$$\begin{aligned} & [\![f_2]\!] \circ (\mathit{assume}(z = 2)) \\ \equiv \quad & z = 3 \wedge z' = 2 \end{aligned}$$

We now formalize the above intuition. Recall that $V(k)$ stands for the set of variables in the scope of a location $k$. Let $l$ be the return location of a function call to $f_i$, $Q \subseteq \Sigma_{V(l)}$ be a set of valuations at $l$, and the corresponding **ret** edge $\langle \mathbf{ex}_i, l \rangle$ be labeled with $\mathbf{x} := \mathbf{r}_{f_i}$. Then, function

$$\mathit{prop}(\langle \mathbf{ex}_i, l \rangle, Q) \triangleq \mathit{pre}\,[[\mathbf{x} := \mathbf{r}_{f_i};\ (\mathbf{x} \to V(l))\mathbf{var}\ (V(l) \setminus \mathbf{x})]]\,(Q)$$

turns $Q$ into a post-condition of $f_i$. Here, the pre-image w.r.t. **var** undeclares (or removes) all variables that are not changed by the call, and the pre-image w.r.t. **ret** edge turns the post-condition on $\mathbf{x}$ into the one on $\mathbf{r}_{f_i}$.

Let $RS : Loc \to \mathbf{2}^{\Sigma}$ map each location $k$ to a subset of $\Sigma_{V(k)}$, and, as in Section 4.3, let $\varepsilon$ be the semantics environment, mapping each fixpoint variable to a relation of an appropriate type. The on-the-fly algorithm for reachability analysis is the equation system (reach-otf):

$$RS(k) = \begin{cases} \Sigma_{V(k)} & \text{if } k \models p\ (k \in \mathit{Loc}) \\ RS(k) \cup \bigcup_{l \in succ(k)} \mathit{pre}[\hat{r}^{\mathfrak{m}}_{\langle k,l \rangle}]\,(RS(l)) & \text{otherwise} \end{cases} \qquad \text{(reach-otf)}$$

$$\varepsilon(X_{f_i}) = [\![S_{f_i}]\!]_{\varepsilon} \circ \mathit{assume}\left(\bigcup_{l \in succ(\mathbf{ex}_i)} \mathit{prop}\,(\langle \mathbf{ex}_i, l \rangle, RS(l))\right) \quad (i \in [1..n])$$

where *succ* are the successors of a node in the ICFG, $S_{f_i}$ is the body of $f_i$, and for $S \equiv \pi(\langle k, l\rangle)$, $\hat{r}^{\mathfrak{m}}_{\langle k,l\rangle}$ is defined as:

$$\hat{r}^{\mathfrak{m}}_{\langle k,l\rangle} = \begin{cases} (U \to U)\ (\llbracket \mathbf{p}_f := \mathbf{a} \rrbracket \circ \varepsilon(X_{f_i}) \circ \llbracket \mathbf{x} := \mathbf{r}_f \rrbracket) & \text{if } S \equiv (U \to U)\ \mathbf{x} := f(\mathbf{a}) \\ \llbracket S \rrbracket & \text{otherwise} \end{cases}$$

This system is a combination of (nat) and (reach), where *prop* is used to propagate the reachability information to the computation of natural semantics. Since reachability and natural semantics are both least solutions to equations (reach) and (nat), respectively, we need the least solution to the above equation as well.

For example, the following is an instance of the system (reach-otf) for checking $EF(pc = 8)$ on $\texttt{EX}_1$ (for $RS$, only locations 5–7 are shown):

$$\begin{aligned} RS(5) &= RS(5) \cup \\ &\qquad \mathit{pre}[(\{x,y\} \to \{x,y\})(z' = x \circ \varepsilon(X_{f_2}) \circ x' = z)](RS(6)) \cup \\ &\qquad \mathit{pre}[(\{x,y\} \to \{z\})z' = x](RS(\mathbf{en_2})) \\ RS(6) &= RS(6) \cup \mathit{pre}[id(\{x,y\})](RS(7)) \\ RS(7) &= RS(7) \cup \\ &\qquad \mathit{pre}[\mathit{assume}(x = 2 \wedge y \leq 0)](RS(8)) \cup \\ &\qquad \mathit{pre}[\mathit{assume}(\neg(x = 2 \wedge y \leq 0))](RS(10)) \\ \varepsilon(X_{f_2}) &= \llbracket \mu X_w \cdot \mathbf{if}(\mathtt{z} < 0)\ \mathbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w) \rrbracket_\varepsilon \circ \llbracket \mathtt{z} := \mathtt{z} - 1 \rrbracket_\varepsilon \circ \\ &\qquad (\mathit{assume}\,(\mathit{prop}(\langle \mathbf{ex_2}, 6\rangle, RS(6))) \cup \mathit{assume}\,(\mathit{prop}(\langle \mathbf{ex_2}, 9\rangle, RS(9)))) \end{aligned}$$

Note that $RS(\mathbf{ex_2})$ does not appear in the computation of $RS(6)$, because **ret** edges are removed in mixed semantics.

Below, we show a possible fixpoint computation. We use the notation $RS(l)_i$ to denote the value of $RS(l)$ at the $i$th step of the computation. We assume that $RS(l)_i = RS(l)_{i-1}$ unless

stated otherwise.

1. Initially,

$$RS(l)_0 = \begin{cases} \mathsf{true} & \text{if } l = 8 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

2. Computing along the edges $\langle 7, 8\rangle$ and $\langle 7, 10\rangle$:

$$\begin{aligned} RS(7)_1 &= RS(7)_0 \cup \\ &\quad \mathit{pre}[\mathit{assume}(x = 2 \wedge y \leq 0)](RS(8)_0) \cup \\ &\quad \mathit{pre}[\mathit{assume}(\neg(x = 2 \wedge y \leq 0))](RS(10)_0) \\ &= \mathsf{false} \cup \\ &\quad \mathit{pre}[\mathit{assume}(x = 2 \wedge y \leq 0)](\mathsf{true}) \cup \\ &\quad \mathit{pre}[\mathit{assume}(\neg(x = 2 \wedge y \leq 0))](\mathsf{false}) \\ &= (x = 2) \wedge y \leq 0 \end{aligned}$$

3. Computing along the edge $\langle 6, 7\rangle$:

$$\begin{aligned} RS(6)_2 &= RS(6)_1 \cup \\ &\quad \mathit{pre}[id(\{x, y\})](RS(7)_1) \\ &= \mathsf{false} \cup \\ &\quad \mathit{pre}[x' = x \wedge y' = y](x = 2 \wedge y \leq 0) \\ &= (x = 2) \wedge y \leq 0 \end{aligned}$$

4. To compute $RS(5)_3$ along the **func-call** edge $\langle 5, 6\rangle$, we need the value of $\varepsilon(X_{f_2})$, which is a partial natural semantics of $f_2$ w.r.t. the *current* value of $RS(6)$. We first propagate

$RS(6)_2$ along the **ret** edge $\langle \mathbf{ex_2}, 6\rangle$:

$$\begin{aligned}
\mathit{prop}(\langle \mathbf{ex}_2, 6\rangle, RS(6)_2) &= \mathit{pre}[[\![x := z;\ (x \to \{x,y\})\mathbf{var}\ y]\!]](RS(6)_2)\\
&= \mathit{pre}[x' = z\ \circ\ (x \to \{x,y\})decl[y]](RS(6)_2)\\
&= \mathit{pre}[x' = z\ \circ\ (x \to \{x,y\})decl[y]](x = 2 \wedge y \leq 0)\\
&= \mathit{pre}[x' = z](x = 2)\\
&= (z = 3)
\end{aligned}$$

Since $RS(9)_2$ is empty, the computation of $\varepsilon(X_{f_2})$ at this step is restricted only to the post-condition $\mathit{assume}(z = 2)$.

$$\begin{aligned}
\varepsilon(X_{f_2}) &= [\![\mu X_w \cdot \mathbf{if}(\mathtt{z} < 0)\ \mathbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w)]\!]_\varepsilon \circ [\![\mathtt{z} := \mathtt{z} - 1]\!] \circ\\
&\quad assume(z = 2)\\
&= [\![\mu X_w \cdot \mathbf{if}(\mathtt{z} < 0)\ \mathbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w)]\!]_\varepsilon \circ\\
&\quad (z = 3 \wedge z' = 2)\\
&= (z = 3) \wedge z' = 2
\end{aligned}$$

This on-the-fly computation of $\varepsilon(X_{f_2})$ avoided computing the semantics of the loop $[\![\mu X_w \cdot \mathbf{if}(\mathtt{z} < 0)\ \mathbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w)]\!]_\varepsilon$ which is required for the "full" natural semantics of $f_2$.

5. Computing along the **func-call** edge $\langle 5, 6\rangle$:

$$\begin{aligned}
RS(5)_3 &= RS(5)_2 \cup \mathit{pre}[(\{x,y\} \to \{x,y\})(z' = x \circ \varepsilon(X_{f_2}) \circ x' = z)](RS(6)_2) \cup\\
&\quad \mathit{pre}[(\{x,y\} \to \{z\})z' = x](RS(\mathbf{en_2})_2)\\
&= \mathsf{false} \cup \mathit{pre}[(\{x,y\} \to \{x,y\})(z' = x \circ \varepsilon(X_{f_2}) \circ x' = z)](x = 2 \wedge y \leq 0) \cup\\
&\quad \mathit{pre}[(\{x,y\} \to \{z\})z' = x](\mathsf{false})\\
&= \mathit{pre}[x = 3 \wedge x' = 2 \wedge y' = y](x = 2 \wedge y \leq 0)\\
&= (x = 3) \wedge y \leq 0
\end{aligned}$$

The above computation establishes that location 8 is reachable from location 5 when $x = 3$ and $y \leq 0$.

The following theorem shows that the analysis based on equation system (reach-otf) is sound, and computes only the necessary part of natural semantics.

**Theorem 4.5.** *Let $RS_{\downarrow}$ and $\varepsilon_{\downarrow}$ be the least solutions to equation system (*reach-otf*). Then,*

*(1) $RS_{\downarrow}$ is the least solution to the equation (*reach*) on $B^{\mathtt{m}}$;*

*(2) for each $i \in [1..n]$, $\varepsilon_{\downarrow}(X_{f_i})$ is a subset of $[\![f_i]\!]$;*

*(3) for any $\varepsilon$, if $RS_{\downarrow}$ is the least solution to the $RS$ equations in (*reach-otf*) w.r.t. $\varepsilon$, then $\forall i \in [1..n] \cdot \varepsilon_{\downarrow}(X_{f_i}) \subseteq \varepsilon(X_{f_i})$.*

**Proof:**

The proof of (1) follows from the fact that the transfer function induced by the equation (reach-otf) is less than that induced by the equation (reach); therefore, the least solution to (reach-otf) is less than that to (reach). Note that $RS_{\downarrow}$ is also solution to (reach). Hence, $RS_{\downarrow}$ is the least solution to (reach).

The proofs of (2) and (3) follow from the equation of $\varepsilon(X_{f_i})$ in (reach-otf) and the definition of least solution, respectively. □

Part (1) of Theorem 4.5 shows that $RS_{\downarrow}$ is the solution for the reachability analysis; part (2) – that $\varepsilon_{\downarrow}$ is sound w.r.t. natural semantics of $f_i$; and part (3) – that $\varepsilon_{\downarrow}$ only contains the information necessary for the analysis.

Since we need the least solution for both $RS(k)$ and $\varepsilon(X_{f_i})$ equations, it can be obtained by any chaotic iteration [Cou77] and thus is independent of the order of computation of $RS$ and $\varepsilon$. Interestingly, the algorithm derived from (reach-otf) is a pre-image-based variant of the post-image-based reachability algorithm of BEBOP [BR00], and is similar to the backward analysis with $\mathbf{wp}$ described in [Bal04].

### 4.5.2 On-the-Fly Non-Termination

The derivation of the on-the-fly algorithm for the analysis of non-termination, (nt-otf), proceeds similarly, and is a combination of systems (nat) and (non-term):

$$NT(k) = \begin{cases} \emptyset & \text{if } k \not\models p \ (k \in Loc) \\ \bigcup_{l \in succ(k)} \mathit{pre}[\hat{r}^{\mathtt{m}}_{\langle k,l \rangle}](NT(l)) & \text{otherwise} \end{cases} \qquad \text{(nt-otf)}$$

$$\varepsilon(X_{f_i}) = [\![S_{f_i}]\!]_\varepsilon \circ \mathit{assume}\left(\bigcup_{l \in succ(\mathbf{ex}_i)} \mathit{prop}\left(\langle \mathbf{ex}_i, l \rangle, NT(l)\right)\right) \quad (i \in [1..n])$$

where $NT : Loc \to \mathbf{2}^{\Sigma}$ maps each location $k$ to a subset of $\Sigma_{V(k)}$, and *succ*, $S_{f_i}$ and $\hat{r}^{\mathtt{m}}$ are the same as those in (reach-otf). Since non-termination requires the greatest solution to (non-term), and natural semantics – the least solution to (nat), in (nt-otf), we need the greatest solution to $NT(k)$, and the least solution to $\varepsilon(X_{f_i})$ equations, respectively.

For example, the following is an instance of the system (nt-otf) for checking $EG(pc \neq \mathbf{ex}_1)$ on $\mathtt{EX}_1$ (for $NT$, only locations 5–10 are shown).

$$\begin{aligned}
NT(5) &= \mathit{pre}[(\{x,y\} \to \{x,y\})(z' = x \circ \varepsilon(X_{f_2}) \circ x' = z)](NT(6)) \cup \\
&\quad \mathit{pre}[(\{x,y\} \to \{z\})z' = x](NT(\mathbf{en_2})) \\
NT(6) &= \mathit{pre}[(id(\{x,y\})](NT(7)) \\
NT(7) &= \mathit{pre}[\mathit{assume}(x = 2 \wedge y \leq 0)](NT(8)) \cup \\
&\quad \mathit{pre}[\mathit{assume}(\neg(x = 2 \wedge y \leq 0))](NT(10)) \\
NT(8) &= \mathit{pre}[(\{x,y\} \to \{x,y\})(z' = y \circ \varepsilon(X_{f_2}) \circ y' = z)](NT(9)) \\
NT(9) &= \mathit{pre}[(id(\{x,y\})](NT(7)) \\
NT(10) &= \mathit{pre}[\mathit{kill}[\{x,y\}]](NT(\mathbf{ex_1}))
\end{aligned}$$

$$\begin{aligned}
\varepsilon(X_{f_2}) &= [\![\mu X_w \cdot \mathbf{if}(\mathtt{z} < 0)\ \mathbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w)]\!]_\varepsilon \circ [\![\mathtt{z} := \mathtt{z} - 1]\!] \circ \\
&\quad (\mathit{assume}\,(\mathit{prop}(\langle \mathbf{ex_2}, 6 \rangle, NT(6))) \cup \mathit{assume}\,(\mathit{prop}(\langle \mathbf{ex_2}, 9 \rangle, NT(9))))
\end{aligned}$$

A fixpoint computation for this system is shown below.

1. Initially:

$$NT(l) = \begin{cases} \mathsf{false} & \text{if } l \in \{\mathbf{ex_1}, \mathbf{ex_2}\} \\ \mathsf{true} & \text{otherwise} \end{cases}$$

2. Computing along the edge $\langle 10, \mathbf{ex_1} \rangle$:

$$\begin{aligned} NT(10)_1 &= \mathit{pre}[kill[\{x,y\}]](NT(\mathbf{ex_1})_0) \\ &\quad \mathit{pre}[kill[\{x,y\}]](\mathsf{false}) \\ &= \mathsf{false} \end{aligned}$$

3. Computing along the edges $\langle 7, 8 \rangle$ and $\langle 7, 10 \rangle$:

$$\begin{aligned} NT(7)_2 = \mathit{pre}[\mathit{assume}(x = 2 \wedge y \leq 0)](NT(8)_1) \cup \\ \mathit{pre}[assume(\neg(x = 2 \wedge y \leq 0))](NT(10)_1) \\ = \mathit{pre}[x = 2 \wedge y \leq 0 \wedge id(\{x,y\})](\mathsf{true}) \cup \\ \mathit{pre}[\neg(x = 2 \wedge y \leq 0) \wedge id(\{x,y\})](\mathsf{false}) \\ = (x = 2) \wedge y \leq 0 \end{aligned}$$

4. Computing along the edge $\langle 6, 7 \rangle$:

$$\begin{aligned} NT(6)_3 &= \mathit{pre}[(id(\{x,y\})](NT(7)_2) \\ &= \mathit{pre}[id(\{x,y\})](x = 2 \wedge y \leq 0) \\ &= (x = 2) \wedge y \leq 0 \end{aligned}$$

5. Computing along the edge $\langle 9, 7 \rangle$:

$$\begin{aligned} NT(9)_4 &= \mathit{pre}[(id(\{x,y\})](NT(7)_3) \\ &= \mathit{pre}[id(\{x,y\})](x = 2 \wedge y \leq 0) \\ &= (x = 2) \wedge y \leq 0 \end{aligned}$$

6. In order to compute $NT(5)_5$ and $NT(8)_5$, we need $\varepsilon(X_{f_2})$ w.r.t. the current values of $NT(6)$ and $NT(9)$, respectively. We first propagate $NT(6)_4$ and $NT(9)_4$ along the **ret**

edges $\langle \mathbf{ex_2}, 6\rangle$ and $\langle \mathbf{ex_2}, 9\rangle$, respectively:

$$\begin{aligned}
\mathit{prop}(\langle \mathbf{ex_2}, 6\rangle, NT(6)_4) &= \mathit{pre}[[\![x := z;\ (x \to \{x, y\})\mathbf{var}\ y]\!]](NT(6)_4) \\
&= \mathit{pre}[x' = z\ \circ\ (x \to \{x, y\})\mathit{decl}[y]](NT(6)_4) \\
&= \mathit{pre}[x' = z\ \circ\ (x \to \{x, y\})\mathit{decl}[y]](x = 2 \wedge y \leq 0) \\
&= (z = 3)
\end{aligned}$$

$$\begin{aligned}
\mathit{prop}(\langle \mathbf{ex_2}, 9\rangle, NT(9)_4) &= \mathit{pre}[[\![y := z;\ (y \to \{x, y\})\mathbf{var}\ x]\!]](NT(9)_4) \\
&= \mathit{pre}[y' = z\ \circ\ (y \to \{x, y\})\mathit{decl}[x]](NT(9)_4) \\
&= \mathit{pre}[y' = z\ \circ\ (y \to \{x, y\})\mathit{decl}[x]](x = 2 \wedge y \leq 0) \\
&= z \leq 0
\end{aligned}$$

We then compute $\varepsilon(X_{f_2})$ restricted to $\mathit{assume}(z = 2)$ and $assume(z \leq 0)$:

$$\begin{aligned}
\varepsilon(X_{f_2}) &= [\![\mu X_w \cdot \mathbf{if}(\mathtt{z} < 0)\ \mathbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w)]\!]_\varepsilon \circ [\![\mathtt{z} := \mathtt{z} - 1]\!]_\varepsilon \circ \\
&\quad \mathit{assume}(z = 2) \vee \mathit{assume}(z \leq 0) \\
&= [\![\mu X_w \cdot \mathbf{if}(\mathtt{z} < 0)\ \mathbf{then}\ (\mathtt{z} := \mathtt{z} + 1; X_w)]\!]_\varepsilon \circ \\
&\quad (z = 3 \wedge z' = 2) \vee (z' = z - 1 \wedge z' \leq 0) \\
&= (z = 3 \wedge z' = 2) \vee (z = 1 \wedge z' = 0) \vee (z \leq 0 \wedge z' = -1)
\end{aligned}$$

This partial semantics of $f_2$ does not include its behaviors for outputs $z > 2$ and $z = 1$, but it is sufficient for continuing the computation of $NT(6)_5$ and $NT(9)_5$.

The following theorem shows that the non-termination algorithm based on (nt-otf) is sound and computes only the necessary part of natural semantics.

**Theorem 4.6.** *Let $NT_\uparrow$ and $\varepsilon_\downarrow$ be the greatest solution for $NT$ and the least solution for $\varepsilon$ in system (*nt-otf*), respectively. Then,*

*(1) $NT_\uparrow$ is the greatest solution to the equation (*non-term*) on $B^{\mathtt{m}}$;*

*(2) for each $i \in [1..n]$, $\varepsilon_\downarrow(X_{f_i})$ is a subset of $[\![f_i]\!]$;*

*(3) for any $\varepsilon$, if $NT_{\uparrow}$ is the greatest solution to the $NT$ equations in (*nt-otf*) w.r.t. $\varepsilon$, then $\forall i \in [1..n] \cdot \varepsilon_{\downarrow}(X_{f_i}) \subseteq \varepsilon(X_{f_i})$.*

**Proof:**

The proof is similar to that of Theorem 4.5. □

As in Theorem 4.5, part (1) of Theorem 4.6 shows soundness of non-termination, and parts (2) and (3) show that $NT_{\uparrow}$ is sound and only contains necessary information for analysis, respectively.

Unlike reachability, non-termination requires different fixpoint solutions for $NT$ and $\varepsilon$, and thus the order of computation can influence the result. For example, consider checking $EG(pc \neq \mathbf{ex}_1)$ on $\mathtt{EX}_1$. Initially, lines 7, 8, and 9 are associated with all the valuations on $x$ and $y$, i.e., $NT(7) = NT(8) = NT(9) = \Sigma_{\{x,y\}}$, and $\varepsilon(f_2)$ is empty, which is not the partial semantics of $f_2$ restricted to $NT(9)$. If the computation of $NT$ proceeds along the function call `y:=f2(y)` using the initial value of $\varepsilon(f_2)$, $NT(8)$ is assigned $\emptyset$. Eventually, $NT(7) = NT(8) = NT(9) = \emptyset$, i.e., the algorithm incorrectly concludes that any execution starting at lines 7, 8 or 9 terminates.

The correct order for computing the solution is such that the pre-image of a set $Q$ w.r.t. a function call to $f$ has to be delayed until the derivation of $\varepsilon(X_f)$ w.r.t. $Q$ is finished. Nonetheless, since this order is only restricted to **func-call** edges, the order of the computation elsewhere can be arbitrary. This can be used to avoid deriving "full" natural semantics. Going back to the previous example, one can first compute $NT$ along all edges except for **func-call** edges, which will assign $NT(9)$ with $x = 2 \wedge y \leq 0$, and then compute natural semantics of $f_2$ restricted to the post-condition $z \leq 0$. Similarly, although initially $NT(6)$ is assigned $\Sigma_{\{x,y\}}$, $NT(6)$ equals $(x = 2 \wedge y \leq 0)$ after the initial computation of $NT$, which means that only partial natural semantics of $f_2$ restricted to the post-condition $z = 2$ is needed.

In this section, we have presented mixed semantics – a stack-free operational semantics of PL, and showed how it can be used to check reachability and non-termination of programs with a finite data domain. The mixed semantics combines both operational and natural semantics,

allowing us to analyzing recursive programs without dealing with call stacks. Although the basic idea of such combination has been used in other tools, e.g., BEBOP [BR00], we provided a formalization of both reachability and non-termination analysis using equation systems, which characterizes all the operations involved in the analysis. Such formalization enables a natural way to combine the analysis with abstraction, which is described in the next section.

## 4.6 Abstract Analysis

We describe our approach for deriving abstract versions of the concrete analysis described in Section 4.4, which follows abstract interpretation.

### 4.6.1 Abstract Domains and Operations

Abstract interpretation [CC77, CC92] is a theory for systematic derivation of abstract program analysis. In particular, in order to approximate the fixpoint of a function over a concrete domain, according to this theory, we need to define an abstract domain and provide sound abstract counterparts of the operations used in concrete fixpoint computation. The concrete analysis in Section 4.4 computes fixpoints of functions defined by equations (reach-otf) and (nt-otf) over the sets of states in $\mathbf{2}^{\Sigma}$ and relations in $\mathbf{2}^{\Sigma\times\Sigma}$. To derive abstract analysis, we require two abstract domains: abstract sets $A_s$ whose elements approximate sets in $\mathbf{2}^{\Sigma}$, and abstract relations $A_r$ whose elements approximate relations in $\mathbf{2}^{\Sigma\times\Sigma}$. These domains must be equipped with abstract versions of all of the operations used in the equations (reach-otf) and (nt-otf). The theory of abstract interpretation then ensures that the solution to an abstract equation is an approximation of the solution to the corresponding concrete equation. In what follows, we first identify the necessary abstract operations on $A_s$ and $A_r$, and then show how to adapt predicate abstraction for our algorithm.

According to the equations (reach-otf) and (nt-otf), the domain of abstract sets $A_s$ must be equipped with a set union $\cup$ (used in the fixpoint computation) and equality (to detect the fix-

point convergence). The domain of abstract relations $A_r$ must be equipped with (a) a pre-image operator to convert abstract relations to abstract transformers over $A_s$, (b) asynchronous and sequential compositions of abstract relations (used in natural semantics), (c) scope extension (used to instantiate a function call using natural semantics of a function), and (d) equality (to detect the fixpoint convergence). Furthermore, we need an *assume* operator that maps an abstract set to a corresponding abstract relation; and, to apply the abstraction directly to the source code, a computable version of abstract base semantics $[\![\cdot]\!]_\alpha$ that maps each atomic statement $S$ to an abstract relation that approximates $[\![S]\!]$ (the concrete semantics of $S$).

### 4.6.2 Predicate Abstraction

We now show how the domain of predicate abstraction [GS97, BPR03, GWC06a] can be extended with the necessary abstract operations to yield abstract reachability and non-termination algorithms.

*Abstract domains*. Predicate abstraction provides domains for abstracting elements, sets, and relations of valuations. Let $V$ be a set of variables, and $\mathcal{P}$ be a set of predicates over $V$. A *monomial* is a conjunction of literals of $P$. The *elementary* domain of predicate abstraction over $\mathcal{P}$, denoted $\text{Mon}(\mathcal{P})$, is the set of all monomials over $\mathcal{P}$. The soundness relation $\rho_\mathcal{P} \subseteq \Sigma_V \times \text{Mon}(\mathcal{P})$ is defined s.t. $(\sigma, a) \in \rho_P$ iff $\sigma \models a$, i.e., an element $a \in \text{Mon}(\mathcal{P})$ approximates a valuation $\sigma \in \Sigma_V$ iff $\sigma$ satisfies all literals in $a$. For example, if $\mathcal{P} = \{x > 0, x < y\}$, then $a_1 = (x > 0)$ and $a_2 = (x > 0) \wedge \neg(x < y)$ are in $\text{Mon}(\mathcal{P})$. A valuation $\sigma = \langle x \mapsto 2, y \mapsto 2\rangle$ is approximated by $a_1$, and is also more precisely approximated by $a_2$.

The elementary domain is lifted to abstract sets and relations to approximate the concrete domains $\mathbf{2}^{\Sigma_V}$ and $\mathbf{2}^{\Sigma_V \times \Sigma_V}$, respectively. In exact-approximating predicate abstraction [GC06, GWC06a] used by YASM, the abstract sets and relations are defined over $\mathbf{4}$-valued logic, that is, the abstract domains $\mathcal{A}_s$ and $\mathcal{A}_r$ are $\mathbf{4}^{\text{Mon}(\mathcal{P})}$ and $\mathbf{4}^{\text{Mon}(\mathcal{P})\times\text{Mon}(\mathcal{P})}$, corresponding to $\mathbf{4}$-valued sets and transition relations, respectively. The approximation relation between an abstract set $X \in \mathbf{4}^{\text{Mon}(\mathcal{P})}$ and a concrete set $Q \in \mathbf{2}^{\Sigma_V}$ is characterized by the information ordering on the

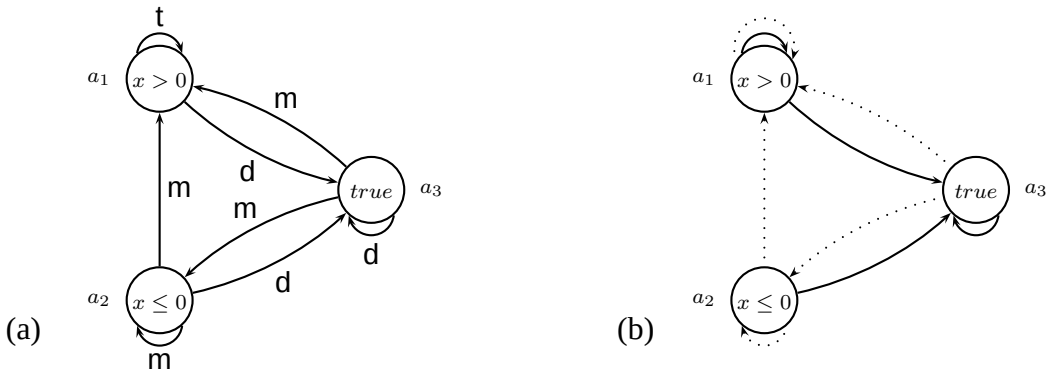


Figure 4.4: (a) A $\mathbf{4}$-valued transition relation $r$, and (b) a mixed transition relation $r'$ transformed from $r$, where solid and dotted lines represent *must* and *may* transitions, respectively.

truth domain, i.e., $X$ approximates $Q$ iff for any $\sigma \in \Sigma_V$ and $a \in \text{Mon}(\mathcal{P})$, $\rho(\sigma, a) \Rightarrow X(a) \preceq Q(\sigma)$. That is, if $a$ is inside (resp. outside) of $X$, then all the concrete elements approximated by $a$ are inside (resp. outside) of $Q$. The approximation relation between a $\mathbf{4}$-valued relation in $A_r$ and a concrete relation in $2^{\Sigma_V \times \Sigma_V}$ is based on mixed simulation.

*Abstract Operations*. The exact-approximating abstract base semantics of an atomic statement has also been defined in [GWC06a, GC06]. For example, let $x$ be an integer variable and $\mathcal{P} = \{x > 0\}$. For the statement $\texttt{x} := \texttt{x} + \texttt{1}$, a $\mathbf{4}$-valued transition relation $r$ approximating its base semantics is shown in Figure 4.4(a), where the abstract states $a_1$ and $a_2$ approximate positive and negative integer numbers, respectively, and $a_3$ approximates all the integer numbers. Figure 4.4(b) shows the mixed transition relation $r'$ transformed from $r$. The abstract transition relation $r$ (or, equivalently, $r'$) shows that if $x$ is a positive number, $x + 1$ is also positive; if $x$ is a non-positive or an arbitrary number, the sign of $x + 1$ is unknown; and $x + 1$ is always an integer number for any $x$. Such abstraction describes both possible and necessary behaviors expressed using the predicate $x > 0$, which is an exact-approximation of the concrete behaviors defined by $\texttt{x} := \texttt{x} + \texttt{1}$.

[GWC06a, GC06] also show that abstract versions of set union, set and relation equality, and pre-image operations for $A_s$ and $A_r$ are defined in the same way as the ones over $\mathbf{2}^{\Sigma_V}$ and

$\mathbf{2}^{\Sigma_V \times \Sigma_V}$ except that the logical operations of conjunctions and disjunctions are interpreted over $\mathbf{4}$-valued logic. For example, if $X_1, X_2 \in \mathbf{4}^{\text{Mon}(\mathcal{P})}$ are two *abstract* sets that approximate sets $Q_1, Q_2 \in \mathbf{2}^{\Sigma_V}$, their abstract union $X_1 \cup_\alpha X_2 \triangleq \lambda z \cdot X_1(z) \vee X_2(z)$ approximates $Q_1 \cup Q_2$.

For the missing abstract relational operations *assume*$_\alpha$, asynchronous ($\vee_\alpha$), and sequential ($\circ_\alpha$) compositions, we define them similarly using the corresponding definitions from Section 4.2, e.g., if $r_1$ and $r_2$ are abstract relations, then their abstract asynchronous composition is $r_1 \vee_\alpha r_2 \triangleq \lambda s, t \cdot r_1(s,t) \vee r_2(s,t)$, where $\vee$ is interpreted over $\mathbf{4}$-valued logic. The soundness of these operations follows from the soundness of the abstract sets and relations associated with them [GWC06a].

Scope extension in concrete semantics is used to extend a relation to additional variables. That is, if $r$ is a relation of type $U \to V$, then $(U \to U)r$ is an extension of $r$ to variables in $U \setminus V$. In the abstract semantics, relations are defined over predicates; thus, abstract scope extension must extend a relation to additional predicates. To do this, we assume that the elementary abstract domain $\text{Mon}(\mathcal{P})$ corresponding to $U$ can be decomposed into two independent abstract domains: one for $V$ and the other – for $U \setminus V$, i.e., $\text{Mon}(\mathcal{P})$ is defined using predicates that either range only over $V$, or only over $U \setminus V$. Then, abstract scope extension $(\cdot \to \cdot)_\alpha$, defined as in Table 4.1, is a sound approximation of concrete scope extension.

In the context of our on-the-fly algorithms, the assumption on abstract scope extension means that predicates that are used to abstract valuations at a return location $l$ of a function call $\mathbf{x} := f(\mathbf{a})$ are either defined only over $\mathbf{x}$, or only over other variables in the scope of $l$. For example, predicates $x = 2$ and $y \leq 0$ can be used to abstract valuations at line 6 in the program $\texttt{EX}_1$, but predicate $x > y$ cannot. This is not a severe restriction in practice since a function can always be extended to accept additional parameters and return them without modification.

We have described how to derive abstract domains and the necessary abstract operations to obtain abstract reachability and non-termination analysis based on exact-approximating predicate abstraction. This approach can be applied to over- and under-approximating predicate

(a)

```
int g;

void main(){
 level_1();
 if (g<0){
   ERROR: ;
 }
 END: ;
}
```

```
void level_i(){
 int t = 0;
 g = -1 * g;
 if (g<=0){
   t = t+1;
 } else {
   level_i+1();
   g = -1 * g;
   level_i+1();
 }
 g = -1 * g;}
```

```
void level_n(){
 int t = 0;
 g = -1 * g;
 if (g<=0){
   t = t+1;
 } else {

   <stmt>

 }
 g = -1 * g;}
```

(b)

```
<stmt>:=
  g = -1 * g;
```

(c)

```
<stmt>:=
  level_n();
  g = -1 * g;
  level_n();
```

Figure 4.5: (a) The template for experiments. (b) `<stmt>` for template **T1(**$n$**)**. (c) `<stmt>` for **T2(**$n$**)**.

abstractions [BPR03] in a similar way with the difference that $A_s$ and $A_r$ are now classical sets and relations over the elementary abstract domain, and conjunctions and disjunctions are now interpreted over Boolean logic. To summarize, both over- and under-approximating predicate abstractions can be used to soundly abstract reachability and non-termination analysis. The choice depends on the desired algorithm. For example, over-approximation is necessary to establish unreachability, whereas under-approximation – to establish non-termination. Since exact predicate abstraction combines them, it can be used for both verification and refutation.

## 4.7 Experiments

The technique described in this chapter has been implemented in our symbolic software model checker YASM [GWC06b]. YASM is written in JAVA; it uses CVC Lite [BB04] to approximate program statements and CUDD [Som01] as a decision diagram engine. We have also extended our proof-based refinement approach [GC06] to handle natural semantics of functions. In

| | **T1($n$)** | **T2($n$)** | |
|---|---|---|---|
| $n$ | $EF\ (pc = \texttt{ERROR})$ (reach) | $EG\ (pc \neq \texttt{END})$ (non-terminate) | $\neg EF\ (pc = \texttt{ERROR})$ (unreach) |
| 20 | 6.5 | 4.9 | 4.3 |
| 50 | 11.7 | 8.9 | 6.3 |
| 100 | 20.3 | 20.3 | 11.1 |
| 200 | 36.7 | 25.2 | 27.6 |
| 300 | 47.6 | 34.4 | 42.1 |
| 400 | 68.1 | 43.2 | 64.5 |
| 500 | 105.2 | 60.6 | 86.6 |

Table 4.3: Experimental results: overall analysis time in seconds.

the rest of this section, we report on a preliminary evaluation of this implementation. All of the experiments have been conducted on a 2xP4Xeon-3.6GHz server, which demonstrate YASM's ability to analyze reachability and non-termination of recursive programs using exact-approximation. In summary:

1. We run YASM on template programs similar to those in the BEBOP and MOPED benchmarks. The experiment shows that the analysis time for *both* reachability and non-termination increases linearly w.r.t. the number of functions in a program.

2. We show that abstract analysis based on exact-approximation supports both verification and refutation.

3. We compare YASM with MOPED and VERA (BEBOP does not do non-termination), and show that YASM can prove non-termination of the original buggy `Quicksort` algorithm, whereas MOPED and VERA do not support non-termination analysis.

To evaluate the reachability algorithm, we have used the template program **T1($n$)** which is a variant of the one used for BEBOP in [BR00]. **T1($n$)** is the result of replacing `<stmt>` in the template shown in Fig. 4.5(a) with the statements in Fig. 4.5(b). It consists of a `main` function and $n$ sub-functions, where `main` calls `level_1`, and `level_i` calls `level_i+1`

twice if the global variable `g` is positive. Since `g` is not initialized, its initial value is arbitrary. Although this program has no recursion, inlining function calls increases its size exponentially, making the analysis infeasible for a sufficiently large $n$. We checked the reachability property *EF* $(pc = \texttt{ERROR})$ with values of $n$ ranging between 20 and 500, and measured the *overall* analysis time (including parsing, abstraction, model-checking, and refinement). The results are shown in the second column of Table 4.3. Since our technique analyzes each function separately, the analysis time increases linearly w.r.t. the number of functions ($n$), as expected. In all these cases, YASM was successful in proving reachability, and discovered predicates $g < 0$, $g > 0$ and $g \leq 0$. While the template **T1($n$)** is similar to the one used in [BR00], there is a fundamental difference: BEBOP assumes an over-approximating abstract semantics of Boolean programs and cannot *conclusively verify* that the `ERROR` label is reachable with these predicates. YASM uses exact-approximation which results in a conclusive analysis.

We also checked the template program **T2($n$)**, obtained by replacing `<stmt>` in the template in Fig. 4.5(a) with statements in Fig. 4.5(c). Non-termination and unreachability results are presented in the third and fourth columns of Table 4.3, respectively. As expected, the analysis time increases linearly with the number of functions.

For non-termination, we have also applied YASM to several examples inspired by [CPR06a], in particular, to programs `Ack` and `Shift`, shown in Fig. 4.6(a) and (b), respectively. YASM was able to automatically prove non-termination of `Ack` in 2.1 seconds and discovered predicates $y > 0$, $n > 0$, $x > 0$, $mx > 0$ and $my > 0$. Analysis of `Shift` took 1.9 seconds and yielded predicates $y < 0$, $x < 0$, $x > 3$, $x = 0$ and $x = 3$. Finally, we have compared YASM to MOPED [ES01] and VERA [ACEM05] on the buggy `Quicksort` example from [ES01] in Fig. 4.6(c), where `nondet()` represents non-deterministic choice. YASM has established non-termination of `Quicksort` in 10 seconds, finding 7 predicates. Note that both MOPED and VERA only apply to programs with finite data domain, and the analysis in [ACEM05] and [ES01] had to restrict the number of bits used by each variable, while YASM did not need any such restriction.

(a)

```
void main (){
 int mx, my;
 ack (mx, my);
 END:; }

int ack (int x, int y){
 int r1,  n;
 if (x > 0) {
   if (y > 0) {
     y = y - 1;
     n = ack (x, y);
   } else { n = 1; }
   r1 = ack (x, n);
   return r1;
 } else {
   r1 = y + 1;
   return r1;
 }}
```

(b)

```
void main(){
 int x;
 foo(x);
 while(x!=0) {
   if (x<0) {
     x = -1 * x;
     x = x+2;
   } else {
     x = -1 * x;
     x = x+3;
   }}
 END: ;}

void foo (int y){
 y = -1 * y;
 if (y < 0) {
   foo (y);
 }}
```

(c)

```
void main (){
 int mleft, mright;
 quicksort (mleft, mright);
 END:;}

void quicksort (int left, int right){
 int lo, hi;
 if (left >= right) return;
 lo = left; hi = right;
 while (lo <= hi) {
   if (nondet()) {
     lo = lo+1;
   } else {
     if(lo!=left || hi!=right)
       hi = hi-1;
   }}
 quicksort (left, hi);
 quicksort (lo, right); }
```

Figure 4.6: Non-terminating programs: (a) `Ack`; (b) `Shift`; (c) Buggy `Quicksort`.

## 4.8 Related Work

Recursive programs have been studied widely in the context of interprocedural program analysis. In general, there are two main approaches for this analysis, *functional* approach [SP81, KS92] and *operational* approach [SP81]. The function approach uses natural semantics to summarize the computational effects of functions, which are applied at the locations of function calls to update program states along the execution paths. The operational approach adopts the operational program semantics that uses an unbounded call stack for recursive calls. In terms of interprocedural program analysis, our approach is *functional* since it uses natural semantics to handle function calls.

Most other model-checking approaches for recursive programs, (e.g., [BR00, ACEM05,

BCP06, PSW05]) are functional as well. The input to BEBOP [BR00] and VERA [ACEM05] is a finite recursive program, which is equivalent to a pushdown system. These tools apply *over-approximation* when analyzing recursive programs on infinite domains. The reachability analysis in them uses the RHS algorithm presented in [RHS95]. The RHS algorithms is developed based on the functional approach in [SP81, KS92], providing a forward on-the-fly summarization of functions using graph reachability techniques. Our reachability algorithm can be seen as a pre-image-based (backward) variant of the RHS algorithm. Function summary is also used in [BCP06, PSW05] to prove termination of recursive programs, where only over-approximation is considered, and computation of function summary is not driven by property analysis.

In the operational approach, MOPED [ES01] is developed based on a symbolic approach to reachability analysis of pushdown systems [BEM97, EHRS00]. This approach relies on the result that the set of reachable states in a pushdown system is a regular language. Therefore, instead of computing the reachable states directly, which may not converge, the approach defines a procedure to construct a finite automata that recognizes these states, which is guaranteed to terminate. MOPED accepts a finite recursive program as input, and also uses over-approximation for infinite programs.

Both MOPED [ES01] and VERA [ACEM05] can detect cycles in programs with *finite* data domains, which can be used for non-termination checking. When used for analysis of programs with infinite data domain, they assume an over-approximating semantics of finite abstractions of original programs. Note that an ability to detect non-termination of over-approximating boolean programs is of limited utility since over-approximation often introduces *spurious* non-terminating computations. Thus, non-termination of an over-approximation says nothing about non-termination of the concrete program[1]. It is unclear how to combine MOPED and VERA with exact-approximation, whereas it is quite natural in our approach, resulting in both sound

[1]In [CH08], Charlton and Huth showed that using only over-approximating information they can prove reachability in limited cases by exploiting the seriality and partial determinism of programs. It is possible to apply this technique to prove non-termination in those cases as well.

verification and refutation of the property over programs with *infinite* data domain.

In [JS04], Jeannet and Serwe propose abstract analysis of recursive programs by different abstractions of the call stack, which provides a method to combine the function and operational approaches. Their method can be parameterized by an arbitrary data abstraction. However, the authors restrict their attention to reachability (i.e., invariance) properties, and do not report on an implementation.

The stack-free semantics we proposed in this chapter allows us to analyze stack-independent properties of recursive programs including reachability and non-termination. More general properties of recursive programs require inspection of the stack. A decidable class of such properties can be expressed by the temporal logic CaRet [AEM04] that allows for matching of calls and returns of procedures. Properties expressible in CaRet can be analyzed using *visibly pushdown automata* (VPA) [AM04]. VPA are a special class of pushdown automata where the set of input symbols is partitioned into calls, returns, and local symbols, and the push and pop operations on the stack are determined by the type of the input symbol. Exposing the matching structure of calls and returns makes VPA an appropriate model for algorithmic verification of recursive programs with respect to properties via stack inspection. We leave investigation of combining exact-approximation and VPA for future work.

## 4.9 Conclusion

In this chapter, we have presented a technique for analyzing reachability and non-termination properties of recursive programs. The technique is based on a stack-free mixed program semantics that eliminates call stacks while preserving stack-independent properties. We showed how to compute only the necessary part of function summary during the analysis, leading to on-the-fly algorithms for analysis of reachability and non-termination of finite programs. We then used the framework of abstract interpretation to combine our algorithms with data abstractions, making them applicable to programs with infinite data domains as well. We have implemented a combination of this approach with exact predicate abstraction in YASM [GWC06b] which

supports both verification and refutation of properties. Our experiments indicate that YASM scales to programs with a large number of functions and is able to analyze reachability and non-termination of non-trivial examples.

Our interest in non-termination is motivated by the work on *termination* (e.g., [CPR06a]). A termination tool typically can prove termination of programs, but requires manual inspection of non-terminating paths. We view our approach as complementary to that. As illustrated by our experiments, YASM can prove non-termination of non-trivial programs. In the future, we plan to investigate how the strengths of the two approaches can be combined together for better analysis. The refinement strategies that are currently implemented in YASM were originally developed for reachability analysis only. While they were sufficient for our non-termination experiments, we would like to investigate strategies in the future that are specifically tailored to the non-termination analysis.

# Chapter 5

# Analysis of Partial Modeling Formalisms

Partial modeling formalisms support abstract model checking of complex temporal properties by combining both over- and under-approximating abstractions into a single model. In this chapter, we investigate three families of these modeling formalisms represented by *Kripke Modal Transition Systems*, *Mixed Transition Systems*, and *Generalized Kripke Modal Transition Systems*. Following the two fundamental ways of using these partial transition systems for abstracting concrete programs and for temporal logic model checking, we study the connection between semantic and logical consistency of partial transition systems, compare expressive power of the three families of formalisms, and discuss the precision of model checking over them.

## 5.1 Introduction

Partial models play a fundamental role in exact-approximation frameworks. A temporal property is interpreted over a partial model in one of four ways: *true* or *false*, if the partial model is precise enough to prove or disprove the property, *unknown*, if the model is imprecise, and *inconsistent* otherwise. The modeling formalism of partial models, i.e., *partial transition systems* (Page 31), usually have two types of transitions, *may* and *must*, representing *possible* (or over-approximating), and *necessary* (or under-approximating) behaviours, respectively. There

are three families of partial transition systems identified in the literature, represented by *Kripke Modal Transition Systems* (KMTSs) [HJS01], *Mixed Transition Systems* (MixTSs) [DGG97], and *Generalized KMTSs* (GKMTSs) [SG04] (Definition 2.9), respectively:

1. KMTSs are equivalent to $\mathbf{3}$-*valued Transition Systems* [CDEG03], *Modal Transition Systems* [LT88], and *Partial Kripke Structures* [BG00]. KMTSs require that every *must* transition is also a *may* transition. They were introduced as computational models for partial specifications of reactive systems [LT88] and then adapted for model checking [BG00, HJS01, CDEG03].

2. MixTSs [DGG97] are equivalent to $\mathbf{4}$-*valued Transition Systems* [GWC06a]. MixTSs extend KMTSs by allowing *must only* transitions (i.e., transitions that are *must* but not *may*). MixTSs were introduced in [DGG97] as abstract models for $L_\mu$, and have been used for predicate abstraction and software model checking in [GC06].

3. GKMTSs [SG04] are equivalent to *Abstract Transition Systems* [dAGJ04] and *Disjunctive Modal Transition Systems* [Lar91]. GKMTSs extend MixTSs by allowing *must hyper-transitions*, (i.e., transitions into sets of states).

In this chapter, we study these formalisms from two points of view: a semantic one, using partial transition systems as objects for abstracting concrete programs, and a logical one, using partial transition systems for temporal logic model checking. A partial transition system is *semantically consistent* if it abstracts at least one concrete program. A partial transition system is *logically consistent* if it gives consistent interpretation to all temporal logic formulas. For semantic consistency, note that we investigate partial transition systems from the perspective of abstract model checking in this chapter, where a partial transition system and its concrete refinement are related through the soundness relation of abstract and concrete states. The notion of semantic consistency in this setting (formally defined in Section 5.4) is slightly different from the notion of *implementability* where partial transition systems are used as specifications of a system's behavior. A discussion of this difference is given in Section 5.9. Specifically, in

this chapter we first study the connection between semantic and logical consistency of partial transition systems. We then compare the expressive power of the formalisms, i.e. what abstractions can be captured using them. Finally, we discuss the analysis power of these formalisms, i.e., the cost and precision of model checking. This chapter includes the following technical contributions:

1. We define *monotone* partial transition systems, and show that they are as expressive as their regular counterparts: for any partial transition system there exists a monotone one such that both of them approximate the same set of concrete systems. The monotonicity condition ensures that all information that can be derived from existing *may* and *must* transitions is made explicit in the transition system, which allows for more effective local reasoning about partial transition systems.

2. We show that while in general semantic and logical consistency of partial transition systems are not equivalent, they are equivalent over monotone ones. Thus, for every partial transition system, there is an equivalent monotone one where semantic and logical consistency coincide. For monotone transition systems, we also define a structural condition that captures both notions of consistency.

3. We show that despite the structural difference, the three families of partial modeling formalisms, KMTSs, MixTSs and GKMTSs, are equally expressive. That is, for any partial transition system from one formalism, there exists a partial transition system in the other such that the two partial transition systems approximate the same set of concrete systems . We prove the equivalence of these formalisms by defining semantics-preserving translations from GKMTSs to MixTSs, and from MixTSs to KMTSs. Our results show that both GKMTSs and KMTSs can be converted to semantically equivalent MixTSs of equal or (possibly exponentially) smaller size.

4. We show that under the traditional inductive semantics of temporal logic, which is referred to as *standard*, a GKMTS can prove/disprove more properties than a MixTS ob-

tained by semantics-preserving translation. However, directly applying symbolic model checking over GKMTSs has been hampered by the difficulty of encoding hyper-transitions symbolically. We then develop a *reduced* inductive semantics that is more precise than the standard one. We show that GKMTSs and MixTSs are equivalent with respect to the reduced semantics, and give a symbolic model checking procedure for it. We apply this algorithm to MixTSs constructed using predicate abstraction, and evaluate it empirically against an algorithm for the standard semantics.

The rest of this chapter is organized as follows. Section 5.2 presents preliminaries and fixes the notation used in this chapter. In Section 5.3, we define *monotone* partial transition systems and show that they are as expressive as their regular counterparts. In Section 5.4, we investigate semantic and logical consistency of partial transition systems. In Section 5.5, we prove that KMTSs, MixTSs and GKMTSs are equally expressive by developing semantics-preserving translations from GKMTSs to MixTSs, and from MixTSs to KMTSs. In Section 5.6, we introduce *reduced* inductive semantics for $L_\mu$. In Section 5.7, we present a symbolic model checking algorithm with respect to the reduced semantics in the context of predicate abstraction. We report on our experience with this algorithm in Section 5.8. We discuss related work in Section 5.9 and conclude this chapter in Section 5.10.

## 5.2 Preliminaries

*Convention*. In this chapter, we use the following naming convention. Roman capital letters denote transition systems (TSs), which are built out of states and transitions: $M$ for a MixTS, $K$ for a KMTS, $G$ for a GKMTS, and $B$ for a BTS. Subscripts indicate a particular transition system. For example, $M_1$ is a MixTS (see Figure 5.1), whereas $G_2$ is a GKMTS (see Figure 5.2). Script capital letters denote models, which extend transition systems with interpretations of atomic propositions: $\mathcal{M}$ for a MixTS model, $\mathcal{K}$ for a KMTS model, $\mathcal{G}$ for a GKMTS model, and $\mathcal{B}$ for a BTS (or a concrete) model. We use subscripts to indicate a model corresponding

to a particular transition system. For example, $\mathcal{M}_1$ is a model whose underlying transition system is the MixTS $M_1$ (see Figure 5.1). The letter $L$ is used exclusively to indicate a labeling function of a model.

*Concrete and Abstract Statespace*. Recall that for a concrete statespace $C$, an *abstract* statespace approximating $C$ is a set of states $S$ equipped with a *soundness* relation $\rho : C \times S$, a *concretization* function $\gamma(s) \triangleq \{c \mid (c, s) \in \rho\}$. Abstract states are related with each other through the concrete states approximated by them. In this chapter, we investigate partial models based on the exploration of approximation abilities of abstract states. To this end, we assume that $S$ is equipped with an approximation ordering $\preceq_a$ s.t. $s \preceq_a t \Leftrightarrow \gamma(s) \supseteq \gamma(t)$. That is, $s \preceq_a t$ if $s$ is *less precise* (more approximate) than $t$. Following [CC92], we require that $\preceq_a$ be a partial order, that is, there are no redundant abstract states that approximate the same set of concrete states. We also require that there exists an abstract state that gives the most precise approximation of a concrete state, i.e., $\forall c \in C \cdot \exists s \in S \cdot (\rho(c, s) \wedge \forall s' \in S \cdot \rho(c, s') \Rightarrow \gamma(s') \supseteq \gamma(s))$ . We use an *abstraction* function $\alpha : C \to S$ to map each concrete element to its best approximation. The image of $\alpha$ is denoted by $\alpha[S] \triangleq \{\alpha(c) \mid c \in C\}$. We use the notation $\langle C, \rho, \gamma, \alpha, S\rangle$ to denote that a concrete statespace $C$ is abstracted by $S$ with $\rho$, $\gamma$, and $\alpha$ as defined above.

An abstract state $s \in S$ is *consistent* iff $\gamma(s) \neq \emptyset$. We require that any state labeling function $L$ over $S$ is *locally consistent*, i.e., for any consistent abstract state $s$ and proposition $p$, at most one of $p$ and $\neg p$ belongs to $L(s)$. We also require that any state labeling function $L$ over $S$ is *monotone* w.r.t. $\preceq_a$: $s_1 \preceq_a s_2 \Rightarrow L(s_1) \subseteq L(s_2)$.

*Predicate Abstraction*. Let $n$ be a natural number, and $P = \{p_1, \ldots, p_n\}$ be a set of quantifier-free first-order boolean predicates. Recall that a *monomial* is a conjunction of literals of $P$, and a *minterm* is a monomial in which each variable $p_i$ appears exactly once (either positively or negatively). We write $\text{Mon}(P)$ and $\text{MT}(P)$ for the set of all monomials and minterms of $P$, respectively. The set $\text{Mon}(P)$ is the domain of predicate abstraction. The soundness relation $\rho_P$ is defined s.t. $(c, s) \in \rho_P$ iff $c \models s$, i.e., $c$ satisfies all literals in $s$; the abstraction $\alpha_P(c) \triangleq (\bigwedge_{c \models p_i} p_i) \wedge (\bigwedge_{c \not\models p_i} \neg p_i)$; $\alpha_P[\text{Mon}(P)] = \text{MT}(P)$; and the approximation ordering

is reverse implication, $s \preceq_a t$ iff $s \Leftarrow t$.

*Semantics over Partial Models.* This chapter discusses several semantics of $L_\mu$. Let $\mathcal{M}$ be a partial model over an abstract statespace $S$, and $\varphi$ be an $L_\mu$ formula. We refer the traditional inductive semantics $L_\mu$ over partial models (Definition 2.10) as the *Standard Inductive Semantics* (SIS), and use a subscript $i$ to indicate this, e.g., the SIS of $\varphi$ over $\mathcal{M}$ is denoted by $\|\varphi\|_i^{\mathcal{M}}$. Recall that $\mathbb{C}[\mathcal{M}]$ is the set of all concrete refinements of $\mathcal{M}$ (Page 35). Intuitively, $\mathbb{C}[\mathcal{M}]$ is the semantic meaning of $\mathcal{M}$. A *thorough* semantics of $\varphi$ over $\mathcal{M}$, denoted $\|\varphi\|_t^{\mathcal{M}}$, is defined with repect to respect to the semantic meaning of $\mathcal{M}$.

**Definition 5.1** (Thorough Semantics)**.** *[BG00] Let $\mathcal{M}$ be a partial model over an abstract statespace $S$. The* thorough semantics *of an $L_\mu$ formula $\varphi$ over $\mathcal{M}$ is defined as $\|\varphi\|_t^{\mathcal{M}} = \langle U, O\rangle$, where*

$$U = \{a \in S \mid \forall \mathcal{B} \in \mathbb{C}[\mathcal{M}] \cdot \gamma(a) \subseteq \mathsf{U}(\|\varphi\|_i^{\mathcal{B}})\}$$

$$O = \{a \in S \mid \exists \mathcal{B} \in \mathbb{C}[\mathcal{M}] \cdot (\gamma(a) \cap \mathsf{O}(\|\varphi\|_i^{\mathcal{B}})) \neq \emptyset\}$$

In order to compare different semantics of $L_\mu$, we introduce two ordering relations on the space $2^S \times 2^S$.

**Definition 5.2** (Information and Semantics Orderings)**.** *Let $S$ be an abstract statespace. Let $e_1 = \langle U_1, O_1\rangle$ and $e_2 = \langle U_2, O_2\rangle$ be two elements in $2^S \times 2^S$. $e_1$ is* less informative *than $e_2$, written $e_1 \preceq_i e_2$, if and only if*

$$U_1 \subseteq U_2 \quad \textit{and} \quad O_2 \subseteq O_1$$

*$e_1$ is* semantically less precise *than $e_2$, written $e_1 \preceq_a e_2$, if and only if*

$$\gamma(U_1) \subseteq \gamma(U_2) \quad \textit{and} \quad \gamma(\overline{O_1}) \subseteq \gamma(\overline{O_2})$$

Note that we use the same notation $\preceq_a$ to denote the precision orderings, defined w.r.t. concretization, for both the elements in $S$ and the ones in $2^S \times 2^S$.

*Semantic Equivalence and Expressive Equivalence.* We define semantic equivalence between partial models (TSs) and expressive equivalence between modeling formalisms as follows.

**Definition 5.3** (Semantic Equivalence)**.** *Two partial models $\mathcal{M}$ and $\mathcal{M}'$ are* semantically equivalent*, if and only if they have the same set of concrete refinements, i.e.,* $\mathbb{C}[\mathcal{M}] = \mathbb{C}[\mathcal{M}']$*. Two partial transition systems, $M$ and $M'$, are* semantically equivalent*, if and only if* $\mathbb{C}[M] = \mathbb{C}[M']$.

**Definition 5.4** (Expressive Equivalence)**.** *Two partial modeling formalisms are* expressively equivalent *if and only if for every TS $M$ from one formalism, there exists a TS $M'$ from the other, s.t. $M$ and $M'$ are semantically equivalent.*

## 5.3 Monotone Partial Transition Systems

In this section, we define *monotone* partial TSs. We show that monotone partial TSs are expressively equivalent (Definition 5.4) to their regular counterparts: for any partial TS there exists an equivalent monotone one, i.e., they approximate the same set of concrete systems. The monotonicity condition simply ensures that all information that can be derived from existing *may* and *must* transitions is made explicit in the TS. As we show in later sections, this condition allows us to perform local reasoning of partial TSs more effectively.

For simplicity, we present the results w.r.t. MixTSs. They can be easily adapted to GKMTSs as well.

**Definition 5.5.** *A MixTS $M = \langle S, R^{\text{may}}, R^{\text{must}} \rangle$ is* monotone *iff*

*(a)* $\forall s, t_1, t_2 \in S \cdot t_2 \preceq_a t_1 \Rightarrow ((s, t_2) \in R^{\text{may}} \Rightarrow (s, t_1) \in R^{\text{may}}) \wedge$
$((s, t_1) \in R^{\text{must}} \Rightarrow (s, t_2) \in R^{\text{must}})$

*(b)* $\forall s_1, s_2, t \in S \cdot s_1 \preceq_a s_2 \Rightarrow ((s_2, t) \in R^{\text{may}} \Rightarrow (s_1, t) \in R^{\text{may}}) \wedge$
$((s_1, t) \in R^{\text{must}} \Rightarrow (s_2, t) \in R^{\text{must}})$

A model $\mathcal{M} = \langle M, L \rangle$ is *monotone* iff its MixTS component $M$ is monotone.

Intuitively, a transition system is monotone iff the information captured by its transition relation is monotone with respect to the approximation ordering $\preceq_a$ of its states. For example,

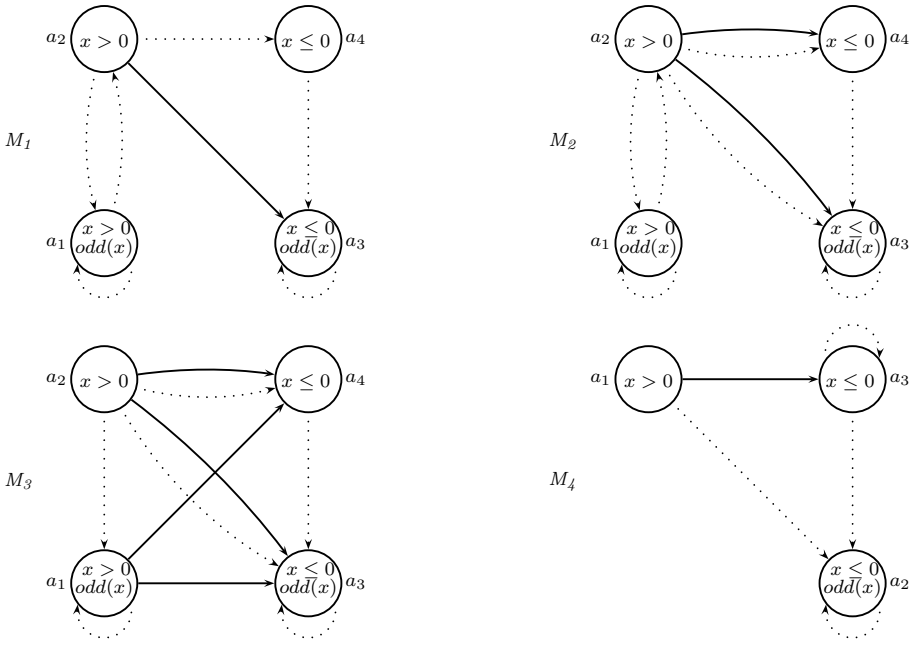


Figure 5.1: Four MixTSs: $M_1$, $M_2$, $M_3$, and $M_4$, where $M_1$ and $M_4$ are monotone. Solid and dashed lines represent *must* and *may* transitions, respectively.

let $M$ be a transition system, $s_1$ and $s_2$ be two states of $M$ such that $s_1 \preceq_a s_2$. (1) Suppose there is a *may* transition from $s_2$ to some other state $t$. The meaning of this transition is that any system that refines $M$ can have a transition from a state in $\gamma(s_2)$ to a state in $\gamma(t)$. Recall that we assumed that $s_1 \preceq_a s_2$; hence, $\gamma(s_1) \supseteq \gamma(s_2)$. Thus, the same behavior is allowed from the states in $\gamma(s_1)$. For $M$ to be monotone with this information, it must have a *may* transition from $s_1$ to $t$. (2) Similarly, suppose there is a *must* transition from $s_1$ to some other state $t$. Then, every state in $\gamma(s_1)$ must have a transition to some state in $\gamma(t)$. Since $\gamma(s_1) \supseteq \gamma(s_2)$, the same is true for the states in $\gamma(s_2)$. Therefore, for $M$ to be monotone with this information, it should have a *must* transition from $s_2$ to $t$.

For example, the MixTS $M_3$ shown in Figure 5.1 is monotone; the MixTS $M_1$ in the same figure is not monotone: for the states $a_1$ and $a_2$, where $a_2$ is less precise than $a_1$, we have $a_2 \xrightarrow{\text{must}} a_3$, but there is no *must* transition from $a_1$ to $a_3$, and for the states $a_3$ and $a_4$, where $a_4$

is less precise than $a_3$, we have $a_2 \xrightarrow{\text{may}} a_4$, but there is no *may* transition from $a_2$ to $a_3$.

In the rest of this section, we show that every partial TS (or model) can be translated into a semantically equivalent (Definition 5.3) monotone one. We first define such translation for MixTSs. The translation consists of two steps, DSTT (destination translation) and SRCT (source translation), which produce a monotone transition system preserving the behaviors of the original one.

**Definition 5.6** (Translation DSTT)**.** *Let $M = \langle S, R_M^{\text{may}}, R_M^{\text{must}} \rangle$ be a MixTS. The result of translation $\text{DSTT}(G)$ is a MixTS $N = \langle S, R_N^{\text{may}}, R_N^{\text{must}} \rangle$, such that*

$$R_N^{\text{may}} \triangleq \{(a,b) \in S \times S \mid \exists b' \in S \cdot b' \preceq_a b \wedge (a,b') \in R_M^{\text{may}}\}$$
$$R_N^{\text{must}} \triangleq \{(a,b) \in S \times S \mid \exists b' \in S \cdot b \preceq_a b' \wedge (a,b') \in R_M^{\text{must}}\}$$

The translation DSTT checks the transition from each state in its input TS and adds missing transitions derived from the approximation ordering over abstract states, ensuring that the result satisfies the condition (a) of Definition 5.5. A *may* transition is added between states $a$ and $b$ in the resulting TS if the source TS has a *may* transition between states $a$ and some state $b'$ that is less precise than $b$. Similarly, a *must* transition between states $a$ and $b$ is added to the resulting TS if the source TS has a *must* transition between $a$ and some state $b'$ that is more precise than $b$. For example, for transition systems in Figure 5.1, the translation $\text{DSTT}(M_1)$ results in the MixTS $M_2$: since $a_4$ is less precise than $a_3$ and there exists a *may* transition $a_2 \xrightarrow{\text{may}} a_4$ in $M_1$, $M_2$ contains a *may* transition $a_2 \xrightarrow{\text{may}} a_3$; furthermore, since there exists a *must* transition $a_2 \xrightarrow{\text{must}} a_3$ in $M_1$, $M_2$ contains a *must* transition $a_2 \xrightarrow{\text{must}} a_4$.

**Lemma 5.7.** *Let $M$ be a MixTS. The translation $\text{DSTT}(M)$ results in a MixTS which satisfies condition (a) of Definition 5.5.*

**Definition 5.8** (Translation SRCT)**.** *Let $M = \langle S, R_M^{\text{may}}, R_M^{\text{must}} \rangle$ be a MixTS. The result of the*

*translation* $\text{SRCT}(G)$ *is a MixTS* $N = \langle S, R_N^{\text{may}}, R_N^{\text{must}} \rangle$*, such that*

$$R_N^{\text{may}} \triangleq \{(a,b) \in S \times S \mid \forall a' \in S \cdot a' \preceq_a a \Rightarrow (a',b) \in R_M^{\text{may}}\}$$

$$R_N^{\text{must}} \triangleq \{(a,b) \in S \times S \mid \exists a' \in S \cdot a \preceq_a a' \wedge (a',b) \in R_M^{\text{must}}\}$$

The translation SRCT ensures that its output, $N$, satisfies the condition (b) of Definition 5.5. It guarantees that the transitions from more precise states are more defined: for each state $a$, it has a *must* transition to a state $b$ in $N$ if a less precise state $a'$ already has a *must* transition to $b$ in its input, $M$; it has a *may* transition to $b$ in $N$ only when all the states that are less precise than it already have *may* transitions to $b$ in $M$. For example, $M_3$ in Figure 5.1 is the result of $\text{SRCT}(M_2)$: because $a_2$ is less precise than $a_1$ and there are *must* transitions $a_2 \xrightarrow{\text{must}} a_3$ and $a_2 \xrightarrow{\text{must}} a_4$ in $M_2$, two *must* transitions $a_1 \xrightarrow{\text{must}} a_3$ and $a_1 \xrightarrow{\text{must}} a_4$ are added to $M_3$; on the other hand, the *may* transition $a_1 \xrightarrow{\text{may}} a_2$ is removed from $M_3$ because $a_2$ has no *may* transition to $a_2$ in $M_2$.

**Lemma 5.9.** *Let* $M$ *be a MixTS. The translation* $\text{SRCT}(M)$ *results in a MixTS which satisfies condition (b) of Definition 5.5.*

We now define the monotone translation MONOT be the composition of the translations for source and destination states, i.e., $\text{MONOT} \triangleq \text{SRCT} \circ \text{DSTT}$. The following theorem shows that MONOT translates a MixTS into an equivalent monotone one.

**Theorem 5.10.** *Let* $M$ *be a MixTS. The translation* $\text{MONOT}(M)$ *results in a MixTS which is monotone and semantically equivalent to* $M$.

**Proof:**

(1) Let $N_1 = \text{DSTT}(M)$ and $N_2 = \text{SRCT}(N_1)$. According to Lemmas 5.7 and 5.9, $N_1$ and $N_2$ satisfy conditions (a) and (b) of Definition 5.5, respectively. To show that $\text{MONOT}(M)$ is monotone, we only need to show that $N_2$ also satisfies condition (a). Proof of this follows from the definition of SRCT.

(2) To prove that $M$ and $N_2$ are semantically equivalent, we show that any concrete BTS $B = \langle C, R \rangle$ refines $M$ iff it refines $N$. It is equivalent to showing that (i) the soundness relation

$\rho \subseteq C \times S$ is a mixed simulation between $B$ and $M$ iff it is a mixed simulation relation between $B$ and $N_1$; and (ii) $\rho$ is a mixed simulation between $B$ and $N_1$ iff it is a mixed simulation relation between $B$ and $N_2$. This follows from the definitions of DSTT and SRCT. □

The translation MONOT can also be used to convert a partial model into its monotone equivalent one.

**Corollary 5.11.** *Let $\mathcal{M} = \langle M, L_M \rangle$ be a MixTS model, $N = \text{MONOT}(M)$, and $L_N = L_M$. Then the model $\mathcal{N} = \langle N, L_N \rangle$ is monotone and semantically equivalent to $\mathcal{M}$.*

In this section, we have shown that monotone partial TSs are as expressive as their "regular" counterparts. The monotone conditions make hidden transitions explicit, allowing us to do better local reasoning about partial TSs, which is illustrated in the following sections.

## 5.4 Consistency

There are two alternatives for defining consistency of a partial TS: either based on satisfaction of temporal logic formulas (*logical consistency*), or based on possible concrete refinements (*semantic consistency*). While semantic consistency implies logical consistency, the converse is not true. There exists a logically consistent TS that has no concrete refinements. In this section, we investigate these two notions, show when they coincide, and provide a new structural condition which is necessary and sufficient to ensure that a TS is consistent.

### 5.4.1 Logical and Semantic Consistency for Consistent Statespaces

Throughout this section, let $C$ be a concrete statespace, and $S$ be the corresponding abstract statespace such that $\langle C, \rho, \gamma, \alpha, S \rangle$. In addition, in this subsection, we assume that every state $a \in S$ is consistent, i.e., $\gamma(a) \neq \emptyset$. We extend our definitions to deal with inconsistent states in Section 5.4.2.

A model $\mathcal{M}$ is logically consistent if it gives a consistent interpretation, i.e., either *true*, *false*, or *unknown*, to every temporal formula. Formally,

**Definition 5.12.** *A model $\mathcal{M}$ is* logically consistent *iff for every $\varphi \in L_\mu$, $\mathsf{U}(\|\varphi\|_i) \subseteq \mathsf{O}(\|\varphi\|_i)$.*

Logical consistency naturally extends from models to transition systems: a transition system $M$ is logically consistent iff for any labeling function $L$ the model $\langle M, L\rangle$ is logically consistent.

A transition system $M$ is semantically consistent iff there exists at least one BTS that refines it:

**Definition 5.13.** *A transition system $M$ is* semantically consistent *iff $\mathbb{C}[M] \neq \emptyset$.*

Semantic consistency extends naturally from transition systems to models. A model $\mathcal{M} = \langle M, L\rangle$ is semantically consistent iff the transition system $M$ is semantically consistent. Because we require that the labeling function $L$ be monotone with respect to $\preceq_a$, this is equivalent to requiring that the model $\mathcal{M}$ has a consistent refinement.

Semantic consistency implies logical consistency:

**Theorem 5.14.** *Any semantically consistent transition system is also logically consistent.*

**Proof:**

Let $M$ be a consistent transition system. We show that $M$ is logically consistent by contradiction.

Assume $M$ is not logically consistent. Then, there exists a labeling function $L$ and a temporal formula $\varphi$ such that $\varphi$ is inconsistent in some state of the model $\mathcal{M} = \langle M, L\rangle$. Formally, there exists a state $a$ of $M$ such that $a$ is in $\mathsf{U}(\|\varphi\|_i^{\mathcal{M}}) \setminus \mathsf{O}(\|\varphi\|_i^{\mathcal{M}})$.

Let $\mathcal{B}$ be a concrete (BTS) model refining $\mathcal{M}$. Since $\mathcal{M}$ is semantically consistent, such $\mathcal{B}$ is guaranteed to exist. By Theorem 2.14, $\gamma(\mathsf{U}(\|\varphi\|_i^{\mathcal{M}})) \subseteq \mathsf{U}(\|\varphi\|_i^{\mathcal{B}})$, and $\gamma(\overline{\mathsf{O}(\|\varphi\|_i^{\mathcal{M}})}) \subseteq \overline{\mathsf{O}(\|\varphi\|_i^{\mathcal{B}})}$. Then, there exists a concrete state $c \in \gamma(a)$ s.t. $c \in \mathsf{U}(\|\varphi\|_i^{\mathcal{B}})$ and $c \in \overline{\mathsf{O}(\|\varphi\|_i^{\mathcal{B}})}$.

Since $\mathcal{B}$ is concrete, $\mathsf{U}(\|\varphi\|_i^{\mathcal{B}}) = \mathsf{O}(\|\varphi\|_i^{\mathcal{B}})$. Hence, $c \in \mathsf{U}(\|\varphi\|_i^{\mathcal{B}})$ and $c \in C \setminus \mathsf{U}(\|\varphi\|_i^{\mathcal{B}})$ — a contradiction. Thus, $\mathcal{M}$ is logically consistent. □

Interestingly, the converse of Theorem 5.14 is not true in general. We illustrate this on an example. Consider the MixTS $M_2$ in Figure 5.1. In $M_2$, every *must* transition is matched

by a *may* transition, i.e., $R^{\text{must}} \subseteq R^{\text{may}}$. Thus, by [HJS01, dAGJ04], it is logically consistent. However, $M_2$ is not semantically consistent as we show using a proof by contradiction. Assume there is a BTS $B$ that refines $M_2$. Let $c_1 : \langle x = 1 \rangle$ be a state of $B$; $c_1$ is approximated by both $a_1$ and $a_2$. Because $B$ refines $M_2$, and $M_2$ has a *must* transition $a_2 \xrightarrow{\text{must}} a_3$, $B$ has a transition from $c_1$ to a state approximated by $a_3$, say, $c_2 : \langle x = -1 \rangle$. Since $M_2$ approximates $B$, by the definition of mixed simulation (Definition 2.11), $a_1$ must have a *may* transition to a state that approximates $c_2$, i.e., either $a_3$ or $a_4$. There is no such *may* transition in $M_2$, contradicting the assumption. Thus, $M_2$ is not semantically consistent.

Below, we show that monotone MixTSs is a class of systems for which logical and semantic consistency coincide. Intuitively, the reason is that the approximation ordering, $\preceq_a$, of the statespace of monotone MixTSs is "pushed" down to its transitions. This gives rise to the following theorem:

**Theorem 5.15.** *Let $M$ be a monotone MixTS $(S, R^{\text{may}}, R^{\text{must}})$, and assume that every state in $S$ is consistent. Then, the following are equivalent:*

*(a) $M$ is semantically consistent (Definition 5.13),*

*(b) $M$ is logically consistent (Definition 5.12),*

*(c)* $\forall a, b_1 \in S \cdot a \xrightarrow{\text{must}} b_1 \Rightarrow \exists b_2 \in S \cdot b_1 \preceq_a b_2 \wedge a \xrightarrow{\text{may}} b_2$.

**Proof:**

We show that $(a) \Rightarrow (b)$, $(b) \Rightarrow (c)$, and $(c) \Rightarrow (a)$.

Part 1. $(a) \Rightarrow (b)$ The proof follows from Theorem 5.14.

Part 2. $(b) \Rightarrow (c)$ Let $a$ and $b_1$ be two states in $S$ such that $a \xrightarrow{\text{must}} b_1$ is a transition in $R^{\text{must}}$. We show that (i) there exists a labeling function $L$, and (ii) there exists a formula $\varphi$, such that $\varphi$ is consistent in the state $a$ of the model $\mathcal{M} = \langle M, L \rangle$ only if $M$ has a transition $a \xrightarrow{\text{may}} b_2$ for some state $b_2$ that is more precise than $b_1$.

(i) To define $L$, we partition the statespace $S$ into sets $S_1$, $S_2$, and $S_3$:

$$S_1 \triangleq \{s \in S \mid b_1 \preceq_a s\}$$
$$S_2 \triangleq \{s \in S \mid \exists t \in S_1 \cdot s \preceq_a t\} \setminus S_1$$
$$S_3 \triangleq S \setminus (S_1 \cup S_2)$$

$S_1$ is the set of all states that are more precise than $b_1$. $S_2$ is the set of all states that are not in $S_1$, but are less precise than some state in $S_1$. $S_3$ contains all states that are neither in $S_1$ nor $S_2$.

Let $AP = \{p\}$. $L$ is defined as follows:

$$L(s) \triangleq \begin{cases} \{p\} & \text{if } s \in S_1 \\ \{\} & \text{if } s \in S_2 \\ \{\neg p\} & \text{if } s \in S_3. \end{cases}$$

$L$ is consistent. We need to show that $L$ is monotone, i.e., if $s \preceq_a t$ then $L(s) \subseteq L(t)$. Let $s$ and $t$ be two states such that $s \preceq_a t$. Then, either $s$ and $t$ belong to the same partition or $s \in S_2$ and $t \in S_1 \cup S_3$. In both cases, monotonicity follows trivially.

(ii) We define $\varphi$ as the formula $\Diamond p$. Note that because of the must transition $a \xrightarrow{\text{must}} b_1$, $a$ is in $\mathsf{U}(\|\Diamond p\|_i^{\mathcal{M}})$. And, because $\mathcal{M}$ is logically consistent, $a \in \mathsf{O}(\|\Diamond p\|_i^{\mathcal{M}})$ as well. We use this fact to show existence of $b_2$, needed for condition (c) of the theorem.

$$
\begin{array}{ll}
 & a \xrightarrow{\text{must}} b_1 \\
\Rightarrow & \text{(by the definition of } L\text{, } \|p\|_i^{\mathcal{M}} = \langle S_1, S_1 \cup S_2 \rangle\text{)} \\
 & a \xrightarrow{\text{must}} b_1 \wedge b_1 \in \mathsf{U}(\|p\|_i^{\mathcal{M}}) \\
\Rightarrow & \text{(by SIS of } \Diamond p\text{)} \\
 & a \in \mathsf{U}(\|\Diamond p\|_i^{\mathcal{M}}) \\
\Rightarrow & \text{(since } \mathcal{M} \text{ is logically consistent, } \Diamond p \text{ is consistent at } a\text{)} \\
 & a \in \mathsf{O}(\|\Diamond p\|_i^{\mathcal{M}}) \\
\Rightarrow & \text{(by SIS of } \Diamond p\text{)} \\
 & \exists b_2 \in S_1 \cup S_2 \cdot a \xrightarrow{\text{may}} b_2 \\
\Rightarrow & \text{(logic)} \\
 & (\exists b_2 \in S_1 \cdot a \xrightarrow{\text{may}} b_2) \vee (\exists b_2 \in S_2 \cdot a \xrightarrow{\text{may}} b_2)
\end{array}
$$

In the first case, $b_2 \in S_1$. By definition of $S_1$, $b_1 \preceq_a b_2$. This fulfills condition (c) of the theorem.

In the second case, $b_2 \in S_2$.

$$
\begin{array}{ll}
 & \exists b_2 \in S_2 \cdot a \xrightarrow{\text{may}} b_2 \\
\Rightarrow & \text{(by the definition of } S_2\text{)} \\
 & \exists b_2 \in S_2 \cdot a \xrightarrow{\text{may}} b_2 \wedge \exists b' \in S_1 \cdot b_2 \preceq_a b' \\
\Rightarrow & \text{(by assumption, } M \text{ is monotone)} \\
 & \exists b' \in S_1 \cdot a \xrightarrow{\text{may}} b'
\end{array}
$$

Hence, $b'$ fulfills the condition (c) of the theorem.

Thus, if $M$ is logically consistent, then

$$\forall a, b_1 \in S \cdot a \xrightarrow{\text{must}} b_1 \Rightarrow \exists b_2 \in S \cdot b_1 \preceq_a b_2 \wedge a \xrightarrow{\text{may}} b_2 \,.$$

Part 3. $(c) \Rightarrow (a)$ The proof proceeds by constructing a concrete BTS $B$ that refines $M$. Let $C$ be a concrete statespace approximated by $S$. Let $\rho \subseteq C \times S$ be the corresponding soundness relation with the abstraction function $\alpha : C \to S$. Let $B$ be a BTS $\langle C, R \rangle$, where

$$R \triangleq \{(c, d) \in C \times C \mid \exists b \in S \cdot (\alpha(c), b) \in R^{\text{may}} \wedge (d, b) \in \rho\}$$

We show that $\rho$ is a mixed simulation relation between $M$ and $B$, i.e., $M \preceq_\rho B$. Let $c \in C$, and $a \in S$ be two states such that $(c, a) \in \rho$. Recall that this implies that $a \preceq_a \alpha(c)$.

First, we show that $\rho$ satisfies condition (a) of Definition 2.11. Let $b$ be a state in $M$ such that there is a must transition $a \xrightarrow{\text{must}} b$. Then,

$$
\begin{aligned}
& (a, b) \in R^{\text{must}} \\
\Rightarrow \quad & \text{(by assumption, } M \text{ is monotone and } a \preceq_a \alpha(c)\text{)} \\
& (\alpha(c), b) \in R^{\text{must}} \\
\Rightarrow \quad & \text{(by assumption of condition (c) of the theorem)} \\
& \exists b' \in S \cdot b \preceq_a b' \wedge (\alpha(c), b') \in R^{\text{may}} \\
\Rightarrow \quad & \text{(by the definition of } B\text{)} \\
& \exists b' \in S \cdot \exists d \in C \cdot b \preceq_a b' \wedge (c, d) \in R \wedge (d, b') \in \rho \\
\Rightarrow \quad & \text{(by monotonicity of } \rho\text{)} \\
& \exists d \in C \cdot (c, d) \in R \wedge (d, b) \in \rho
\end{aligned}
$$

Second, we show that $\rho$ satisfies condition (b) of Definition 2.11. Let $d$ be a state in $B$ such that there is a transition $c \rightarrow d$. Then,

$$
\begin{aligned}
& (c, d) \in R \\
\Rightarrow \quad & \text{(by the definition of } B\text{)} \\
& \exists b \in S \cdot (\alpha(c), b) \in R^{\text{may}} \wedge (d, b) \in \rho \\
\Rightarrow \quad & \text{(by assumption, } M \text{ is monotone, and } a \preceq_a \alpha(c)\text{)} \\
& \exists b \in S \cdot (a, b) \in R^{\text{may}} \wedge (d, b) \in \rho
\end{aligned}
$$

Thus, we have constructed a BTS $B$ and produced $\rho$ which is a mixed simulation between $M$ and $B$. Hence, $M$ is semantically consistent. □

In the rest of this section, we highlight some of the consequences of Theorem 5.15. First, note that Theorem 5.15 does not extend to monotone partial models! For example, consider a monotone MixTS $M_3$ in Figure 5.1. By Theorem 5.15, $M_3$ is inconsistent: there is a *must* transition $a_1 \xrightarrow{\text{must}} a_3$, but no *may* transition $a_1 \xrightarrow{\text{may}} a$ to a state $a$ such that $a_3 \preceq_a a$. Let $p$ be an atomic proposition: "$x$ is a prime number". Let $L_3$ be a labeling function: for any state $s$ of $M_3$, $L_3(s) = \emptyset$. That is, $p$ is unknown at all the states in $M_3$. The model $\mathcal{M}_3 = \langle M_3, L_3 \rangle$

is semantically inconsistent. But, $\mathcal{M}_3$ is logically consistent – there does not exists a formula $\varphi$ such that $\mathsf{U}(\|\varphi\|_i^{\mathcal{M}_3}) \setminus \mathsf{O}(\|\varphi\|_i^{\mathcal{M}_3}) \neq \emptyset$. Intuitively, the labeling function $L_3$ is too coarse to detect the inconsistency logically.

Second, part (c) of Theorem 5.15 gives a necessary and sufficient structural condition for a monotone MixTS to be consistent. Let us compare it with the previously known condition to ensure logical consistency [HJS01, dAGJ04]:

$$\forall a, b \in S \cdot (a \xrightarrow{\text{must}} b) \Rightarrow (a \xrightarrow{\text{may}} b)\,.$$

Our new condition is weaker. Thus, there is a consistent monotone MixTS which has a *must* transition that is not a *may* transition. For example, consider the MixTS $M_4$ in Figure 5.1. Note that the *must* transition $a_1 \xrightarrow{\text{must}} a_3$ is not matched by any *may* transition. Let $B$ be a BTS $(\mathbb{Z}, R)$, where $\mathbb{Z}$ is the set of integers, and $R$ is defined as follows:

$$R \triangleq \{(x, x') \in \mathbb{Z} \times \mathbb{Z} \mid (x > 0 \wedge x' = -1) \vee (x \leq 0 \wedge x' = x - 2)\}\,.$$

$B$ refines $M_4$. Thus, by definition, $M_4$ is semantically consistent. By Theorem 5.14, $M_4$ is logically consistent as well.

Third, by definition, a KMTS always satisfies condition (c) of Theorem 5.15. Existing work on KMTSs [HJS01] often implicitly assumes that the abstract domain is flat (i.e., the abstract ordering $\preceq_a$ on $S$ is discrete). This assumption ensures that every KMTS is monotone. For such TSs, semantic and logical consistency coincide. Yet the assumption about the flatness of the abstract domain is too restrictive. For example, it is not true in a typical application of predicate abstraction (e.g., in [GC06]). By looking at a wider range of transition systems and considering not only flat abstract domains, we have uncovered the subtle but important differences between logical and semantic consistency.

### 5.4.2 Logical and Semantic Consistency for Arbitrary Statespaces

In Section 5.4.1, we have assumed that the abstract statespace $S$ does not contain any inconsistent states. That is, if $a$ is in $S$, then its concretization $\gamma(a)$ is non-empty. We now lift this

restriction, i.e., we aim to redefine (i) logical consistency, (ii) semantic consistency and (iii) the structural condition of Theorem 5.15.

(i) An inconsistent state does not abstract any concrete states, so a temporal formula can have any value in that state, including being both satisfied and refuted. We thus strengthen Definition 5.12 as follows:

**Definition 5.16.** *A model $\mathcal{M}$ is* logically consistent *iff for every $\varphi \in L_\mu$*

$$a \in (\mathsf{U}(\|\varphi\|_i) \setminus \mathsf{O}(\|\varphi\|_i)) \Rightarrow \gamma(a) = \emptyset\,.$$

If the abstract statespace $S$ has no inconsistent states, this definition reduces to Definition 5.12.

(ii) Semantic consistency does not need a new definition: a transition system is *semantically consistent* iff there is a BTS that refines it, independently of the structure of the abstract statespace.

(iii) We now need to strengthen the structural condition to match the new Definition 5.16. Specifically, we add the requirement that every *must* transition from a *consistent* state must be matched by a *may* transition into a *consistent* state.

Under these conditions, we now restate Theorem 5.15 to handle inconsistent states:

**Theorem 5.17.** *Let $M = \langle S, R^{\text{must}}, R^{\text{may}}\rangle$ be a monotone MixTS. Then, the following are equivalent:*

*(a) $M$ is semantically consistent (Definition 5.13),*

*(b) $M$ is logically consistent (Definition 5.16),*

*(c)*
$$\begin{aligned}&\forall a, b_1 \in S \cdot (\gamma(a) \neq \emptyset \wedge a \xrightarrow{\text{must}} b_1) \Rightarrow \\ &\qquad (\exists b_2 \in S \cdot b_1 \preceq_a b_2 \wedge \gamma(b_2) \neq \emptyset \wedge a \xrightarrow{\text{may}} b_2).\end{aligned}$$

**Proof:**

The proof is similar to that of Theorem 5.15. □

In this section, we have investigated the connection between semantic and logical consistency of partial models. Semantic consistency is important for when partial TSs are used as objects for abstracting concrete TSs. Logical consistency is important when partial models are used to interpret temporal logic formulas. In the following two sections, we first compare the expressive power of the different TS formalisms, i.e., what can be modeled and what abstractions can be captured (Section 5.5). Second, we compare the analyzability of the formalisms, i.e., the cost and precision of model checking (Section 5.6).

## 5.5 Expressiveness

We show that GKMTSs, MixTSs, and KMTSs are expressively equivalent (Definition 5.4). The equivalence of the three formalisms is proved by defining semantics-preserving translations from GKMTSs to MixTSs, and from MixTSs to KMTSs. Since GKMTSs syntactically subsume KMTSs, the translation from KMTSs to GKMTSs is basically an identity map.

### 5.5.1 GTOM: Translation from GKMTSs to MixTSs

We present the translation GTOM that converts a GKMTS into a semantically equivalent MixTS. First, we illustrate the translation on a GKMTS $G_1$ in Figure 5.2. $G_1$ is not a MixTS because of *must* hyper-transition $a_1 \xrightarrow{\text{must}} \{a_2, a_3\}$. This transition ensures that in every concrete BTS refining $G_1$, all states in $\gamma(a_1)$, i.e., those satisfying $(x \leq 0 \wedge even(x))$, must have a transition to a state in $\gamma(\{a_2, a_3\})$, i.e., satisfying $(x > 0)$. No single state of $G_1$ represents $(x > 0)$. Thus, this requirement can only be captured either by a hyper transition (as done in $G_1$), or by extending $G_1$ with a new state, say $a_5$, such that $\gamma(a_5) = (x > 0)$. In the latter case, the *must* hyper-transition $a_1 \xrightarrow{\text{must}} \{a_2, a_3\}$ can be replaced by (regular) *must* transition $a_1 \xrightarrow{\text{must}} a_5$. The result is a MixTS $M_5$ in Figure 5.2. Since $a_5$ replaces a "hyper-state" $\{a_2, a_3\}$, $a_5$ needs to preserve its *may* behaviours. This is done by adding $a_5 \xrightarrow{\text{may}} a_4$ and $a_5 \xrightarrow{\text{may}} a_2$ corresponding to $a_2 \xrightarrow{\text{may}} a_4$ and $a_3 \xrightarrow{\text{may}} a_2$, respectively. There are no outgoing *must* transi-

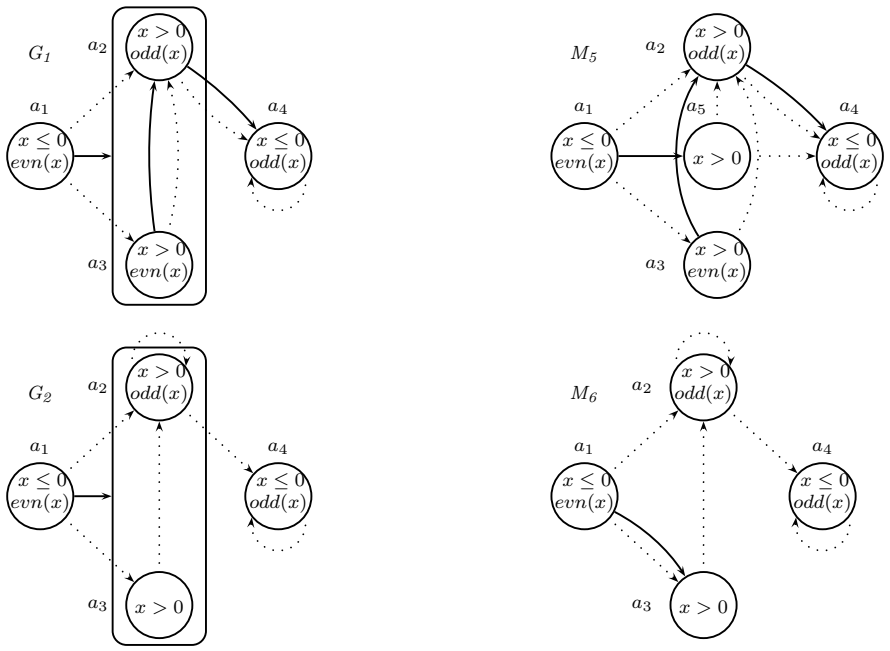


Figure 5.2: Two GKMTSs: $G_1$, $G_2$, and two MixTSs: $M_5$, $M_6$, where $G_1$ and $G_2$ are semantically equivalent to $M_5$ and $M_6$, respectively.

tions from $a_5$ since the existing *must* transitions from $a_2$ and $a_3$ are sufficient. $G_1$ and $M_5$ are semantically equivalent: any BTS that refines $G_1$ also refines $M_5$, and vice versa.

In our example, a new state was added to encode a hyper-transition by a regular one. This isn't always necessary. For example, TSs $G_2$ and $M_6$ in Figure 5.2 are semantically equivalent. The hyper-transition $a_1 \xrightarrow{\text{must}} \{a_2, a_3\}$ is encoded by $a_1 \xrightarrow{\text{must}} a_3$ in $M_6$ since the hyper-state $\{a_2, a_3\}$ is equivalent to an existing state $a_3$, i.e., $\gamma(\{a_2, a_3\}) = \gamma(a_3) = (x > 0)$.

In summary, a GKMTS $G$ is translated to a MixTS $M$ in two steps:

(i) every *must* hyper-transition $a \xrightarrow{\text{must}} U$ of $G$ is replaced by a regular *must* transition $a \xrightarrow{\text{must}} b$, where $b$ is a (possibly new) state s.t. $\gamma(b) = \gamma(U)$;

(ii) *may* transitions are added for every state introduced in the first step, if any.

We formalize this below.

**Definition 5.18** (GTOM)**.** *Let* $G = \langle S_G, R_G^{\text{may}}, R_G^{\text{must}} \rangle$ *be a GKMTS. The translation* $\text{GTOM}(G)$ *is a MixTS* $M = \langle S_M, R_M^{\text{must}}, R_M^{\text{may}} \rangle$*, such that*

$$S_M \triangleq S_G \cup S^+$$

$$S^+ \triangleq \{a \mid \exists (s,U) \in R_G^{\text{must}} \cdot \gamma(a) = \gamma(U) \wedge (\forall t \in S_G \cdot \gamma(t) \neq \gamma(U))\}$$

$$R_M^{\text{may}} \triangleq R_G^{\text{may}} \cup \{(a,b) \mid a \in S^+ \wedge b \in S_G \wedge \exists s \in S_G \cdot (s,b) \in R_G^{\text{may}} \wedge \gamma(s) \subseteq \gamma(a)\}$$

$$R_M^{\text{must}} \triangleq \{(a,b) \mid a \in S_G \ \wedge \ b \in S_M \wedge \exists U \subseteq S_G \cdot (a,U) \in R_G^{\text{must}} \ \wedge \ \gamma(b) = \gamma(U)\}$$

The translation GTOM is semantics-preserving.

**Theorem 5.19.** *Let* $G$ *be a GKMTS, and* $M = \text{GTOM}(G)$*. Then,* $M$ *is a MixTS, and* $G$ *and* $M$ *are semantically equivalent.*

**Proof:**

(1) According to the construction in Definition 5.18, every must hyper-transition is replaced by a regular one. Therefore, $M$ is a MixTS. (2) To prove that $G$ and $M$ are semantically equivalent, we show that any concrete BTS $B = \langle C, R \rangle$ refines $G$ iff it refines $M$. It is equivalent to showing that the soundness relation $\rho_G \subseteq C \times S_G$ is a mixed simulation between $B$ and $G$ iff the soundness relation $\rho_M \subseteq C \times S_M$ is a mixed simulation between $B$ and $M$. This follows from the construction of transition relations given in Definition 5.18. □

A corollary of Theorem 5.19 is that GKMTSs and MixTSs are equivalent w.r.t. thorough semantics. Let $L_G$ be a labeling function for $G$. We extend the translation GTOM to a GKMTS model $\langle G, L_G \rangle$ such that $\text{GTOM}(\langle G, L_G \rangle) \triangleq \langle M, L_M \rangle$, where $M = \text{GTOM}(G)$, and $L_M$ is a labeling function for $S_M$ defined as follows:

$$L_M(a) \triangleq \begin{cases} L_G(a) & \text{if } a \in S_G \\ \bigcap_{\{s \in S_G \mid \gamma(s) \subseteq \gamma(a)\}} L_G(s) & \text{if } a \in S^+ \end{cases}$$

That is, if $a$ is a state belonging to the original statespace $S_G$, the labels on $a$ are the same as before. For a new state $a$ added by the translation, since the concrete states approximated by $a$ are the *union* of the ones approximated by a set of states in $S_G$, the labels on $a$ are the literals

that are true in all the concrete states; therefore, $L_M(a)$ is defined as the intersection of the labels on the states in $S_G$ that are more precise than $a$.

**Theorem 5.20.** *The state labeling $L_M$ above is well-defined and approximates the same labellings as $L_G$.*

**Proof:**

The proof immediately follows from the approximation defined for state labeling and construction of $L_M$. □

As a result, $\langle G, L_G\rangle$ and $\langle M, L_M\rangle$ satisfy the same properties under thorough semantics.

**Corollary 5.21.** *Let $\langle G, L_G\rangle$ be a GKMTS model and $\langle M, L_M\rangle = \text{GTOM}(\langle G, L_G\rangle)$. Then, $\langle G, L_G\rangle$ and $\langle M, L_M\rangle$ are equivalent w.r.t. thorough semantics.*

*Complexity*. We show that the translation GTOM does not increase the size of the model. Let $G$ be a GKMTS with the statespace $S_G$, and $M = \text{GTOM}(G)$. The size of $G$ is at most $|S_G \times 2^{S_G}|$. Each new state added by GTOM corresponds to a subset of $S_G$, i.e., $|S^+| \leq |2^{S_G}|$. Furthermore, no transitions between the states in $S^+$ are added. Thus, the size of $M$ is also at most $|S_G \times 2^{S_G}|$.

Sometimes GTOM can reduce a GKMTS exponentially. For example, assume that $S_G$ is a disjunctive completion [CC92], i.e., for every subset $U$ of $S_G$ there exists an equivalent element $s$ in $S_G$ such that $\gamma(U) = \gamma(s)$. In this case, GTOM does not add any new states, i.e., $S^+ = \emptyset$. This makes the size of the output MixTSs be $|S_G \times S_G|$, which is exponentially smaller than that of the input GKMTS.

### 5.5.2 MTOK: Translation from MixTSs to KMTSs

We present the translation MTOK that converts a MixTS into a semantically equivalent KMTS. First, we illustrate the translation using a MixTS $M_7$ in Figure 5.3. $M_7$ is not a KMTS because of the two *must only* transitions, $a_1 \xrightarrow{\text{must}} a_2$ and $a_2 \xrightarrow{\text{must}} a_4$. One way to turn $M_7$ into a KMTS

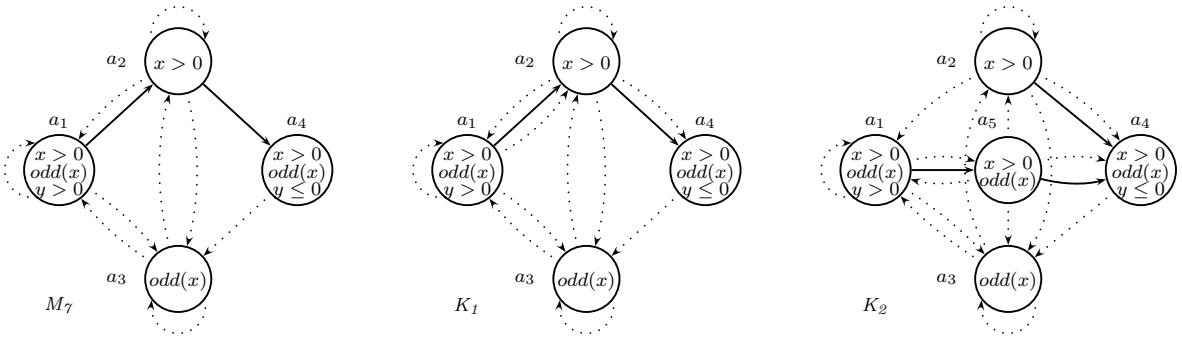


Figure 5.3: One MixTSs: $M_7$, and two KMTSs: $K_1$, $K_2$, where $M_7$ and $K_4$ are semantically equivalent.

is to add *may* transitions $a_1 \xrightarrow{\text{may}} a_2$ and $a_2 \xrightarrow{\text{may}} a_4$, resulting in $K_1$ in Figure 5.3. This naive transformation is not semantics-preserving, i.e., $K_1$ and $M_7$ are not semantically equivalent. For example, the concrete system

$$((y > 0) \wedge (x > 0) \wedge odd(x) \wedge x' = x + 1 \wedge y' = y) \vee$$
$$((x > 0) \wedge odd(x) \wedge x' = x \wedge y' = -1 \times x) \vee$$
$$((x > 0) \wedge \neg odd(x) \wedge x' = x + 1 \wedge y' = -1 \times x)$$

refines $K_1$, but not $M_7$: the transition $\langle x = 1, y = 1\rangle \rightarrow \langle x = 2, y = 1\rangle$ cannot be simulated by any *may* transition of $M_7$.

The *must only* transition $a_1 \xrightarrow{\text{must}} a_2$ of $M_7$ ensures that in any concrete BTS refining $M_7$, all states in $\gamma(a_1)$, i.e., those satisfying $(x > 0 \wedge odd(x) \wedge y > 0)$, must have a transition to a state in $\gamma(a_2)$, i.e., satisfying $(x > 0)$. This is further restricted by the *may* transitions from $a_1$ that ensure that states in $\gamma(a_1)$ have transitions only to states in $\gamma(\{a_1, a_3\})$. Hence, in any BTS refining $M_7$, every state in $\gamma(a_1)$ must (and may) have a transition to a state in $\gamma(a_2) \cap \gamma(\{a_1, a_3\})$. That is, the restrictions posed by a *must only* transition from $a_1$ are further restricted by the set of all of the *may* transitions from $a_1$. In general, for abstract states $b_0, \ldots, b_k$, a *must only* transition $b_0 \xrightarrow{\text{must}} b_1$, and a set of *may* transitions $b_0 \xrightarrow{\text{may}} b_2, \ \ldots, \ b_0 \xrightarrow{\text{may}} b_k$ ensure that every state in $\gamma(b_0)$ has a transition to a state in $\gamma(b_1) \cap \gamma(\{b_2, \ldots, b_k\})$.

The *must only* transition $a_2 \xrightarrow{\text{must}} a_4$ in $M_7$ is equivalent to a pair of *may* and *must* transitions from $a_2$ to $a_4$, since $\gamma(a_4) \cap \gamma(\{a_1, a_2, a_3\}) = \gamma(a_4)$. The *must only* transition $a_1 \xrightarrow{\text{must}} a_2$ can be equivalently represented by (a) adding a new state $a_5$ such that $\gamma(a_5) = \gamma(a_2) \cap \gamma(\{a_1, a_3\}) = (x > 0 \wedge odd(x))$, and (b) adding a *must* and a *may* transition from $a_1$ to $a_5$. Moreover, since $a_5$ approximates some of the same states as $a_2$, i.e., $\gamma(a_5) \subseteq \gamma(a_2)$, $a_5$ inherits the transitions from $a_2$: $a_5 \xrightarrow{\text{may}} a_1$, $a_5 \xrightarrow{\text{may}} a_2$, $a_5 \xrightarrow{\text{may}} a_3$, $a_5 \xrightarrow{\text{must}} a_4$, $a_5 \xrightarrow{\text{may}} a_4$. The final result is the KMTS $K_2$ in Figure 5.3, which is semantically equivalent to $M_7$.

In summary, a MixTS $M$ is translated to a KMTS $K$ in two steps:

(i) every *must only* transition $a \xrightarrow{\text{must}} b$ of $M$ is replaced by a pair of *must* and *may* transitions $a \xrightarrow{\text{must}} \widehat{a \to b}$ and $a \xrightarrow{\text{may}} \widehat{a \to b}$, where $\widehat{a \to b}$ is a (possibly new) abstract state such that $\gamma(\widehat{a \to b}) = \gamma(b) \cap \gamma(R_M^{\text{may}}(a))$;

(ii) *may* and *must* transitions are added for all states introduced in the first step.

We formalize this below.

**Definition 5.22** (MTOK)**.** *Let $M = \langle S_M, R_M^{\text{may}}, R_M^{\text{must}} \rangle$ be a MixTS. The translation* MTOK$(M)$ *is a KMTS $K = \langle S_K, R_K^{\text{may}}, R_K^{\text{must}} \rangle$, s.t.*

$$S_K \triangleq S_M \cup S^+$$

$$R_K^{\text{may}} \triangleq R_M^{\text{may}} \cup \text{REPL} \cup \text{IMAY} \cup \text{IMO}$$

$$R_K^{\text{must}} \triangleq (R_M^{\text{must}} \cap R_M^{\text{may}}) \cup \text{REPL} \cup \text{IMUST} \cup \text{IMO},$$

*where*

$$
\begin{aligned}
S^+ &\triangleq \{\widehat{a \to b} \mid \exists (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \cdot \forall s \in S_M \cdot \gamma(s) \neq \gamma(\widehat{a \to b})\} \\
\text{REPL} &\triangleq \{(a, \widehat{a \to b}) \mid \exists (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}})\} \\
\text{IMAY} &\triangleq \{(\widehat{a \to b}, b') \mid \exists a,b,b' \in S_M \cdot \\
&\qquad\qquad (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \wedge (b,b') \in R_M^{\text{may}} \wedge \widehat{a \to b} \in S^+\} \\
\text{IMUST} &\triangleq \{(\widehat{a \to b}, b') \mid \exists a,b,b' \in S_M \cdot \\
&\qquad\qquad (a,b) \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \wedge (b,b') \in (R_M^{\text{must}} \cap R_M^{\text{may}}) \wedge \widehat{a \to b} \in S^+\} \\
\text{IMO} &\triangleq \{(\widehat{a \to b}, \widehat{b \to b'} \mid \exists a,b,b' \in S_M \cdot \\
&\qquad\qquad (a,b),(b,b') \in (R_M^{\text{must}} \setminus R_M^{\text{may}}) \wedge \widehat{a \to b} \in S^+\}
\end{aligned}
$$

In Definition 5.22, REPL denotes transitions that replace *must only* transitions, and IMAY, IMUST and IMO denote transitions from newly added states in $S^+$ that correspond to *may*, *must*, and *must only* transitions of the original system, respectively. In our example of MTOK$(M_7)$, we have

$$
\begin{aligned}
S^+ &= \{a_5\}, \\
\text{REPL} &= \{(a_1,a_5),(a_2,a_4)\}, \\
\text{IMUST} &= \emptyset, \\
\text{IMO} &= \{(a_5,a_4)\}, \\
\text{IMAY} &= \{(a_5,a_1),(a_5,a_2),(a_5,a_3)\}.
\end{aligned}
$$

The result of the translation MTOK is a KMTS: every *must* transition is matched by a *may* transition.

**Theorem 5.23.** *Let $M$ be a MixTS, and $K = \text{MTOK}(M)$. Then $K$ is a KMTS, and $M$ and $K$ are semantically equivalent.*

**Proof:**

(1) The construction in Definition 5.22 ensures that every *must* transition in $K$ is matched by a *may* transition. Therefore, $K$ is a KMTS. (2) To prove that $M$ and $K$ are semantically equivalent, we show that for any concrete BTS $B = \langle C, R\rangle$, the soundness relation $\rho_M \subseteq C \times S_M$ is a mixed simulation between $B$ and $M$ iff the soundness relation $\rho_K \subseteq C \times S_K$ is a

mixed simulation between $B$ and $K$. This follows from the construction of transition relations in Definition 5.22. □

A corollary of Theorem 5.23 is that MixTSs and KMTSs are equivalent w.r.t. thorough semantics. Let $L_M$ be a labeling function for $M$. We extend MTOK to $\langle M, L_M\rangle$ such that $\text{MTOK}(\langle M, L_M\rangle) \triangleq \langle K, L_K\rangle$, where $K = \text{MTOK}(M)$, and $L_K$ is a labeling function for $S_K$ defined as follows:

$$L_K(a) \triangleq \begin{cases} L_M(a) & \text{if } a \in S_M \\ \bigcup_{\{s \in S_M | \gamma(a) \subseteq \gamma(s)\}} L_M(s) & \text{if } a \in S^+ \end{cases}$$

In this case, if $a$ is a new state added by the translation, the concrete states approximated by $a$ correspond to the *intersection* of the concrete states approximated by a set of states in $S_G$; the labels on $a$ are all the literals which are true on the concrete states. Therefore, $L_K(a)$ is defined as the union of the labels on the states in $S_M$ that are less precise than $a$.

**Theorem 5.24.** *The state labeling $L_K$ above is well-defined and approximates the same labellings as $L_M$.*

**Proof:**

The proof immediately follows from the approximation defined for state labeling and the construction of $L_K$. □

As a result, $\langle M, L_M\rangle$ and $\langle K, L_K\rangle$ satisfy the same properties under thorough semantics.

**Corollary 5.25.** *Let $\langle M, L_M\rangle$ be a MixTS model and $\langle K, L_K\rangle = \text{MTOK}(\langle M, L_M\rangle)$. Then, $\langle M, L_M\rangle$ and $\langle K, L_K\rangle$ are equivalent w.r.t. thorough semantics.*

*Complexity.* Let $M = \langle S_M, R_M^{\text{may}}, R_M^{\text{must}}\rangle$ be a MixTS, and $K$ be a KMTS such that $K = \text{MTOK}(M)$. The size of $M$ is bounded by $O(|S_M \times S_M|)$. In the worst case, the translation adds a new state for each *must only* transition in $R_M^{\text{must}} \setminus R_M^{\text{may}}$. Thus, the number of new states $|S^+|$ is bounded by $|S_M \times S_M|$, and $|K|$ is bounded by $O(|S_M \times S_M|^2)$.

MixTSs are more succinct than KMTSs: over a fixed statespace $S$, the set of MixTSs is more expressive than the set of KMTSs. This holds because $S^+$ may not be empty in some

cases, i.e., new states have to be added by MTOK. The following theorem shows that if $S$ is a powerset abstract domain [BHZ06], then MTOK does not add new states, and therefore, MixTS and KMTSs over $S$ are equally expressive.

**Theorem 5.26.** *Let $S$ be an abstract statespace satisfying the assumption of the existence of best abstraction. For any abstract state $a \in S$ and a subset $Q \subseteq S$, there exists a subset $V \subseteq S$ s.t. $\gamma(V) = \gamma(a) \cap \gamma(Q)$.*

**Proof:**

Let $V \triangleq \{b \in S \mid \exists c \cdot c \in \gamma(a) \cap \gamma(Q) \wedge b = \alpha(c)\}$. The proof of $\gamma(V) \supseteq \gamma(a) \cap \gamma(Q)$ follows from the definition of $V$. To prove $\gamma(V) \subseteq \gamma(a) \cap \gamma(Q)$, we show that for each $b \in V$, $\gamma(b) \subseteq \gamma(a)$ and $\gamma(b) \subseteq \gamma(Q)$, which follows from the definition of abstraction function. □

## 5.6 Reduced Inductive Semantics

GKMTSs and MixTSs are equally expressive: a GKMTS model and its equivalent MixTS model satisfy the same properties under thorough semantics. However, thorough model checking is expensive. In practice, model checking of partial models is done w.r.t. a more tractable inductive semantics, SIS. GKMTSs are more precise than MixTSs w.r.t. SIS: for any $\varphi \in L_\mu$, model checking $\varphi$ in a GKMTS model $\mathcal{G}$ w.r.t. SIS is more precise than model checking it in the MixTS model $\mathcal{M} = \text{GTOM}(\mathcal{G})$. However, the direct use of GKMTSs in symbolic model checkers has been hampered by the difficulty of encoding hyper-transitions into BDDs. In this section, we propose a new semantics, called *reduced inductive semantics* (RIS), that is inductive while being strictly more precise than SIS. We show that GKMTSs and MixTSs are equivalent w.r.t. RIS. In Section 5.7, we give a symbolic model checking procedure for computing RIS over MixTSs. The outcome is an algorithm that combines the benefits of the symbolic encoding of MixTSs with the better model checking precision of GKMTSs.

In Section 5.6.1, we illustrate the differences between GKMTSs and MixTSs w.r.t. SIS; define RIS in Section 5.6.2; and show how to effectively model check w.r.t. RIS in Section 5.6.3.

### 5.6.1 Example

Let $p$ and $q$ denote predicates $(x > 0)$ and $odd(x)$, respectively. Consider the model $\mathcal{G}_1 = \langle G_1, L_{G_1} \rangle$, where $G_1$ is shown in Figure 5.2, and $L_{G_1}$ is a labeling function that labels each abstract state as follows:

$$L_{G_1}(a_1) = \{\neg p, \neg q\} \qquad L_{G_1}(a_2) = \{p, q\}$$
$$L_{G_1}(a_3) = \{p, \neg q\} \qquad L_{G_1}(a_4) = \{\neg p, q\}\,.$$

Let $\mathcal{M}_5 = \langle M_5, L_{M_5} \rangle$ be the model obtained from $\mathcal{G}_1$ by GTOM, where $M_5$ is shown in Figure 5.2 and $L_{M_5}(s) \triangleq$ **if** $s = a_5$ **then** $\{p\}$ **else** $L_{G_1}(s)$.

Compare the value of $\varphi \triangleq \Diamond(q \vee \neg q)$ under SIS on $\mathcal{G}_1$ and $\mathcal{M}_5$:

$$\begin{aligned} \|\varphi\|_i^{\mathcal{G}_1} &= \langle \{a_1, a_2, a_3\}, \{a_1, a_2, a_3, a_4\} \rangle \\ \|\varphi\|_i^{\mathcal{M}_5} &= \langle \{a_2, a_3\}, \{a_1, a_2, a_3, a_4, a_5\} \rangle \end{aligned}$$

According to $\mathcal{G}_1$, in all states corresponding to $a_1$, $\varphi$ is true . According to $\mathcal{M}_5$, the value of $\varphi$ is unknown in exactly the same states. Since $\mathcal{M}_5 = \text{GTOM}(\mathcal{G}_1)$, $\mathcal{G}_1$ and $\mathcal{M}_5$ are semantically equivalent. Thus, although $\mathcal{M}_5$ and $\mathcal{G}_1$ are semantically equivalent, $\mathcal{M}_5$ is less precise than $\mathcal{G}_1$ for model checking w.r.t. SIS.

Let us reexamine the above example. First, there is no precision loss during the evaluation of $q \vee \neg q$:

$$\begin{aligned} e_1 = \|q \vee \neg q\|_i^{\mathcal{G}_1} &= \langle \{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_4\} \rangle \\ e_2 = \|q \vee \neg q\|_i^{\mathcal{M}_5} &= \langle \{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_4, a_5\} \rangle \end{aligned} \qquad (\star)$$

Since $\gamma(\mathsf{U}(e_1)) = \gamma(\mathsf{U}(e_2))$ and $\gamma(\overline{\mathsf{O}(e_1)}) = \gamma(\overline{\mathsf{O}(e_2)}) = \gamma(\emptyset)$, $e_1 \equiv_a e_2$. However, there is a subtle difference between $e_1$ and $e_2$. In state $a_5$ of $M_5$, $q \vee \neg q$ is unknown even though it is true in both $a_2$ and $a_3$, and $\gamma(a_5) = \gamma(a_2) \cup \gamma(a_3)$. This minor imprecision is then magnified by the $\Diamond$ operator.

This loss of precision is not limited to tautologies. For example, a formula $\mu Z \cdot (\neg p \wedge q) \vee \Diamond Z$, i.e, $EF(\neg p \wedge q)$ in CTL, is true in state $a_1$ of $\mathcal{G}_1$, but is unknown in the same state of $\mathcal{M}_5$.

### 5.6.2 Reduced Inductive Semantics for Partial Models

In this section, we define the reduced inductive semantics (RIS). The new semantics is inductive and is *strictly more precise* than SIS. The key idea is to eliminate any local imprecision by using a special *reduction* operator

*Reduction Operator*. Let $S$ be an abstract statespace, and $e, e' \in 2^S \times 2^S$ be two abstract elements. Recall that in the information order $e$ is less than $e'$, i.e., $e \preceq_i e'$, if $\mathsf{U}(e)$ is contained in $\mathsf{U}(e')$, and $\mathsf{O}(e)$ contains $\mathsf{O}(e')$. We define the *reduction* operator as follows:

$$\mathtt{RED}(e) \triangleq \langle \mathtt{RED}_{\mathsf{U}}(U), \mathtt{RED}_{\mathsf{O}}(O) \rangle$$

where $\mathtt{RED}_{\mathsf{U}}(U) \triangleq \{s \mid \gamma(s) \subseteq \gamma(U)\}$ and $\mathtt{RED}_{\mathsf{O}}(O) \triangleq \{s \mid \gamma(s) \nsubseteq \gamma(\overline{O})\}$. Intuitively, $\mathtt{RED}(e)$ increases $\mathsf{U}(e)$ and decreases $\mathsf{O}(e)$ as much as possible without affecting the semantic meaning of $e$. That is, $\mathtt{RED}(e)$ is the largest element w.r.t. information ordering that is semantically equivalent to $e$. For example, consider $\mathtt{RED}(e_2)$, where $e_2$ is as defined by ($\star$) above. Then,

$$e_3 = \mathtt{RED}(e_2) = \langle \{a_1, a_2, a_3, a_4, a_5\}, \{a_1, a_2, a_3, a_4, a_5\} \rangle \qquad (\star\star)$$

$e_3$ differs from $e_2$ only in the addition of $a_5$ to $\mathsf{U}(e_3)$. Since $\gamma(\mathsf{U}(e_2)) = \gamma(\mathsf{U}(e_3))$ and $\gamma(\overline{\mathsf{O}(e_2)}) = \gamma(\overline{\mathsf{O}(e_3)})$, $e_2 \equiv_a e_3$; but $e_3$ is more informative, since $\mathsf{U}(e_2) \subset \mathsf{U}(e_3)$.

An element $e = \langle U, O \rangle \in 2^S \times 2^S$ is *monotone* iff

$$s_1 \preceq_a s_2 \Rightarrow (s_1 \in U \Rightarrow s_2 \in U \wedge s_1 \notin O \Rightarrow s_2 \notin O)$$

The monotonicity of elements is preserved under propositional operations. That is, if $e$ and $e'$ are monotone elements, so are $\sim e$ and $e \sqcap e'$. Moreover, $\mathtt{RED}(e)$ is monotone for any $e$, and commutes with propositional operations on monotone elements. That is, let $e$ and $e'$ be monotone elements of $2^S \times 2^S$. Then, $\sim e \equiv_a \sim \mathtt{RED}(e)$, and $e \sqcap e' \equiv_a \mathtt{RED}(e) \sqcap \mathtt{RED}(e')$.

*Reduced Inductive Semantics*. RIS is defined by applying the RED operator before and after $\Diamond$ to prevent it from propagating imprecision.

**Definition 5.27** (RIS)**.** *Let $\mathcal{M} = \langle M, L\rangle$ be a model, s.t. $M = \langle S, R^{\text{may}}, R^{\text{must}}\rangle$ and $\sigma : Var \to 2^S \times 2^S$. The* reduced inductive semantics *of $\varphi \in L_\mu$ is defined as follows:*

$$
\begin{aligned}
||p||^{\mathcal{M}}_{r,\sigma} &\triangleq \langle \{s \mid p \in L(s)\}, \{s \mid \neg p \notin L(s)\}\rangle \\
||\neg\varphi||^{\mathcal{M}}_{r,\sigma} &\triangleq \sim||\varphi||^{\mathcal{M}}_{r,\sigma} \\
||\varphi \wedge \psi||^{\mathcal{M}}_{r,\sigma} &\triangleq ||\varphi||^{\mathcal{M}}_{r,\sigma} \sqcap ||\psi||^{\mathcal{M}}_{r,\sigma} \\
||\Diamond\varphi||^{\mathcal{M}}_{r,\sigma} &\triangleq \mathtt{RED}(\langle pre_{\mathsf{U}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(||\varphi||^{\mathcal{M}}_{r,\sigma}))), pre_{\mathsf{O}}(\mathtt{RED}_{\mathsf{O}}(\mathsf{O}(||\varphi||^{\mathcal{M}}_{r,\sigma})))\rangle) \\
||Z||^{\mathcal{M}}_{r,\sigma} &\triangleq \sigma(Z) \\
||\mu Z \cdot \varphi||^{\mathcal{M}}_{r,\sigma} &\triangleq \langle \mathit{lfp}^{\sqsubseteq}\left(\lambda Q \cdot \mathsf{U}(||\varphi||^{\mathcal{M}}_{r,\sigma[Z\mapsto Q]})\right), \mathit{lfp}^{\sqsubseteq}\left(\lambda Q \cdot \mathsf{O}(||\varphi||^{\mathcal{M}}_{r,\sigma[Z\mapsto Q]})\right)\rangle
\end{aligned}
$$

The only difference between RIS (Definition 5.27) and SIS (Definition 2.10) is the semantics of $\Diamond$. Since we assume that state-labelings are monotone, applying RED to other operators as well does not improve precision. We now show that RIS is sound.

**Theorem 5.28.** *Let $C$ be a concrete statespace approximated by an abstract statespace $S$ via $\langle C, \rho, \gamma, \alpha, S\rangle$. Let $\mathcal{B} = \langle B, L_B\rangle$ be a concrete model over $C$, and $\mathcal{M} = \langle M, L_M\rangle$ be a partial model over $S$. If $\mathcal{M}$ approximates $\mathcal{B}$, then, for any $L_\mu$ formula $\varphi$:*

$$
\gamma(\mathsf{U}(\|\varphi\|^{\mathcal{M}}_r)) \subseteq \mathsf{U}(\|\varphi\|^{\mathcal{B}}_r), \qquad \textit{and} \qquad \gamma(\overline{\mathsf{O}(\|\varphi\|^{\mathcal{M}}_r)}) \subseteq \overline{\mathsf{O}(\|\varphi\|^{\mathcal{B}}_r)}\ .
$$

**Proof:**

The only difference between RIS and SIS is the application of the RED operator before and after $\Diamond$. Since RED is semantics-preserving, following Theorem 2.14, the result holds. □

Returning to our running example, RIS of $\varphi$ on $\mathcal{M}_5$ is computed as follows: RIS of $q$, $\neg q$, and $q \vee \neg q$ is the same as SIS. Thus, $\|q \vee \neg q\|^{\mathcal{M}_5}_r = e_2$. To compute $\Diamond$, recall from ($\star\star$) that $\mathtt{RED}(e_2) = e_3$; thus, $\|\varphi\|^{\mathcal{M}_5}_r = \langle\{a_1, a_2, a_3, a_5\}, \{a_1, a_2, a_3, a_4, a_5\}\rangle$. Hence, $\|\varphi\|^{\mathcal{M}_5}_r$ is more precise than $\|\varphi\|^{\mathcal{M}_1}_i$.

**Theorem 5.29.** *RIS is more precise than SIS: $\|\varphi\|_i \preceq_a \|\varphi\|_r$.*

**Proof:**

The proof is by structural induction on $\varphi$. For the base case, it is obvious that for any atomic

proposition $p$, $\|p\|_i \equiv_a \|p\|_r$. In the following, we show the inductive case for $\Diamond\varphi$; the proofs of other cases are trivial.

We show that $||\varphi||_i \preceq_a ||\varphi||_r \Rightarrow ||\Diamond\varphi||_i \preceq_a ||\Diamond\varphi||_r$, which is equivalent to proving the following two statements:

$(a)$ $||\varphi||_i \preceq_a ||\varphi||_r \Rightarrow \gamma(\mathsf{U}(||\Diamond\varphi||_i)) \subseteq \gamma(\mathsf{U}(||\Diamond\varphi||_r))$

$(b)$ $||\varphi||_i \preceq_a ||\varphi||_r \Rightarrow \gamma(\overline{\mathsf{O}(||\Diamond\varphi||_i)}) \subseteq \gamma(\overline{\mathsf{O}(||\Diamond\varphi||_r)})$

The proof of (a) is as follows. First, note that for any two sets $Q_1$, $Q_2$, we have that

$$\gamma(Q_1) \subseteq \gamma(\mathtt{RED}_{\mathsf{U}}(Q_2)) \Rightarrow Q_1 \subseteq \mathtt{RED}_{\mathsf{U}}(Q_2) \qquad \textbf{(P1)}$$

This follows from the following derivation: suppose $Q_1 \nsubseteq \mathtt{RED}_{\mathsf{U}}(Q_2)$. Then there exists a state $s$ s.t. $s \in Q_1$ and $s \notin \mathtt{RED}_{\mathsf{U}}(Q_2)$. By the definition of $\mathtt{RED}_{\mathsf{U}}$, $\gamma(s) \nsubseteq \gamma(Q_2)$; on the other hand, since $\gamma(Q_1) \subseteq \gamma(\mathtt{RED}_{\mathsf{U}}(Q_2)) = \gamma(Q_2)$, $\gamma(s) \subseteq \gamma(Q_2)$, reaching a contradition.

We then have the following:

$||\varphi||_i \preceq_a ||\varphi||_r$

$\Rightarrow$ (by the definition of $\preceq_a$)

$\gamma(\mathsf{U}(||\varphi||_i) \subseteq \gamma(\mathsf{U}(||\varphi||_r))$

$\Rightarrow$ (by the definition of $\mathtt{RED}_{\mathsf{U}}$, $\gamma(Q) = \gamma(\mathtt{RED}_{\mathsf{U}}(Q))$)

$\gamma(\mathsf{U}(||\varphi||_i) \subseteq \gamma(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(||\varphi||_r)))$

$\Rightarrow$ (by **(P1)**)

$\mathsf{U}(||\varphi||_i)) \subseteq \mathtt{RED}_{\mathsf{U}}(\mathsf{U}(||\varphi||_r))$

$\Rightarrow$ (by monotonicity of $pre$ )

$pre_{\mathsf{U}}(\mathsf{U}(||\varphi||_i)) \subseteq pre_{\mathsf{U}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(||\varphi||_r)))$

$\Rightarrow$ (by monotonicity of $\gamma$)

$\gamma(pre_{\mathsf{U}}(\mathsf{U}(||\varphi||_i))) \subseteq \gamma(pre_{\mathsf{U}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(||\varphi||_r))))$

$\Rightarrow$ (by the definition of $\mathtt{RED}_{\mathsf{U}}$, $\gamma(Q) = \gamma(\mathtt{RED}_{\mathsf{U}}(Q))$)

$\gamma(pre_{\mathsf{U}}(\mathsf{U}(||\varphi||_i))) \subseteq \gamma(\mathtt{RED}_{\mathsf{U}}(pre_{\mathsf{U}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(||\varphi||_r)))))$

$\Rightarrow$ (by the definitions of SIS and RIS)

$\gamma(\mathsf{U}(||\Diamond\varphi||_i)) \subseteq \gamma(\mathsf{U}(||\Diamond\varphi||_r))$

Proof of (b) is dual of the one above. □

The previous example illustrates another important point: GKMTSs and MixTSs are equivalent w.r.t. RIS. For example, $\|\varphi\|_r^{\mathcal{M}_5}$ is equivalent to $\|\varphi\|_r^{\mathcal{G}_1}$. The following theorem formalizes this.

**Theorem 5.30.** *Let $\mathcal{G}$ be a GKMTS model, and $\mathcal{M} = \text{GTOM}(\mathcal{G})$. Then, $\mathcal{G}$ and $\mathcal{M}$ are equivalent w.r.t. RIS:* $\forall \varphi \in L_\mu \cdot \|\varphi\|_r^{\mathcal{G}} \equiv_a \|\varphi\|_{r.}^{\mathcal{M}}$

**Proof:**

The proof is by structural induction on $\varphi$. For the base case, according to the definition of $L_M$, $\|p\|_r^{\mathcal{G}} \equiv_a \|p\|_r^{\mathcal{M}}$ for any atomic proposition $p$. In the following, we show the inductive case for $\Diamond\varphi$; the proofs of the other cases are trivial.

We show that $\|\varphi\|_r^{\mathcal{G}} \equiv_a \|\varphi\|_r^{\mathcal{M}} \Rightarrow \|\Diamond\varphi\|_r^{\mathcal{G}} \equiv_a \|\Diamond\varphi\|_r^{\mathcal{M}}$, which is equivalent to proving the following two statements:

$(a)$ $\|\varphi\|_r^{\mathcal{G}} \equiv_a \|\varphi\|_r^{\mathcal{M}} \Rightarrow \gamma(\mathsf{U}(\|\Diamond\varphi\|_r^{\mathcal{G}})) = \gamma(\mathsf{U}(\|\Diamond\varphi\|_r^{\mathcal{M}}))$

$(b)$ $\|\varphi\|_r^{\mathcal{G}} \equiv_a \|\varphi\|_r^{\mathcal{M}} \Rightarrow \gamma(\overline{\mathsf{O}(\|\Diamond\varphi\|_r^{\mathcal{G}})}) = \gamma(\overline{\mathsf{O}(\|\Diamond\varphi\|_r^{\mathcal{M}})})$

The proof of (a) is as follows. First, note that for any concrete state $c$ and a set of abstract states $Q$,

$$c \in \gamma(\texttt{RED}_{\mathsf{U}}(Q)) \Leftrightarrow \exists a \in Q \cdot c \in \gamma(a) \qquad \textbf{(P2)}$$

We then have that, for any concrete state $c$,

$$
\begin{aligned}
& c \in \gamma(\mathsf{U}(\|\Diamond\varphi\|_r^{\mathcal{G}})) \\
\Leftrightarrow\ & \text{(by the definition of RIS)} \\
& c \in \gamma(\mathtt{RED}_{\mathsf{U}}(pre_{\mathsf{U}}^{\mathcal{G}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{G}}))))) \\
\Leftrightarrow\ & \text{( (}\Rightarrow\text{) let } a \text{ be the abstract state in (}\mathbf{P2}\text{),} \\
& \text{ (}\Leftarrow\text{) since } \gamma(Q) = \gamma(\mathtt{RED}_{\mathsf{U}}(Q))\text{)} \\
& c \in \gamma(a) \wedge a \in pre_{\mathsf{U}}^{\mathcal{G}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{G}}))) \\
\Leftrightarrow\ & \text{(by the definition of } pre_{\mathsf{U}}\text{)} \\
& c \in \gamma(a) \wedge \exists Q \subseteq \mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{G}})) \cdot R_{\mathcal{G}}^{\text{must}}(a, Q) \\
\Leftrightarrow\ & \text{(by the definition of GTOM)} \\
& c \in \gamma(a) \wedge \exists b \cdot \gamma(b) \subseteq \gamma(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{G}}))) \wedge R_{\mathcal{M}}^{\text{must}}(a, b) \\
\Leftrightarrow\ & \text{(since } \|\varphi\|_r^{\mathcal{G}} \equiv_a \|\varphi\|_r^{\mathcal{M}}\text{, } \gamma(\mathsf{U}(\|\varphi\|_r^{\mathcal{G}})) = \gamma(\mathsf{U}(\|\varphi\|_r^{\mathcal{M}}))\text{)} \\
& c \in \gamma(a) \wedge \exists b \cdot \gamma(b) \subseteq \gamma(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{M}}))) \wedge R_{\mathcal{M}}^{\text{must}}(a, b) \\
\Leftrightarrow\ & \text{(since } \gamma(Q) = \gamma(\mathtt{RED}_{\mathsf{U}}(Q))\text{, by the definition of } \mathtt{RED}_{\mathsf{U}}\text{)} \\
& c \in \gamma(a) \wedge \exists b \in \mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{M}})) \cdot R_{\mathcal{M}}^{\text{must}}(a, b) \\
\Leftrightarrow\ & \text{(by the definition of } pre_{\mathsf{U}}\text{)} \\
& c \in \gamma(a) \wedge a \in pre_{\mathsf{U}}^{\mathcal{M}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{M}}))) \\
\Leftrightarrow\ & \text{ ( (}\Rightarrow\text{) since } \gamma(Q) = \gamma(\mathtt{RED}_{\mathsf{U}}(Q))\text{,} \\
& \text{ (}\Leftarrow\text{) let } a \text{ be the abstract state in (}\mathbf{P2}\text{))} \\
& c \in \gamma(\mathtt{RED}_{\mathsf{U}}(pre_{\mathsf{U}}^{\mathcal{M}}(\mathtt{RED}_{\mathsf{U}}(\mathsf{U}(\|\varphi\|_r^{\mathcal{G}}))))) \\
\Leftrightarrow\ & \text{(by the definition of RIS)} \\
& c \in \gamma(\mathsf{U}(\|\Diamond\varphi\|_r^{\mathcal{M}}))
\end{aligned}
$$

The proof of (b) is similar to the one above, based on the observation that for any concrete state $c$ and a set of abstract states $Q$, $c \in \gamma(\overline{\mathtt{RED}_{\mathsf{O}}(Q)}) \Leftrightarrow \exists a \in \overline{Q} \cdot c \in \gamma(a)$. □

Our new semantics RIS is both inductive and precise enough to make GKMTSs and MixTSs equivalent. However, the definition of RED operator is based on concretization, $\gamma$, of abstract elements. In practice, reasoning directly about concrete elements may be undecidable or inefficient. We address this limitation next.

### 5.6.3 Reduced Inductive Semantics for Monotone Models

We study the reduction operator RED of RIS in the context of monotone models. As shown in Section 5.3, monotone models are as expressive as their regular counterparts. Furthermore, models built by automated predicate abstraction [GC06] are monotone by construction. Thus, restricting RED to monotone models is neither a theoretical nor a practical restriction.

Note that in any monotone model and any formula $\varphi$, $\|\varphi\|_r$ is a monotone element. This holds because of the monotonicity of the state labeling and the transition relation. For monotone elements, RED can be computed effectively, as we show below.

For a state $s \in S$, the *upset* of $s$ is defined as

$$\uparrow s \triangleq \{t \in \alpha[S] \mid s \preceq_a t\}.$$

Thus, $\uparrow s$ is the set of all those states in $\alpha[S]$ that are more precise than $s$. For example, let $S_1$ be the statespace of $M_5$ shown in Figure 5.2. Then, $\alpha[S_1] = \{a_1, a_2, a_3, a_4\}$, and $\uparrow a_5 = \{a_2, a_3\}$. A state $s$ and a set $\uparrow s$ approximate the same set of concrete states, i.e., $\gamma(s) = \gamma(\uparrow s)$. For example, $\gamma(\uparrow a_5) = \gamma(a_5) = (x > 0)$.

We now show that the $\uparrow s$ computes a canonical representation of an element $s$ of the abstract statespace.

**Theorem 5.31.** *Let $S$ be an abstract statespace, $e = \langle U, O\rangle$ be a monotone element of $2^S \times 2^S$, and $s \in S$. Then, $\gamma(s) \subseteq \gamma(U)$ iff $\uparrow s \subseteq U$ and $\gamma(s) \not\subseteq \gamma(\overline{O})$ iff $\uparrow s \not\subseteq \overline{O}$.*

**Proof:**

First, we show that $\gamma(s) \subseteq \gamma(U) \Leftrightarrow \uparrow s \subseteq U$. The ($\Leftarrow$) direction follows directly from the definition of $\gamma$. We prove the ($\Rightarrow$) direction by contradiction. Let $C$ be the concrete statespace

approximated by $S$. Suppose that $\uparrow s \not\subseteq U$. Then,

$$
\begin{array}{ll}
 & \uparrow s \not\subseteq U \\
\Rightarrow & \exists a \in S \cdot a \in \uparrow s \wedge a \notin U \\
\Rightarrow & \text{(by the definition of } \uparrow s\text{, } a \in \alpha[S]\text{)} \\
 & \exists a \in S \cdot s \preceq_a a \wedge a \notin U \wedge \exists c \in C \cdot a = \alpha(c) \\
\Rightarrow & \text{(since } s \preceq_a a\text{, } \gamma(a) \subseteq \gamma(s)\text{; since } \gamma(s) \subseteq \gamma(U)\text{)} \\
 & \exists a \in S \cdot a \notin U \wedge \exists c \in C \cdot a = \alpha(c) \wedge c \in \gamma(U) \\
\Rightarrow & \text{(by the definition of } \gamma\text{)} \\
 & \exists a \in S \cdot a \notin U \wedge \exists c \in C \cdot a = \alpha(c) \wedge \exists b \in U \cdot c \in \gamma(b) \\
\Rightarrow & \text{(by the definition of } \alpha\text{)} \\
 & \exists a \in S \cdot a \notin U \wedge \exists b \in U \cdot b \preceq_a a \\
\Rightarrow & \text{(by monotonicity of } e\text{, } a \in U\text{)} \\
 & \exists a \in S \cdot a \notin U \wedge a \in U \\
\Rightarrow & \text{false}
\end{array}
$$

The proof of $\gamma(s) \not\subseteq \gamma(\overline{O}) \Leftrightarrow \uparrow s \not\subseteq \overline{O}$ is dual of the one above. □

We now define a new operator red for monotone elements. Let $e = \langle U, O \rangle$ be a monotone element of $2^S \times 2^S$. red is defined as

$$\mathtt{red}(e) \triangleq \langle \mathtt{red_U}(U), \mathtt{red_O}(O) \rangle$$

where $\mathtt{red_U}(U) \triangleq \{s \mid \uparrow s \subseteq U\}$ and $\mathtt{red_O}(O) \triangleq \{s \mid \uparrow s \not\subseteq \overline{O})\}$. A corollary of Theorem 5.31 is that red and RED are equivalent.

**Corollary 5.32.** *Let $S$ be an abstract statespace, and $e$ be a monotone element in $2^S \times 2^S$. Then,* $\mathtt{red}(e) = \mathtt{RED}(e)$.

For example, the element $e_2$ defined in ($\star$) is monotone. We have that $\mathtt{red}(\mathsf{U}(e_2)) = \{a_1, a_2, a_3, a_4, a_5\}$ since $\uparrow a_5 = \{a_2, a_3\} \subseteq \mathsf{U}(e_2)$, and $\mathtt{red}(\mathsf{O}(e_2))$ is the same as $\mathsf{O}(e_2)$ since $\overline{\mathsf{O}(e_2)}$ is empty. Therefore, $\mathtt{red}(e_2)$ and $\mathtt{RED}(e_2)$ are equal. Note that red can be computed effectively since it does not reason about concrete elements directly.

In this section, we have introduced a new inductive semantics RIS, and shown that it is more precise than SIS, and that GKMTSs and MixTSs are equivalent w.r.t. RIS. RIS can be computed effectively on monotone models, which is not a limitation since monotone models are as expressive as their non-monotone counterparts.

## 5.7 Symbolic Model Checking of RIS using BDDs

In this section, we describe a symbolic algorithm RIS that implements the RIS semantics for monotone models constructed using predicate abstraction. These are the models used by existing software model checkers [GWC06b].

Our implementation is based on the following observation. Let $S$ be an abstract statespace. Then, for any monotone element of $2^S \times 2^S$, there exists a *semantically equivalent* element in $2^{\alpha[S]} \times 2^{\alpha[S]}$. For example, the monotone element $e_2$ defined in ($\star$) is semantically equivalent to $\langle\{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_4\}\rangle$.

**Theorem 5.33.** *Let $e_1 = \langle U_1, O_1\rangle$ be a monotone element of $2^S \times 2^S$, and $e_2 = \langle U_2, O_2\rangle$ be in $2^{\alpha[S]} \times 2^{\alpha[S]}$. If $U_1 \cap \alpha[S] = U_2$ and $O_1 \cap \alpha[S] = O_2$, then $e_1 \equiv_a e_2$.*

**Proof:**

This is proved by showing that $\texttt{RED}(e_1) = \texttt{RED}(e_2)$; since RED is semantics-preserving, the result holds. □

This theorem allows us to restrict the algorithm to computing sets over $\alpha[S]$ instead of sets over $S$. Another consequence of Theorem 5.33 is that the transition relations can be simplified as well, since we only need the result of the pre-image in the states of $\alpha[S]$.

**Theorem 5.34.** *Let $R^{\text{may}} \subseteq S \times S$ and $R^{\text{must}} \subseteq S \times S$ be the* may *and* must *transition relations of a monotone MixTS, respectively, and $e = \langle U, O\rangle$ be a monotone element of $2^S \times 2^S$. Define $\hat{U} \triangleq U \cap \alpha[S]$, $\hat{O} \triangleq O \cap \alpha[S]$, $\hat{R}^{\text{must}} \triangleq R^{\text{must}} \cap (\alpha[S] \times S)$, and $\hat{R}^{\text{may}} \triangleq R^{\text{may}} \cap (\alpha[S] \times \alpha[S])$. Then,*

$$\langle \mathit{pre}[R^{\text{must}}](\texttt{RED}_{\mathsf{U}}(U)), \mathit{pre}[R^{\text{may}}](\texttt{RED}_{\mathsf{O}}(O))\rangle \equiv_a \langle \mathit{pre}[\hat{R}^{\text{must}}](\texttt{RED}_{\mathsf{U}}(\hat{U})), \mathit{pre}[\hat{R}^{\text{may}}](\hat{O})\rangle$$

**Proof:**

By the definition of $\equiv_a$, the theorem is equivalent to proving that the following results hold:

$(a)\ \ \gamma(\mathit{pre}[R^{\text{must}}](\texttt{RED}_{\mathsf{U}}(U))) = \gamma(\mathit{pre}[\hat{R}^{\text{must}}](\texttt{RED}_{\mathsf{U}}(\hat{U})))$

$(b)\ \ \gamma(\overline{\mathit{pre}[R^{\text{may}}](\texttt{RED}_{\mathsf{O}}(O))}) = \gamma(\overline{\mathit{pre}[\hat{R}^{\text{may}}](\hat{O})})$

(1) We first show that (a) holds. The proof of $\gamma(\mathit{pre}[R^{\text{must}}](\texttt{RED}_{\mathsf{U}}(U))) \subseteq \gamma(\mathit{pre}[\hat{R}_1](\texttt{RED}_{\mathsf{U}}(\hat{U})))$ is shown below. For any concrete state $c$,

$$
\begin{aligned}
& c \in \gamma(\mathit{pre}[R^{\text{must}}](\texttt{RED}_{\mathsf{U}}(U))) \\
\Rightarrow\ & \exists a \in S \cdot c \in \gamma(a) \wedge a \in \mathit{pre}[R^{\text{must}}](\texttt{RED}_{\mathsf{U}}(U)) \\
\Rightarrow\ & \text{(by the definition of } pre\text{)} \\
& \exists a \in S \cdot c \in \gamma(a) \wedge \exists b \in \texttt{RED}_{\mathsf{U}}(U) \cdot R^{\text{must}}(a,b) \\
\Rightarrow\ & \text{(let } a' = \alpha(c)\text{; by the definition of } \alpha\text{)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \exists a \in S \cdot a \preceq_a a' \wedge \exists b \in \texttt{RED}_{\mathsf{U}}(U) \cdot R^{\text{must}}(a,b) \\
\Rightarrow\ & \text{(by monotonicity of the transition relations)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \exists b \in \texttt{RED}_{\mathsf{U}}(U) \cdot R^{\text{must}}(a',b) \\
\Rightarrow\ & \text{(by the definition of } \hat{R}^{\text{must}}\text{)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \exists b \in \texttt{RED}_{\mathsf{U}}(U) \cdot \hat{R}^{\text{must}}(a',b) \\
\Rightarrow\ & \text{(since } e \text{ is a monotone element, } \gamma(U) = \gamma(\hat{U})\text{)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \exists b \in \texttt{RED}_{\mathsf{U}}(\hat{U}) \cdot \hat{R}^{\text{must}}(a',b) \\
\Rightarrow\ & \text{(by the definition of } pre\text{)} \\
& c \in \gamma(a') \wedge a' \in \mathit{pre}[\hat{R}^{\text{must}}](\texttt{RED}_{\mathsf{U}}(\hat{U})) \\
\Rightarrow\ & c \in \gamma(\mathit{pre}[\hat{R}^{\text{must}}](\texttt{RED}_{\mathsf{U}}(U)))
\end{aligned}
$$

The proof of $\gamma(\mathit{pre}[R^{\text{must}}](\texttt{RED}_{\mathsf{U}}(U))) \supseteq \gamma(\mathit{pre}[\hat{R}^{\text{must}}](\texttt{RED}_{\mathsf{U}}(\hat{U})))$ follows from the definitions of $\hat{R}^{\text{must}}$ and $\hat{U}$.

(2) We now show that $\gamma(\overline{\mathit{pre}[R^{\text{may}}](\texttt{RED}_{\mathsf{O}}(O))}) = \gamma(\overline{\mathit{pre}[\hat{R}^{\text{may}}](\hat{O})})$. The proof of

$\gamma(\overline{\mathit{pre}[R^{\text{may}}](\texttt{RED}_{\mathsf{O}}(O))}) \subseteq \gamma(\overline{\mathit{pre}[\hat{R}^{\text{may}}](\hat{O})})$ is shown below. For any concrete state $c$,

$$
\begin{aligned}
& c \in \gamma(\overline{\mathit{pre}[R^{\text{may}}](\texttt{RED}_{\mathsf{O}}(O))}) \\
\Rightarrow\ & \exists a \in S \cdot c \in \gamma(a) \wedge a \in \overline{\mathit{pre}[R^{\text{may}}](\texttt{RED}_{\mathsf{O}}(O))} \\
\Rightarrow\ & \text{(by the definition of } pre\text{)} \\
& \exists a \in S \cdot c \in \gamma(a) \wedge R^{\text{may}}(a) \subseteq \overline{\texttt{RED}_{\mathsf{O}}(O)} \\
\Rightarrow\ & \text{(let } a' = \alpha(c)\text{; by the definition of } \alpha\text{)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \exists a \in S \cdot a \preceq_a a' \wedge R^{\text{may}}(a) \subseteq \overline{\texttt{RED}_{\mathsf{O}}(O)} \\
\Rightarrow\ & \text{(by monotonicity of the transition relations)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \exists a \in S \cdot R^{\text{may}}(a') \subseteq R^{\text{may}}(a) \subseteq \overline{\texttt{RED}_{\mathsf{O}}(O)} \\
\Rightarrow\ & \text{(by the definition of } \hat{R}^{\text{may}}\text{, } R^{\text{may}}(a') \cap \alpha[S] = \hat{R}^{\text{may}}(a')\text{)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \hat{R}^{\text{may}}(a') \subseteq (\overline{\texttt{RED}_{\mathsf{O}}(O)} \cap \alpha[S]) \\
\Rightarrow\ & \text{(since } e \text{ is a monotone element, } \forall s \in \alpha[S] \cdot s \in \texttt{RED}_{\mathsf{O}}(O) \Leftrightarrow s \in O\text{ )} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \hat{R}^{\text{may}}(a') \subseteq (\overline{O} \cap \alpha[S]) \\
\Rightarrow\ & \text{(by the definition of } \hat{O}\text{)} \\
& c \in \gamma(a') \wedge a' \in \alpha[S] \wedge \hat{R}^{\text{may}}(a') \subseteq \alpha[S] \setminus \hat{O} \\
\Rightarrow\ & \text{(by the definition of } pre\text{)} \\
& c \in \gamma(a') \wedge a' \in \gamma(\overline{\mathit{pre}[\hat{R}^{\text{may}}](\hat{O})}) \\
\Rightarrow\ & c \in \gamma(\overline{\mathit{pre}[\hat{R}^{\text{may}}](\hat{O})})
\end{aligned}
$$

The proof of $\gamma(\overline{\mathit{pre}[R^{\text{may}}](\texttt{RED}_{\mathsf{O}}(O))}) \supseteq \gamma(\overline{\mathit{pre}[\hat{R}^{\text{may}}](\hat{O})})$ is similar to the one above. □

The algorithm RIS is shown in Figure 5.4. It uses BDDs to symbolically represent and manipulate sets of states and transition relations. Functions that are prefixed with "BDD" are the standard BDD operations. The algorithm works recursively on the structure of the input formula $\varphi$. The fixpoints are computed as usual, by iterating until convergence. We describe the details of the implementation below.

Let $P = \{p_1, \ldots, p_n\}$ be a set of $n$ predicates. Recall that Mon$(P)$ denotes the set of monomials over $P$, and MT$(P)$ — the set of minterms over $P$. Furthermore, $\alpha[\text{Mon}(P)] = \text{MT}(P)$. The input to the algorithm is a MixTS model $\langle M, L_M \rangle$, s.t. $M = (S, R^{\text{may}}, R^{\text{must}})$, $S = \text{Mon}(P)$, and $L_M(s) = Lit(s)$, and an $L_\mu$ property $\varphi$. Without loss of generality, by

```
1:  global var Rmay, Rmust : BDD
2:  func RIS(Expr φ) : BDD
3:    match φ with
4:      ATOMIC(p) : return ABSV(BDDVAR("p"),
                               BDDVAR("p"))
5:      ¬ψ : return ABSNOT(RIS(ψ))
6:      ψ1 ∧ ψ2 : return ABSAND(RIS(ψ1), RIS(ψ2))
7:      ψ1 ∨ ψ2 : return ABSOR(RIS(ψ1), RIS(ψ2))
8:      ◇ψ : return ABSPRE(Rmay, Rmust, RIS(ψ))
9:      μψ : return RIS_lfp(ψ)
10:     νψ : return RIS_gfp(ψ)
11:
12: func ABSV(BDD u, BDD o) : BDD
13:   sel := BDDVAR("sel")
14:   return BDDITE(sel, u, o)
15:
16: func ABSO(BDD v) = v[0/sel]
17: func ABSU(BDD v) = v[1/sel]
18: func ABSAND(BDD v1, BDD v2) = BDDAND(v1, v2)
19: func ABSOR(BDD v1, BDD v2) = BDDOR(v1, v2)
20: func ABSEQ(BDD v1, BDD v2) = BDDEQ(v1, v2)
21:
22: func ABSNOT(BDD v) : BDD
23:   o := ABSO(v), u := ABSU(v)
24:   return ABSV(BDDNOT(o), BDDNOT(u))
25:
26: func ABSREDU(BDD v) : BDD
27:   if (BDDISCONST(v)) return v
28:   b := BDDROOTVAR(v), h := UVAR(b)
29:   T := ABSREDU(v[1/b]), F := ABSREDU(v[0/b])
30:   tmp := BDDITE(b, T, F)
31:   return BDDITE(h, BDDAND(T, F), tmp)
32:
33: func ABSPRE(BDD Rmay, BDD Rmust, BDD v) : BDD
34:   o := ABSO(v), u := ABSREDU(ABSU(v))
35:   return ABSV(BDDPRE(Rmust, u), BDDPRE(Rmay, o))
```

Figure 5.4: The RIS algorithm and its supporting functions.

Theorem 5.34, we assume that the transition relations are restricted s.t. $R^{\text{may}} \subseteq \text{MT}(P) \times \text{MT}(P)$, and $R^{\text{must}} \subseteq \text{MT}(P) \times \text{Mon}(P)$.

The algorithm uses the following sets of BDD variables: $B = \{b_i \mid p_i \in P\}$ – the current state Boolean variables, $B' = \{b'_i \mid b_i \in B\}$ – the next state Boolean variables, $H = \{h_i \mid p_i \in P\}$ – the current state unknown variables, and $H' = \{h'_i \mid h_i \in H\}$ – the next state unknown variables. In what follows, we do not distinguish between the BDDs and the corresponding propositional formulas.

A set of minterms $X \subseteq \text{MT}(P)$ is encoded by a propositional formula over $B$, as usual. For example, let $P = \{p_1, p_2, p_3\}$. Then $b_1 \wedge \neg b_2$ encodes the set $\{p_1 \wedge \neg p_2 \wedge p_3, p_1 \wedge \neg p_2 \wedge \neg p_3\}$. A set of monomials $X \subseteq \text{Mon}(P)$ is encoded by a formula over $B \cup H$. Intuitively, for a monomial $m$, a variable $h_i$ indicates whether $p_i$ is present in $m$, and a variable $b_i$ specifies the

polarity of the occurrence. Formally, the encoding is

$$\bigvee_{m \in X} \left( \left( \bigwedge_{p_i \in Lit(m)} \neg h_i \right) \wedge b_i \right) \wedge \left( \left( \bigwedge_{\neg p_i \in Lit(m)} \neg h_i \right) \wedge \neg b_i \right) \wedge \left( \bigwedge_{p_i \in P \setminus Lit(m)} h_i \right) \right)$$

For example, $(\neg h_1 \wedge b_1) \wedge (\neg h_2 \wedge \neg b_2) \wedge h_3$ represents a singleton set $\{p_1 \wedge \neg p_2\}$.

An abstract value $e = \langle U, O \rangle$ is encoded in a single BDD by a formula $(\texttt{sel} \wedge U) \vee (\neg \texttt{sel} \wedge O)$, where `sel` is a designated BDD variable. This encoding is implemented by function ABSV. The $U$ and $O$ elements of value $e$ are extracted using ABSU and ABSO, respectively. Abstract intersection (ABSAND), union (ABSOR), and equality (ABSEQ) are done using the corresponding BDD operations. Abstract negation (ABSNOT) is implemented following its definition on Page 33.

The may transition relation $R^{\text{may}} \subseteq \text{MT}(P) \times \text{MT}(P)$ is encoded by a formula over $B \cup B'$ as usual. Similarly, the must relation $R^{\text{must}} \subseteq \text{MT}(P) \times \text{Mon}(P)$ is encoded by a formula over $B \cup B' \cup H'$, where the primed variables are used to encode the destination state. For example, a *must* transition from a state $(p_1 \wedge p_2 \wedge p_3)$ to a state $(p_1 \wedge \neg p_2)$ is represented by $(b_1 \wedge b_2 \wedge b_3) \wedge ((\neg h'_1 \wedge b'_1) \wedge (\neg h'_2 \wedge \neg b'_2) \wedge h'_3)$.

Function ABSREDU implements the $\texttt{red}_\cup$ reduction operator of Section 5.6.3. It takes a set of minterms as input, and returns a set of monomials for the computation of pre-image over *must* transitions. A monomial is added to the output iff its upset is contained in the input. The implementation of ABSREDU uses the following observation: let $Q \subseteq \text{MT}(P)$ be a set of minterms, and $a \in \text{Mon}(P)$. If $a \in \text{MT}(P)$, then $\uparrow a = \{a\}$, and $\uparrow a \subseteq Q \Leftrightarrow a \in Q$; otherwise, some predicate $p$ is not present in $a$, and in this case $\uparrow a \subseteq Q$ iff $\uparrow(a \wedge p) \subseteq Q$ and $\uparrow(a \wedge \neg p) \subseteq Q$. For example, suppose $P = \{p_1, p_2, p_3\}$ and $Q = \{p_1 \wedge p_2 \wedge p_3, p_1 \wedge p_2 \wedge \neg p_3\}$. For the monomial $a = p_1 \wedge p_2$, we have that $\uparrow a \subseteq Q$ because $\uparrow(a \wedge p_3) = \uparrow(p_1 \wedge p_2 \wedge p_3) = \{p_1 \wedge p_2 \wedge p_3\} \subseteq Q$ and $\uparrow(a \wedge \neg p_3) = \uparrow(p_1 \wedge p_2 \wedge \neg p_3) = \{p_1 \wedge p_2 \wedge \neg p_3\} \subseteq Q$. Function ABSREDU applies this reasoning recursively on the input diagram, using function UVAR to find a variable $h_i \in H$ for each variable $b_i \in B$. Function ABSPRE implements the pre-image computation based on Theorem 5.34.

**Theorem 5.35.** *For a monotone MixTS $\mathcal{M}$ and $\varphi \in L_\mu$, algorithm* RIS($\varphi$) *in Figure 5.4 returns the symbolic representation of* $\|\varphi\|_r^{\mathcal{M}}$.

**Proof:**

The proof follows from Theorem 5.32, Theorem 5.33, and Theorem 5.34. In particular, Theorem 5.33 is used to show that in the interpretation of $\Diamond\varphi$ in Definition 5.27, removing the application of RED after $pre_{\forall}$ and $pre_{\exists}$ does not affect precision. □

The main difference between the symbolic implementations of SIS and our RIS is the extra ABSREDU operation in function ABSPRE (line 29 in Figure 5.4). ABSREDU is similar to existential quantification (BDDEXISTS) of BDDs, with one exception: BDDEXISTS uses BDDOR in each iteration, but ABSREDU uses one BDDAND and two BDDITE operations. Thus, ABSREDU has the same complexity as BDDEXISTS, and symbolic implementations of RIS and SIS also have the same complexity. This means that the extra precision of RIS comes "for free", without a penalty in complexity.

## 5.8 Experiments

To empirically evaluate the cost and performance of RIS versus SIS, we have implemented symbolic algorithms for computing both of them using the CUDD [Som01] library, and analyzed reachability and non-termination properties over a realistic model. While our algorithm in Figure 5.4 can analyze any $\mu$-calculus formula, our experiments considered just reachability and non-termination properties because of their practical interest.

For the model, we used a template program $\texttt{Prog}_1$ based on an example from [SG04], which uses $n$ ($n$ is a natural number) integer variables $\{\texttt{x}[0], \texttt{x}[1], \ldots, \texttt{x}[\texttt{n}-1]\}$, and is built out of a sequence of $n$ blocks. Figures 5.5(a) and 5.5(b) show the code for each of the first $n-1$ blocks, and the code for the last block, respectively. The method of [GC06] was applied to build an abstract MixTS using the set of predicates

$$\{\texttt{x}[0] > 0, \texttt{x}[1] > 0, \ldots, \texttt{x}[\texttt{n}-1] > 0\} \cup \{\texttt{odd}(\texttt{x}[0]), \texttt{odd}(\texttt{x}[1]), \ldots, \texttt{odd}(\texttt{x}[\texttt{n}-1])\}$$

(a)

```
if (x[i]>5)
  x[i]=x[i]+1
else if (x[i]>0)
  x[i]=x[i]+2
else
  x[i]=x[i]-2

while  (x[i]>0)
  if (odd(x[i]))
    x[i]=-1
   else
    x[i]=x[i]+1
```

(b)

```
if (x[n-1]>5)
  x[n-1]=x[n-1]+1
else if (x[n-1]>0)
  x[n-1]=x[n-1]+2
else
  x[n-1]=x[n-1]-2

while  (x[n-1]>0)
  if (odd(x[n-1]))
    x[n-1]=-1
   else
    x[n-1]=x[n-1]+1
L:
while  (x[n-1]<=0)
  x[n-1]=x[n-1]-1
END:
```

(c)

```
if (nondet)
  x[i]=x[i]+1
  if (nondet)
    x[i]=x[i]+1
   else
    x[i]=x[i]*x[i]-10
else
  x[i]=x[i]*x[i]-10

if (x[i]>0)
  x[i]=x[i]+1
else
  x[i]=x[i]-1
```

Figure 5.5: Code examples for experiments. `nondet` denotes non-deterministic choice.

We model checked the following reachability (least fixed-point) and non-termination (greatest fixed-point) properties w.r.t. the standard and the reduced semantics:

$$\texttt{Prop}_1 \ : EF(pc = \texttt{L})$$

$$\texttt{Prop}_2 \ : EG(pc \neq \texttt{END})$$

$$\texttt{Prop}_3 \ : EG(pc \neq \texttt{END} \wedge (\texttt{x}[0] > 0 \vee \texttt{x}[1] > 0 \vee \cdots \vee \texttt{x}[\texttt{n} - 1] > 0)).$$

For both SIS and RIS, we measured the size of the abstract models using the number of BDD nodes, the total analysis time, the number of iterations of the fixpoint computation, and the time spent in the ABSREDU operation for RIS. To compare the precision of the results, we considered two sets of initial states

$$\texttt{I}_1 : (\texttt{x}[0] \leq 0 \wedge \texttt{x}[1] \leq 0 \wedge \cdots \wedge \texttt{x}[\texttt{n} - 1] \leq 0)$$

$$\texttt{I}_2 : (\texttt{x}[0] > 0 \wedge \texttt{x}[1] > 0 \wedge \cdots \wedge \texttt{x}[\texttt{n} - 1] > 0) \ .$$

| | $n$ | SIS | | | | RIS | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Model Size | 100 | 370,070 | | | | 216,689 | | | | |
| | 200 | 1,460,270 | | | | 853,389 | | | | |
| | 250 | 2,275,196 | | | | 1,329,215 | | | | |
| Prop. | $n$ | Analysis (sec.) | Iter. | $\mathtt{I}_1$ | $\mathtt{I}_2$ | Analysis (sec.) | ABSREDU (sec.) | Iter. | $\mathtt{I}_1$ | $\mathtt{I}_2$ |
| $\mathtt{Prop}_1$ | 100 | 2.20 | 301 | t | m | 3.60 | 0.74 | 401 | t | t |
| | 200 | 15.36 | 601 | | | 27.77 | 6.45 | 801 | | |
| | 250 | 28.92 | 751 | | | 55.19 | 13.40 | 1001 | | |
| $\mathtt{Prop}_2$ | 100 | 3.60 | 203 | t | m | 0.03 | $< 10^{-4}$ | 2 | t | t |
| | 200 | 27.16 | 403 | | | 0.12 | $< 10^{-4}$ | 2 | | |
| | 250 | 54.62 | 503 | | | 0.19 | $< 10^{-4}$ | 2 | | |
| $\mathtt{Prop}_3$ | 100 | 33.96 | 400 | f | f | 21.24 | 4.5 | 400 | f | f |
| | 200 | 395.24 | 800 | | | 258.72 | 42.44 | 800 | | |
| | 250 | 1108.67 | 1000 | | | 546.88 | 101.20 | 1000 | | |

Table 5.1: Experimental results for SIS and RIS over $\mathtt{Prog}_1$.

and checked whether conclusive results can be obtained over them.

The results are summarized in Table 5.1, where t, f and m denote *true*, *false*, and *unknown*, respectively. The top part of the table shows that RIS models enjoy significantly smaller encodings than their SIS counterparts, due to restricted transition relations (see Theorem 5.34). RIS is more precise than SIS: for the two sets of initial states, RIS produces conclusive results for both of them w.r.t. the three properties being checked, whereas SIS cannot decide whether $\mathtt{Prop}_1$ and $\mathtt{Prop}_2$ hold in $\mathtt{I}_2$. As expected, the extra precision of RIS does not cause a complexity penalty: the experiments show that the increases of the analysis time w.r.t. the size of the models for both RIS and SIS are comparable. In all of the cases, the time spent in ABSREDU, which represents the main difference between the two semantics, comprises roughly 20% - 25% of the total time.

Note that RIS and SIS may require different numbers of iterations of fixpoint computation:

| | $n$ | SIS | | | | RIS | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Model Size | 100 | 245,584 | | | | 145,284 | | | | |
| | 200 | 971,062 | | | | 570,462 | | | | |
| | 250 | 1,513,796 | | | | 888,046 | | | | |
| Prop. | $n$ | Analysis (sec.) | Iter. | $\mathtt{I}_1$ | $\mathtt{I}_2$ | Analysis (sec.) | ABSREDU (sec.) | Iter. | $\mathtt{I}_1$ | $\mathtt{I}_2$ |
| $\mathtt{Prop}_4$ | 100 | 0.48 | 603 | | | 0.27 | $< 10^{-4}$ | 403 | | |
| | 200 | 2.15 | 1203 | m | t | 0.97 | $< 10^{-4}$ | 803 | t | t |
| | 250 | 3.46 | 1503 | | | 1.44 | 0.01 | 1003 | | |

Table 5.2: Experimental results for SIS and RIS over $\mathtt{Prog}_2$.

in the above experiments, RIS required more iterations than SIS for the reachability property $\mathtt{Prop}_1$, but less iterations than SIS for the non-termination property $\mathtt{Prop}_2$. These differences are determined by the structure of the model and by the fixpoint type (least or greatest) being computed.

As another example, we checked the reachability property on a different program, $\mathtt{Prog}_2$, built of a sequence of $n$ blocks. The code for the $i$th block is shown in Figure 5.5(c). An abstraction of the template is built using the set of predicates

$$\{\mathtt{x}[0] > 0, \mathtt{x}[1] > 0, \ldots, \mathtt{x}[\mathtt{n}-1] > 0\}$$

The property checked was $\mathtt{Prop}_4 : EF(pc = \mathtt{END})$, where END is the last location of $\mathtt{Prog}_2$. model checking was evaluated on the same initial sets of states, $\mathtt{I}_1$ and $\mathtt{I}_2$. The results are summarized in Table 5.2. In this case, while still more precise, RIS requires fewer iterations than SIS.

These experiments suggest that using the more precise RIS semantics improves the overall performance of model checking, making it a viable alternative to SIS in practice.

## 5.9 Related Work

*Consistency*. In this chapter, we investigated partial TSs and models from the perspective of abstract model checking. Partial TSs are also used as specifications of a system's behavior [LT88, LNW07]. In this case, semantic consistency is replaced by implementability. A partial transition system $M$ is *implementable* iff there exists a BTS $B$ that refines $M$ through some mixed simulation. Such a BTS is called *an implementation*. There is a subtle, but crucial, difference between implementability and semantic consistency as defined in this chapter. We assume that the statespace of an abstract transition system is an abstract domain, and that it is related to the concrete domain by a given soundness relation $\rho$. In our case, a partial TS $M$ is semantically consistent iff there exists a BTS that refines $M$ via this $\rho$. On the other hand, the definition of implementability leaves the choice of the mixed simulation relation open. Thus, semantic consistency is stronger than implementability.

For example, the MixTS $M_2$ in Figure 5.1 is not semantically consistent. It is, however, implementable. Let $B$ be a BTS $(\mathbb{Z}, R)$, where $\mathbb{Z}$ is the set of integers, and $R$ is defined as follows:

$$\begin{aligned} R \triangleq\ & \{(x, x') \mid (x > 0 \wedge \mathit{odd}(x) \wedge x' = 2)\} \cup \\ & \{(x, x') \mid (x > 0 \wedge \mathit{even}(x) \wedge x' = -3)\} \cup \\ & \{(x, x') \mid (x > 0 \wedge \mathit{even}(x) \wedge x' = -2)\} \cup \\ & \{(x, x') \mid (x < 0 \wedge x' = -3)\} \quad . \end{aligned}$$

Then, $B$ refines $M_2$ through the following mixed simulation relation:

$$\{(c, a_1) \mid c > 0 \wedge \mathit{odd}(c)\} \cup \{(c, a_2) \mid c > 0 \wedge \mathit{evn}(c)\} \cup$$
$$\{(c, a_3) \mid c \leq 0 \wedge \mathit{odd}(c)\} \cup \{(c, a_4) \mid c \leq 0 \wedge \mathit{evn}(c)\}$$

Note that in this case, no concrete state in $B$ is approximated by both $a_1$ and $a_2$. Therefore, the source of inconsistency discussed in Section 5.4 does not exist.

In [HJS04], Huth et al. provided the *mix condition* (MC) on MixTSs to ensure implementability. A MixTS $M = \langle S, R^{\text{may}}, R^{\text{must}} \rangle$ satisfies MC iff for all $(a, b) \in R^{\text{must}}$, there exists some $b' \in S$ such that $b'$ refines $b$, and $(a, b') \in R^{\text{must}} \cap R^{\text{may}}$. For example, the MixTS $M_2$ in Figure 5.1 satisfies this condition, whereas $M_4$ does not. However, $M_2$ is semantically inconsistent, and $M_4$ is consistent. Therefore, MC is neither sufficient nor necessary for semantic consistency.

The complexity of deciding implementability of a partial transition system is EXPTIME-complete [AHL$^+$08, AHL$^+$09, Ant08]. On the other hand, semantic consistency can be decided in time polynomial in the size of the system; this is immediate from Theorem 5.17. This result is not surprising since semantic consistency is stronger than implementability.

Huth et al. showed that the KMTS models are logically consistent [HJS01]. To ensure logical consistency of GKMTSs, de Alfaro et al. defined the condition that requires that every destination of a *must* hyper-transition intersects with the destination of a *may* transition from the same state [dAGJ04]. This can be viewed as an analogue of the condition $R^{\text{must}} \subseteq R^{\text{may}}$ required by KMTSs. In this chapter, we showed that such a condition is not necessary for logical consistency. We fixed this problem by defining a relaxed structural condition which captures both logical consistency and semantic consistency of partial models.

Other forms of partial model consistency, in addition to the one based on mixed simulation, are possible, e.g. [LNW07]. For example, a partial model may be built for abstract model checking of temporal logic properties without the next operator. Exploring connections between semantic and logical consistency in this case and providing algorithms for deciding them are interesting questions which we leave for future work.

*Expressiveness.* The work of Godefroid and Jagadeesan [GJ03], and Gurfinkel and Chechik [GC05] showed that the models in the KMTS family have the same expressive power and are equally precise for SIS. Dams and Namjoshi [DN05] showed that the three families considered in this chapter are subsumed by tree automata. We completed the picture by proving that the three families are equivalent as well. Specifically, we showed that KMTSs, MixTSs and GKMTSs

are relatively complete (in the sense of [DN05]) with one another.

We did not consider Hyper TSs (HTSs) [SG06] which allow for both *must* and *may* hyper-transitions. As pointed out in [SG06], *may* hyper-transitions can be eliminated by increasing the abstract statespace, making HTSs exactly as expressive as GKMTSs.

Our results bring forth several interesting research directions. Since the three modeling formalisms are equally expressive, it would be interesting to study how to relate the results of model checking w.r.t. thorough semantics for one formalism, e.g., for KMTSs [BG00, GP09], to the ones for another formalism. Another direction is formalizing our translations within the abstract interpretation framework using Galois connections [CC92].

*Reduced Inductive Semantics*. Our reduction operator RED is an instance of normalization from Abstract Interpretation [CC92]. It is often used to provide a canonical representation of equivalent abstract properties. The symbolic implementation ABSREDU is similar to the semantic minimization of 3-valued propositional formulas [RLS02].

Regarding the ability to improve model checking results, the reduction operator is similar to the focus and defocus operations defined in [DN04]. According to the definition of RED, a formula holds in an abstract state $a$ if (i) $\gamma(a)$ can be split into (i.e., focused) different parts approximated by more precise states than $a$, and the formula holds in each of these states, or (ii) $\gamma(a)$ can be covered (i.e., defocused) by a set approximated by a state less precise than $a$, and the formula holds in it. In particular, if the partial model is monotone, then the reduction operator resembles the focus operation only.

For a partial modeling formalism, the ability to support the monotonic abstraction refinement framework allows us to define a best model over an abstract statespace s.t. model checking on it is more precise than on other models over the statespace. In the context of SIS, as shown in [SG04], KMTSs is inappropriate for monotonic abstraction refinement — extra *may* transitions required by the condition $R^{\text{must}} \subseteq R^{\text{may}}$ introduce a loss of precision, and therefore, a best KMTS model over an abstract statespace may not exist. However, this is not a problem for MixTSs [DGG97, GWC06a] which support monotonic abstraction refinement by allowing

must-only transitions. GKMTSs achieve the same goal by using *must* hyper-transitions [SG04], which essentially ensure that no extra *may* transitions are added. The following theorem shows that our new inductive semantics, RIS, preserves the precision order of partials models w.r.t. SIS. Therefore, the best abstract model for SIS is also the best one for RIS, and both MixTSs and GKMTSs still support monotonic abstraction refinement under RIS.

**Theorem 5.36.** *Let $\mathcal{M} = \langle M, L\rangle$ and $\mathcal{M}' = \langle M', L'\rangle$ be two partial models, where $M = \langle S, R^{\text{may}}, R^{\text{must}}\rangle$ and $M' = \langle S, R^{\text{may}'}, R^{\text{must}'}\rangle$ are two (underlying) transition systems defined over the same abstract statespace $S$. Then, if $\mathcal{M}$ is less precise than $\mathcal{M}'$ under SIS, i.e., $\forall\varphi \in L_\mu \cdot \|\varphi\|_i^{\mathcal{M}} \preceq_a \|\varphi\|_i^{\mathcal{M}'}$, $\mathcal{M}$, then is also less precise than $\mathcal{M}'$ under RIS.*

**Proof:**

The proof is by structural induction. In particular, the inductive case for $\Diamond\varphi$ follows from the definition of RED and the monotonicity of *preimage*. □

Regarding precision of model checking, one interesting area is the investigation of self-minimizing temporal formulas whose inductive and thorough semantics coincide [GH05]. Through a semantic minimization process, every $L_\mu$ formula can be transformed into an equivalent formula that is self-minimizing, but may be exponentially larger than the original one. Several results along this line, based on the comparison of SIS and thorough semantics, have been reported, e.g., [GH05, GC05, NGC06, AH06]. In this chapter, we have used a reduction operator to improve precision of inductive semantics based on the exploration of approximation ordering over abstract domain. Our approach is orthogonal to semantic minimization. For example, consider the model $\mathcal{M}_5$ defined in Section 5.6.1 (its transition system $M_5$ is shown in Figure 5.2) and the formula $\psi \triangleq EF(\neg p \wedge q)$, where $p$ and $q$ denote predicates $(x > 0)$ and $odd(x)$, respectively. $\psi$ is self-minimizing. However, its value in $a_1$ is unknown under SIS, but is true under RIS. We leave further investigation of the relation between RIS and semantic minimization of temporal logic formulas for future work.

We have shown that symbolic model checking of RIS and SIS have the same complexity. An interesting question left for future study is whether there exists an inductive semantics that

is more precise than RIS, and whether it can be symbolically model checked with the same complexity as RIS.

## 5.10 Conclusion

Several types of partial transition systems have been developed over the years to support abstract model checking of complex temporal formulas. Some were claimed to be more precise; some had a more efficient decision procedure; some were more succinct. In this chapter, we have studied these formalisms, partitioned into three families – KMTSs, MixTSs and GKMTSs. We have compared them w.r.t. two fundamental ways of using partial transition systems: as objects for abstracting concrete systems, and as models for checking temporal properties.

Specifically, we studied the connection between semantic and logical consistency of partial transition systems, which is necessary to ensure meaningful abstract model checking. We showed that these notions are not equivalent. However, we proved that they coincide for monotone partial transition systems and provided an effective structural condition which is necessary and sufficient to guarantee consistency.

We have also compared the expressive power of the three families of partial transition systems w.r.t. their ability to capture abstractions. We showed, by defining semantics-preserving transformations between the formalisms, that while there are structural differences, all three formalisms are equally expressive. Thus, neither hyper-transitions nor restrictions on *may* and *must* transitions affect expressiveness. They do, of course, affect the succinctness of the resulting transition systems.

We then turned to looking at the power of these formalisms w.r.t. the cost and precision of model checking. We have introduced a new inductive semantics, RIS, for partial transition systems and showed not only that it is more precise than the standard semantics, SIS, but also that model-checking under this semantics for MixTSs and GKMTSs has the same results. We

have further described a symbolic implementation of model checking w.r.t. RIS. The outcome is an algorithm that combines the symbolic encoding of MixTSs with the model checking precision of GKMTSs. The symbolic algorithm was evaluated empirically, and our preliminary experiments suggest that RIS should be a good alternative to SIS for predicate abstraction-based model checkers. We leave further experimental comparisons between the two semantics for future work.

We hope that the results of our investigation help clear out the confusion about the expressive power of the different partial transition systems and enable their increasing usage as underlying formalisms for abstract model checking.

# Chapter 6

# Conclusion

In this chapter, we summarize the contributions made in this thesis, and discuss limitations of our work and future research directions.

## 6.1 Summary of The Thesis

In this thesis, we have studied abstraction in model checking based on exact-approximation, which combines over- and under-approximations, allowing us to verify and refute properties in the same abstraction framework. Our work is driven by problems from both practical and theoretical aspects of exact-approximation.

We started with symmetry reduction that exploits symmetry in programs for abstraction. It can be seen as a strong exact-approximation technique, performing abstract model checking using a symmetry-reduced structure that is bisimilar to the original program. In Chapter 3, we studied symmetry reduction with respect to full virtual symmetry, addressing two challenges of effectively using it in practice, i.e., identification of virtually symmetry and support for symbolic model checking. We first characterized symmetry reduction from the perspective of abstraction. Based on that, we reduced identifying full virtual symmetry to satisfiability of quantifier-free Presburger formulas built directly from program specifications. This satisfiability problem is NP-complete and can be solved by existing decision procedures. We also

extended counter abstraction to fully virtually symmetric programs, which avoids the bottleneck problem of building orbit relations in symbolic symmetry reduction.

A software model checker often uses predicates over program variables for data abstraction. In our previous work, we have developed a software model checker YASM based on exact-approximating semantics of predicate abstraction, which builds abstract models that are exact-approximation of original programs, supporting proving and disproving properties with equal effectiveness. In Chapter 4, we extended YASM to reachability and non-termination analysis of recursive programs. To avoid explicitly dealing with call stacks, we proposed a stack-free program semantics that effectively reduce to the analysis of reachability and non-termination of recursive programs to that of non-recursive ones. This allows us to reuse existing abstract analysis in YASM to handle recursive programs. We also developed on-the-fly algorithms that improve the performance of the analysis.

Exact-approximation can be achieved using different partial modeling formalisms. Our third study focused on the analysis of three families of partial transition systems for this, represented by KMTSs, MixTSs, and GKMTSs. In Chapter 5, we investigated these formalisms from two fundamental ways of using them – as objects for abstracting concrete programs, and as models for checking temporal properties. We first proved equivalence between semantic and logical consistency of partial transition systems over the class of monotone ones, and provided a structural condition that is both necessary and sufficient to guarantee consistency. We then developed semantic-preserving translations between KMTSs, MixTSs, and GKMTs, which shows that despite the structural difference, the three families of formalisms have the same abstraction ability. We also defined a new inductive semantics of temporal logic for them, which is more precise than the standard one. Based on that, we developed an algorithm that combines the benefits of a symbolic encoding of MixTSs with a better model checking precision of GKMTSs.

Abstraction in model checking uses a smaller abstract model to analyze properties of computer programs. Exact-approximation provides a uniform abstraction framework for proving

correctness and detecting errors. Choice of designing components for abstract model checking depends on the programs and the properties being analyzed. In this thesis, we have investigated abstract analysis of virtually symmetric programs and extended exact-approximating predicate abstraction to recursive programs, which increases the applicability of existing exact-approximation techniques. We also reported the results of the analysis of partial modeling formalisms, which provides a better understanding of exact-approximating abstraction framework and its application in practice.

## 6.2 Limitations and Future Work

In this section, we discuss the limitations of our work and point out future research directions.

### 6.2.1 Extended Symmetry Reduction

In this thesis, we have developed techniques for identification and counter abstraction of full virtual symmetry. Extending them to handle industrial-sized programs is still a challenge. A major limitation of our techniques is that the specification language is too restrictive, where transition guards can only be expressed using counters of local process states. It is of practical interest to investigate how to extend our techniques to more general specification languages. For example, counter abstraction has been applied to fully symmetric programs where processes communicate using shared variables [EW03]. We plan to explore this for full virtual symmetry as well.

The symmetry reduction techniques we studied in this thesis is induced by a single permutation group. We have shown that a program is symmetric with respect to this group if and only if all the local transitions in the program are symmetric with respect to it. That is, the symmetric relations between processes needs to be preserved on every local transition. Such restriction may be relaxed by considering *transition dependent* symmetry reduction (e.g., [SWZ07, Wah07]). In this case, symmetric relations between processes can dynamically

change over local transitions. Therefore, we can apply different permutation groups to a program, which enables better statespace reduction than the static one using a single group. We leave further investigation of combining such symmetry reduction with virtual symmetry for future work.

We have only investigated symmetry reduction techniques based on process symmetry, where symmetry is induced by permutations of process indices. In practice, symmetry can also be induced by data [ID96, EW05], where symmetry permutations act directly on the values of variables. For example, Murphi [ID96] defines a special scalarset data structure in its description language, and requires only symmetric operations over scalarset variables. This ensures that permutations of these variables in all states correspond to an automorphism of the statespace. In the future, we would like to extend our investigation of virtual symmetry reduction to data symmetry.

To ensure a bisimilar symmetry-reduced quotient, we have required that the transition system of the program be invariant under symmetry permutations. An interesting extension of this is only to require that the transitive closure of the transition system be invariant under permutations, which is called *architectural symmetry* [TW09]. Although the quotient structure induced by architectural symmetry is not bisimilar to the original program, it preserves many interesting properties, e.g., reachability. Architectural symmetry extends space reduction with respect to the full symmetry group to programs that are not even fully virtually symmetric. We would like to explore how to identify and apply counter abstraction to those programs. Our techniques for full virtual symmetry are based on the analysis of individual transitions in a program. Since architectural symmetry is defined based on the transitive closure of transition relations rather than individual transitions, we need to look for other directions to handle architectural symmetry.

### 6.2.2 Termination and Non-Termination Analysis

We have developed abstract analysis of the non-termination property of programs. Since exact-approximation supports both verification and refutation, when non-termination checking fails,

it means that the program terminates. However, our abstract analysis is based on predicate abstraction. It is known that, as a finite state abstraction, the ability of predicate abstraction for proving termination is limited. The reason is that any computation of a program that is longer than the number of abstract states results in an abstract computation containing a loop. Therefore, we cannot always prove termination using predicate abstraction. Because of this limitation, when analyzing non-termination of a program, our analysis may get stuck on the terminating part of the program.

To overcome this limitation, it is necessary to develop solutions for both termination and non-termination analysis. One approach is using abstraction framework that is complete for the modal $\mu$-calculus [DN04]. For example, Fecher and Huth proposed an abstraction framework that extends predicate abstraction to ranked predicate abstractionin [FH06] so that liveness properties including termination can be analyzed as well. They showed that ranked predicate abstractions are increment and thus can be possibly integrated with counterexample-guided abstraction refinement. It is an interesting future work to develop refinement heuristics that are appropriate for termination and non-termination analysis within this framework.

Another approach is to synthesize the existing techniques for termination and non-termination analysis. Recently, there has been a lot of work on termination analysis based on automatically synthesizing ranking functions for a program [PR04, PR05, CPR06a, BCC$^+$07]. These methods are biased towards proving termination. When termination analysis fails, they require users to manually check where a program is non-terminating. We would like to develop techniques to combine this analysis with non-termination analysis into a single framework so that the two analysis can benefit from each other. One possible direction that we intend to explore is based on the approach used by TERMINATOR [CPR06a, CPR06b], where through a program transformation, termination of the original program is equivalently analyzed using over-approximating predicate abstraction over a transformed program. This approach provides a way to combine the termination analysis methods above with predicate abstraction. Following this approach, we would like to develop analysis for both termination and non-termination by applying exact-

approximation over the transformed program.

### 6.2.3 Partial Modeling Formalisms

In this thesis, we have defined a more precise reduced inductive semantics than the standard one, and compared the performance of the two semantics over programs abstracted using the same predicates that we provided manually. On the other hand, in software model checking, the predicates for abstraction are usually computed through iterative refinement. Currently, we do not know whether the more precise semantics always leads to better performance of the overall abstraction refinement process, or whether the result depends on properties and programs being analyzed. We would like to design experiments to find this out.

Fairness is often required for analyzing concurrent programs with interleaving semantics. A fairness constraint associated with a transition system partitions the infinite computations in the system into fair and unfair ones. In this thesis, we did not consider fairness in abstract analysis. In particular, the partial modeling formalisms we studied are related with concrete systems through mixed simulation that does not distinguish fair and unfair computations. The notion of mixed simulation can be adapted to transition systems with fairness constraints [DN04]. In the future, we would like to investigate partial modeling formalisms with fair mixed simulation, and see how this will affect our results presented in this thesis.

### 6.2.4 Combination of Abstraction and Testing

Our work in this thesis has focused on abstract model checking that builds an abstract model for property analysis. A drawback of this abstraction approach is that it does not handle large programs with complex statements well, due to the difficulty of constructing precise abstract models of these programs. As a dynamic analysis method, testing directly executes programs, which is fast and easily scales to large programs. Recently, there has been a growing interest in combining abstract analysis and testing [KGC04, PPV05, YBS06, GHK$^+$06, BNRS08] to

take advantage of the strength of both methods. For example, SYNERGY [GHK+06] provides algorithms that combine over-approximation and testing, where testing results are used to explore program statespace and detect bugs, and abstract analysis results are used to guide the generation of test inputs and prove correctness.

We intend to study such combinations over our exact-approximation framework. In particular, we would like to see how the additional under-approximation in the framework can improve existing techniques such as the SYNERGY approach. Our observation is that although testing can also be seen as a way to under-approximate program behaviors, it is different from the abstract under-approximating analysis. The program behaviors discovered by testing are restricted by the test inputs, whereas the abstract under-approximating analysis explores all the possible program behaviors in an abstract way, which complements the ones explored by testing. Moreover, unlike testing, which can only be executed in a forward way, the abstract under-approximating analysis can be conducted backward. Such backward analysis can provide us more information about error-reachable states, which may be used to guide generation of test inputs that lead to quick discovery of program bugs. We plan to work on implementation of the above ideas in the future to obtain more efficient approaches for program analysis.

# Bibliography


[ACEM05] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. "On-the-Fly Reachability and Cycle Detection for Recursive State Machines". In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 61–76. Springer, 2005.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. "A Temporal Logic of Nested Calls and Returns". In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 467–481. Springer, March 2004.

[AH06] A. Antonik and M. Huth. "Efficient Patterns for Model Checking Partial State Spaces in CTL *intersection* LTL". *Electronic Notes in Theoretical Computer Science*, 158:41–57, 2006.

[AHL+08] A. Antonik, M. Huth, K. G. Larsen, U. Nyman, and A. Wasowski. "Complexity of Decision Problems for Mixed and Modal Specifications". In *Proceedings of the 11th International Conference of Foundations of Software Science and Computational Structures (FOSSACS'08)*, volume 4962 of *LNCS*, pages 112–126, March 2008.

[AHL+09] A. Antonik, M. Huth, K. G. Larsen, U. Nyman, and A. Wasowski. "EXPTIME-complete Decision Problems for Modal and Mixed Specifications". *Electronic*

*Notes in Theoretical Computer Science*, 242(1):19–33, 2009.

[AM04] R. Alur and P. Madhusudan. "Visibly Pushdown Languages". In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04)*, pages 202–211. ACM, June 2004.

[Ant08] A. Antonik. *"Decision Problems for Partial Specifications: Empirical and Worst-Case Complexity"*. PhD thesis, Imperial College, London, UK, September 2008.

[ARMS02] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. "PBS: A Backtrack Search Pseudo-Boolean Solver". In *Proceedings of Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, pages 346–353, Cincinnati, Ohio, 2002.

[Bal04] T. Ball. "Formalizing Counterexample-Driven Refinement with Weakest Preconditions". Technical Report 134, Microsoft Research, 2004.

[BB04] C. Barrett and S. Berezin. "CVC Lite: A New Implementation of the Cooperating Validity Checker". In *Proceedings of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 515–518, Boston, MA, July 2004. Springer.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. "Property Preserving Simulations". In *Proceedings of the 4th International Workshop on Computer Aided Verification (CAV'92)*, volume 663 of *Lecture Notes in Computer Science*, pages 260–273. Springer, June 1992.

[BCC+07] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. "Variance Analyses from Invariance Analyses". In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 211–224, 2007.

[BCP06] I. Balaban, A. Cohen, and A. Pnueli. “Ranking Abstraction of Recursive Programs”. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’06)*, volume 3855 of *LNCS*, pages 267–281. Springer, January 2006.

[BDEGW03] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. “Model Checking at IBM”. *Formal Methods in System Design*, 22(2):101–108, 2003.

[BDH02] D. Bosnacki, D. Dams, and L. Holenderski. “Symmetric Spin”. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):92–106, 2002.

[Bel77] N.D. Belnap. “A Useful Four-Valued Logic”. In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. Reidel, 1977.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. “Reachability Analysis of Pushdown Automata: Application to Model-Checking”. In *Proceedings of 8th International Conference on Concurrency Theory (CONCUR’2004)*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997.

[BG99] G. Bruns and P. Godefroid. “Model Checking Partial State Spaces with 3-Valued Temporal Logics”. In *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV’99)*, volume 1633 of *LNCS*, pages 274–287, July 1999.

[BG00] G. Bruns and P. Godefroid. “Generalized Model Checking: Reasoning about Partial State Spaces”. In *Proceedings of 11th International Conference on Concurrency Theory (CONCUR’00)*, volume 1877 of *LNCS*, pages 168–182. Springer, 2000.

[BG02] S. Barner and O. Grumberg. “Combining Symmetry Reduction and Under-Approximation for Symbolic Model Checking”. In *Proceedings of the 14th In-*

*ternational Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 93–106, July 2002.

[BHZ06] R. Bagnara, P. Hill, and E. Zaffanella. "Widening Operators for Powerset Domains". *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4-5):449–466, 2006.

[BK07] T. Ball and O. Kupferman. "Better Under-Approximation of Programs by Hiding Variables". In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, volume 4349 of *LNCS*, pages 314–328. Springer, January 2007.

[BKY05] T. Ball, O. Kupferman, and G. Yorsh. "Abstraction for Falsification". In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 67–81. Springer, July 2005.

[BMMR01] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. "Automatic Predicate Abstraction of C Programs". In *Proceedings of ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 203–213. ACM, 2001.

[BNRS08] N. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. "Proofs from Tests". In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 3–14, July 2008.

[BPR01] T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In *Proceedings of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 268–283, April 2001.

[BPR02] T. Ball, A. Podelski, and S. Rajamani. "Relative Completeness of Abstraction Refinement for Software Model Checking". In *Proceedings of 8th International*

*Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 158–172, Grenoble, France, April 2002. Springer.

[BPR03] T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):49–58, November 2003.

[BR00] T. Ball and S. Rajamani. "Bebop: A Symbolic Model Checker for Boolean Programs". In *Proceedings of SPIN 2000 Workshop on Model Checking of Software (SPIN'00)*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[BR01] T. Ball and S. Rajamani. "The SLAM Toolkit". In *Proceedings of 13th International Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[Bra91] J. C. Bradfield. *"Verifying Temporal Properties of Systems with Applications to Petri Nets"*. PhD thesis, University of Edinburgh, UK, July 1991.

[Bry92] R. E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams". *Computing Surveys*, 24(3):293–318, September 1992.

[CC77] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proceedings of the 4th Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, 1977.

[CC92] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CCG+04] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. "Modular Verification of Software Components in C". *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[CCGR99] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. "NUSMV: A New Symbolic Model Verifier". In N. Halbwachs and D. Peled, editors, *Proceedings of 11th Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499, Trento, Italy, July 1999. Springer.

[CCK+02] P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. "Automated Abstraction Refinement for Model Checking Large State Spaces Using SAT Based Conflict Analysis". In *Proceedings of 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 33–51, 2002.

[CDEG03] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM Transactions on Software Engineering and Methodology*, 12(4):1–38, October 2003.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. "Bandera: Extracting Finite-state Models from Java Source Code". In *Proceedings of 22nd International Conference on Software Engineering (ICSE'00)*, pages 762–765, June 2000.

[CE81] E. M. Clarke and E. A. Emerson. "Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic". In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[CEJS98] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. "Symmetry Reductions in Model Checking". In *Proceedings of the 10th International Conference on*

*Computer-Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 147–158, June 1998.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January 1983.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". *Journal of the ACM*, 50(5):752–794, September 2003.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. "Model Checking and Abstraction". *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGMP99] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. "State Space Reduction Using Partial Order Techniques". *STTT*, 2(3):279–287, 1999.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CH08] N. Charlton and M. Huth. "Falsifying Safety Properties Through Games on Over-approximating Models". *Electronic Notes in Theoretical Computer Science*, 223:71–86, 2008.

[CIY95] R. Cleaveland, S. P. Iyer, and D. Yankelevich. "Optimality in Abstractions of Model Checking". In *Proceedings of International Symposium on Static Analysis (SAS'95)*, volume 983 of *LNCS*, pages 51–63, Glasgow, UK, September 1995. Springer.

[CJEF96] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. “Exploiting Symmetry in Temporal Logic Model Checking”. *Formal Methods in System Design*, 9(1-2):77–104, 1996.

[CKSY05] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. “SATABS: SAT-Based Predicate Abstraction for ANSI-C”. In *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, April 2005.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. “Compositional Model Checking”. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS’89)*, pages 464–475, June 1989.

[CM84] K. M. Chandy and J. Misra. “The drinking philosophers problem”. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[Cou77] P. Cousot. “Asynchronous Iterative Methods for Solving a Fixed Point System of Monotone Equations in a Complete Lattice”. Research report, Laboratoire IMAG, University of Grenoble, Septemeber 1977.

[CPR06a] B. Cook, A. Podelski, and A. Rybalchenko. “Termination Proofs for System Code”. In *Proceedings of ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI’06)*, pages 415–426. ACM, 2006.

[CPR06b] B. Cook, A. Podelski, and A. Rybalchenko. “Terminator: Beyond Safety”. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer-Aided Verification (CAV’06)*, volume 4144 of *LNCS*, pages 415–418, Seattle, WA, 2006. Springer.

[CTV06] E. M. Clarke, M. Talupur, and H. Veith. “Environment Abstraction for Parameterized Verification”. In *Proceedings of 7th International Conference on Verifi-*

*cation, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 126–141, 2006.

[dAGJ04] L. de Alfaro, P. Godefroid, and R. Jagadeesan. "Three-Valued Abstractions of Games: Uncertainty, but with Precision". In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 170–179, 2004.

[Dam03] D. Dams. "Comparing Abstraction Refinement Algorithms". *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. "Abstract Interpretation of Reactive Systems". *ACM Transactions on Programming Languages and Systems*, 2(19):253–291, 1997.

[DN03] D. Dams and K. S. Namjoshi. "Shape Analysis through Predicate Abstraction and Model Checking". In *Proceedings of 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *LNCS*, pages 310–324. Springer, January 2003.

[DN04] D. Dams and K. S. Namjoshi. "The Existence of Finite Abstractions for Branching Time Model Checking". In *Proceedings of 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 335–344, 2004.

[DN05] D. Dams and K. S. Namjoshi. "Automata as Abstractions". In *Proceedings of 6th International Conference on Verification, Model-Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 216–232. Springer, 2005.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. "Efficient Algorithms for Model Checking Pushdown Systems". In *Proceedings of the 12th International*

*Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*. Springer, July 2000.

[EHT00] E. A. Emerson, J. W. Havlicek, and R. J. Trefler. "Virtual Symmetry Reduction". In *Proceedings of 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 121–131, Santa Barbara, CA, USA, June 2000. IEEE Computer Society.

[EK00] E. A. Emerson and V. Kahlon. "Reducing Model Checking of the Many to the Few". In *Proceedings of 17th International Conference on Automated Deduction (CADE'00)*, volume 1831 of *LNCS*, pages 236–254. Springer, June 2000.

[Eme08] E. A. Emerson. "The Beginning of Model Checking: A Personal Perspective". In *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *LNCS*, pages 27–45. Springer, 2008.

[EN95] E. A. Emerson and K. S. Namjoshi. "Reasoning about Rings". In *POPL*, pages 85–94, 1995.

[ES96] E. A. Emerson and A. P. Sistla. "Symmetry and Model Checking". *Formal Methods in System Design*, 9(1-2):105–131, 1996.

[ES01] J. Esparza and S. Schwoon. "A BDD-Based Model Checker for Recursive Programs". In *Proceedings of 13th International Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 324–336. Springer, 2001.

[ET99] E. A. Emerson and R. J. Trefler. "From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking". In *Proceedings of 9th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 142–157. Springer, 1999.

[EW03] E. A. Emerson and T. Wahl. "On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking". In *Proceedings of 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 216–230. Springer, 2003.

[EW05] E. A. Emerson and T. Wahl. "Dynamic Symmetry Reduction". In *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 382–396, April 2005.

[FH06] H. Fecher and M. Huth. "Ranked Predicate Abstraction for Branching Time: Complete, Incremental, and Precise". In *Proceedings of 4th International Symposium on Automated Technology for Verification and Analysis (ATVA'06)*, volume 4218 of *Lecture Notes in Computer Science*, pages 322–336. Springer, October 2006.

[GC03] A. Gurfinkel and M. Chechik. "Proof-like Counterexamples". In *Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 160–175. Springer, 2003.

[GC05] A. Gurfinkel and M. Chechik. "How Thorough is Thorough Enough". In *Proceedings of 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 65–80, Saarbrücken, Germany, October 2005. Springer.

[GC06] A. Gurfinkel and M. Chechik. "Why Waste a Perfectly Good Abstraction?". In *Proceedings of 12th International Conference on Tools and Algorithms for the*

*Construction and Analysis of Systems (TACAS'06)*, volume 212-226 of *LNCS*, page 3920. Springer, April 2006.

[GH05] P. Godefroid and M. Huth. "Model Checking v.s. Generalized Model Checking: Semantic Minimizations for Temporal Logics". In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS'05)*, LICS, pages 158–167, June 2005.

[GHJ01] P. Godefroid, M. Huth, and R. Jagadeesan. "Abstraction-Based Model Checking using Modal Transition Systems". In *Proceedings of 12th International Conference on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 426–440, Aalborg, Denmark, 2001. Springer.

[GHK+06] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. "SYNERGY: A New Algorithm for Property Checking". In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, pages 117–127. ACM, November 2006.

[GJ03] P. Godefroid and R. Jagadeesan. "On the Expressiveness of 3-Valued Models". In *Proceedings of 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *LNCS*, pages 206–222. Springer, January 2003.

[GL91] O. Grumberg and D. E. Long. "Model Checking and Modular Verification". In *Proceedings of 2nd International Conference on Concurrency Theory (CONCUR'91)*, volume 527 of *LNCS*, pages 250–265. Springer, August 1991.

[God97] P. Godefroid. "Model Checking for Programming Languages using VeriSoft". In *Proceedings of 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186, January 1997.

[GP09] P. Godefroid and N. Piterman. “LTL Generalized Model Checking Revisited”. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’09)*, volume 5403 of *LNCS*, pages 89–104, January 2009.

[GS97] S. Graf and H. Saïdi. “Construction of Abstract State Graphs with PVS”. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV’97)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[GW94] P. Godefroid and P. Wolper. “A Partial Approach to Model Checking”. *Information and Computation*, 110(2):305–326, 1994.

[GWC06a] A. Gurfinkel, O. Wei, and M. Chechik. “Systematic Construction of Abstractions for Model-Checking”. In *Proceedings of 7th International Conference on Verification, Model-Checking, and Abstract Interpretation (VMCAI’06)*, volume 3855 of *LNCS*, pages 381–397. Springer, 2006.

[GWC06b] A. Gurfinkel, O. Wei, and M. Chechik. “YASM: A Software Model-Checker for Verification and Refutation”. In *Proceedings of 18th International Conference on Computer-Aided Verification (CAV’06)*, volume 4144 of *LNCS*, pages 170–174. Springer, 2006.

[GWC08] A. Gurfinkel, O. Wei, and M. Chechik. “Model Checking Recursive Programs with Exact Predicate Abstraction”. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA’08)*, volume 5311 of *LNCS*, pages 95–110. Springer, 2008.

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. “Abstractions from Proofs”. In *Proceedings of 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 232–244, Venice, Italy, January 2004. ACM.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. “Lazy Abstraction”. In *Proceedings of 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’02)*, pages 58–70, Portland, Oregon, January 2002. ACM.

[HJS01] M. Huth, R. Jagadeesan, and D. A. Schmidt. “Modal Transition Systems: A Foundation for Three-Valued Program Analysis”. In *Proceedings of 10th European Symposium on Programming (ESOP’01)*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.

[HJS04] M. Huth, R. Jagadeesan, and D. Schmidt. “A Domain Equation for Refinement of Partial Systems”. *Mathematical Structures in Computer Science*, 14(4):469–505, 2004.

[ID96] C. Norris Ip and David L. Dill. “Better Verification Through Symmetry”. *Formal Methods in System Design*, 9(1-2):41–75, 1996.

[IYG+05] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. “F-Soft: Software Verification Platform”. In *Proceedings of 17th International Conference on Computer-Aided Verification (CAV’05)*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005.

[JS04] B. Jeannet and W. Serwe. “Abstracting Call-Stacks for Interprocedural Verification of Imperative Programs”. In *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST’2004)*, volume 3116 of *LICS*, pages 258–273. Springer, 2004.

[KGC04] D. Kroening, A. Groce, and E. Clarke. “Counterexample Guided Abstraction Refinement via Program Execution”. In *Proceedings of 6th International Conference on Formal Engineering Methods (ICFEM’04)*, volume 3308 of *LNCS*, pages 224–238. Springer, November 2004.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.

[KM95] R. P. Kurshan and K. L. McMillan. "A Structural Induction Theorem for Processes". *Information and Computation*, 117(1):1–11, 1995.

[Koz83] D. Kozen. "Results on the Propositional $\mu$-calculus". *Theoretical Computer Science*, 27:334–354, 1983.

[KP00] Y. Kesten and A. Pnueli. "Verification by Augmented Finitary Abstraction". *Information and Computation*, 163(1):203–243, 2000.

[KS92] J. Knoop and B. Steffen. "The Interprocedural Coincidence Theorem". In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, pages 125–140, London, UK, 1992. Springer-Verlag.

[Kur89] R. P. Kurshan. "Analysis of Discrete Event Coordination". In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 414–453. Springer, May 1989.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes : The Automata-Theoretic Approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[Kur08] R. P. Kurshan. "Verification Technology Transfer". In *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *LNCS*, pages 46–64. Springer, 2008.

[Lar91] P. Larsen. "The Expressive Power of Implicit Specifications". In *Proceedings of the 18th International Conference on Automata, Languages, and Programming (ICALP'91)*, volume 510 of *LNCS*, pages 204–216. Springer, 1991.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. "Property Preserving Abstractions for the Verification of Concurrent Systems". *Formal Methods in System Design*, 6:1–35, 1995.

[LNW07] K. G. Larsen, U. Nyman, and A. Wasowski. "On Modal Refinement and Consistency". In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR'07)*, volume 4703 of *LNCS*, pages 105–119, September 2007.

[LT88] K. G. Larsen and B. Thomsen. "A Modal Process Logic". In *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE Computer Society Press, 1988.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[NGC06] S. Nejati, M. Gheorghiu, and M. Chechik. "Thorough Checking Revisited". In *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD'06)*, pages 106–116, November 2006.

[NK00] K. S. Namjoshi and R. P. Kurshan. "Syntactic Program Transformations for Automatic Abstraction". In *Proceedings of the 12th Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 435–449, Chicago, IL, USA, July 2000. Springer.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992.

[NNH05] H. R. Nielson, F. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[Pap81] C. H. Papadimitriou. “On the Complexity of Integer Programming”. *Journal of ACM*, 28(4):765–768, 1981.

[Par81] D. Park. “Concurrency and Automata on Infinite Sequences”. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer.

[PDV01] C. Pasareanu, M. B. Dwyer, and W. Visser. “Finding Feasible Counter-examples when Model Checking Abstracted Java Programs”. In *Proceedings of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’01)*, volume 2031 of *LNCS*, pages 284–298, April 2001.

[Pnu77] A. Pnueli. “The Temporal Logic of Programs”. In *Proceedings of 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[Pnu84] A. Pnueli. “In Transition from Global to Modular Temporal Reasoning about Programs”. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1984.

[PPV05] C. Pasareanu, R. Pelanek, and W. Visser. “Concrete Model Checking with Abstract Matching and Refinement”. In *Proceedings of 17th International Conference on Computer-Aided Verification (CAV’05)*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005.

[PR04] A. Podelski and A. Rybalchenko. “A Complete Method for the Synthesis of Linear Ranking Functions”. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’04)*, volume 2937 of *LNCS*, pages 239–251. Springer, January 2004.

[PR05] A. Podelski and A. Rybalchenko. “Transition Predicate Abstraction and Fair Termination”. In *Proceedings of 32th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05)*, pages 132–144, 2005.

[PSW05] A. Podelski, I. Schaefer, and S. Wagner. “Summaries for While Programs with Recursion”. In *Proceedings of the 14th European Symposium on Programming Languages and Systems (ESOP’05)*, volume 3444 of *LNCS*, pages 94–107. Springer, April 2005.

[Pug92] W. Pugh. “The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis”. *Comm. of the ACM*, August 1992.

[PXZ02] A. Pnueli, J. Xu, and L. D. Zuck. “Liveness with (0, 1, $\infty$)-Counter Abstraction”. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV’02)*, volume 2404 of *LNCS*, pages 107–122. Springer, 2002.

[QS82] J. Queille and J. Sifakis. “Specification and Verification of Concurrent Systems in CESAR”. In *Proceedings of 5th International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, April 1982.

[RHS95] T. W. Reps, S. Horwitz, and M. Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages(POPL’95)*, pages 49–61, San Francisco, CA, 1995.

[RLS02] T. W. Reps, A. Loginov, and S. Sagiv. “Semantic Minimization of 3-Valued Propositional Formulae”. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS’02)*, pages 40–54, July 2002.

[Sch98] D. A. Schmidt. “Data Flow Analysis is Model Checking of Abstract Interpretations”. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL’98)*, 1998.

[Sch04] D. A. Schmidt. “Closed and Logical Relations for Over- and Under-Approximation of Powersets”. In *Proceedings of 11th International Symposium*

*on Static Analysis (SAS'04)*, volume 3148 of *LNCS*, pages 22–37, Verona, Italy, August 2004. Springer.

[SG03] S. Shoham and O. Grumberg. "A Game-Based Framework for CTL Counter-Examples and 3-Valued Abstraction-Refinement". In *Proceedings of the 15th Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 275–287. Springer, July 2003.

[SG04] S. Shoham and O. Grumberg. "Monotonic Abstraction-Refinement for CTL". In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 546–560. Springer, 2004.

[SG06] S. Shoham and O. Grumberg. "3-Valued Abstraction: More Precision at Less Cost". In *Proceedings of 21th IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 399–410. IEEE Computer Society, 2006.

[SGE00] A. P. Sistla, V. Gyuris, and E. A. Emerson. "SMC: A Symmetry-Based Model Checker for Verification of safety and liveness properties". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(2):133–166, 2000.

[Som01] F. Somenzi. "CUDD: CU Decision Diagram Package Release", 2001.

[SP81] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter "Two Approaches to Interprocedural Data Flow Analysis", pages 189–233. Prentice-Hall, 1981.

[SRW99] M. Sagiv, T. W. Reps, and R. Wilhelm. "Parametric Shape Analysis via 3-Valued Logic". In *Proceedings of 26th Annual ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 105–118, New York, NY, 1999. ACM.

[SWZ07] A. P. Sistla, X. Wang, and M. Zhou. "Checking extended *TL* properties using guarded quotient structures". *Formal Methods in System Design*, 31(3):197–219, 2007.

[TW09] R. J. Trefler and T. Wahl. "Extending Symmetry Reduction by Exploiting System Architecture". In *Proceedings of 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*, volume 5403 of *LNCS*, pages 320–334. Springer, January 2009.

[Wah07] T. Wahl. "Adaptive Symmetry Reduction". In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 393–405. Springer, July 2007.

[WB95] P. Wolper and B. Boigelot. "An Automata-Theoretic Approach to Presburger Arithmetic Constraints". In *Proceedings of International Symposium on Static Analysis (SAS'95)*, volume 1785 of *LNCS*, pages 21–32. Springer-Verlag, 1995.

[WGC05] O. Wei, A. Gurfinkel, and M. Chechik. "Identification and Counter Abstraction for Full Virtual Symmetry". In *Proceedings of 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 285–300. Springer, 2005.

[WGC09a] O. Wei, A. Gurfinkel, and M. Chechik. "Mixed Transition Systems Revisited". In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*, volume 5403 of *LNCS*, pages 349–365, January 2009.

[WGC09b] O. Wei, A. Gurfinkel, and M. Chechik. "On the Consistency, Expressiveness, and Precision of Partial Modeling Formalisms". May 2009. Submitted for publication.

[WL89] P. Wolper and V. Lovinfosse. “Verifying Properties of Large Sets of Processes with Network Invariants”. In *Proceedings of International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer, June 1989.

[YBS06] G. Yorsh, T. Ball, and M. Sagiv. “Testing, Abstraction, Theorem Proving: Better Together!”. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’06)*, pages 145–156. ACM, July 2006.

[Zad87] L.A. Zadeh. “Fuzzy Sets”. In *Fuzzy Sets and Applications: Selected Papers by L.A. Zadeh*, pages 29–44, New York, 1987. John Wiley & Sons, Inc.