STATIC AND DYNAMIC FORMAL ANALYSIS OF CONCURRENT SYSTEMS AND LANGUAGES: A SEMANTICS-BASED APPROACH

BY

AZADEH FARZAN

B.S., Sharif University of Technology, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois
Abstract

Concurrency is ubiquitous in modern software. The computing base of systems software, including operating systems and databases, has always been highly concurrent and with the introduction of language-level thread primitives in languages like Java and C# and the advent of distributed web services, concurrency has become commonplace even in application software. The design of concurrent software is notoriously error-prone due to the nondeterministic interaction among concurrently executing threads. Therefore, it is important to develop techniques for specifying correctness properties of concurrent software and tools for automatically checking these properties.

A burning problem in program verification today is how to model the world of concurrent systems. The excellent models of sequential behavior that have evolved during the past thirty years of sequential program verification do not adequately reflect the nature of a concurrent universe. In this work, we present solutions to the commonly known problems of concurrency by using appropriate semantic models for concurrency, namely rewriting logic, and traces and Petri nets. We argue that using the appropriate semantic model helps one to define the problem in a simple, intuitive, and effective way in the first place, and then provides sound solutions for these now well-defined problems. This work addresses some verification problems for concurrent programs and systems. The approaches presented here fall into two main category of static versus dynamic verification.

In the dynamic category, we use rewriting logic as a true concurrency model to specify concurrent programs and systems. We established the tool JavaFAN based on rewriting logic semantics of Java at both language and bytecode level. We provide a methodology in which formal semantics is then used as a basis to develop formal analyses tools (interpreter, model checker, and safety property checker) for analyzing programs in any language, in the case of JavaFAN tool for Java programs at both levels. We furthermore advanced these ideas by developing a generic partial order reduction module that with minimum effort can be added to any language specification and automatically enrich it with POR capabilities. We took this idea even further by providing reduction methods that can work on the specification of any concurrent system. Experimental result suggest that these methods are effective in practice, despite their generality and relative effortless implementations.

In the static category, the focus of this work is to provide an appropriate notion of static abstraction for concurrent programs, called control net, based on Petri nets which captures the abstraction of
program into a control only (no data) structure and at the same time captures the interaction mechanism of the threads and preserves the independence of their execution where applicable. This model is then used to show how two very important static analyses problems in the context of concurrency, namely atomicity and dataflow analyses can be cleanly defined for the program traces (partially-ordered runs) generated by the control net, and how clean algorithmic solutions can be provided to solve these problems. Experimental results suggest that these solutions are feasible in practice.
To my parents
and Bardia
It is a pleasure to be able express my deepest gratitudes to my adviser, José Meseguer, without whose sincere care and guidance this work would not have been possible. His scholarliness and admirable sense of responsibility will always be my aspiration.

Special thanks are due to Grigore Rosu for his helps with my very first project towards my PhD work and invaluable contributions to some of my publications. Same gratitude must go to Madhusudan Parthasarathy for contributing to many ideas presented here, and for his support and friendship during my last year of graduate studies. I would also like to thank my other thesis committee members Gul Agha and Edmund Clarke, for their constructive comments and ideas.

Special thanks should go to Feng chen for closely collaborating with me on the JavaFAN project and being my coauthor. I would like to thank Joe Hendrix for his friendship, many useful discussions, and making the never ending hours of office work more pleasant. I would also like to thank other formal methods students in UIUC for helping me with many things throughout my studies.

Most of all, I wish to express my deepest and most sincere love and gratitude to my dearest Bardia. Having is love and support throughout the years have pulled me through all the difficult situations.
# Table of Contents

**List of Figures** ................................................................. xii

**List of Tables** ................................................................. xiv

**Chapter 1 Introduction** ......................................................... 1

1.1 Causality for Static Analysis of Concurrent Programs ....... 2

1.1.1 Causal Atomicity .......................................................... 3

1.1.2 Causal Dataflow Analyses for Concurrent Programs ...... 4

1.2 Semantics-based Formal Analysis of Concurrent Programs ... 6

1.3 State Space Reduction Techniques ................................. 7

1.3.1 State Space Reduction for Programming Languages ....... 7

1.3.2 State Space Reduction for Rewrite Theories ................. 8

**Chapter 2 Preliminaries** ....................................................... 9

2.1 Petri Nets and Traces ...................................................... 9

2.1.1 Petri Nets ................................................................. 9

2.1.2 Traces ................................................................. 9

2.1.3 Traces of a Petri Net .................................................. 11

2.1.4 Net Unfoldings .......................................................... 12

2.2 Rewriting Logic Operational Semantics ....................... 12
Chapter 8  State Space Reduction for Programming Languages  

8.1 Language-independent Partial Order Reduction Tools  

8.2 Partial Order Reduction for Language Definitions  

8.2.1 Some Assumptions  

8.2.2 The Theory Transformation  

8.2.3 The Partial Order Reduction Module  

8.2.4 Correctness of the Theory Transformation  

8.3 Methodology: POR for JavaFAN  

8.3.1 A Case Study for the Java Language  

8.3.2 The JVM POR Unit  

8.4 Applications of the Method and Experiments  

8.5 Related Work and Conclusions  

8.5.1 Related Work  

8.5.2 Conclusions  

Chapter 9  Conclusions and Future Research Directions  

9.1 Modeling Concurrency  

9.1.1 Static Analysis  

9.1.2 Dynamic Analysis  

9.2 Language-independent Techniques for Software Analysis  

9.3 Notions of Atomicity  

9.4 Static Analysis for Concurrency
# List of Figures

2.1 Sample Petri net and its associated traces. ................................................. 11
2.2 Net unfolding. .................................................................................................. 13
2.3 Rewriting Logic Inference Rules. .................................................................... 14
2.4 Maude Specification for Natural Numbers. ..................................................... 15
3.1 SML syntax ........................................................................................................ 20
3.2 Model Construction ............................................................................................ 21
3.3 A program and its control net. .......................................................................... 22
4.1 Not serializable but causally atomic ................................................................. 25
4.2 Two Programs ..................................................................................................... 26
4.3 Non-Atomic Trace .............................................................................................. 26
4.4 Example .............................................................................................................. 29
5.1 Flow of facts over a trace. .................................................................................. 41
5.2 Representation relation. ..................................................................................... 51
5.3 Augmented Control Net. .................................................................................. 52
6.1 Restaurant State Space ....................................................................................... 62
7.1 The iadd instruction. ......................................................................................... 83
7.2 The invokevirtual instruction ........................................... 83
7.3 The monitorenter instruction ............................................. 84
7.4 ...................................................................................... 84
7.5 ...................................................................................... 87
7.6 Property Specification Language ........................................ 90
8.1 Two Partial Order Reduction Heuristics .............................. 102
List of Tables

4.1 Programs and Performances ........................................... 35

5.1 Programs and Performances ........................................... 59

6.1 MOP Performance Results ........................................... 68

6.2 Performance Results ................................................ 71

7.1 Thread Game Times ................................................... 90

7.2 Dining Philosophers Times .......................................... 91

7.3 Readers/Writers Performance ...................................... 92

8.1 Time and Space Reduction Comparisons for Sieve of Eratosthenes ................................................... 109

8.2 Dining Philosophers in the JVM .................................. 109

8.3 Changing Dependency Relation – JVM ......................... 110

8.4 Java POR Unit Performance ......................................... 110

8.5 Partial Order Reduction Results for Various JVM Programs ................................................... 110
Concurrency is ubiquitous in modern software. The computing base of systems software, including operating systems and databases, has always been highly concurrent. With the introduction of language-level thread primitives in languages like Java and C# and the advent of distributed web services, concurrency has become commonplace even in application software. The application programmer of today deals with concurrency issues just as much as the systems programmer of yesterday.

The design of concurrent software is notoriously error-prone due to the nondeterministic interaction among concurrently executing threads. Therefore, it is important to develop techniques for specifying correctness properties of concurrent software and tools for automatically checking these properties. To improve the reliability of concurrent software, a variety of analysis techniques are required, both static such as model checking, program analysis, theorem proving and type systems, as well as dynamic such as run-time analysis and testing. This work aims at addressing the problem of concurrency by proposing a collection of static and dynamic formal analysis techniques. In this chapter, we give an overview of these techniques which are in turn discussed in detail in the rest of this proposal.

A burning problem in program verification today is how to model the world of concurrent systems. The excellent models of sequential behavior that have evolved during the past thirty years of sequential program verification do not adequately reflect the nature of a concurrent universe. The reason for this is twofold: one, a variety of “testing scenarios” reveals situations where sequential models yield a visibly wrong answer and hence an unsound logic, and two, trivial modifications of sequential models in order to make them sound for concurrent systems usually prove to be very inefficient. This has spurred interest in true concurrency, as it has come to be called; namely, in modeling concurrency in a way that is faithful to all currently understood modes of interaction of system components, particularly those beyond the reach of sequential models.

The work presented in this dissertation can be categorized under two main themes: (1) static formal analysis of concurrent programs, and (2) dynamic formal analysis of concurrent programs. The purpose of this thesis is to show how using true concurrency models can give new insights into how to properly define the appropriate properties (correctness criteria) for the concurrent
programs on one hand, and how to use these models to come up with state-of-the-art algorithms and techniques to check these properties, and, moreover, come up with innovative formal analysis tools.

The true concurrency model of rewriting logic \cite{Mes92a} provides an excellent framework for modeling concurrent systems and languages. We have developed a set of language-independent techniques \cite{FMR04,FCMR04,FM06b} which exploit the advantages of this model in order to build dynamic formal analysis tools for concurrent software systems. Moreover, the framework supports a seamless combination of different techniques used for ensuring software reliability including model checking and theorem proving. \cite{FM06c} demonstrates an excellent instance of such combination.

On the static analysis side, we believe that causality is an appropriate notion to capture the flow of control in concurrent programs, and consequently a nice basis for static analyses. Hence, (Mazurkiewicz) Traces and Petri nets, which depict the causal structure of concurrent program in natural way, are proper candidates to model concurrent programs. Mapping a static analysis problem from the sequential setting to the concurrent (causal) setting is in many cases a nontrivial issue, and demonstrates how Petri nets serve very well as models to provide algorithms for finding the causally defined solutions to these static analysis problems. Rewriting logic, in fact, works as a unifying framework for a wide range of Petri net models \cite{SMO01} (including the simple 1-safe nets and the colored Petri nets used in this thesis). Therefore, rewriting logic in a nice way unifies the different concurrent semantics that are used for formal analyses throughout this thesis.

In the rest of this Chapter, we present a very high level summary of research problems that have been investigated in the framework of concurrency. In the following chapters, a detailed account of achievements in each direction is presented together with the planned future work for each direction appearing at the end of the corresponding chapter. The main results presented in this work have been published in a more summarized form in \cite{FM06c,FM06a,FM07,FMR04,FCMR04,FM06b}.

1.1 Causality for Static Analysis of Concurrent Programs

Despite the growing importance of concurrency in programming, little progress has been made in static analysis of concurrent programs. For instance, there is no standard notion of a control-flow graph for concurrent programs, while the analogous notion in sequential programs has existed for a long time \cite{Hec77}. For example, dataflow analysis problems (arguably the simplest of static analyses problems) have not been clearly understood for programs with concurrency.

In this work, by investigating two important and relevant static analysis problems, namely atomicity and dataflow analyses, we argue that causality is the right notion to define these concepts in the
concurrent setting, and that Petri nets serve very well as models to provide algorithms for finding the causally defined solutions to these static analysis problems.

We present the control net model, that captures the control flow in a concurrent program, and give a translation from programs to Petri nets that explicitly abstracts data and captures the control flow in the program, as a candidate for standard notion of control flow graph in the concurrent setting. Given a program \( P \) with multiple threads, we exhibit a clean control model of the program as a Petri net \([\text{NPWS81}, \text{LMRT90}]\). This modeling is aimed at capturing control and abstracting away the data values used in the program. Moreover, the Petri net explicitly captures the independence of execution of threads, and the interaction mechanism of the threads (using shared variables, locks, etc.). The model for the program is independent of any notion of atomicity, and captures just the dependence and independence of control in the threads of the program.

We introduce causal atomicity as a new notion of atomicity based on causality and show how the control net model enriched with colors can be used to check causal atomicity. We introduce the causal concurrent dataflow (CCD) framework to provide a theoretical basis for a wide range of dataflow analyses in the concurrent setting, and show how an augmented version of the control net can be used to find this solution within reasonable time complexity.

1.1.1 Causal Atomicity

A practical approach to programming threads is to develop techniques that allow the programmer to specify and verify disciplined interaction between threads. The lack of race conditions is such a discipline. A race condition occurs when two threads simultaneously access a global variable, and one of the accesses is a write. Depending on when the write event gets scheduled, the program could take very different paths, which is a cause of concern. While the lack of races does seem to be a natural discipline to adhere to, it has been observed and argued that it is not a strong enough condition \([\text{FFL03}]\).

A stronger discipline is to require methods or blocks of code to be atomic. The general definition of atomicity is: a code block is atomic \([\text{FFL03}]\) if for every interleaved execution of the program in which the code block is executed, there is an equivalent run of the program where the code block is executed sequentially (without interleaving with other threads). Intuitively, since for every interleaved execution \( t \), there is an equivalent execution \( t' \) where the block occurs sequentially, if the block had a logical error making \( t \) incorrect, then \( t' \) would be incorrect as well, arguing that the error in the block was not because of its interleaving with other threads. Consequently, a block being atomic greatly simplifies writing and understanding code: the programmer can pretend that the code block is executed sequentially, which simplifies its correctness argument. Note that the

\(^1\) Atomicity is not a strictly stronger notion than race-freedom; see Figure 4.4 for an example.
The problem of checking atomicity of program blocks is obviously undecidable.

Atomicity is a well-argued principle of programming: in the database literature, it is known as serializability [EGLT76 Pap86 BHG87 AMP00 SKS05], and in the software analysis literature, it has been argued that many errors in threaded code can be traced back to blocks being non-atomic [FFL03 FFQ05 Fla04 WS05 HRD04 FF04 WS03]. There has been extensive work on detecting atomicity of program blocks: static analysis techniques that use type-checking to detect transactions [Lip75 FFL03 FFQ05], where a transaction is a strong notion that implies atomicity; dynamic analysis that checks atomicity of tests of the program at run-time [FF04 WS03]; and model checking for atomicity where a monitor that detects non-atomic blocks runs in parallel with the system, which is then model checked [HRD04 Fla04].

In order to get effective algorithms, atomicity checkers aim for soundness (i.e. if the tool reports a block to be atomic, then the block should indeed be atomic), and the generic way to achieve this is to abstract the program in a sound fashion, as well as to define a sound notion of equivalence between abstract traces. In other words, the equivalence relation between traces of the abstract model should imply that the concrete traces represented by them are equivalent as well.

While atomicity checkers in the literature do assure that their analyses are sound, they do not define precisely the abstraction they use, nor define precisely the notion of equivalence they assume. For example, static atomicity checking using types are based on transactions (transactions imply atomicity), but transactions require knowing what kind of “mover” each statement is, which is achieved using a separate race-checking analysis (which again is not precisely defined). The algorithm for checking for transactions is then implemented using types, and again it is not argued whether the classification of statements as movers, the type-checking approach is complete or not. All in all, though every step is sound, and the soundness of the whole algorithm is assured, the precise abstraction and notion of equivalence used is not clear, making it hard to evaluate and compare the formalisms.

The main contributions of Chapter 4 is a new notion of atomicity for programs based on causality that has precise definitions of the abstraction mechanism and the notion of equivalence between abstract runs.

### 1.1.2 Causal Dataflow Analyses for Concurrent Programs

Classical sequential optimizations are known to fail for concurrent programs, as they perform unsound transformations [MP90]. While it is certainly easy to formulate dataflow analyses for concurrent programs using the global product state space of the individual threads, the usefulness of doing so is questionable, as algorithms working on the global state space will not scale up.
Consequently, the literature in flow analysis for threaded programs concentrates on finding tractable problem definitions for dataflow analysis. A common approach has been to consider programs where the causal relation between events is static and apparent from the structure of the code (such as fork-join formalisms), making feasible an analysis that works by finding fixpoints on the union of the individual sequential control flow graphs. These approaches are often highly restrictive (for example, they require programs to have no loops [SW] or at least to have no loops with concurrent fork-join constructs [LMP97, LPM99]), and cannot model even simple shared-memory program models. Another approach is to restrict the dataflow problems that are solved, and develop mechanisms that handle complex concurrency (i.e. where causality is dynamic and not static) by exploiting the structure of the dataflow problem (like unidirectional dataflow analysis [KSV96]). In fact, a coherent formulation of control flow that can capture programs with dynamic concurrency (including those with shared memory) and a general definition of dataflow analysis problems for these programs has not been formulated in the literature (see Section 5.0.1).

The goals of this work are (a) to develop a formal control-flow model for programs using Petri nets, (b) to propose a novel definition of dataflow analyses based on causal flows in a program, (c) to develop algorithms for solving causal flow analyses when the domain of flow facts is a finite set $D$ by exploring the partially-ordered runs of the program as opposed to its interleaved executions, and (d) to provide provably efficient algorithms for the class of distributive CCD problems, and support the claim with demonstrative experiments. The framework we set forth here is the first one we know of that defines a formal general definition of dataflow analysis for concurrent programs.

Moreover, for the more common and important distributive subclass of dataflow analysis problems (which includes many important problems such as reaching definitions, live variables, uninitialized variables, copy constant propagation, available expressions, etc), we present efficient algorithms based on partial order model checking methods. We also prove a complexity bound for the distributive subclass: given a flow analysis problem over a finite set $D$, we prove that our analysis is no more complex than $|D|$ times the time required to generate the partially ordered control behaviors of the program.

To summarize, our new casual dataflow analysis definition seems very appealing as it does not destroy the concurrency and independence of events in a program, and yet can capture all common dataflow analysis problems. Using the classic concurrency model of Petri nets, we can model the concurrency and the dataflow analysis in a simple, compact machine. The net can represent dynamic concurrency and can model shared-memory concurrency. However, it cannot be treated as a graph and must be unfolded in order for the relevant information to be extracted from it. On the other hand, unfolding generates partial orders as opposed to interleavings, and hence can be analyzed efficiently. This leads to efficient solutions for flow analysis problems for concurrent programs. The formalism and algorithms work for a large class of important dataflow analysis problems. Finally, the control net modeling of programs using Petri nets (see Chapter 3) emerges
as a fine model that can capture control-flow in concurrent programs, and perhaps can be considered as the robust cousin of control-flow graphs for sequential programs.

In Chapter 5, we define a novel formulation of dataflow analysis for concurrent programs, where the flow of facts is along the causal dependencies of events. We capture the control flow of concurrent programs using a Petri net (called control net), and develop methods that solve the causal dataflow analysis problem using efficient algorithms based on partially-ordered unfoldings of Petri nets. We show that our causal dataflow definition is powerful enough to capture common dataflow analysis problems, and present experimental results that show the efficacy of dataflow analysis using partial orders. For the subclass of distributive problems, we prove that complexity of checking data flow is linear in the size of the problem and in the unfolding of the control net.

1.2 Semantics-based Formal Analysis of Concurrent Programs

There is a general belief in the algebraic specification community that all traditional programming language features can be described with equational specifications [BWP87, GM96, Wan80]. What is less known, or tends to be ignored, is that concurrency, which is a feature of almost any current programming language, cannot be naturally handled by equational specifications, unless one makes deterministic restrictions on how the different processes or threads are interleaved. While some of these restrictions may be acceptable, as most programming languages also provide thread or process scheduling algorithms, most of them are unacceptable in practice, because concurrent execution typically depends upon the external environment, which is unpredictable. Rewriting logic provides a suitable framework to alleviate this problem. Rewriting logic [Mes92a] extends equational logic with rewriting rules and has been mainly introduced as a unified model of concurrency; indeed, many formal theories of concurrency have been naturally mapped into rewriting logic during the last decade.

A next natural challenge is to define mainstream concurrent programming languages in rewriting logic and then use those definitions to build formal analysis tools for such languages. There is already a substantial body of case studies, of which we only mention [TSMO02, ST02, VMO], backing up one of the key claims of this work, namely, that rewriting logic can be fruitfully used as a unifying framework for defining programming languages. Further evidence on this claim includes modeling of a wide range of programming language features that has been developed and tested as part of a recent course taught at the University of Illinois [Ros]. In this work, we give detailed evidence for a second key claim, namely, that rewriting logic specifications can be used in practice to build simulators and formal analysis tools for mainstream programming languages such as Java with competitive performance.
In Chapter 7 we explain in detail how a concurrent language (in this case, Java bytecode) can be formally specified in rewriting logic, and how the formal language specification can be used to formally analyze programs in that language.

We have developed a tool, JavaFAN (Java Formal Analyzer), that works based on the above paradigm. Future work in this direction includes enhancing the JavaFAN tool by: (a) performing more experimentation with the tool including more comparisons with other Java model checkers, and (b) making the tool available to public through a web interface.

1.3 State Space Reduction Techniques

Although model checking is one of the most successful automated verification techniques, there are real limitations to its applicability in practice. These limitations are mostly related to the state space explosion problem. For example, as the number of processes considered in a distributed system grows, the associated state space may easily grow exponentially, particularly due to the system’s concurrency. This can sometimes make it unfeasible to model check these systems.

For this reason, a host of techniques to tame the state space explosion problem, which could be collectively described as state space reduction techniques, have been investigated: bisimulation techniques, partial order reduction (POR) techniques, abstraction techniques, and so on (see for example [KP00, SS99, LGS+95, CGL94, CU98, DGG97, GW91, Val90, Pel94, ABH+97, KLM+98, FG05]). The general idea is to transform the original system into a simpler one (typically bisimilar or at least similar to the original one) whose state space is small enough to model check properties. Transfer results then ensure that the same property holds in the original system.

In this dissertation, we present state space reduction techniques that fit in the following two major categories:

1.3.1 State Space Reduction for Programming Languages

To improve the performance of the generic tools mentioned above, we proposed a new approach to build language-independent partial order reduction (POR) [FM06b] in the above framework. In this approach, getting the POR capabilities does not require making any changes to the underlying model checking algorithms. The question that this work raises and answers affirmatively is: can semantics-based tools be equipped with state space reduction techniques that are language-independent in the strong sense of being generic, that is, being applicable not just to a few but to an unlimited class of languages satisfying some very basic abstract requirements?
In Chapter 8, we propose a new method to make software model checking tools language-independent; and to substantially improve their performance using *partial order reduction* (POR) (see, for example, [Pel93, God96, God97, Val90, ABH+97, FG05, CGMP99, KLM+98]). The key insight of POR is that the state space explosion caused by the many interleaving computations of a concurrent program can be tamed using the fact that many such computations are semantically *equivalent*, because they are different descriptions of the same partial order of events. This means that only a representative *subset* of all the interleaving computations has to be model checked, without losing completeness: the model checking results are the same as if all computations had been analyzed, but performance can thus be greatly increased.

Our generic POR unit can be customized for any language \( L \), under very minimal assumptions, with minimal interface, and with relatively little effort. Our experiments with several languages including Java, JVM, Promela, and Maude, indicate that significant state space reductions and time speedups can be gained for tools generated this way. In fact, comparisons with the POR unit of the SPIN model checker for programs in Promela show that our generic POR unit works as well as (and in some cases slightly better than) the SPIN’s specific POR unit designed for Promela.

### 1.3.2 State Space Reduction for Rewrite Theories

We propose a new state space reduction technique within the rewriting logic semantic framework, in which concurrent systems are formally specified as rewrite theories [Mes92b]. In such specifications, the set of states is specified as an algebraic data type by an equational theory \((\Sigma, E)\), and the system’s transitions are specified by rewrite rules \( R \) that are applied *modulo* the equations \( E \). The rewrite theory specifying the system is then the triple \( \mathcal{R} = (\Sigma, E, R) \). The fact that rewriting logic has been shown to be a very general and expressive semantic framework to specify concurrent systems [Mes99, MOM02] makes our proposed state space reduction technique applicable to a very wide range of concurrent systems. Also, achieving a state space reduction typically requires discharging *proof obligations* to verify that the reduction is correct. In this regard, the fact that the state space is itself axiomatized by an equational theory \((\Sigma, E)\) makes the tool-assisted discharging of such proof obligations using equational theorem proving techniques and tools much easier than if a non-logical specification formalism had been used instead. In Chapter 6, we discuss this technique and possible extensions in detail.
Chapter 2

Preliminaries

2.1 Petri Nets and Traces

2.1.1 Petri Nets

Definition 1 A Petri net\footnote{Petri nets can be more general, but in this paper we restrict to 1-safe Petri nets where each place gets at most one token.} is a triple $N = (P, T, F)$, where $P$ is a set of places, $T$ (disjoint from $P$) is a set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation.

For a transition $t$ of a (Petri) net, let $\bullet t = \{ p \in P | (p, t) \in F \}$ denote its set of pre-conditions and $t \bullet = \{ p \in P | (t, p) \in F \}$ its set of post-conditions.

A marking of the net is a subset $M \subseteq P$ of places. A marked net is a structure $(N, \text{Init})$, where $N$ is a net and $\text{Init}$ is an initial marking. A transition $t$ is enabled at a marking $M$ if $\bullet t \subseteq M$. The transition relation is defined on the set of markings: $M \xrightarrow{t} M'$ if a transition $t$ is enabled at $M$ and $M' = (M \setminus \bullet t) \cup t \bullet$. Let $\rightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$. A marking $M'$ covers a marking $M$ if $M' \subseteq M$.

A firing sequence is a sequence of transitions $t_1 t_2 \ldots$ provided we have a sequence of markings $M_0 M_1 \ldots$ such that $M_0 = \text{Init}$ and for each $i$, $M_i \xrightarrow{t_{i+1}} M_{i+1}$. We denote the set of firing sequences of $(N, \text{Init})$ as $FS(N, \text{Init})$. A firing sequence can be viewed as a sequential execution of the Petri net. However, we are interested in the partially-ordered traces that the Petri net exhibits; we will define these using Mazurkiewicz traces.

2.1.2 Traces

A trace alphabet is a pair $(\Sigma, I)$ where $\Sigma$ is an alphabet describing finite set of actions and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric relation over $\Sigma$ called the independence relation. The induced relation
\( D = (\Sigma \times \Sigma) \setminus I \) is called the \textit{dependence} relation. A Mazurkiewicz trace is a behavior that describes a partially-ordered execution of events in \( \Sigma \) (when \( I = \emptyset \), then it is simply a word).

**Definition 2** [DR95] A (Mazurkiewicz) \textit{trace} over the trace alphabet \((\Sigma, I)\) is a \( \Sigma \)-labeled poset \( t = (E, \preceq, \lambda) \) where \( E \) is a finite or countable set of events, \( \preceq \) is a partial order on \( E \), called the \textit{causal order} (\( \prec \) is the immediate causal relation defined as: \( e \prec e' \) iff \( e \prec e' \) and there is no event \( e'' \) such that \( e \prec e'' \prec e' \)), and \( \lambda : E \rightarrow \Sigma \) is a labeling function such that the following hold, where \( \downarrow e = \{ e' \in E \mid e' \preceq e \} \) and \( \downarrow e = \downarrow e - \{ e \} = \{ e' \in E \mid e' \prec e \} \):

- \( \forall e \in E, \downarrow e \) is finite, so we demand that there are only finitely many events causally before \( e \).
- \( \forall e, e' \in E, e \prec e' \Rightarrow \lambda(e) D \lambda(e') \). Events that are immediately causally related must correspond to dependent actions.
- \( \forall e, e' \in E, \lambda(e) D \lambda(e') \Rightarrow (e \preceq e' \lor e' \preceq e) \). Any two events with dependent labels must be causally related.

\( T(\Sigma, I) \) denotes the set of all traces over \((\Sigma, I)\). We identify traces that are isomorphic (i.e. if there is a correspondence between events that preserves the causality and labeling predicates).

A \textit{linearization} of a trace \( t = (E, \preceq, \lambda) \) is a linearization of its events that respects the partial order; in other words, it is a word structure \((E, \preceq', \lambda)\) where \( \preceq' \subseteq \preceq \).

Let us define an equivalence on words over \( \Sigma \): \( \sigma \sim \sigma' \) if and only if for every pair of letters \( a, b \in \Sigma \), with \( aDb, \sigma \downarrow \{a, b\} = \sigma' \downarrow \{a, b\} \), where \( \downarrow \) is the projection operator that drops all symbols not belonging the second argument. Then, \( \sigma \) and \( \sigma' \) are linearizations of the same trace iff \( \sigma \sim \sigma' \). We denote the equivalence class that \( \sigma \) belongs to as \([\sigma]\).

Let \((\Sigma, I)\) be a trace alphabet and \( \sim \) be the associated equivalence relation. Let us now formally associate the (unique) trace that corresponds to a word \( \sigma \) over \( \Sigma \).

A finite word \( \sigma a \) is said to be \textit{prime} if for every \( \sigma' \sim \sigma a \), \( \sigma' \) is of the form \( \sigma'' a \) (i.e. all words equivalent to \( \sigma \) end with \( a \)).

Let \( \sigma \) be a finite or infinite word over \( \Sigma \). The trace associated with \( \sigma \), \( Tr(\sigma) = (E, \preceq, \lambda) \) is defined as:

- \( E = \{ [\sigma'] \mid [\sigma'] \text{ is prime}, \exists \sigma'' \sim \sigma, \sigma' \text{ is a prefix of } \sigma'' \} \),
- \( [\sigma] \preceq [\sigma'] \) if there exists \( \sigma_1 \in [\sigma], \sigma'_1 \in [\sigma'] \) such that \( \sigma_1 \) is a prefix of \( \sigma'_1 \),
- \( \lambda([\sigma']a) = a \) for each \( \sigma' \in E \).
It is easy to see that $Tr(\sigma)$ is a trace, and that $\sigma$ is a linearization of it.

The concatenation of two traces is defined as taking the union of the two traces and causally relating the events of the first trace to the events of the second trace that have dependent labels. Let $t_1, t_2 \in Tr(\Sigma, I)$ with $t_1 = (E_1, \preceq_1, \lambda_1)$ and $t_2 = (E_2, \preceq_2, \lambda_2)$. Assume without loss of generality that $E_1$ and $E_2$ are disjoint. Then, $t_1.t_2 = (E, \preceq, \lambda)$ where $E = E_1 \cup E_2$, $\preceq = \preceq_1 \cup \preceq_2 \cup \{(e_1, e_2) | e_1 \in E_1, e_2 \in E_2, (\lambda(e_1), \lambda(e_2)) \notin I\}$, and the labeling function is given by $\lambda(e) = \lambda_i(e_i)$ if $e_i \in E_i$, where $i \in \{1, 2\}$. The subtrace relation is defined as $t \preceq t'$ if there exists $t_1$ such that $t.t_1 = t_2$.

![Figure 2.1: Sample Petri net and its associated traces.](image)

### 2.1.3 Traces of a Petri Net

Let us now define the set of traces generated by a Petri net. Given a marked net $(N, Init)$, $N = (P, T, F)$, we consider the trace alphabet $(\Sigma, I)$ where $\Sigma = T$, and $(t, t') \in I$ if and only if the neighborhoods of $t$ and $t'$ are disjoint, i.e. $(\bullet t \cup t') \cap (\bullet t' \cup t'') = \emptyset$.

Now the traces generated by the net is defined as $\{Tr(\sigma) \mid \sigma \in FS(N, Init)\}$. It is easy to see that the set of all linearizations of the traces generated by the net is precisely the set of firing sequences of the net. A single trace, however, identifies several linearizations into a single structure and represents a partially-ordered behavior that explicitly encodes the causal dependence between events, which we will exploit to define atomicity. For example, Figure 2.1 displays a marked Petri net (Init is denoted by black dots) and its two traces $T_1$ and $T_2$. 

![Figure 2.1: Sample Petri net and its associated traces.](image)
2.1.4 Net Unfoldings

Consider the control net in Figure 2.2(a). Places and transitions have unique labels, and the initial marking is \( \{p_1, p'_1, Y_1, Y_2\} \). To build the unfolding of this net (Figure 2.2(b)), we start by the set of places in the initial marking. At each step, a copy of transition \( t \), for which the set of places in \( \bullet t \) are available, is added to the unfolding, and the set of places in \( t^* \) are as a result added. We call these copies events. These events are labeled by their corresponding transitions. Note that for each place \( p \) in \( t^* \), a new place is added to the unfolding even if \( p \) already existed. For example, transition \( t_1 \) can be added after the initial marking, and the places \( p_2, Y_1, Y_2 \) are added after that, although \( Y_1 \) and \( Y_2 \) already existed as part of the initial marking. The only constraint for adding a transition \( t \) is that the set of places in \( \bullet t \) should all be concurrent; that is not causally related and is not in conflict. Also, the same set of places cannot be used more than once to generate the same transition. This way, the unfolding for the control net in Figure 2.2(a) is infinite, part of which is illustrated in Figure 2.2(b).

An interesting property of a net unfolding is that it contains (exactly) the set of all reachable markings of the original net \[McM95, ERV96\]. Therefore, all the queries of coverability/reachability can be answered on the unfolding instead of the net itself.

Another interesting property of net unfoldings is that if the original Petri net is 1-safe then there exists a finite prefix of the unfolding of the net which contains all the reachable markings of the original net \[McM95, ERV96\]. It is also shown in \[ERV96\] that this finite prefix has size \( O(n) \) where \( n \) is the number of reachable markings of the original net. In Figure 2.2(b), the unshaded area contains the finite prefix of the unfolding of the net in Figure 2.2(a). The double-lined squares indicate the so-called cut-off events which mark the boundaries of the finite prefix.

2.2 Rewriting Logic Operational Semantics

In this section, we present an introduction to rewriting logic and the Maude system. We indicate how rewriting logic operational semantics of programming languages can be a desirable approach to build formal analysis frameworks for multithreaded programs.

2.2.1 Rewriting Logic

Equational semantics and structural operational semantics are two well-known semantics frameworks. In equational semantics formal definitions take the form of semantic equations, typically

---

2A Petri net is 1-safe if no place can contain more than one token in it. Control nets are by definition 1-safe.
satisfying the Church-Rosser property \[\text{BN99}\]. Equational semantics has a proof-theoretic operational semantics given by equational reduction with the semantic equations. Semantic definitions can be easily turned into efficient interpreters, thanks to efficient higher-order and first-order equational languages. Equational semantics, although well-suited for deterministic languages (e.g. conventional sequential languages), are quite poorly fit to define the semantics of concurrent languages.

**Structural Operational Semantics (SOS)** \[\text{Plo81}\] is also a proof-theoretic approach, focusing on giving a detailed step-by-step formal description of a program’s execution. SOS is fairly well suited to model concurrent languages and can also deal with the detailed execution of deterministic languages. SOS has some drawbacks; so far, it has no commonly accepted model-theoretic semantics. SOS imposes a centralized *interleaving semantics* of concurrent computations in its standard formulation, which may be unnatural in some cases. Also, the tool support for SOS is considerably less developed than for equational semantics.

**Rewriting logic** \[\text{Mes92a}\] is a very simple computational logic. It unifies equational semantics and SOS within a common semantic framework, combining the best features of both while overcoming their respective limitations. It has several high-performance implementations (Maude \[\text{CDE}^+03\], CafeOBJ \[\text{CDEM00}\], ELAN \[\text{BKKM02}\]) of which the most comprehensive so far, in expressiveness and in range of features, is probably the Maude system. There is a natural mapping of SOS definitions into rewriting logic \[\text{BHMM02}\].

Figure 2.2: Net unfolding.
A rewrite theory \( \mathcal{R} \) is a triple \( \mathcal{R} = (\Sigma, E, R) \) where \( (\Sigma, E) \) is an equational theory with signature \( \Sigma \) and equations \( E \), and \( R \) is a set of labeled rewrite rules of the form \( l : t \rightarrow t' \) if cond, with \( l \) a label, \( t \) and \( t' \) \( \Sigma \)-terms, and cond a condition (i.e. a guard). Intuitively, \( \mathcal{R} \) specifies a concurrent system, whose states are elements of the initial algebra \( T_{\Sigma/E} \) specified by \( (\Sigma, E) \), and whose concurrent transitions are specified by rules \( R \). Note that when the set \( R \) of rules is empty, \( \mathcal{R} \) becomes an equational theory. Therefore, equational logic appears as a special, degenerate case of rewriting logic. Rewriting logic has rules of deduction that are sound and complete with respect to its model-theoretic semantics, which is a true concurrency semantics (as opposed to interleaving). Figure 2.3 presents the inference rules of rewriting logic for the case of unconditional rules.

Rewriting logic provides a crucial distinction between semantic equations \( E \) and semantic rules \( R \), that is a distinction between deterministic and concurrent computations, which is not available in either equational semantics or SOS.

---

**1. Reflexivity.** For each \( [t] \in T_{\Sigma/E}(X) \),

\[
[t] \rightarrow [t].
\]

**2. Congruence.** For each \( f \in \Sigma^n, n \in \mathbb{N} \),

\[
[t_1] \rightarrow [t'_1], \ldots, [t_n] \rightarrow [t'_n], \quad [f(t_1, \ldots, t_n)] \rightarrow [f(t'_1, \ldots, t'_n)].
\]

**3. Replacement.** For each rule \( r : t(x_1, \ldots, x_n) \rightarrow t'(x_1, \ldots, x_n) \) in \( R \),

\[
[w_1] \rightarrow [w'_1], \ldots, [w_n] \rightarrow [w'_n], \quad [t(w_1/x) \ldots w_n/x] \rightarrow [t'(w'_1/x) \ldots w'_n/x]
\]

**4. Transitivity.**

\[
[t_1] \rightarrow [t_2], [t_2] \rightarrow [t_3] \quad \rightarrow \quad [t_1] \rightarrow [t_3]
\]

**Figure 2.3: Rewriting Logic Inference Rules.**

One can define the semantics of a concurrent programming language, say Java, by specifying a rewrite theory \( \mathcal{R}_{\text{Java}} = (\Sigma_{\text{Java}}, E_{\text{Java}}, R_{\text{Java}}) \) where:

- the signature \( \Sigma_{\text{Java}} \) specifies the Java syntax and auxiliary operators needed in semantic definitions (memory, environment, etc),

- the equations \( E_{\text{Java}} \) specify the semantics of all the deterministic features of Java and of the auxiliary semantic operations.

- the rewrite rules \( R_{\text{Java}} \) specify the semantics of all the concurrent features of Java.
More generally, one can use a rewrite theory to formally specify any concurrent system, such as a distributed algorithm, a network protocol, or a web service. Furthermore, under reasonable assumptions, one can program such systems directly with rewrite rules. This is the point of Maude language that we explain below.

### 2.2.2 Maude

We use Maude \cite{CDE02} as the rewriting language and system to specify the semantics of Java and the JVM in JavaFAN, and to also specify various distributed algorithms and hardware architectures in examples we present later in this thesis. Maude is a high-level language and a high-performance system supporting executable specification and declarative programming in rewriting logic. Since rewriting logic contains equational logic, Maude also supports equational specification and programming in its sub-language of functional modules and theories. The underlying equational logic chosen for Maude is membership equational logic, that has sorts, sub-sorts, operator overloading, and partiality definable by membership and equality conditions. Using Maude and its associated tools, formal executable specification of a programming language (e.g. Java) can then be analyzed in a variety of ways, including symbolic simulation and debugging, searching for safety violations, and flexible forms of model checking.

Figure 2.4 shows a simple Maude module that implements the Peano notation of natural numbers. Keywords `sort` and `op` refer to types and the operations respectively. In the example, we have the sort `Nat` for natural numbers, with the successor operator `s()` and the addition operator `+`. The zero is declared as an operator without arguments, which is regarded as a constant in the rewriting logic.

The addition operator “`op _+_ : Nat Nat -> Nat [assoc comm id: 0]`” takes two natural numbers (of sort `Nat`) and returns another natural number. Maude allows attributes such as associativity (`assoc`), commutativity (`comm`), identity (`id:`), and precedence to be associated with operators. For example, the addition operation is associative and commutative with the constant 0 as the identity.

The equations and rewrite rules in Maude are specified by the keywords `eq` and `rl`, respectively. Equations in this example define the semantics of the addition operation.

```maude
fmod PEANO-NAT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  vars N M : Nat .
  eq s(N) + M = s(N + M) .
  eq 0 + M = M .
endfm
```

Figure 2.4: Maude Specification for Natural Numbers.
2.3 Partial Order Reduction

A finite transition system is a tuple \((S, S_0, T, AP, L)\), where \(S\) is a finite set of states, \(S_0 \subseteq S\) is the set of initial states, \(T\) is a finite set of transitions such that \(\alpha \in T\) is a partial function \(\alpha : S \rightarrow S\), \(AP\) is a finite set of propositions and \(L : S \rightarrow 2^{AP}\) is the labeling function. A transition \(\alpha\) is enabled in a state \(S\) if \(\alpha(s)\) is defined. Denote by \(\text{enabled}(s)\) the set of transitions enabled in \(s\). The main goal of partial order reductions is to find a subset of enabled transitions \(\text{ample}(s) \subseteq \text{enabled}(s)\) that is used to construct a reduced state space that is behaviorally equivalent to that of the original program. Partial order reduction is based on several observations about the nature of concurrent computations. The first observation is that concurrent transitions are often commutative, which is expressed in terms of an independence relation, \(I \subseteq T \times T\), that is, a symmetric and anti-reflexive relation which satisfies the following condition: for each \((\alpha, \beta) \in I\), and for each state \(s\), if \(\alpha, \beta \in \text{enabled}(s)\) then: (1) \(\alpha \in \text{enabled}(\beta(s))\) and \(\beta \in \text{enabled}(\alpha(s))\), and (2) \(\alpha(\beta(s)) = \beta(\alpha(s))\).

Note that \(D = (T \times T) \setminus I\) is the dependence relation. The second observation is that in many cases only a few transitions can change the value of the propositions, which suggests the concept of visibility; a transition \(\alpha \in T\) is invisible if for each \(s \in S\), if \(s' = \alpha(s)\) then \(L(s) = L(s')\).

There are several existing heuristics to compute \(\text{ample}(s)\). [CGP01] gives a set of four conditions that, if satisfied by \(\text{ample}(s)\), guarantee a correct reduction of the given state transition system. In Section 8.2.3, we present a special case of the conditions in [CGP01] which are used in this paper.

2.4 Termination, Confluence and Coherence in Rewrite Theories

A rewrite theory \([\text{Mes92b}]\) is a triple \(R = (\Sigma, E, R)\) where \((\Sigma, E)\) is an equational theory with signature \(\Sigma\) and equations \(E\), and where \(R\) is a set of conditional rewrite rules of the form \(l \rightarrow r\) if \(C\). In this paper we assume that \(C\) is always an equational condition. Intuitively, if a concurrent system is modeled as a rewrite theory \(R = (\Sigma, E, R)\), then the equational theory \((\Sigma, E)\) defines the system states (terms in \(T_{\Sigma/E}\)) and the set of rewrite rules \(R\) specify the system’s concurrent transitions.

Given two terms \(u, v \in T_{\Sigma/E}\), a one-step rewrite \(u \xrightarrow{\tau} v\) means that there is a rule \(\tau : l \rightarrow r\) if \(C\) in \(R\) that can be applied to a subterm of \(u\) with a ground substitution \(\theta\) such that \(E \models \theta C\) and \(u\) rewrites to \(v\) by replacing the subterm \(\theta(l)\) by the subterm \(\theta(r)\). We write \(u \xrightarrow{R} v\) to mean that there is a rule \(\tau \in R\) such that \(u \xrightarrow{\tau} v\). The notation \(\xrightarrow{R}^*\) denotes the reflexive and transitive closure of the relation \(\xrightarrow{R}\). We define \(\xrightarrow{S}_{\equiv} = \{(x, y) \mid t \xrightarrow{S}^* y \land y \in \text{Can}(S)\}\). Rewriting over equivalence classes modulo equations \(E\) is defined as follows: \([t]_E \xrightarrow{R} [t']_E\) if and only if there are terms \(u\) and \(v\) such that \(u \in [t]_E\) and \(v \in [t']_E\) and \(u \xrightarrow{R} v\). We define \(\xrightarrow{R/E}\) by the equivalence
$[t]_E \xrightarrow{R/E} [t']_E \iff t \xrightarrow{R} t'$.

A set $S \subseteq R$ of rewrite rules is confluent modulo $E$ in the theory $(\Sigma, E, R)$ if and only if $\forall t, t', t'' \in T_{\Sigma/E} : ([t]_E \xrightarrow{*} [t']_E \wedge [t]_E \xrightarrow{*} [t'']_E) \Rightarrow (\exists [w]_E \in [t]_E \wedge [t']_E \xrightarrow{R} [w]_E \wedge [t'']_E \xrightarrow{R} [w]_E)$. $S$ is terminating if for all $[t]_E$ there exists no infinite chain of rewriting $[t]_E \xrightarrow{R/A} \ldots \xrightarrow{R/A} \ldots$.

**Definition 3** [Vir02] In a rewrite theory $R = (\Sigma, E, R)$, where $E = E_0 \cup A$ with $E_0$ a set of equations and $A$ a set of equational axioms, $R$ is called locally strongly coherent with respect to $E_0$ modulo $A$ if

\[
(t \xrightarrow{R/A} t_1 \wedge t \xrightarrow{E_0/A} t_2) \Rightarrow (\exists t_3, t_4 : t_2 \xrightarrow{E_0/A} t_3 \wedge t_3 \xrightarrow{R/A} t_4 \wedge t_4 \xrightarrow{E_0/A} t_1)
\]

Strong local coherence is the main property to check to ensure executability of a rewrite theory $R = (\Sigma, E_0 \cup A, R)$ when we have matching algorithms for the equational axioms $A$. Viry shows that if the equations $E_0$ are confluent and terminating modulo $A$, then strong local coherence implies a more general strong coherence property [Vir02]. Strong coherence ensures that we can achieve the effect of rewriting with $R$ in $E_0 \cup A$-equivalence classes by first computing the $E_0 \cup A$-canonical form modulo $A$, and then rewriting that canonical form with $R$ modulo $A$.

### 2.5 Stuttering Simulations

Let us assume that the equational part $(\Sigma, E)$ of a rewrite theory $R = (\Sigma, E, R)$ defines, among other things a set $P$ of state predicates on the initial algebra $T_{\Sigma/E}$. We can then associate to $R$ a Kripke structure [CGP01] whose states are the set $T_{\Sigma/E,\text{State}}$ for some designated sort $\text{State}$ of states, whose labeling function assigns to each state the predicates $p \in P$ that provably hold in it using $E$, and whose transition relation is the total closure of $\xrightarrow{R}$, that is, we make $\xrightarrow{R}$ into a total relation by adding identity transitions for each deadlock state. We can then interpret any temporal logic formula, say in $CTL^*$ in $R$, namely by interpreting it in its associated Kripke structure. For a more detailed presentation on the relations between rewrite theories, Kripke structures and temporal logic, with applications to model checking in Maude see [EMS03].

**Definition 4** A Kripke structure is a triple $A = (A, \xrightarrow{A}, L_A)$, where $(A, \xrightarrow{A})$ is a transition system with $\xrightarrow{A}$ a total relation, and $L_A : A \to \mathcal{P}(AP)$ is a labeling function associating to each state the set of atomic propositions (a subset of $AP$) that hold in it.
We use the notation $a \xrightarrow{A} b$ to say that $(a, b) \in A \rightarrow$. Note that the transition relation of a Kripke structure must be total, that is, for each $a \in A$ there is a $b \in A$ such that $a \xrightarrow{A} b$. Given an arbitrary relation $\rightarrow$, we write $\rightarrow \bullet$ for the total relation that extends $\rightarrow$ by adding a pair $a \rightarrow \bullet a$ for each $a$ such that there is no $b$ with $a \rightarrow b$.

We now present some basic notions and results about transition systems, Kripke structures, and stuttering (bi-)simulations between them that will apply in particular to the Kripke structures associated to rewrite theories and that will be needed later. In the following definitions, we assume that the Kripke structures are total.

**Definition 5** Let $A = (A, \xrightarrow{A})$ and $B = (B, \xrightarrow{B})$ be transition systems and let $H \subseteq A \times B$ be a relation. Given a path $\pi$ in $A$ and a path $\rho$ in $B$, we say that $\rho H$-matches $\pi$ if there are strictly increasing functions $\alpha, \beta : \mathbb{N} \to \mathbb{N}$ with $\alpha(0) = \beta(0) = 0$ such that, for all $i, j, k \in \mathbb{N}$, if $\alpha(i) \leq j < \alpha(i + 1)$ and $\beta(i) \leq k < \beta(i + 1)$, it holds that $\pi(j) H \rho(k)$.

**Definition 6** Given transition systems $A$ and $B$, a stuttering simulation of transition systems $H : A \rightarrow B$ is a binary relation $H \subseteq A \times B$ such that if $a H b$, then for each path $\pi$ in $A$ starting at $a$ there is a path $\rho$ in $B$ starting at $b$ that $H$-matches $\pi$.

**Definition 7** Given Kripke structures $A = (A, \xrightarrow{A}, L_A)$ and $B = (B, \xrightarrow{B}, L_B)$ over a set of predicates $P$, a stuttering $P$-simulation $H : A \rightarrow B$ is a stuttering simulation of transition systems $H : (A, \xrightarrow{A}) \rightarrow (B, \xrightarrow{B})$ such that if $a H b$ then $L_B(b) \subseteq L_A(a)$. We call the stuttering $P$-simulation strict if $a H b$ implies $L_B(b) = L_A(a)$. $H$ is called a stuttering $P$-bisimulation if both $H$ and $H^{-1}$ are stuttering $P$-simulations.

By results from [CGP01, Man01, PMMO05] we know that stuttering $P$-simulations preserve the satisfaction of $ACTL^*_{\neg X}(P)$ formulas and that stuttering $P$-bisimulations preserve the satisfaction of $CTL^*_{\neg X}(P)$ formulas.
Chapter 3

The Control Net

Given a program $P$ with multiple threads, we exhibit a clean control model of the program as a Petri net \[\text{LMRT90}\] that explicitly captures: (a) the independence of execution of threads, (b) the interaction mechanism of the threads (using shared variables, locks, etc.), and (c) the abstraction of the program into a structure that loses track of data in the program, retaining only the control structure of $P$. The model for the program is independent of any notion of atomicity, and captures just the dependence and independence of control in the threads of the program.

This model of a Petri net generates, in a standard way, partially-ordered runs of $P$ that depict possible control interactions between threads \[\text{DR95}\]; we call these partially-ordered runs the control traces of $P$. The partially ordered control traces depict the set of events that have occurred and also define the causal relation between these events; such a causal structure is not evident in the linear runs of $P$.

We use this model to model and check atomicity of code blocks in concurrent programs \[\text{FM06}\] as explained in Chapter [4] as well as computing the causal dataflow solutions for concurrent programs \[\text{FM07}\] in Chapter [5]. We believe that the control net model is an excellent candidate for capturing control flow in concurrent programs, and can emerge as the robust analog of control-flow graphs for sequential programs. The size of a control net is linear in the size of the program it models, and can be efficiently computed just from the structure of the program without any analysis, except for parsing.

It statically captures certain obvious dependencies and independencies of statements. However, the global causal dependency of statements is apparent neither syntactically nor through simple graph reachability on the control net. For instance, two statements in different threads acquiring the same lock are obviously dependent in every context where they are mutually enabled, and this information is syntactically captured in the net. However, finding whether a definition of a global variable $x$ flows (without being redefined) to a use of $x$ in another thread is not easy, as it crucially depends on the locks protecting these definitions, and cannot be determined without analyzing the runs of the program. This is markedly different from static concurrency in simple fork-join programs where the causal dependence is not dynamic, and is apparent readily from code \[\text{LMP97, LPM99}\].
In this chapter, we explain the control net model for a simple language called SML. The ideas are completely general, so the control net can be defined for many program constructs in many languages.

### 3.1 The Language for Programs

We base our formal development on the language SML (Simple Multithreaded Language). Figure 3.1 presents the syntax of SML. The number of threads in an SML program is fixed and preset. There are two kinds of variables: local and global, respectively identified by the sets $GVar$ and $LVar$. All the variables that appear at the definition list of the program are global and shared among all threads. Any other variable that is used in a thread is assumed to be local to the thread.

It is assumed that all variables are integers and are initialized to zero. We use small letters (resp. capital letter) to denote local (resp. global) variables. $Lock$ is a global set of locks that the threads can use for synchronization purposes through acquire and release primitives.

| $P$ ::= | defn thlist | (program) |
| $thlist$ ::= | null | stmt || thlist | (thread list) |
| $defn$ ::= | int $x$ | lock $l$ | defn ; defn | (variable declaration) |
| $stmt$ ::= | $x$ ::= $e$ | (expression) |
| | while ($b$) { stmt } | begin stmt end |
| | if ($b$) { stmt } else { stmt } | skip |
| | acquire($l$) | release($l$) | stmt ; stmt | (statement) |
| $e$ ::= | $i$ | $x$ | $Y$ | $e + e$ | $e * e$ | $e / e$ |
| $b$ ::= | true | false | $e$ op $e$ | $b \lor b$ | $\neg b$ | (boolean expression) |

$op \in \{<, \leq, >, \geq, =, ! =\}$

$x \in LVar, Y \in GVar$

$i \in Integer, l \in Lock$

Figure 3.1: SML syntax

begin and end primitives are used to mark the beginning and end of a block that is intended to be checked for atomicity. The goal of the atomicity checker is to check whether all such blocks are indeed atomic. We will assume all blocks marked atomic occur only in one thread.
Figure 3.2: Model Construction
3.2 Modeling programs using Petri Nets

We model the flow of control in SML programs by Petri nets. This modeling formally captures the concurrency between threads using the concurrency of the Petri net, captures synchronizations between threads (e.g., locks, accesses to global variables) using appropriate mechanisms in the net, and finally formalizes the fact that data is abstracted in a sound manner.

Figure 3.2 illustrates the function $N$ that models statements using nets (inductively, for a fixed number of threads $n$). $N(S)$ is defined to have a unique entry place $p^S_m$ and one or more exit transitions $tx^S_1, \ldots, tx^S_m$. In this natural way of modeling the control of a program, transitions correspond to program statements, and places are used to control the flow, interdependencies, and synchronization primitives. Figure 2.1 illustrates the Petri net model for the program (a) from Figure 4.2.

There is a place $l$ associated to each lock $l$ which initially has a token in it. To acquire a lock, this token has to be available which then is taken and put back when the lock is released. This ensures that at most one thread can hold the lock at any time.

For each defined global variable $Y$, there are $n$ (the preset number of threads in the program) places $Y_0, \ldots, Y_n$, one per each thread. Every time the thread $T_i$ reads the variable $Y$ ($Y$ appears in an expression), it takes the token from the place $Y_i$ and puts it back immediately. If $T_i$ wants to write $Y$ ($Y$ is on the left side of an assignment), it has to take one token from each place $Y_j$, $1 \leq j \leq n$. This is to ensure causality: two read operations of the same variable by different threads will be
independent (as their neighborhoods will be disjoint), but a read and a write, or two writes are declared dependent. If \( N_i = (P_i, T_i, F_i) \) is the Petri net model for statement \( S_i \) (1 ≤ \( i \) ≤ \( n \)), then the Petri net model for \( S_1||\ldots||S_n \) is the net \((P_1 \cup \cdots \cup P_n, T_1 \cup \cdots \cup T_n, F_1 \cdots \cup \ldots \cup F_n)\), assuming \( T_i \)s are disjoint. Note the over-abstraction: if a read and a write of two threads are simultaneously enabled (i.e., if there is a race condition), then the order on their accesses may be crucial. Since we are not keeping track of data in any manner, we declare them to be causally dependent and hence will consider the two runs as two inequivalent traces. The dependency relation defined in the model will lead to the appropriate notion of causality in the traces of the Petri net. Figure 3.3 shows examples of multithreaded programs written in SML and its control net.

For a firing sequence \( \sigma \) of the net corresponding to a program, the sequence \( \sigma \) may not be feasible in the concrete program (because of the abstraction of data values). However, note that for every feasible sequence of the concrete program, its control trace is a trace of the net. Moreover, if \( \sigma \) is a firing sequence of the net which is feasible in the program (say by a concrete run \( r \)), then it is easy to see that for each firing sequence \( \sigma' \) such that \( \sigma' \in [\sigma] \), there is a concrete run \( r' \) corresponding to it in the program that is equivalent to \( r \) (in terms of resulting in the same valuation of concrete variables). This property is key in ensuring that our entire approach is sound, as we will use trace equivalence as the equivalence over runs in defining atomicity in Chapter 4.
Chapter 4

Causal Atomicity

A code block is atomic \cite{FFL03} if for every interleaved execution of the program in which the code block is executed, there is an equivalent run of the program where the code block is executed sequentially (without interleaving with other threads). We propose a new notion of atomicity for programs based on causality. Causal atomicity is then defined using the causal structure of the control traces (partially-ordered runs) generated by the program. Intuitively, we consider two sequential executions of a program to be equivalent if and only if they correspond to linearizations of the same partially-ordered trace of the program. Causal atomicity reduces to a very simple property on partially ordered traces: a block $B$ of a thread is causally atomic if there is no control trace of the program where an event of another thread occurs causally after the beginning of $B$ and causally before another event that is within the same block $B$.

Our notion of causal atomicity is simple and yet powerful enough to capture common interaction disciplines in correct programs. Our notion is certainly stronger than looking for patterns of transactions \cite{Lip75,FFL03}, and can handle programs that do not explicitly use locks. More importantly, we believe that our notion gives a clear definition of atomicity for static-analysis, making explicit the data-abstraction, emphasizing control and interaction between threads.

The notion of transactions is a local property of programs that assures atomicity of blocks in the program no matter in which context the program is placed in. In contrast, causal atomicity is a global notion that takes into account the entire program in order to determine whether blocks are atomic. Consequently, transactions can be seen as sound local checks that ensure global causal atomicity. We make this intuition precise by defining the notion of transactions in our setup, and in fact show a nice property of transactions: transactions are the weakest local property that ensure causal atomicity of a program, no matter which context it is put in. This is the first result we know that formally argues why the notion of transactions is the best local notion of atomicity.

Turning to algorithms for checking atomicity, we show how causal atomicity can be checked using partial-order techniques based on unfoldings of Petri nets. Our algorithm is sound and complete in checking for causal atomicity of the net. Given a Petri net model of $P$ with a block marked to be checked for atomicity, we show how to reduce the problem of checking causal atomicity of $P$ to a coverability problem for an associated colored Petri net $Q$ \cite{Jen96}. The latter problem is
decidable and there are several tools that can efficiently check coverability in colored Petri nets. In particular, the tools that use unfolding techniques \cite{McM95,ERV96} of nets are useful, as they exploit the structure of the net to give automatic reduction in state-space (akin to partial-order reduction that has been exploited in model checking concurrent systems).

Finally, we show that causal atomicity is a common paradigm in many programs by considering several examples. We report on experiments that reduce checking causal atomicity to coverability, where the latter is solved using the PEP tool (Programming Environment based on Petri nets \cite{Gra97}). The experiments show that causal atomicity lends itself to fast partial-order based model-checking.

When there is only one block that is being checked for atomicity, our notion of atomicity is the same as the notion of serializability studied for database transactions \cite{FR82,AMP00}. However, when there are multiple blocks, serializability demands that for every execution, there is one execution where all the atomic blocks are executed serially, while our notion demands that for every execution and every block, there is some execution where that block occurs sequentially. We believe our notion is more appropriate for threaded software. Figure 4.1 shows an example of a trace of a program with four threads and two blocks which intuitively ought to be declared atomic. For instance, any pre-post condition of the block \(B\) (or \(B'\)) that depends only on the variables used in the block holds on all interleaved runs provided it holds in runs where the block is executed sequentially. Note that a program with such a trace would be declared non-serializable, but is causally atomic.

![Figure 4.1: Not serializable but causally atomic](image)

While we believe the jury is still out on which of these notions of atomicity is useful and accurate for checking programs, note that our contributions hold equally well for serializability: we can define a notion of serializability using the causal edges in the Petri net model and check for it using unfolding algorithms (however, checking serializability seems more complex than checking causal atomicity).
4.1 Causal Atomicity

Recall the general notion of atomicity: a block is atomic if for every sequential execution in which it is executed, there is another equivalent sequential execution where the block is executed without being interleaved with other threads. Given our abstraction using a Petri net, the only reasonable definition of equivalence of two sequential executions is that they are linearizations of the same control trace (see Section 3.2).

Let us first illustrate the concept of causal atomicity by a simple example. Consider the two programs in Figure 4.2. Although the first thread (on the left) is the same in both versions, the block within begin and end is atomic in 4.2(b) and not atomic in 4.2(a).

The Petri net model of a program $P$ induces the traces that correspond to the partially ordered runs of the program. We call these traces the control traces of program $P$. Causal atomicity is then defined as a property of these control traces. Figure 4.3 shows a control trace of the non-atomic program of Figure 4.2(a). Here labels of the events (transitions given by $\lambda$) are mentioned instead of the event names themselves to make the trace more readable. The arrows depict the immediate causality relation. The trace is a witness for non-atomicity, since $x := Y - 2$ is causally after $Y := 5$ and causally before $Y := 3$, and therefore in all linearizations of this trace, $x := Y - 2$ has to appear in the middle of the block.

A notational remark: when we denote a transition as $t^T_i$, we mean that it is a transition that belongs to the thread $T_i$.

Definition 8 A code block $B = \text{begin } S \text{ end}$ of the program $P$ is causally atomic if and only
if the Petri net model of $P$ induces a trace $Tr = (E, \preceq, \lambda)$ for which the following does not hold:

$$\exists e_1, e_2, f \in E : e_1 \preceq f \preceq e_2 \quad \text{where} \quad \lambda(e_1) = t^T_{\text{begin}}, \lambda(e_2) = t^T_T, \lambda(f) = t^{T'}_3 \quad \text{such that} \quad T \neq T' \quad \text{and} \quad \not\exists e \in E : (\lambda(e) = t^T_{\text{end}} \land e_1 \preceq e \preceq e_2)$$

The above definition says that a block declared atomic is not causally atomic if the block begins, and there are two events, $e_2$ belonging to the same thread (and $e_2$ occurs before the block finishes) and $f$ belonging to another thread such that $f$ occurs causally between the beginning of the block and $e_2$. Note that traces that witness non-atomicity may not even execute the block completely (and we do not require any termination requirement for blocks).

The following theorem captures the intuition of why the above defines causal atomicity; it argues that if a trace of the program is not of the kind mentioned in the definition above, then there is indeed some linearization of events that executes the atomic block without interleaving.

**Theorem 9 (a)** A code block $B = \text{begin } S \text{ end}$ of the program $P$ is causally atomic if and only if for all finite traces induced by the Petri net model of $P$, there is a linearization the trace where all atomic blocks occur sequentially (without interleaving with other threads).

**Theorem 9 (b)** If a code block $B = \text{begin } S \text{ end}$ of the program $P$ is causally atomic then for all infinite traces induced by the Petri net model of $P$, there is a linearization of a subset of the events of the trace which contains all events belonging to the thread containing the atomic block, in which all atomic blocks occur sequentially.

**Proof.**

(a) If a code block is not causally atomic, then the witness trace (as given in the definition of causal atomicity) is such that every linearization of the trace includes the event $f$ within the atomic block. Conversely, assume that a program is causally atomic and let $t$ be a finite trace of it. Let $E_T$ be the set of events of the thread $T$, where $T$ is the thread that contains the atomic blocks. Note that $E_T$ is finite and linearly ordered. Consider the first time the block begins in $E_T$ (if there is none, we are done); let this event be $e_1$. Let $F$ be the set of all events in $E_T$ that are causally after $e_1$ (including $e_1$), up until the first event after $e_1$ that denotes the end of the block. Let $G$ be the set of events immediately causally before the events in $F$ that are in threads other than $T$. We can now find a linearization where we execute the causal closure of $e_1$ (excluding $e_1$), and then the causal closure of $G$. Note that while executing the causal

---

1See Section 2.1.2 for a formal definition of a trace.
closure of \( G \), we will never encounter an event \( f \) which is causally after \( e_1 \); if we do, the trace violates the condition required for causal atomicity. Hence \( e_1 \) would not have been executed in the above linearization. After the above linearization, we can proceed to execute the events in \( F \) serially starting from \( e_1 \), as all events that \( F \) causally depends on in other threads have been executed. Having finished this occurrence of the atomic block, we can inductively move to the next instance where the block is executed in \( E_T \) and grow the linearization in the same manner. Finally we would construct a linearization of the trace where thread \( T \) executes the block each time in a serial fashion.

(b) Assume that a code block is indeed causally atomic, and let \( t \) be an infinite trace of the program. The argument above does not work for infinite traces: for example, if there are two completely independent threads, one which has an atomic block that never ends, and the other thread has an infinite number of events, then there is no \( \omega \)-linearization of the trace where the block executes without intervention. So we weaken the claim and argue that there is at least a sub-trace of \( t \) that contains all events of the thread \( T \), which has a linearization that executes the block atomically.

Again, let \( E_T \), \( F \), and \( G \) be defined as in the earlier argument. Notice that if \( F \) and \( G \) are both infinite, our strategy of executing \( G \) before beginning \( F \) will not work. We argue that \( G \) must be finite.

Recall that there is an underlying trace alphabet. Assume \( G \) is infinite. Then there would be a thread \( T' \) (\( T' \neq T \)) that has infinitely many immediate predecessors of \( F \). By finiteness of the labels, it follows that there must be two labels \( a \) and \( b \) such that there are two events \( e_1 \) and \( e_2 \) in \( F \) labeled \( a \) such that their immediate predecessors are in thread \( T' \) and are labeled \( b \), where \( aDb \). Now by symmetry of the dependence relation\(^2\), it follows that these four events witness a trace for non causal atomicity.

Having established that the set of immediate causal predecessors of \( F \) is finite, we can go ahead and construct the linearization exactly as in the previous argument.

\[ \blacksquare \]

Note that Theorem 9 yields soundness of our approach: if a code block \( B \) is causally atomic, then by the above theorem and by the fact that either every linearization of a trace of the net is feasible in the concrete program or none are, it follows that the block \( B \) is atomic in the concrete program as well.

The program in Figure 4.4 distinguishes causal atomicity from other static notions of atomicity in the literature. According to the notion of causal atomicity the code block in thread \( T_2 \) is atomic. However, since there are races on both global variables \( X \) and \( Y \), both statements \( X = 1 \) and \( Y = 1 \)

---

\(^2\)See Section 2.1.2 to recall the definition of the dependence relation.
are non-mover statements and this block is not a transaction, and therefore will not be detected as atomic by methods in [FFL03]. Our notion of causal atomicity is also behavioral and more geared towards model checking, as it depends on the partial-order executions of the program, not on the static structure of the code.

Commit-atomicity [Fla04] is a dynamic notion of atomicity which is different from our static notion. The presence of data in commit-atomicity allows a more precise detection of atomicity according to the general definition of atomicity, and there are examples (e.g. see Bluetooth3 in Section 4.3) that can be detected as atomic by commit-atomicity, but fail the causal atomicity check. On the other hand, the presence of data limits commit-atomicity to finite state space programs, and hinders scalability (specifically, in terms of number of threads). Instead, causal atomicity can deal with infinite data since the data is completely abstracted. Also, the commit-atomicity method requires the atomic blocks to be terminating, while we do not need such an assumption.

**Local Causal Atomicity**

Causal atomicity is a global notion in that one has to look at all threads and their interactions to decide whether a code block in a given thread is causally atomic. Here, we define a stronger notion of causal atomicity, which we call local causal atomicity. It is stronger in the sense that local causal atomicity always implies causal atomicity. It is also easier to check, since one can decide local causal atomicity by merely looking at the code block under consideration. We then show that the notion of local causal atomicity is equivalent to the notion of a transaction [FFL03, FFQ05, FF04], which makes a transaction a special case of causal atomicity.

**Definition 10** In the control net $P$, a transition $t$ (equally an instruction) of a thread is called a NRM (respectively NLM) if and only if $t$ has a nonlocal\(^3\) place $p$ as a successor (respectively

---

\(^3\)A local place in the control net is one that is used to model a control point in a thread. A nonlocal place, is anything that is used to model interaction between the threads, including places associated to global variables, and locks.
predecessor).

**Definition 11** A code block $B$ is locally causally atomic if and only if in the Petri net model of the thread containing $B$, there is no NRM transition $t \in B$ flowing into a NLM transition $t' \in B$.

**Proposition 12** If a code block $B$ is locally causally atomic then it is causally atomic.

**Proof.** (sketch): If $B$ is not causally atomic, then there is an induced trace $Tr = (E, \preceq, \lambda)$ containing events $e_1, e_2,$ and $f$ such that $e_1 \prec f \prec e_2$, where $e_1 = \lambda(t^1_T)$, $e_2 = \lambda(t^2_T)$, and $f = \lambda(t^{T'})$. Since $e_1 \prec f$, there are events $e'_1$ and $f'$ such that $e'_1$ belongs to thread $T$ and $f'$ belongs to thread $T'$ and $e_1 \preceq e'_1 \prec f' \preceq f$. This implies that the transition $t'_1 = \lambda(e'_1)$ is a NRM. Similarly, we can show that there is an NLM transition $t'_2 = \lambda(e'_2)$ where $e_1 \prec e'_2 \preceq e_2$. This means that block $B$ is not locally causally atomic. ■

Proposition 12 shows that local causal atomicity is stronger than causal atomicity. One can observe that each transition in the Petri net can be an NRM, an NLM, both (NM), or neither (BM). Assuming a four-symbol alphabet \{NRM, NLM, NM, BM\} (representing the four possibilities), one can express the set of all possible locally causally atomic code blocks as a regular language represented by the following regular expression:

$$L_{lca} = (NLM + BM)^*(NM + e)(NRM + BM)^*$$

which covers all the strings in which a NRM (or a NM) is not followed by a NLM (or a NM).

**Remark 13** If a transition $t^T$ in the Petri net model is a NRM, then there is an induced trace of the net in which there are events $e$ ($\lambda(e) = t^T$) and $e'$ ($\lambda(e') = t'^{T'}$) such that $e \prec e'$. This means that in any execution associated with this trace, $t$ cannot move to the right of $t'$; in other words it is not a right-mover instruction.

Similarly, one can observe that a NLM instruction is not a left-mover instruction. If an instruction is both NLM and NRM, then it must be a non-mover instruction, and if it is neither NLM nor NRM, then it has to be a both-mover instruction. Recall the four-letter alphabet \{R, L, B, N\} that is the basis of the transaction model. Based on the above observations, the following hold:

$$\begin{align*}
NRM & = L + N \\
NLM & = R + N \\
BM & = B \\
NM & = N
\end{align*}$$

\[\text{Recall from Section 2.1.3 that } \preceq \text{ is the immediate causal ordering relation.}\]
Now, let us rewrite the regular expression considering the above equations:

\[
L_{\text{lea}} = ((NLM + BM)^*(NM + \epsilon)(NRM + BM)^*
\]

\[
= ((R + B) + B)^*(N + \epsilon)((L + B) + B)^*
\]

\[
= (R + B)^*(N + \epsilon)(L + B)^*
\]

which gives us the regular expression that defines transactions. This shows that the two concepts of transactions and local causal atomicity are equivalent.

We showed that local causal atomicity (similarly transactions) is a stronger notion than causal atomicity. The question that naturally comes up is whether it is too strong. We show that local causal atomicity is a necessary local condition to ensure causal atomicity — we have already shown that it is sufficient. To do this, we introduce the concept of a context. A context \( C \) for a thread \( T \) is a set of threads running concurrently with \( T \). A sufficient local condition on a code block \( B \) has to ensure the causal atomicity of \( B \) regardless of the context \( C \) for its enclosing thread \( T \).

**Proposition 14** If a code block \( B \) of thread \( T \) is causally atomic in every context \( C \), then \( B \) is locally causally atomic.

Proof. (sketch): Assume \( B \) is not locally causally atomic. We show that there is a context \( C \) for \( T \) in which \( B \) is not causally atomic. This means that in the Petri net model of \( T \), there are transitions \( t \) and \( t' \) in \( B \) such that \( t \) is an NRM and \( t' \) is an NLM and \( t \) flows to \( t' \). According to the Petri net model construction method, an NRM instruction should be one of the following: (1) \text{release}(1), (2) \text{read}(Y), or (3) \text{write}(Y), where \( Y \) is a global variable. Similarly, an NLM instruction should be one of the following: (1) \text{acquire}(1), (2) \text{read}(X), or (3) \text{write}(X), where \( Y \) is a global variable. One can easily verify that the piece of code on the right, if run in a separate thread, can make \( B \) causally non-atomic for all the nine possibilities of \( t \) and \( t' \).

Theorem 15 Local causal atomicity (and consequently the transaction model) is the strongest (meaning necessary and sufficient) local condition that ensures causal atomicity.


4.2 Checking Atomicity

In this section, we show how causal atomicity defined on traces can be reduced to coverability in a Petri net, a problem most existing Petri net analysis tools support.
4.2.1 Colored Petri Nets

Colored Petri nets are a subclass of higher level Petri nets [Jen96]. We explain how atomicity checks can be done by checking very simple properties on the colored nets. This does not imply any complications theoretically, since the results in [Jen96] show that each colored net has an equivalent Petri net, and practically since most Petri net analysis tools today support higher level nets.

We define here a very simplified definition of colored Petri nets. Essentially, in a colored Petri net, the markings are colored from a finite range of colors, and the transitions can determine how colors change when a transition is executed.

**Definition 16** A marked colored Petri net is a tuple $CPN = (P, T, F, C, E, IC)$ satisfying the requirements below:

- $(P, T, F)$ is a net
- $C$ is a finite set of colors
- For a transition $t \in T$, let $G^t_{pre}$ be the set of functions $g : \bullet t \rightarrow C$ and $G^t_{post}$ be the set of functions $g : t^* \rightarrow C$. Then $E$ is a set of color transformers $\{e_t\}_{t \in T}$, one for each transition, where $e_t : G^t_{pre} \rightarrow G^t_{post}$.
- $IC = \langle Init, c \rangle$ is a pair where $Init \subseteq P$ is an initial marking and $c : Init \rightarrow C$ is a function that determines the colors of the initial marking.

Intuitively, $G^t_{pre}$ is a coloring of markings of the pre-conditions of $t$, and $e_t$ determines how the post-conditions of $t$ will get colored when $t$ fires. A colored marking is a pair $\langle M, c \rangle$ where $M \subseteq P$ is a marking of the underlying net, and $c : M \rightarrow C$ gives colors to these markings. A transition $t$ is enabled at a colored marking $\langle M, c \rangle$ if it is enabled in the underlying net at the marking $M$, and when it fires it results in a colored marking $\langle M', c' \rangle$ where $M'$ is determined as usual, and $c'$ is determined using the coloring $e_t$ recommends on reading the coloring $c$.

Given a model of a program $P$ as a net $N = (P, T, F)$ and an initial marking $Init$, we define a colored Petri net $N' = (P', T', F', C, E, IC)$ as follows:

- $(P', T', F') = (P, T, F)$, i.e., the underlying net is the same,
- $C = \{A, B, Y, R\}$.
- $IC = \langle Init, c \rangle$ inherits the initial marking and colors all of them $A$ (achromatic).
The evolution of colors is determined by $E$, which is described below. Note, however, that since the colored net has the same underlying net, inherits the independence relation, and generates the same traces as the net modeling the program.

We use colors to monitor the executions. When the monitor detects a non-atomic execution, it can alert of this fact by raising a flag which is the appearance of a red ($R$) token in this case.

Tokens are of one of the colors: achromatic ($A$), blue ($B$), yellow ($Y$), and red ($R$). All tokens are initially achromatic ($A$). These colors change depending on the observation of certain events. If all tokens in $\bullet t$ are achromatic ($A$), it means nothing interesting has happened so far, and therefore $t$ puts achromatic tokens in all places in $t^*$. 

The begin transition, regardless of the incoming tokens, generates blue ($B$) tokens which the end transition will set back to achromatic later on. This way the block, and every resource it accesses is distinguished by the blue color. A transitions $t$ in the block will propagate the blue color to the all places in $t^*$, and as a result, to the places related to the global variables and locks that $t$ accesses. If one of these blue places is in $\bullet t'$ for a transition $t'$ of another thread — meaning that it is accessing the same global variable or lock — it will generate yellow ($Y$) output tokens recording the fact that it is causally later than the beginning of the block. In the same spirit, if a transition $t''$ of the block sees a yellow ($Y$) token in a place in $\bullet t''$, it then knows that it is a causal successor of a so-called yellow transition $t'$, which was itself a causal successor of a blue transition $t$. This implies that the block cannot be atomic, since the trio $t \preceq t' \preceq t''$ contradicts Definition 8 for causal atomicity. To indicate the observation of this violation a red ($R$) token is generated in the post-conditions.

The following simple rules show how the transitions determine the color of the outgoing tokens, based on the color of the incoming ones. Let $\bullet t = \{p, p_1, \ldots, p_n\}$, where $p$ is the local place in the thread.

- **Transitions within the atomic block:**
  - If $p$ contains a blue token, then if there is a $p_i$ with a yellow token in it, put red tokens in all places in $t^*$; otherwise put blue tokens in all places in $t^*$.
  - If $p$ contains a red token then put red tokens in all places in $t^*$.

- **Transitions outside the atomic block in the thread containing the atomic block:**
  - Regardless of the incoming token colors, put achromatic tokens in all places in $t^*$.

- **Transitions of all other threads:**
  - If all places in $\bullet t$ have achromatic tokens, put achromatic tokens in all places in $t^*$.
  - If there is a place in $\bullet t$ with a yellow token in it, put yellow tokens in all places in $t^*$.
– If there is a place in \( t \) with a blue token in it, put yellow tokens in all places in \( t' \).
– If there is a place in \( t \) with a red token in it, put red tokens in all places in \( t' \).

Thus the problem of checking atomicity in the Petri net model of a program is reduced to the problem of checking whether in the associated net, there is a reachable marking that contains a red (\( R \)) token.

**Theorem 17** A marking containing a red (\( R \)) token is reachable in CPN if and only if the block \( B \) is not causally atomic.

For implementing the above check on the colored net, we introduce new events and a new place \( p^* \); these events will detect a red token anywhere in the net and will propagate it to \( p^* \); hence we can check the transformed net for a reachable marking that contains a red-marking on \( p^* \), which is a coverability query on colored nets that Petri net tools can algorithmically solve.

### 4.3 Experiments

We have applied the method in Section 4.2.1 to check the causal atomicity of several example programs taken from [Fla04]. This section presents a brief description of each program and the performance results of our technique.

**Dekker’s mutual exclusion algorithm:** Dekker’s algorithm is a classic algorithm for mutual exclusion between two threads that uses subtle synchronization. The mutual exclusion is modeled by means of two boolean variables. We check whether the critical sections of the threads are indeed atomic. They do turn out to be causally atomic.

**Busy-waiting acquire lock:** In this example, a busy waiting while loop is used to acquire a lock. There is a forever loop that acquires the mutex using this method, then a global variable \( \text{data} \) is updated, and the mutex is released. The correctness specification requires the updating of the data to be done atomically. We have checked two different versions of this example. In one, there is only one line modifying the data (Acquire1), and the other one with several lines manipulating the data (Acquire2). Using our technique there is negligible difference in the size of the unfolding between the two cases, since the partial order semantics would not interleave the internal statements that modify the data. In contrast, the model-checking algorithm in [Fla04] uses interleavings, and they see a large increase in the time taken for (Acquire2). One can make the block in Acquire1 non-atomic by adding an extra thread that manipulates the data without acquiring the mutex. nAcquire1 refers to this case.
In Acquire1* and Dekker*, multiple blocks (one per thread) are checked for causal atomicity. This is in contrast to the rest of the benchmarks where one block is checked at a time.

**Bluetooth Device Driver:** We used a simplified version of the bluetooth device driver in Windows NT, similar to the one used in [QW04, Fla04]. There are two dispatch functions; let us call them Add and Stop. Any process that wants to use the driver calls Add, and any process that wants to stop the driver calls Stop. The correctness specification requires these two dispatch functions to run atomically. The Add function is not causally atomic, which can be verified using only two threads where one calls Add and the other one calls Stop. This turns out to be a real cause for concern in the code, as interleaving events from other threads while executing Add does make the program behave unexpectedly; this was already reported in [QW04, Fla04]. There is a fixed version of Bluetooth from [CCK+06] (Bluetooth3) which is still not causally atomic despite the fact that it is correct. However, commit-atomicity [Fla04] can detect this as atomic since it can keep track of the value of the counter in the program.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Causally Atomic?</th>
<th>#Threads</th>
<th>Unfolding #Places</th>
<th>Unfolding #Events</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker</td>
<td>Yes</td>
<td>2</td>
<td>52</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>Dekker*</td>
<td>Yes</td>
<td>2</td>
<td>795</td>
<td>409</td>
<td>0.01</td>
</tr>
<tr>
<td>Acquire1</td>
<td>Yes</td>
<td>4</td>
<td>81</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>Acquire1</td>
<td>Yes</td>
<td>30</td>
<td>2135</td>
<td>1022</td>
<td>0.03</td>
</tr>
<tr>
<td>Acquire1</td>
<td>Yes</td>
<td>100</td>
<td>21105</td>
<td>10402</td>
<td>3.71</td>
</tr>
<tr>
<td>Acquire1</td>
<td>Yes</td>
<td>150</td>
<td>46655</td>
<td>23102</td>
<td>22.31</td>
</tr>
<tr>
<td>Acquire1*</td>
<td>Yes</td>
<td>4</td>
<td>146</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>Acquire1*</td>
<td>Yes</td>
<td>30</td>
<td>4202</td>
<td>1200</td>
<td>0.35</td>
</tr>
<tr>
<td>Acquire1*</td>
<td>Yes</td>
<td>50</td>
<td>10582</td>
<td>2985</td>
<td>5.02</td>
</tr>
<tr>
<td>Acquire1*</td>
<td>Yes</td>
<td>100</td>
<td>40486</td>
<td>10784</td>
<td>635.56</td>
</tr>
<tr>
<td>nAcquire1</td>
<td>No</td>
<td>4</td>
<td>97</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>nAcquire1</td>
<td>No</td>
<td>6</td>
<td>171</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>nAcquire1</td>
<td>No</td>
<td>8</td>
<td>261</td>
<td>119</td>
<td>0</td>
</tr>
<tr>
<td>Acquire2</td>
<td>Yes</td>
<td>4</td>
<td>73</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>Acquire2</td>
<td>Yes</td>
<td>6</td>
<td>146</td>
<td>74</td>
<td>0</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>No</td>
<td>2</td>
<td>235</td>
<td>116</td>
<td>0</td>
</tr>
<tr>
<td>Bluetooth3</td>
<td>No</td>
<td>2</td>
<td>223</td>
<td>109</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Programs and Performances

**Experiment Results**

Table 4.1 shows the results of evaluating the above benchmarks using PEP [Gra97]. Each program is modeled as a colored Petri net as described in Section 4.2.1. The unfolding of the colored net is generated. Then, with a simple query, we checked whether the place before the end of the atomic
block can contain a red token. The table reports the size of the unfolding, the time taken to check for causal atomicity (in seconds), and whether the atomicity checker detected causal atomicity. We performed these experiments under Linux on a 1.7GHz Pentium M laptop with 384MB of memory. The output time is reported with the precision of 10 milliseconds, therefore all the experiments with 0 reported time were done in less than 10 milliseconds.

Note that in the (Acquire1) example, the size of the unfolding grows only linearly with the number of threads; this reflects the space savings obtained through unfoldings. In contrast, the model-checking algorithm in [Pla04], which reasons using sequential traces, started to fail at around four threads. Note, however, that this isn’t a fair comparison as the notion of atomicity (called commit atomicity) in [Pla04] is quite different, more accurate, and harder to check. However, in all the examples, their notion of atomicity agreed with ours.

4.4 Conclusions

We have defined a notion of atomicity based on the causal structure of events that occur in executions of programs. The causal structure is obtained using a straightforward data abstraction of the program that captures control interactions between threads using the concurrent model of Petri nets. We have demonstrated the usefulness of the notion of causal atomicity and shown that it can be effectively checked using unfolding-based algorithms on Petri nets.

This work is part of a bigger project whose aim is to identify sound control abstractions for concurrent programs that can be used for static analysis (for example, causal dataflow analysis in Chapter 5). We believe that true concurrent models (such as Petri nets) and true concurrent behaviors like traces and event structures should prove to be effective for this purpose. This paper demonstrates the efficacy of a truly concurrent behavior model (traces) in identifying atomicity.

There are several future directions. Our method of checking atomicity is a global analysis involving all threads simultaneously, while methods based on types and transactions work locally on each thread independently. Since local algorithms are likely to scale better, it would be interesting to find the weakest local property that ensures global causal atomicity. Also, finding compositional algorithms that derive information from each program locally and combine these to check for global atomicity would be interesting to study as they would scale better than global analysis and be more accurate than local analysis. Finally, we would also like to study extensions of atomicity defined in the literature (for example, purity [FFQ05]), in the causal setting.
Chapter 5

Causal Dataflow Framework for Concurrent Programs

Our definition of causal concurrent dataflow (CCD) analysis is not in terms of the global state-space, but is based on the partially ordered causal structures the program generates. More precisely, we require flows to be defined only using causal dependencies of events in the program. This assumption is only natural, as one expects dataflow facts to be changed by two statements if those statements are related or dependent on each other. All common dataflow analysis problems considered for sequential programs can be formulated in our setting.

The causal concurrent dataflow (CCD) framework is in the flavor of a meet-over-all-paths formulation for sequential programs. We assume a set of dataflow facts \( D \) and each statement of the program is associated with a flow transformer that changes a subset of facts, killing some old facts and generating new facts. However, we demand that the flow transformers respect the concurrency in the program: we require that if two independent (concurrent) statements transform two subsets of facts, \( D \) and \( D' \), then the sets \( D \) and \( D' \) must be disjoint. For instance, if there are two local variable accesses in two different threads, these statements are independent, and cannot change the same dataflow fact, which is a very natural restriction. For example, if we are tracking uninitialized variables, two assignments in two threads to local variables do affect the facts pertaining to these variables, but do not modify the same fact. We present formulations of most of the common dataflow analysis problems in our setting.

The structural restriction of requiring transformers to respect causality ensures that dataflow facts can be inferred using partially ordered traces of the control net. We define the dataflow analysis problem as a meet over partially ordered traces that reach a node, rather than the traditional meet-over-paths definition. The meet-over-traces definition is crucial, as it preserves the concurrency in the program, allowing us to exploit it to solve flow analysis using partial-order-based methods, which do not explore all interleavings of the program.

Our next step is to give a solution for the general causal dataflow analysis problem when the set of facts \( D \) is finite by reducing the problem to a reachability problem of a Petri net, akin to the classic approach of reducing meet-over-paths to graph reachability for sequential recursive programs [RHS95]. Given a control net of a program and a causal dataflow analysis problem, we describe an effective method for constructing a Petri net that augments the control net with a
monitor that tracks the evolution of dataflow facts. Finally, the reachability/coverability problem is solved using an optimized partial-order unfolding [McM95, ERV96] based tool called PEP [Gra97].

For the important subclass of distributive dataflow analysis problems, we develop a more efficient algorithm for checking flows. If $N$ is a control net of a program and the size of its finite unfolding is $n$, then we show that any distributive CCD problem over a domain $D$ of facts results in an augmented net of size $n|D|$ (and hence in an algorithm working within similar bounds of time and space). This is a very satisfactory result, since it proves that the causal definition does not destroy the concurrency in the net (as that would result in a blow-up in $n$), and that we are exploiting distributivity effectively (as we have a linear dependence on $|D|$). The analogous result for sequential recursive programs also creates an augmented graph of size $n|D|$, where $n$ is the size of the control-flow graph.

### 5.0.1 Related Work

Program flow analysis was originally developed for sequential programs to enable optimizing compilers to generate efficient code [ASU86]. These analyses are performed over Control Flow Graphs (CFG) which are structures that explicitly model the control in the program and abstract the data. Although the majority of flow analysis research has been focused on sequential software [ASU86, NN99, Muc97, RSJM05], flow analysis for concurrent software has also been studied to some extent. Flow-insensitive analyses can of course be directly adapted to the concurrent setting. Existing methods for flow-sensitive analyses have at least one of the following restrictions: (a) the programs handled have simple static concurrency and can be handled precisely using the union of control flow graphs of individual programs, or (b) the analysis is sound but not complete, and solves the dataflow problem using heuristic approximations.

A body of work on flow-sensitive analyses exists in which the model for the program is essentially a collection of CFGs of individual threads (tasks, or components) together with additional edges among the CFGs that model inter-thread synchronization and communication [MR93, NA98, SR01]. These analyses are usually restricted to a class of behaviors (such as detecting deadlocks) and their models do not require considering the set of interleavings of the program. More general analyses based on the above type of model include [?] which presents a unidirectional bit-vector dataflow analysis framework based on abstract interpretation (where the domain $D$ is a singleton). This framework comes closest to ours in that it explicitly defines a meet-over-paths definition of dataflow analysis, can express a variety of dataflow analysis problems, and gives sound and complete algorithms for solving them. However, it cannot handle dynamic synchronization mechanisms (such as locks), and the restriction to having only one dataflow fact is crucially (and cleverly) used, making multidimensional analysis impossible. For example, this framework cannot handle the problem of solving uninitialized variables, since this requires keeping track of all undefined variables and
propagating their effects. See also [SW] for a form of dataflow analysis that uses flow along causal edges but disallows loops in programs and requires them to have static concurrency. The works in [LMP97, LPM99] use the extension of the static single assignment form [CFR+91] for concurrent programs with emphasis on optimizing concurrent programs as opposed to analyzing them.

In [DCCN04], concurrent models are used to represent interleavings of programs, but the initial model is coarse and refined to obtain precision, and efficiency is gained by sacrificing precision. This is in contrast to our model, which precisely captures all the partially ordered traces that include all the possible interleavings of the program, and the efficient reasoning based on these partially ordered traces ensures that precision does not have to be sacrificed for efficiency. Petri nets are used as control models for Ada programs in [DC96], although the modeling is completely different from ours. In [CC96], the authors combine reachability analysis with symbolic execution to prune the infeasible paths in order to achieve more efficient results. Our program model does not generate any infeasible concurrent paths; this is of course in the static sense, since presence of data can even make some paths in a sequential program infeasible. A point that distinguishes our work from these works is that while we build net unfoldings [ERV96, McM95], the work above use reachability graphs, which can be exponentially larger than the unfoldings.

### 5.1 The Causal Concurrent Dataflow Framework

We now formulate our framework for dataflow analysis of concurrent programs based on causality, called the Causal Concurrent Dataflow (CCD) framework.

**Definition 18** A property space is a subset lattice \((\mathcal{P}(\mathbb{D}), \sqsubseteq, \sqcup, \bot)\) where \(\mathbb{D}\) is a finite set of dataflow facts, \(\bot \subseteq \mathbb{D}\) is the set of facts that initially hold, and where \((\sqcup, \sqsubseteq)\) is \((\cup, \subseteq)\) or \((\cap, \supseteq)\) depending on the specific problem.

Intuitively, \(\mathbb{D}\) is the set of dataflow facts of interest, \(\bot\) is the initial set of facts, and \(\sqcup\) is the meet operation that will determine how we combine dataflow facts along different paths reaching the same control point in a program. “May” analysis is formulated using \(\sqcup = \cup\), while “must” analysis uses the \(\sqcup = \cap\) formulation. The property space of an IFDS (interprocedural finite distributive subset) problem [RHS95] for a sequential program (i.e. the subset lattice) is exactly the same lattice as above.

For every transition \(t\) of the control net, we associate two subsets of \(\mathbb{D}\), \(D_t\) and \(D^*_t\). Intuitively, \(D^*_t\) is the set of dataflow facts relevant at \(t\), while \(D_t \subseteq D^*_t\) is the subset of relevant facts that \(t\) may modify when it executes. The transformation function associated with \(t\), \(f_t\), maps every subset of \(D_t\) to a subset of \(D_t\), reflecting how the dataflow facts change when \(t\) is executed.
Definition 19 A causal concurrent dataflow (CCD) problem is a tuple \((N, S, F, D, D^*)\) where:

- \(N = (P, T, F)\) is the control net model of a concurrent program,
- \(S = (P(D), \sqsubseteq, \sqcup, \bot)\) is a property space,
- \(D = \{D_t\}_{t \in T}\) and \(D^* = \{D_t^*\}_{t \in T}\) are \(T\)-indexed families of sets, where each \(D_t \subseteq D_t^* \subseteq D\),
- \(F\) is a set of functions \(\{f_t : P(D_t) \rightarrow P(D_t)\}_{t \in T}\) such that:

\[
(\ast) \forall t, t' : (t, t') \in I_N \Rightarrow (D_t \cap D_t^* = D_t^* \cap D_{t'} = \emptyset) \tag{\text{\[1\]}}
\]

Recall that the independence relation \(I_N\) of the net \(N\) is defined as \((t, t') \in I_N \iff (\bullet t \cup \bullet t') \cap (\bullet t' \cup t^*) = \emptyset\).

We call a CCD problem **distributive** if all transformation functions in \(F\) are distributive, that is \(\forall f_t \in F, \forall X, Y \subseteq D_t : f_t(X \sqcup Y) = f_t(X) \sqcup f_t(Y)\).

Remark 20 Condition \((\ast)\) above is to be specially noted. It demands that for any two concurrent events \(e\) and \(e'\), \(e\) cannot change a dataflow fact that is relevant to \(e'\). Note that if \(e\) and \(e'\) are events in a trace such that \(D_{\lambda(e)} \cap D_{\lambda(e')}^*\) is non-empty, then they will be causally related.

5.1.1 Meet Over All Traces Solution

In a sequential run of a program, every event \(t\) has at most one predecessor \(t'\). Therefore, the set of dataflow facts that hold before the execution of \(t\) (let us call this \(\text{in}(t)\)) is exactly the set of dataflow facts that hold after the execution of \(t'\) (\(\text{out}(t')\)). This is not the case for a trace (a partially ordered run). Consider the example in Figure 5.1. Assume \(t_1\) generates facts \(d_1\) and \(d_2\), \(t_2\) generates \(d_3\) and \(t_3\) kills \(d_2\) and generates \(d_4\). The corresponding \(D_t\) sets appear in Figure 5.1. Trying to evaluate the “\(\text{in}\)” set of \(t_4\), one can see three important scenarios here that never happen in the case of a sequential run: (1) \(t_4\) inherits **independent** facts \(d_3\) and \(d_4\) respectively from its immediate predecessors \(t_2\) and \(t_3\), (2) \(t_4\) inherits fact \(d_1\) from \(t_1\) which is not its immediate predecessor, and (3) \(t_4\) does not inherit \(d_2\) from \(t_1\) because \(t_3\), which is a (causally) later event and the last event to modify \(d_2\), kills \(d_2\).

This example demonstrates that in a trace the immediate causal predecessors do not specify the “\(\text{in}\)” set of an event. The indicating event is actually the (causally) last event that can change a dataflow fact.
dataflow fact (e.g., $t_3$ for fact $d_2$ in computing $\text{in}(t_4)$). We formalize this concept by defining the operator $\text{maxc}_d^d(\text{Tr})$, for a trace $\text{Tr} = (E, \preceq, \lambda)$ as $\text{maxc}_d^d(\text{Tr}) = \text{max}_{\preceq}(\{e \mid e \in E \land d \in D_{\lambda(e)}\})$. Note that this function is undefined on the empty set, but well-defined on non-empty sets, because all events that affect a dataflow fact $d$ are causally related due to (*) in Definition 19.

Remark 20 suggests that for each event $e$ it suffices to only look at the facts that are in the “out” set of events in $\Downarrow e$ (events that are causally before $e$\(^2\)), since events that are concurrent with $e$ will not change any fact that is relevant to $e$.

**Definition 21** For any trace $\text{Tr} = (E, \preceq, \lambda)$ of the control net and for each event $e \in E$, we define the following dataflow sets:

$$
\begin{align*}
\text{in}^{\text{Tr}}(e) &= \bigcup_{d \in D_{\lambda(e)}} (\text{out}^{\text{Tr}}(\text{maxc}_d^d(\Downarrow e)) \cap \{d\}) \\
\text{out}^{\text{Tr}}(e) &= f_{\lambda(e)}(\text{in}^{\text{Tr}}(e) \cap D_t)
\end{align*}
$$

where $\text{in}^{\text{Tr}}(e)$ (respectively $\text{out}^{\text{Tr}}(e)$) indicates the set of dataflow facts that hold before (respectively after) the execution of event $e$ of trace $\text{Tr}$.

In the above definition, $\text{maxc}_d^d(\Downarrow e)$ may be undefined (if $\Downarrow e = \emptyset$), in which case we assume $\text{in}^{\text{Tr}}(e)$ evaluates to the empty set.

We can now define the **meet over all traces** solution for a program $Pr$, assuming the $T(N)$ denotes the set of all traces induced by the control net $N$.

**Definition 22** The set of dataflow facts that hold before the execution of a transition $t$ of a control net $N$ is

$$
\text{MOT}(t) = \bigcup_{\text{Tr} \in T(N), e \in \text{Tr}, \lambda(e) = t} \text{in}^{\text{Tr}}(e).
$$

\(^2\)See Section 2.1.2 for the formal definition.
The above formulation is the concurrent analog of the meet-over-all-paths formulation for sequential programs. Instead of the above definition, we could formulate the problem as a meet-over-all-paths problem, where we take the meet over facts accumulated along the sequential runs (interleavings) of the concurrent program. However, due to the restriction (*) in Definition 19, we can show that the dataflow facts accumulated at an event of a trace are precisely the same as those accumulated using any of its linearizations. Consequently, for dataflow problems that respect causality by satisfying condition (*), the meet-over-all-paths and the meet-over-traces formulations coincide. The latter formulation, however, yields faster algorithms based on partial-order methods that use unfoldings to solve the dataflow analysis problem.

5.1.2 The Global Meet Over All Paths Solution

A run \( \sigma \) of the program is a linearization of some trace depicted by the control net of the program. For each event \( e \) appearing in \( \sigma \), let \( \text{pre}_\sigma(e) \) (respectively \( \text{post}_\sigma(e) \)) be the event happening immediately before (respectively immediately after) \( e \) in \( \sigma \). For the first event \( e_0 \) of each run, we assume \( \text{pre}(e_0) = \bot \) and we also assume that \( \text{out}^\sigma(\bot) = \bot \).

**Definition 23** For each run \( \sigma \) of the program and for each event \( e \) appearing in \( \sigma \), we define the following two sets of dataflow facts:

\[
\begin{align*}
\text{in}^\sigma(e) &= \text{out}^\sigma(\text{pre}_\sigma(e)) \\
\text{out}^\sigma(e) &= f^\sigma(e) + \text{in}^\sigma(e) \mid D^\sigma(e) \cup \text{in}^\sigma(e) \setminus D^\sigma(e)
\end{align*}
\]

where \( \text{in}^\sigma \) (respectively \( \text{out}^\sigma \)) is the set of dataflow facts which hold before (respectively after) execution of event \( e \) in the run \( \sigma \) and where we have \( \text{in}^\sigma(e) \mid D^\sigma_t = \text{in}^\sigma(e) \cap D^\sigma_t \).

This definition states that each event \( e \) changes the subset of the facts that reach \( e \) and pass the rest unchanged to the next event in \( \sigma \). Let \( \mathbb{R} \) denote the set of all (linear) runs of the program. Based on the definition of dataflow sets for a single run, we can define the *meet over all paths solution*:

**Definition 24** The set of all dataflow facts that hold before the execution of a transition \( t \) of a Petri net is \( \text{MOP}(t) = \bigcup_{\sigma \in \mathbb{R}, e \in \sigma, \lambda(e) = t} \text{in}^\sigma(e) \mid D^\sigma_t \).

The restriction of the set \( \text{in}^\sigma(e) \) to \( D^\sigma_t \) ensures that only information relevant to an event can reach the event. If statement \( s \) of thread \( T \) is scheduled right after statement \( s' \) of thread \( T' \) in a run, but \( T \) and \( T' \) do not interact in any way, information from \( s' \) should not reach \( s \) merely because \( s' = \text{pre}(s) \) in that particular run.

---

3See Chapter 2.1.2 for more background knowledge on traces.
The Relation between MOP and MOT solutions

The following theorem shows that the two definitions MOP and MOT are equivalent.

**Theorem 25** For every CCD problem, we have $\forall t \in T, \text{MOT}(t) = \text{MOP}(t)$.

This leads to an alternative way of computing the MOP solution, that is, to computing the MOT instead. Considering the fact that there may be exponentially many runs for a single trace of the program, computing MOT can be substantially faster than computing MOP directly. To prove Theorem 25, we need the following lemma:

**Lemma 26** For every trace $Tr$ of the control net of a program, and for every event $e \in Tr$, and for every linearization $\sigma$ of the trace $Tr$, $in_{Tr}(e) = in_{\sigma}(e)|_{D^*_{\lambda(e)}}$.

**Proof.** (sketch)

(a) $d \in in_{Tr}(e) \Rightarrow \forall \sigma d \in in_{\sigma}(e)|_{D^*_{\lambda(e)}}$.

Since $in_{Tr}(e) \subseteq D^*_{\lambda(e)}$ by Definition 21, it suffices to show that $in_{Tr}(e) \subseteq in_{\sigma}(e)$, for all $\sigma$. We prove this by induction using the subtrace relation $\sqsubseteq$. The base case clearly holds for an empty trace. Assume that for trace $Tr'$ such that $Tr' \sqsubset Tr$, the above statement holds. We prove then that it holds for $Tr$.

Consider any linearization $\sigma$ of $Tr$. Consider the prefix of $\sigma$ ending in $e'$, and call it $\sigma'$. Consider the corresponding trace of $\sigma'$, $Tr(\sigma')$. Clearly, $Tr(\sigma') \sqsubset Tr$. Therefore, by induction and the fact that $d \in out_{Tr}(e')$, we have $d \in out_{\sigma'}(e')$, or, equivalently, (since $\sigma'$ is a prefix of $\sigma$) $d \in out_{\sigma}(e')$. We argue that there are no events in $\sigma \setminus \sigma'$ that can cancel $d$. Assume that there is such an event $e''$. Since $e''$ changes $d$, it has to be causally related to $e'$. Since $e''$ appears later than $e'$ in $\sigma$, the only option is that $e' \preceq e''$. But this is in contradiction with $e' = max_d(\downarrow e)$. Therefore such event $e''$ cannot exist. If no event in $\sigma \setminus \sigma'$ cancels $d$, then $d$ propagates to $e$ and therefore, we have $d \in in_{\sigma}(e)$.

(b) $d \in in_{\sigma}(e)|_{D^*_{\lambda(e)}} \Rightarrow d \in in_{Tr(\sigma)}(e)$.

Assume $d \in in_{\sigma}(e)|_{D^*_{\lambda(e)}}$ for some linearization $\sigma$ (ending in $e$) of trace $Tr$. This means that $d \in D^*_{\lambda(e)}$ and also, $d \in out_{\sigma}(\text{pre}(e))$. There are two possibilities:

(i) $\text{pre}(e)$ generates $d$, and therefore $d \in D_{\lambda(\text{pre}(e))}$. By assumption (*) in Definition 19, $\lambda(\text{pre}(e))$ and $\lambda(e)$ should be dependent transitions and therefore $\text{pre}(e) \preceq e$ in trace $Tr$.

By reasoning similar to the one given in part (a), we can deduce that there is no event
that can change the fact $d$ and $\text{pre}(e) \preceq e'$, and $\text{pre}(e) = \max_{\preceq}^{d}(\downarrow e)$. Therefore, by Definition 21, $d \in \text{in}^{Tr}(e)$, since $d \in D_{\lambda(e)}^{*}$.

(ii) $\text{pre}(e)$ is just passing $d$ along without changing it, and therefore $d \in \text{in}^\sigma(\text{pre}(e))$. Then we just keep going back in $\sigma$ until we reach an event $e'$ such that $e' \preceq e$ in $Tr$ and there is no $e''$ changes $d$ and $e' \preceq e'' \preceq e$, and therefore $e' = \max_{\preceq}^{d}(\downarrow e)$, and $d \in \text{in}^{Tr}(e)$.

Now, the proof of Theorem 25

Proof. (sketch)

(a) $d \in \text{MOT}(t) \Rightarrow d \in \text{MOP}(t)$:

$d \in \text{MOT}(t)$ implies that there is a trace $Tr$ and an event $e \in Tr$, $\lambda(e) = t$ such that $d \in \text{in}^{Tr}(e)$. Since $d \in \text{in}^{Tr}(e)$, for all linearizations $\sigma$ of $Tr$, we have $d \in \text{in}^\sigma(e)|_{D_{\lambda(e)}^{*}}$ by Lemma 26, therefore $d \in \text{MOP}(t)$ by definition.

(b) $d \in \text{MOP}(t) \Rightarrow d \in \text{MOT}(t)$:

$d \in \text{MOP}(t)$ implies that there is a run $\sigma$ and an event $e \in \sigma$ where $\lambda(e) = t$ such that $d \in \text{in}^\sigma(e)|_{D_{\lambda(e)}^{*}}$. Consider the corresponding trace of $\sigma$, $\text{Tr}(\sigma)$; clearly, $\text{Tr}(\sigma)$ is a valid trace of the program. By Lemma 26 we know that $d \in \text{in}^{\text{Tr}(\sigma)}(e)$, and therefore $d \in \text{MOT}(t)$.

This leads to an alternative way of computing the MOP solution, that is to computing the MOT instead. Considering the fact that there may be exponentially many runs for a single trace of the program, computing MOT can be substantially faster than computing MOP directly.

5.1.3 Backward Flow Analysis

In the presence of concurrency, backward flow analysis cannot be handled as a trivial dual of the forward analysis, as it is handled in the sequential case. The main reason for this, is that, in contrast to the sequential case, not every position in the control net is reachable; presence of synchronization mechanisms such as locks can make certain positions unreachable.

Consider a class of CCD problems in which the transformation functions in $F$ are interpreted backward, meaning that based on the facts that hold after the execution of a transition $t$, they return the facts that hold before the execution of $t$. To distinguish the backward case form the
forward case, we call these problems bCCD (backward Causal Concurrent Dataflow) problems. Here, we discuss the general MOT solution for bCCD problems.

**Definition 27** \( bTr \) is a backward trace of a control net \( N \) if and only if there is a finite trace \( Tr = (E, \preceq, \lambda) \) of \( N \) such that \( bTr = (E, \succeq, \lambda) \) where \( e_1 \preceq e_2 \iff e_2 \succeq e_1 \).

Now, using the backward trace \( bTr \) and the trace solution of Definition 23, we can define the set of dataflow facts that hold before/after each event \( e \) in a trace \( Tr \).

**Definition 28** For a bCCD problem \((N, S, F)\) and for a trace \( Tr = (E, \preceq, \lambda) \) of the control net and for each event \( e \in \mathcal{E} \), we define the following set of dataflow facts:

\[
\begin{align*}
\text{out}^{Tr}(e) &= \text{in}^{bTr}(e) \\
\text{in}^{Tr}(e) &= f_{\lambda}(\text{out}^{Tr}(e))
\end{align*}
\]

where \( \text{in}^{bTr}(e) \) is defined according to Definition 21, assuming all the maximal events of \( Tr \) (minimal events of \( bTr \)) have an initial value of \( \bot \).

### 5.1.4 Formulation of Specific Problems in the CCD Framework

A wide variety of dataflow analysis problems can be formulated using the CCD framework, including reaching definitions, uninitialized variables, live variables, available expressions, very busy expressions, etc. Some of these are backward flow analysis problems that can be formulated using the bCCD framework.

**Reaching Definitions.** The reaching definitions analysis determines:

\[\text{For each control point, which relevant assignments may have been made and not overwritten when program execution reaches that point along some path.}\]

The relevant assignments are the assignments to variables that are referred to in that control point. Given the control net \( N = (P, T, F) \) for a program \( Pr \), define \( Defs = \{ (v, t) \mid t \in T, v \in (GVar \cup LVar) \text{, and } v \text{ is assigned in } t \} \). The property space is \( (Defs, \subseteq, \cup, \emptyset) \), where presence of \( (v, t) \) in \( D_{in}(t') \) means that the definition of \( v \) at \( t \) may reach \( t' \). We have \( D_t = \{ (v, t') \mid v \text{ is assigned in } t \} \), and \( D_t^* = \{ (v, t') \mid v \text{ is assigned or accessed by } t \} \). For each transition \( t \) and each set \( S \subseteq D_t \):

\[
f_t(S) = \begin{cases} 
S & \text{if } t \text{ is not an assignment} \\
S - \{(v, t') \mid t' \in T\} \cup \{(v, t)\} & \text{if } t \text{ is of the form } v := e
\end{cases}
\]
The construction of the control net ensures that two accesses of a variable \( v \), where one of them is a write, are dependent (neighborhoods intersect). This guarantees that condition (*) of Definition 19 holds, i.e., our formulation of reaching-definitions ensures that information is inherited only from causal predecessors. Note that the above formulation is also distributive.

**Available Expressions.** The available expressions analysis determines:

For a program point containing \( x := \text{Exp}(x_1, \ldots, x_k) \) whether \( \text{Exp} \) has already been computed and not later modified on all paths to this program point.

In the standard (sequential) formulation of available expressions analysis, dataflow facts are defined as pairs \((t, \text{Exp})\), where \( \text{Exp} \) is computed at \( t \). This formulation does not work for the concurrent setting. To see why not, consider the trace on the right, where \( x \) is a local variable in \( T \) and \( Y \) is a global variable. Events \( e_2 \) and \( e_3 \) are independent (concurrent), but they both can change (kill) the dataflow fact associated with \( x + Y \), which is not in accordance with condition (*) of Definition 19. The natural remedy is to divide this fact into two facts, one for \( x \) and another for \( Y \). Let us call these two facts \( x + Y : x \) and \( x + Y : Y \). The fact \( x + Y : x \) (respectively \( x + Y : Y \)) starts to hold when the expression \( x + Y \) is computed, and stops to hold when a definition to \( x \) (respectively \( Y \)) is seen. The problem is that \( x + Y \) holds when \( x + Y : x \) holds and \( x + Y : Y \) holds, which makes the framework non-distributive. Although we can solve non-distributive problems in the CCD framework, distributive problems yield faster algorithms (see Section 5.2.1).

The analysis can however be formulated as a distributive CCD problem by looking at the dual problem; that is, for unavailability of expressions. The dataflow fact \( x + Y \) indicates the expression being unavailable, and accordingly the presence of \( x + Y : x \) or \( x + Y : Y \) can make it hold. We are now in a distributive framework. Assume \( EXP \) presents the set of all expressions appearing in the program code, and define \( D = \{ \text{exp} : x_i | \text{exp} \in EXP \land x_i \text{ appears in } \text{exp} \} \). The property space is the subset lattice \((\mathcal{P}(D), \subseteq, \cup, \mathbb{D})\), where presence of \( \text{exp} \) in \( D^m(t') \) means that \( \text{exp} \) is unavailable at \( t \). We have \( D_t = D_t^* = \{ \text{exp} : x | x \text{ is assigned in } t \text{ or } \text{exp} \text{ appears in } t \} \). For each transition \( t \) and each set \( S \subseteq \mathbb{D} \):

\[
f_t(S) = \begin{cases} 
S & \text{if } t \text{ is not an assignment} \\
S \cup \{ \text{exp}' : x | \forall \text{exp}' \in EXP, x \in V(\text{exp}') \} \\
- \{ \text{exp} : y | y \in V(\text{exp}) \} & \text{if } t = x := \text{exp} 
\end{cases}
\]

where \( V(\text{exp}) \) denotes the set of variables that appear in \( \text{exp} \).
Uninitialized Variables. The uninitialized variables analysis determines:

For each relevant program point, which variables may be read before having previously been initialized when the program execution reaches that point.

Given the control net \( N = (P, T, F) \) for a program \( Pr \), define \( Vars = GVar \cup LVar \). Formulation of uninitialized variables analysis in the CCD framework is as follows: The property space is \( (Vars, \supseteq, \cap, Vars) \), where presence of \( v \) in \( D^{\text{ini}}(t') \) means that \( v \) may be uninitialized at \( t' \). Note that at the beginning, everything is uninitialized. We have \( D_t = \{ v \mid v \text{ is assigned in } t \} \), and \( D'_t = \{ v \mid v \text{ is assigned or accessed by } t \} \). For each transition \( t \) and each set \( S \subseteq Vars:\)

\[ f_t(S) = \begin{cases} S & \text{if } t \text{ is not an assignment} \\ S - \{v\} & \text{if } t \text{ is of the form } v := e \end{cases} \]

which makes \( D_t = \{ v \mid v \text{ is written in } t \} \).

In this problem, one starts with all variables being uninitialized. If \( t \) is an assignment \( v := e(v_1, \ldots, v_n) \) with some uninitialized argument \( v_i \), then \( v \) is added to the set of uninitialized variables; otherwise, \( v \) is removed from that set. Any transition \( t' \) that can affect the status of \( v_i \) must contain an assignment to it. This creates a similar setting to that of the reaching definitions problem, since a write to \( v_i \) and a read of \( v_i \) (in \( t \)) are always ordered in any trace. Therefore, again considering the flow of facts along the causal edges suffices.

Live Variables A variable \( v \) is live at the exit point a definition to \( v \) if there is path from that point to a use of \( v \) which does not contain any definitions to \( v \). The live variables analysis determines:

For each definition of a variable \( v \), whether \( v \) is live at the exit point of this definition.

Note that this is a variant of the standard textbook definition of live variables analysis, since the standard definition evaluates the liveness of a variable at any program point, not just the relevant ones (the definitions points). We argue that the above version is more sensible for concurrent programs since, if a thread \( T \) does not access global variable \( v \) at all, then information on liveness of \( v \) should not be relevant to an instruction in thread \( T \).

Given the control net \( N = (P, T, F) \) for a program \( Pr \), define \( Vars = GVar \cup LVar \). The live variables problem can then be summarized as: The property space is \( (P(Vars), \subseteq, \cup, \emptyset) \), where presence of \( v \) in \( D^{\text{live}}(t') \) means that \( v \) is live at the exit from \( t' \). We have \( D_t = \{ v \mid v \text{ is assigned in } t \} \), and
\[ D_t^* = \{ v \mid v \text{ is assigned or accessed by } t \}. \] For each transition \( t \) and each set \( S \subseteq Vars \):

\[
f_t(S) = \begin{cases} 
    S & \text{if } t \text{ is not an assignment} \\
    S - \{ v \} \cup \{ v_1, \ldots, v_k \} & \text{if } t \text{ is of the form } v := e(v_1, \ldots, v_k)
\end{cases}
\]

which makes \( D_t = \{ v \mid v \text{ is accessed in } t \} \).

As argued in the previous cases, since variable liveness is changed by definitions and uses of the variable and the control net imposes all the definitions and uses to be causally related, our notion of information flowing through causal edges holds for this problem.

**Very Busy Expressions** An expression is *very busy* at the exit from a transition computing it if, on all paths starting from this transition the expression in used before any of the variables in it is redefined. The very busy expression analysis determines:

*For a assignment point \( x := \exp \) wether \( \exp \) is a very busy expression.*

Similar to the available expressions framework, we look at the dual problem, and the question whether \( \exp \) is not very busy. The dataflow fact \((t, \exp) : x_1 \) means that An expression \( \exp(x_1, \ldots, x_n) \) is not very busy at the exit point from an assignment if, there is \( x_i \) that is redefined before it is used (the use refers to the later appearance of same expression \( \exp \)). The set of facts \( D \) is the same as in the available expressions problem. The property space is the subset lattice \((\mathcal{P}(D), \subseteq, \cup, \emptyset)\), where presence of \( \exp : x \) (for all \( x \)) in \( D^{in}(t') \) means that \( \exp \) is unavailable at \( t \). We have \( D_t = D_t^* = \{ \exp : x \mid x \text{ is assigned in } t \text{ or } \exp \text{ appears in } t \} \). For each transition \( t \) and each set \( S \subseteq EXP: \)

\[
f_t(S) = \begin{cases} 
    S & \text{if } t \text{ is not an assignment} \\
    S - \{ \exp' : x \} & \text{if } t \text{ is } x := \exp'
\end{cases}
\]

### 5.2 Solving the the Distributive CCD Problem

In this section, we show how to solve a dataflow problem in the CCD framework. The algorithm we present is based on augmenting a control net to a larger net based on the dataflow analysis problem, and reduce the problem of checking whether a dataflow fact holds at a control point to a reachability problem on the augmented net. The augmented net is carefully constructed so as to not destroy the concurrency present in the system (crucially exploiting condition (*) in Definition [19]). Reachability on the augmented net is computed using net unfoldings, which is a partial-order-based approach that checks traces generated by the net as opposed to checking its linear runs.
We first present only the solution for the distributive CCD problems where the meet operator is union, and we prove tidy upper bounds that compare the unfolding of the augmented net with respect to the size of the unfolding of the original control net. Later, we extend this to cover the non-distributive CCD problems and the backward flow problems.

In the previous section, we defined the precise solution to the class of CCD problems. Here, we present a general way of obtaining this solution by transforming the problems into coverability queries on augmented control nets. The idea is to show that finding the precise solution is algorithmically possible, and the algorithm suggested here has a reasonable complexity bound. There are three different constructions for the augmented net: one for the class of distributive CCD problems, one for the general non-distributive case, and one for backward problems. We discuss the construction for the distributive case here, and refer the interested reader to Section 5.2.2 for the backward and the non-distributive cases.

5.2.1 Distributive Frameworks

In order to track the dataflow facts, we enrich the control net so that each transition performs the transformation of facts as well. We would like to enrich the structure of the control net such that, besides the flow of control, it can track the dataflow facts as well. We introduce new places which represent the dataflow facts. The key is then to model the transformation functions, for which we use representation relation from [RHS95].

**Definition 29** The representation relation of a distributive function $f : \mathcal{P}(D) \rightarrow \mathcal{P}(D) \ (D \subseteq \mathbb{D})$, $R_f \subseteq (D \cup \{\perp\}) \times (D \cup \{\perp\})$ is a binary relation defined as follows:

$$R_f = \{(\perp, \perp)\} \cup \{(\perp, d) \mid d \in f(\emptyset)\} \cup \{(d, d') \mid d' \in f(\{d\}) \land d' \not\in f(\emptyset)\}$$

The relation $R_f$ captures $f$ faithfully, in that we can show that $f(X) = \{d' \in D \mid (d, d') \in R_f, \text{ where } d = \perp \text{ or } d \in X\}$, for any $X \subseteq D$.

Given a CCD framework $(\mathcal{N}, \mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{D}^*)$ with control net $\mathcal{N} = (P, T, F)$, we define the net representation for a function $f_t$ as below:

**Definition 30** The net representation of $f_t$ is a Petri net $N_{f_t} = (P_{f_t}, T_{f_t}, F_{f_t})$ defined as follows:

- The set of places is $P_{f_t} = \bullet t \cup \bullet \perp \cup \{\perp_m \mid m \in [1, n]\} \cup \bigcup_{d_i \in \mathcal{D}_t} \{p_i, \overline{p}_i\}$ where $n$ is the number of threads in the program and a token in $p_i$ means the dataflow fact $d_i$ holds, while a token in $\overline{p}_i$ means that $d_i$ does not hold. Each thread has a special $\perp$ place.
• Set of transitions \( T_f \) which contains exactly one transition per pair \((d_i, d_j)\) ∈ \( R_{ft} \) is formally defined as:

\[
T_{ft} = \left\{ s^t_{(\perp, \perp)} \right\} \cup \left\{ s^t_{(\perp, j)} \mid (\perp, d_j) \in R_{ft} \right\} \cup \left\{ s^t_{(i, j)} \mid (d_i, d_j) \in R_{ft} \right\}
\]

Note that if \( D_t = \emptyset \) then \( T_{ft} = \left\{ s^t_{(\perp, \perp)} \right\} \).

• The flow relation is defined as follows:

\[
F_{ft} = \bigcup_{s \in T_{ft}} \left( \bigcup_{p \in \bullet t} \{(p, s)\} \cup \bigcup_{p \in t^*} \{(s, p)\} \right) \cup \bigcup_{d_k \in D_t} \left\{ (\overline{p}_k, s^t_{(\perp, \perp)}), (s^t_{(\perp, \perp)}, \overline{p}_k) \right\}
\]

\[
\cup \bigcup_{(\perp, d_j) \in R_{ft}} \left\{ (\perp_m, s^t_{(\perp, j)}) \mid t \in T_m \right\} \cup \left\{ (s^t_{(\perp, j)}, p_j) \right\}
\]

\[
\cup \bigcup_{d_k \in D_t} \left\{ (\overline{p}_k, s^t_{(\perp, j)}) \right\} \cup \bigcup_{k \neq j} \left\{ (s^t_{(i, j)}, \overline{p}_k) \right\}
\]

\[
\cup \bigcup_{(d_i, d_j) \in R_{ft}} \left\{ (p_i, s^t_{(i, j)}), (s^t_{(i, j)}, p_j), (\overline{p}_j, s^t_{(i, j)}), (s^t_{(i, j)}, \overline{p}_i) \right\}
\]

\[
\cup \bigcup_{(d_i, d_i) \in R_{ft}} \left\{ (p_i, s^t_{(i, i)}), (s^t_{(i, i)}, p_i) \right\}
\]

The idea is that each transition \( s^t_{(i, j)} \) is a copy of transition \( t \) that, besides simulating \( t \), models one pair \((d_i, d_j)\) of the relation \( R_{ft} \), by taking a token out of place \( p_i \) (meanwhile, also checking that nothing else holds by taking tokens out of each \( \overline{p}_k, k \neq i \)) and putting it in \( p_j \) (also returning tokens all \( \overline{p}_k, k \neq j \)). Thus if \( d_i \) holds (solely) before execution of \( t \), \( d_j \) will hold afterwards. The transitions \( s^t_{(i, j)} \) generate new dataflow facts, but consume the token \( \perp_m \) associated with the thread. We will engineer the net to initially contain only one \( \perp_m \) marking (for some thread \( m \)), and hence make sure that only one fact is generated from \( \perp \).

For every \( t \), transitions \( s^t_{(i, j)} \) are in conflict, since they have \( \bullet t \) as common predecessors. This means that only one of them can execute at a time, generating a single fact. If we assume that initially nothing holds (i.e., that initial tokens are in every \( \overline{p}_i \)s and no initial tokens in any of \( p_i \)s), then since each transition consumes one token and generates a new token, the following invariant always holds for the system: “At any reachable configuration of the augmented net, exactly one position \( p_i \) corresponding to some dataflow fact \( d_i \) holds”. We use this observation later to argue the complexity of our analysis.

**Definition 31** The augmented marked net \( N^{S,F} \) of a CCD problem \((N,S,F)\) is defined based
on $\bigcup_{f \in \mathcal{F}} N_f$ where the union of two nets $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ is defined as $N_1 \cup N_2 = (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2)$. It is assumed that the $N_f$s have disjoint set of transitions, and that only the common places are identified in the union. Furthermore we add (to $\bigcup_{f \in \mathcal{F}} N_f$) a new position $p^*$, make each $\bar{p}_i$ initial, and also introduce $n$ initial transitions $t_m^*$, one for each thread, that removes $p^*$ and puts a token in $\perp_m$ and a token in the initial positions of each thread.

The above construction only works when $\perp = \emptyset$. When $\perp = D_0$, for some $D_0 \subseteq \mathbb{D}$, we will introduce a new initial set of events (all in conflict) that introduce nondeterministically a token in some $p_i \in D_0$ and remove $\bar{p}_i$.

**Proposition 32** The above construction of the augmented net ensures that at each moment at most one dataflow fact holds. In other words, at most one of the $p_i$s has a token in it.

![Figure 5.2: Representation relation.](image)

**Example:** Consider the reaching definitions problem (which is distributive) for the program on the right and assume we are interested in the global variable $Y$. Figure 5.3(a) illustrates the control net of the program. There are two definitions of $Y$ at $t_1$ and $t_2$, therefore $\mathbb{D} = \{(Y, t_1), (Y, t_2)\}$. $f_{t'_1}$ is the skip function. The representation relation for functions $f_{t_1}$ and $f_{t_2}$ are as in Figure 5.2.

Figures 5.3(b,c) present the net representations of the above relations, considering that initially $\perp$ holds. Figure 5.3(d) shows the union of the three nets in Figures 5.3(a,b,c) which is the augmented net of the program for the reaching definitions problem.

The problem of computing the MOT solution can be reduced to a coverability problem on the augmented net. To be more precise, fact $d_i$ may hold before the execution of transition $t$ of the control net if and only if $\{p_i\} \cup \bullet t$ is coverable from the initial marking of the control net.

**Example:** Consider the example from Figure 5.3. In the augmented net $\{p_\perp\} \cup \bullet t_1 = \{p_\perp, p_1, p_{t_1}, p_{t_2}\}$ is coverable, this means that $\perp$ holds before the execution of $t_1$. Similarly, $\{p_{(Y, t_1)}\} \cup \bullet t_2 = \{p_{(Y, t_1)}, p_2, p_{t_1}, p_{t_2}\}$ means that $(Y, t_1)$ holds before the execution of $t_2$. It is also easy to see that $\{p_{(Y, t_1)}\} \cup \bullet t_1 = \{p_{(Y, t_1)}, p_1, p_{t_1}, p_{t_2}\}$ is not coverable (to have a token in $p_{(Y, t_1)}$, a token from $p_1$ must be consumed) and therefore $(Y, t_1)$ does not hold before execution of $t_1$. 

51
Figure 5.3: Augmented Control Net.
The following theorem formalizes the above idea:

**Theorem 33** A dataflow fact \( d_i \) holds before the execution of a transition \( t \) in the control net \( N \) of a program if and only if \( d_i \in D^*_i \) and the marking \( \{ p_i \} \cup \cdot t \) (**t computed in \( N \)) are coverable from the initial marking in the augmented net \( N^{S,F} \) constructed according to Definition [31]

**Proof.** (sketch)

(a) \( d_i \in \text{MOT}(t) \Rightarrow (d_i \in D^*_i \land \{ p_i \} \cup \cdot t \) is coverable in \( N^{S,F} \)):

\( d_i \in \text{MOT}(t) \) implies \( d_i \in D^*_i \) by definition. We only have to show that \( d_i \in \text{MOT}(t) \) implies that \( \{ p_i \} \cup \cdot t \) is coverable in \( N^{S,F} \). \( d_i \in \text{MOT}(t) \) implies that there is a trace \( Tr \), and an event \( e \in Tr \) such that \( \lambda(e) = t \) and \( d_i \in in^{Tr}(e) \). This trace \( Tr \) corresponds to a configuration \( C \) of the unfolding of the control net \(^4\) Clearly, \( \downarrow e \subseteq C \). Based on configuration \( \downarrow e \{ e \} \), we construct a configuration \( C' \) of the augmented net such that \( \{ p_i \} \cup \cdot t \subseteq C'^* \), which will then mean that \( \{ p_i \} \cup \cdot t \) is coverable. We present the construction by induction on the size of configuration \( \downarrow e \subseteq C \), using the subset relation.

Since \( d_i \in in^{Tr}(e) \), there is an event \( e' = \text{max}^{d_i}_{\uparrow} (\downarrow e) \), and \( e' \subseteq \downarrow e \). Since the transformation functions are distributive, there is a single fact \( d_j \in in^{Tr}(e') \) such that \( (d_j, d_i) \in R^{t}_{S}(e') \); basically, \( d_i \) is generated as the result of \( d_j \) holding before execution of \( e' \) (Note that \( d_j \) could be \( \bot \)).

Since \( d_j \in in^{Tr}(e') \) and \( \downarrow e' \subseteq \downarrow e \), inductively, we can build the configuration \( \overline{C} \) in the unfolding of the augmented net that corresponds to \( \downarrow e' \{ e' \} \), and \( \{ p_j \} \cup \cdot \lambda(e') \subseteq \overline{C}' \).

We extend \( \overline{C} \) to get \( C' \) as follows: first, we add an event corresponding to transition \( s^{\lambda(e')}_{(j,i)} \) to \( \overline{C} \). This transition is enabled because all its predecessor events \( \{ p_j \} \) and \( \cdot \lambda(e') \) are enabled. Since \( \{ p_j \} \subseteq \overline{C}' \), we have \( \{ p_i \} \subseteq \overline{C} \cup \{ s^{\lambda(e')}_{(j,i)} \} \). Note that at this moment \( p_i \) has a token, and according to Proposition [32] none of the other \( p_k \)s can contain a token, and as all the \( \overline{p}_k \)s, \( k \neq i \), have tokens in them.

Then, we add events corresponding to the events \( f \) in \( \downarrow e' \{ e' \} - \downarrow e' \) in the same order that one would expand the configuration \( \downarrow e' \{ e' \} \); which is an order in which they become enabled. For each \( f \), we add a copy of \( s^{\lambda(f)}_{(i,j)} \). These copies are enabled since \( \cdot \lambda(f) \) is enabled as a result of \( f \) being enabled in the unfolding of the control net, and all the \( \overline{p}_k \)s \( (k \neq i) \) have tokens in them. Note that since none of such events \( f \) can further change \( d_i \), then \( d_i \not\in D^{t}_{\lambda(f)} \) and therefore, the \( (\bot, \bot) \) transition of none of these requires a token in \( p_i \). In summary, after we generate \( d_i \) by the corresponding event of \( e' \), we just keep it by executing the rest of the events as skips for \( d_i \).

\(^4\)We refer the reader to [ERV02] for in depth background information on the net unfoldings.
We have, \( \{p_i\} \subset C' \). We also have, \( \bullet t \subseteq C' (t = \lambda(e)) \), since this is the case in the unfolding of the control net, and each \( s^{M}_{(\bot, \bot)} \) that we have added behaves exactly the same as the corresponding \( \lambda f \) by the definition of the augmented net.

(b) \( (d_i \in \text{D}_t^* \land \{p_i \cup \bullet t \text{ is coverable in } N^S,F \} \Rightarrow d_i \in \text{MOT}(t) ) \)

The argument is very similar to the reverse of the argument in part (a) and therefore we skip it for the moment.

Checking coverability: While there are many tools that can check reachability (coverability) properties of Petri nets, tools that use unfolding techniques \[^{[McM95,ERV02]}\] of nets are particularly effective, as they explore the state space using partially ordered unfoldings and give automatic reduction in state-space (akin to partial-order reduction for model checking of concurrent systems).

We use the PEP tool \[^{[Gra97]}\] to check the coverability properties from Theorem 33. Since the PEP tool (and other unfolding-based tools), check the properties on the finite unfolding of the net, the time complexity of the coverability/reachability queries depends (linearly) on the size of the unfolding.

Complexity of distributive CCD: Algorithms for Petri nets which use finite unfoldings essentially produce a finite unfolding of the net, from which coverability of one position can be checked in linear time. For every transition \( t' \in T_f \) and every fact \( d_i \in \text{D}_t^* \), we can create a new transition whose preconditions are those of \( t' \) plus \( p_i \), and outputs a token in a new position \( (t, d_i) \). By Theorem \[^{[33]}\] coverability of this single position is equivalent to fact \( d_i \) holding at \( t \). Furthermore, we can argue that the unfolding of this net introduces at most \( n|\text{D}| \) new events compared to the unfolding of the augmented net.

Let us now analyze the size of the unfolding of the augmented net in terms of the size of the unfolding of the original control net; let us assume that the latter has \( n \) events. We can show that: (a) every marking reachable by a local configuration of the control net has a corresponding event in its finite unfolding that realizes this marking, and (b) for every marking reached by a local configuration of the control net, there are at most \( |\text{D}| \) corresponding local configurations in the augmented net (at most one for each dataflow fact), and this covers all local configurations of the augmented net. Since the number of events in the unfolding is bounded by the number of markings reachable by local configurations, it follows that the size of the unfolding of the augmented net is at most \( |\text{D}| \times n \) times that of the control net. This argues the efficacy of our approach in preserving the concurrency inherent in the control net and in exploiting distributivity to its fullest extent.

**Theorem 34** Let \((N,S,F)\) be a distributive CCD problem, with \( S = (\mathcal{P}(\text{D}), \subseteq, \cup, \emptyset) \). Let \( n \) be the size of the unfolding of \( N \). Then the size of the unfolding of the augmented net \( N^{S,F} \) (and hence the complexity of checking whether a fact holds at a control point) is at most \( O(n|\text{D}|) \).
Lemma 35 for every $e$ in the augmented net, if the marking obtained by the local configuration corresponding to $e$ is $M$, then $M$ with the extra places removed from it is a local configuration of some event $e'$ in the original net.

Proof. Every marking corresponding to a local configuration of a net must be represented by some event in the finite unfolding. For a proof of this, take the normal proof of why the finite unfolding represents all reachable markings [ERV02]. In that proof, note that if one specializes it for markings reachable using local configurations, it shows that there is some event in the finite unfolding which realizes that marking. Coming back to the lemma, the marking obtained from the local configuration of $e$, with the extra places removed, surely corresponds to a local configuration of some event in the infinite unfolding of the original net. It follows that some event in the finite unfolding must represent it. ■

In other words, Lemma 35 is a lower bound on finite unfoldings — the finite unfolding must is at least be as large as the number of reachable local configuration markings.

Proof. (sketch of proof of Theorem 33) Now, let $f$ be an event in the finite unfolding of the original net. There are at most $|D|$ events corresponding to this event in the unfolding of the augmented net. Furthermore, by Lemma 35, this covers all events in the finite unfolding of the augmented net. Hence the number of events in the augmented net is at most $n|D|$, where $n$ is the finite unfolding of the original net. ■

To finish the entire analysis, we must show coverability is checkable in $O(n|D|)$ as well. For each transition $t$ and $d$ relevant at $t$, introduce an event $Test_{t,d}$ which takes the preconditions of $t$ as well as the condition corresponding to $d$, and puts a token in a new position $Post_{t,d}$. Now we can check if $d$ belongs to $in(t)$ by checking if $Post_{t,d}$ is reachable, and this is checkable by just examining the unfolding. So we need to argue about the new unfolding. Each $Test_{t,d}$ is a maximal event; so no events can causally occur after it. But $Test_{t,d}$ can occur many times in the unfolding because it can correspond to many local configurations. But for each of these, there is a corresponding event corresponding to $t$ that induces the same local configuration and hence is present in the unfolding of the augmented net before the $Test_{t,d}$ events were added. In other words, the local configuration of $Test_{t,d}$ is precisely the same as the event corresponding to $t$ that is immediately in conflict with it (which would have happened if $Test_{t,d}$ did not happen). Hence the number of $Test_{t,d}$ events is at most the number of $t$-events in the unfolding of the original augmented net. Hence the number of events in the new net is at most twice that of the old net. Hence checkable in $O(n|D|)$.
5.2.2 Augmented Net for Non-distributive Problems

In the case of non-distributive frameworks, the singletons are not sufficient for modeling the transformation functions. Therefore, the transformation functions have to be defined for all elements of $2^{D_t}$ for each function $f_t$.

**Definition 36** Representation relation of a non-distributive $f : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$ ($D \subseteq \mathbb{D}$), $R_{f}^{nd} \subseteq \mathcal{P}(D) \times \mathcal{P}(D)$ is a binary relation defined as follows:

$$R_{f}^{nd} = \{(S, S') | S, S' \subseteq D \land f(S) = S'\} \cup \{(\emptyset, \emptyset)\}$$

Given an CCD problem with the property space ($\mathcal{P}\mathbb{D}, \subseteq, \cup, \emptyset$) where $\mathbb{D} = \{d_1, \ldots, d_m\}$ and with control net $N = (P, T, F)$, we define the net representation for the set of transformation functions $\{f_t\}_{t \in T}$:

**Definition 37** Net representation of $f_t$ is a Petri net $N_{f_t} = (P_{f_t}, T_{f_t}, F_{f_t})$ defined as follows:

- The set of places is defined as $P_{f_t} = \bigcup_{d_i \in D_t} \{p_i, \overline{p}_i\}$, where a token in place $p_i$ means that the dataflow fact $d_i$ holds, while a token in $\overline{p}_i$ means that $d_i$ does not hold.
- The set of transitions $T_{f_t}$, which contains exactly one transition per pair $(S, S') \in R_{f_t}^{nd}$, is defined as:

  $$T_{f_t} = \left\{ s_t^t \right\} \cup \left\{ s_{(S,S')}^t | (S, S') \in R_{f_t}^{nd} \right\}$$

- The flow relation is defined as follows:

  $$F_{f_t} = \bigcup_k \left\{ (\overline{p}_k, s_{(\emptyset,\emptyset)}^t), (s_{(\emptyset,\emptyset)}^t, \overline{p}_k) \right\} \cup \bigcup_{s \in T_{f_t}} \left( \bigcup_{p \in s^t} \left\{ (p, s) \right\} \cup \bigcup_{p \in s^t} \left\{ (s, p) \right\} \right)$$

  $$\cup \bigcup_{(S,S') \in R_{f_t}^{nd}} \left( \bigcup_{d_i \in S} \left\{ (p_i, s_{(S,S')}^t) \right\} \cup \bigcup_{d_j \in D_t - S} \left\{ (\overline{p}_j, s_{(S,S')}^t) \right\} \right)$$

  $$\cup \bigcup_{d_i \in S'} \left\{ (s_{(S,S')}^t, p_i) \right\} \cup \bigcup_{d_j \in D_t - S'} \left\{ (s_{(S,S')}^t, \overline{p}_j) \right\}$$

Definition of the augmented net remains the same as Definition 31 by replacing $\bot$ with $\emptyset$. Similar to the distributive case, a fact $d_i$ holds at the entry point of a transition $t$ if and only if $^t t \cup \{p_i\}$ is coverable from the initial marking, and Theorem 33 states the correctness of the approach. Also, similar reasoning as Theorem 34 can show that the time/space complexity bound on the size of the unfolding in this case is $2^{||D||} \times |U_N|$ where $U_N$ is the unfolding of the program control net.
5.2.3 Augmented Net for Backward Flow Problems

Here, we present the construction of augmented net for the backward flow only for the distributive framework. In the backward flow problems, the transformation functions are interpreted in the reverse direction, in the sense that \( D_{in}(e) = f_{\lambda(e)}(D_{out}(e)) \) as opposed to \( D_{out}(e) = f_{\lambda(e)}(D_{in}(e)) \). We work with the inverse of these functions (that are not necessarily functions) for the construction of the augmented net. It is easy to see that for each \( f_t, R_{f_t}^{-1} \) (for \( R_{f_t} \) defined as in Definition 29) models the relation \( f_t^{-1} \).

The backward analysis is more tricky than the forward case. Assume we want to check whether \( d \in D_{out}(t) \). By Definition 28, \( d \) holds at this point because there is a trace \( Tr \) (let us call this the witness trace for \( d \)) with events \( e, e' \) such that \( e \preceq e' \) and \( e' \) generates \( d \) and no event \( e'' \) (\( e \preceq e'' \preceq e' \)) changes \( d \).

The first step is to make sure that \( t \) is reachable; if \( t \) is not reachable, there is no interest in dataflow facts that may reach it. After checking reachability of \( t \), the next step is to indicate that we intend to check wether \( d \) holds at the exit from \( t \); since different facts may have different witness traces.

**Definition 38** A \((t, d_i)\)-monitoring net \( N_{(t,d_i)} \) for a transition \( t \) of a program control net \( N \) and a dataflow fact \( d_i \in D_t = \{d_1, \ldots, d_m\} \), is a small Petri net with one transition \( t_{d_i} \), and one place \( p_i^* \) such that \( ^*t_{d_i} = ^*t \cup \{p_1, \ldots, p_m\} \) and \( t_{d_i}^* = t^* \cup \{p_i^*, p_i\} \cup \bigcup_{k \neq i} \{d_k\} \).

The purpose of a \((t, d_i)\)-monitor is to indicate (by putting a token in place \( p_i^* \) which is initially empty) that \( d_i \) is the fact to be checked to be in \( D_{out}(t) \). \( t_{d_i} \) is a special copy of \( t \) that is enabled when \( t \) is enabled and no dataflow fact holds. It then assumes that \( d_i \) holds by putting a token in \( p_i \).

The main difference between the backward case and the forward case (introduced earlier) is that, in the backward analysis, the augmented net is specialized to check for a specific fact at a specific point.

**Definition 39** The augmented net of a backward CCD problem \((N, S, \mathcal{F})\) to check for a data flow fact \( d \) to hold at the exit from transition \( t \), is defined as \( N \cup \bigcup_{f \in \mathcal{F}} N_{f^{-1}} \cup N_{(t,d)} \) where the union of two nets \( N_1 = (P_1, T_1, F_1) \) and \( N_2 = (P_2, T_2, F_2) \) is defined as \( N_1 \cup N_2 = (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2) \) with the exception of identifying each transition \( t \) of \( N \) with transitions \( s_{(\perp, \perp)}^t \) of \( N_{(t,d)} \). It is assumed that the \( N_f \)'s have disjoint sets of transitions, and only the common places are identified in the union.

Note that, although the above definition of the augmented net for the backward case is specific to

\(^5\)Note that in the concurrent setting, in contrast to the sequential case, some transitions may not be reachable.
a fact $d$ and a transition $t$, one can always generalize it by adding (by union) all the $(t, d_i)$-monitors such that $d_i \in D_t$.

**Theorem 40** A dataflow fact $d_i$ holds at the exit from a transition $t$ in the control net $N$ of a program if and only if $d_i \in D_t^*$ and the marking $\{p_i^*\} \cup \{p_1, \ldots, p_m\}$ (for $D_t = \{d_1, \ldots, d_m\}$) is coverable from the initial marking in the augmented net constructed according to Definition 39.

*Proof.* We skip this proof since it is tedious and the ideas behind it are more or less the same as those offered in the proof of Theorem 33. ■

### 5.3 Experiments

We have applied the techniques from Section 5.2 to perform several dataflow analyses for concurrent programs. Unfortunately, there is no standard benchmark for concurrent programs. We have however experimented with our algorithms using sample programs for the primary dataflow analysis problems, and have studied their performances when the number of threads is increased.

The goal of the experiments is to exhibit in practice the advantages of concurrent dataflow analyses that exploit the causal framework set forth in this paper. While the practical efficacy of our approach on large programs is still not validated, we believe that setting up a general framework with well-defined problems that permit reasonable algorithms is a first step towards full-scale flow analysis. Algorithms that work on large code may have to implement approximations and heuristics, and we believe that the framework herein will serve as a standard for correctness.

In many of our examples there is an exponential increase in the set of reachable states as one increases the number of threads, but the partial order methods inherent to these techniques substantially alleviate the problem. We use the PEP tool [Gra97] to check the coverability property on the augmented net to answer the relevant coverability queries.

For each example, we have included the sizes of the unfolding for the program’s control net and of the augmented net. The *construction* time refers to the time to build the unfolding, and the *checking* time refers to the time for checking a single fact. Note the huge differences between the two times in some cases, and also note that the unfolding is only built once and is then used to answer several coverability queries. All experiments were performed on a Linux machine with a 1.7GHz processor and 1GB of memory. The numbers are all in seconds (with a precision of 0.01 seconds); all columns actions reported to finish in “0” time finished in less than 0.01 seconds.

**Uninitialized Variables.** This set of examples contains a collection of $n$ threads with $n$ global variables $x^0, \ldots, x^n$. One uninitialized variable $x^0$ in one thread can consequently make all $x$'s
uninitialized. Concurrency results in many possible interleavings in this example, a few of which can make a certain variable $X_j$ uninitialized.

**Reaching Definitions.** This example set demonstrates how our method can successfully handle synchronization mechanisms. There are two types of threads: (1) those which perform two consequent writes to a global variable $Y$, and (2) those which perform a read of $Y$. There are two variations of this example: (1) one where none of the accesses is protected by a lock, which we call RD, and (2) one where the read, and the two writes combined are protected by the same lock, which we call RDL (the code on the right). The main difference between the two versions is that $Y := 1$ will reach the read in the lock-free version, but cannot reach it in the presence of the locks. In a setting with one copy of $T'$ and $n$ copies of $T$, there are $2n$ definitions where only $n$ of them can reach the line $x := Y + 1$ of $T'$.

**Available Expressions.** The example set AE shows how the unfolding method can fully benefit from concurrency. The threads here do not have any dependencies. Each thread defines the same expression $X + Y$ twice, and therefore, the expression is always available for the second instruction of each thread. Table 5.1 shows that in the case of zero dependencies, the size of the unfolding grows linearly with the number of threads (especially since new threads do not introduce new dataflow facts).

| Example  | $|D|$ | #Threads | Unfolding Control Net | Unfolding Augmented Net | Checking Time (sec) | Construction Time (sec) |
|----------|-----|----------|-----------------------|------------------------|---------------------|------------------------|
| UV(10)   | 11  | 11       | 906                   | 4090                   | 0                   | 0                      |
| UV(20)   | 21  | 21       | 3311                  | 16950                  | 0                   | 0.70                   |
| UV(60)   | 61  | 61       | 40859                 | 156390                 | 0.01                | 60.11                  |
| RD(3)    | 4   | 6        | 410                   | 1904                   | 0                   | 0.03                   |
| RD(4)    | 5   | 8        | 1545                  | 9289                   | 0.01                | 1.5                    |
| RD(5)    | 6   | 10       | 5596                  | 41186                  | 0.01                | 133.16                 |
| RDL(3)   | 6   | 4        | 334                   | 1228                   | 0                   | 0.01                   |
| RDL(4)   | 8   | 5        | 839                   | 3791                   | 0                   | 29                     |
| RDL(5)   | 10  | 6        | 2024                  | 10834                  | 0                   | 5.35                   |
| RDL(6)   | 12  | 7        | 4745                  | 29333                  | 0.01                | 121.00                 |
| AE(50)   | 2   | 50       | 250                   | 650                    | 0                   | 0                      |
| AE(150)  | 2   | 150      | 750                   | 1950                   | 0                   | 0.34                   |
| AE(350)  | 2   | 350      | 1750                  | 4550                   | 0                   | 4.10                   |

Table 5.1: Programs and Performances
5.4 Conclusions

The main contribution of this work lies in the definition of a framework that captures dataflow analysis problems for concurrent program using partial orders that preserves the concurrency in the system. The preserved concurrency has been exploited in the partial-order based analysis, but could instead have been exploited in other ways, for example using partial-order reduction strategies as those used in SPIN.

As for future directions, the first would be to study local or compositional methods to solve the CCD problems and deploy them on large real world programs. This would have to handle (approximately) complex data such as pointers and objects. Our algorithms do not work for programs with recursion, and it is well known that dataflow analysis for concurrent programs with recursion quickly leads to undecidability. Structural restrictions like nested locking would be worth studying to obtain decidable fragments. Studying a framework based on computing minimal fixpoints for concurrent programs would be also interesting. Extending our approach to decide flow problems with infinite domains of finite height is challenging as well (they can be handled in the sequential setting [RSJM05]).
Chapter 6

State Space Reduction for Rewrite Theories

This chapter is an extended version of [FM06c]. We propose a new state space reduction technique within the rewriting logic semantic framework, in which concurrent systems are formally specified as rewrite theories.

Our technique is based on the idea of invisible transitions, that generalize a similar notion in POR techniques (see for example [CGP01]). The basic setting is that we assume a rewrite theory \( R = (\Sigma, E, R) \) in which a certain set \( P \) of state predicates has been equationally axiomatized by some of the equations in \( E \). \( R \) then has an associated Kripke structure, whose labeling function associates to each state (represented as an \( E \)-equivalence class of terms \([t]\) in the initial algebra \( T_{\Sigma/E} \)) all those predicates in \( P \) that hold in \([t]\) according to the equations \( E \). We then call a rewrite rule \( r \) in \( R \) P-invisible if in any rewrite step \([t] \rightarrow [t']\) using \( r \) the states \([t]\) and \([t']\) satisfy the same state predicates, i.e., they are labeled in the same way. Our state space reduction technique is then very simple: we identify a subset \( S \subseteq R \) of rules such that all rules in \( S \) are invisible. We then define the S-reduction of \( R = (\Sigma, E, R) \) as the rewrite theory \( R/S = (\Sigma, E \cup S, R \setminus S) \), that is, we turn all rules in \( S \) into equations, thus collapsing the set of states from \( T_{\Sigma/E} \) to the quotient \( T_{\Sigma/E \cup S} \). The intuitive idea, therefore, is that all states that can be reached from a given state by repeated \( S \)-transitions can be collapsed into a single one. In practice, as we show by means of several case studies in Section 6.2, the reductions obtained this way can be huge.

However, the above technique must meet an important executability requirement. The point is that, for \( E \) an arbitrary set of equations, rewriting modulo \( E \), which is the way transitions take place in the Kripke structure associated to \( R = (\Sigma, E, R) \), is in general undecidable. Therefore, to be able to execute and model check a rewrite theory in a rewriting logic language implementation such as Maude [CDE+02] [CDE+], we must require that the equations \( E \) are ground confluent and terminating (perhaps modulo some axioms \( A \)) and that the rules \( R \) are strongly ground coherent [Vir02]. Intuitively, the coherence requirement means that we can identify a state \([t]\) with the canonical form \( \text{can}_E(t) \) of \( t \) by the equations \( E \), and that rewriting with equations \( E \) and with rules \( R \) commutes in an appropriate sense, so that we can safely restrict our computations with \( R \) to only rewrite \( E \)-canonical forms. Therefore, the executability requirement for our technique is that \( R/S = (\Sigma, E \cup S, R \setminus S) \) should be executable, that is, that \( E \cup S \) should
be confluent and terminating, and that the rules \( R \setminus S \) should be strongly coherent with respect to \( E \cup S \) (perhaps modulo axioms \( A \)).

We show in Section 6.1 that the above-mentioned executability requirements on \( R / S \), besides being absolutely essential to model check \( R / S \) in practice, ensure a further very important property, namely that \( R \) and \( R / S \) are stuttering bisimilar, and therefore they satisfy exactly the same \( CTL^* \) formulas. Furthermore, to make our technique applicable to cases where a suitable set \( S \) may not be available, we generalize it to allow enlarging a set of invisible rules \( S \) by adding new invisible rules not in \( R \) to get a superset \( \hat{S} \supseteq S \). This gives rise to a state space reduction \( \hat{R} / \hat{S} \) that is no longer stuttering bisimilar to \( R \) but is nevertheless stuttering similar to it. This still allows us to verify \( ACTL^* \) formulas for \( R \) if we can model check them for \( \hat{R} / \hat{S} \), but such model checking can now give rise to spurious counterexamples. We illustrate how this more general technique is also quite useful in practice by means of a client-server protocol in Section 6.2.2.

![Figure 6.1: Restaurant State Space](image)

We can make all these ideas concrete by means of an example which models the workflow in a simplistic restaurant with one waiter and two customers. Customers have a flag indicating their status (waiting, ordered, or eating), so a customer is represented as a pair \( C(id, f) \) with \( id \) an identifier and \( f \) the flag. The waiter has also a status flag (free or order-taken). Therefore, the waiter is represented by a term of the form \( W(f) \). The restaurant state is a set with a waiter and two customers, with set union represented by a binary associative and commutative juxtaposition operator “\( \_ \)”. We have the following rewrite rules \( R \) in our theory \( \mathcal{R} = (\Sigma, A, R) \), where \( A \) consists
of the associativity and commutativity axioms for “_”:

\[ s_1 : W(\text{free})C(1, \text{waiting}) \rightarrow W(\text{order-taken})C(1, \text{ordered}) \]
\[ s_2 : W(\text{free})C(2, \text{waiting}) \rightarrow W(\text{order-taken})C(2, \text{ordered}) \]
\[ s_3 : W(\text{order-taken}) \rightarrow W(\text{free}) \]
\[ t_1 : W(\text{free})C(1, \text{ordered}) \rightarrow W(\text{free})C(1, \text{eating}) \]
\[ t_2 : W(\text{free})C(2, \text{ordered}) \rightarrow W(\text{free})C(2, \text{eating}) \]

Figure 6.1 (a) shows the state space induced by the above rewrite rules from an initial state with the waiter free and the two customers waiting.

Let us assume that the property \( \phi \) that we are interested in is: “eventually both customers eat”. This property can be expressed as formula \( \Diamond(e_1 \land e_2) \) where \( e_i \) is true if the \( i \)th customer’s status is “eating” and false otherwise. Rewrite rules \( s_1, s_2, \) and \( s_3 \) do not change the truth value of the predicates \( e_1 \) and \( e_2 \). One can observe that the rules in \( S = \{s_1, s_2, s_3\} \) are confluent and terminating and \( R \setminus S \) is strongly locally coherent \([Vir02]\) with respect to \( S \) modulo axioms. The reduced theory \( R/S \) (see the state space in Figure 6.1 (b), where each state represents an \( S \)-equivalence class) is then stuttering bisimilar to the original theory \( R \).

### 6.1 Invisible Transitions and the \( R/S \) Reduction

**Definition 41** Given a rewrite theory \( R = (\Sigma, E, R) \) having an equationally-defined set of atomic predicates \( P \), a rewrite rule \( \tau : l \rightarrow r \) if \( C \) in \( R \) is called \( P \)-invisible if for any \( [t] \in T_{\Sigma/E} \) and any \( u \in [t] \) such that \( u \xrightarrow{\tau} v \), then for each \( p \in P \) we have \( [t] \models p \iff [v] \models p \). We denote by \( \text{Inv}^P(R) \) the set of all \( P \)-invisible rewrite rules of \( R \).

Given a subset of rules \( S \subseteq R \), in a rewrite theory \( R/S = (\Sigma, E_0 \cup A, R) \), we call \( R/S = (\Sigma, S \cup E_0 \cup A, T = R \setminus S) \) the \( S \)-reduced theory of \( R \). We are particularly interested in the \( S \)-reduced theory of \( R \) when \( S \subseteq \text{Inv}^P(R) \), \( S \cup E_0 \) is confluent and terminating modulo \( A \), and \( T \) is coherent with respect to \( S \cup E_0 \) modulo \( A \).

**Theorem 42** Let \( R = (\Sigma, E_0 \cup A, R) \) be a rewrite theory with \( P \) a set of equationally defined atomic predicates. Let \( S \subseteq R \) be a set of \( P \)-invisible rules such that \( S \cup E_0 \) is confluent and terminating modulo \( A \), and \( T = R \setminus S \) is coherent with respect to \( S \cup E_0 \) modulo \( A \). Then \( R \) and \( R/S \) are stuttering bisimilar.

**Proof.** (sketch)\(^1\) The relation \( H \) on which the bisimilarity is based is defined by the quotient

---

\(^1\)See Section 2.5 to recall the relevant definitions.
homomorphism $H : T_{\Sigma/E} \rightarrow T_{\Sigma/E \cup S}$. We need to prove that: (a) $H$ is a stuttering simulation; and (b) that $H^{-1}$ is so too. Since $H$ maps deadlock states to deadlock states, and $H^{-1}$ of a deadlock state always contains a deadlock state, we can disregard deadlocks.

(a) It suffices to show that for each path $\pi$ in the underlying Kripke structure of the theory $\mathcal{R}$, $(T_{\Sigma/E,k} \xrightarrow{R \circ \bullet} L)$ (where $k$ is a kind for the state), there exists a stuttering equivalent path $\pi'$ in the underlying Kripke structure of the $S$-reduced theory $\mathcal{R}/S$, $(T_{\Sigma/E \cup S,k} \xrightarrow{S \circ \tilde{S}} L)$. $\pi$ must be of the following general form:

$$
\pi : [s_0]_E \xrightarrow{S} [t_0]_E \xrightarrow{S} [t_1]_E \xrightarrow{S} \ldots \xrightarrow{S} [t_n]_E \xrightarrow{S} \ldots
$$

Since the rules in $S$ are $P$-invisible, we know that $L(s_i) = L(t_i)$ for all $i$. Also, observe that by collapsing the $\xrightarrow{S}$, we have $[s_i]_{E \cup S} = [t_i]_{E \cup S}$. Then the following path

$$
\pi' : [t_0]_{E \cup S} \xrightarrow{T} [t_1]_{E \cup S} \xrightarrow{T} \ldots \xrightarrow{T} [t_n]_{E \cup S} \xrightarrow{T} \ldots
$$

is stuttering equivalent to $\pi$ and of course, by construction, it is a path in the underlying Kripke structure of $\mathcal{R}/S$.

(b) It suffices to show that for each path $\rho$ in the underlying Kripke structure $(T_{\Sigma/E \cup S,k} \xrightarrow{R/S} L)$ of the theory $\mathcal{R}/S$, there exists a stuttering equivalent path $\rho'$ in the underlying Kripke structure of the reduced theory $\mathcal{R}$, $(T_{\Sigma/E,k} \xrightarrow{R \circ \bullet} L)$.

Assume that $\rho$ is of the following general form:

$$
\rho : [s_0]_{S \cup E} \xrightarrow{T} [s_1]_{S \cup E} \xrightarrow{T} \ldots \xrightarrow{T} [s_n]_{S \cup E} \xrightarrow{T} \ldots
$$

We show by construction that there exists a stuttering equivalent path $\rho'$ of the following form:

$$
\rho' : [s'_0]_E \xrightarrow{S} [t_0]_E \xrightarrow{S} [s'_1]_E \xrightarrow{S} [t_1]_E \xrightarrow{S} \ldots \xrightarrow{S} [s'_n]_E \xrightarrow{S} [t_n]_E \xrightarrow{S} \ldots
$$

where $s_0 = s'_0$ and for all $i$, $s_i \equiv_{S \cup E} s'_i$, and therefore $L(s_i) = L(s'_i)$. $H^{-1}$ then relates the state $[s_i]_{S \cup E}$ to all the states on $[s'_i]_E \xrightarrow{S} [t_i]_E$ which by invisibility of $S$ all satisfy the same set of predicates.

$[s_i]_{S \cup E} \xrightarrow{T} [s_{i+1}]_{S \cup E}$ implies that there are terms $u_i$ and $u_{i+1}$ such that $s_i \equiv_{S \cup E} u_i \xrightarrow{T} u_{i+1} \equiv_{S \cup E} s_{i+1}$. If $s'_i \equiv_{S \cup E} s_i$ (meaning $s'_i H s_i$), then there is a term $t_i$ such that $s_i \xrightarrow{S \cup E} t_i$ and $s'_i \xrightarrow{S \cup E} t_i$. Since $u_i \equiv_{S \cup E} s_i$, by confluence of $S \cup E$, we have $u_i \xrightarrow{S \cup E} t_i$. Therefore, by $T$ being coherent with respect to $S \cup E$ modulo $A$, there exists a term $s'_{i+1}$ such that $t_i \xrightarrow{T} s'_{i+1}$ and $s'_{i+1} \equiv_{S \cup E} u_{i+1}$. 

64
Since \( u_{i+1} \equiv_{S \cup E} s_{i+1} \), we have \( s'_{i+1} \equiv_{S \cup E} s_{i+1} \) (meaning that \( s'_{i+1} \equiv H s_{i+1} \)).

\[ \begin{array}{c}
\xrightarrow{T} \quad s_i \quad \equiv_{S \cup E} \quad \xrightarrow{T} \quad u_{i+1} \quad \equiv_{S \cup E} \quad s_{i+1} \\
\xrightarrow{\tau} \quad S \quad \equiv \quad \xrightarrow{T} \quad s'_{i+1} \quad \equiv \quad \xrightarrow{T} \quad t_{i+1} \\
\xrightarrow{\star} \quad s'_{i} \quad \equiv \quad \xrightarrow{T} \quad s_{i+1} \quad \equiv \\
\end{array} \]

Start by letting \( s'_0 = s_0 \). Since \( s'_0 = s_0 \), it trivially holds that \( s'_0 \equiv_{S \cup E} s_0 \). Inductively construct the path according to the above diagram. Note that by viewing \( S \) steps as \( \tau \)-transitions, the above argument also shows that \( \equiv_{S \cup E} \) is a branching bisimulation relation \([NV95]\). ■

We have shown that, under the theorem hypothesis, the reduced rewrite theory \( R/S \) is stuttering \( P \)-bisimilar with the original theory \( R \). Therefore, (see Chapter 2) for any \( \phi \in \text{CTL}^*_{\text{X}}(P \Sigma) \) and any initial state \([t]_E\) we have

\[ R, [t]_E \models \phi \iff R/S, [t]_{E \cup S} \models \phi \]

In practice, the reduced theory \( R/S \) can have a drastically smaller state space than \( R \), making model checking of \( R/S \) feasible when model checking of \( R \) is unfeasible.

In cases where the \( R/S \) construction cannot be carried out for lack of a suitable \( S \) satisfying the confluence condition in Theorem 42, we can nevertheless achieve a similar state space reduction with a relation \( H \) that is a stuttering simulation. For example the client-server reduction in Section 6.2.2 is achieved in this manner. The general method is as follows: we assume that we have a set of rules \( S \subseteq R \) which are \( P \)-invisible \( (S \subseteq \text{Inv}_P(R)) \), and \( T = R \setminus S \) is coherent with respect to \( S \cup E_0 \) modulo \( A \), and \( S \cup E_0 \) is terminating but not confluent modulo \( A \). We then extend \( S \) to a set of rules \( \tilde{S} \) with \( S \subseteq \tilde{S}, \tilde{S} \not\subseteq R \), and where \( \tilde{S} \) is still \( P \)-invisible, and \((R \setminus \tilde{S}) \) is coherent with respect to \( \tilde{S} \cup E_0 \) modulo \( A \), and furthermore, \( E_0 \cup \tilde{S} \) is terminating and confluent modulo \( A \). Consider now the rewrite theory \( \hat{R} = (\Sigma, E_0 \cup A, R \cup \tilde{S}) \). Since \( \hat{R} \) has more rules than \( R \), if \( R \) is deadlock-free\(^3\) that is, if any state \([t]_E\) can always be rewritten by \( R \) to a new state \([t']_E\), then the following proposition is easy to prove:

**Proposition 43** The identity homomorphism \( 1_{T \Sigma/E_0 \cup A} : T \Sigma/E_0 \cup A \to T \Sigma/E_0 \cup A \) induces a \( P \)-simulation map from the underlying Kripke structure of \( R \) to that of \( \hat{R} \).

We can now apply Theorem 42 to \( \hat{R} \) to obtain a stuttering \( P \)-bisimilar \( \hat{S} \)-reduced theory \( \hat{R}/\hat{S} \).

\(^2\)Note that the simulation relations are strict in the sense that \( aHb \Rightarrow L(a) = L(b) \) and therefore negation does not have to be excluded.

\(^3\)Given a rewrite theory \( R \), we can always transform it into a bisimilar deadlock-free theory (see \([MPMO03]\)). Therefore, there is no real loss of generality imposed by this requirement.
closed under composition \([\text{PMMO05 Man01}]\), by composing the above simulation from \(R\) to \(\hat{R}\) with the stuttering bisimulation from \(\hat{R}\) to \(\hat{R}/\hat{S}\) generated by Theorem \([42]\) we obtain a stuttering simulation from \(R\) to \(\hat{R}/\hat{S}\) and therefore we have

**Theorem 44** Under the above assumptions for any \(\phi \in ACTL^*_X(P)\) and any initial state \([t]_{E_0 \cup A}\) in \(R\), we have \(\hat{R}/\hat{S}, [t]_{E_0 \cup \hat{S} \cup A} \models \phi \Rightarrow R, [t]_{E_0 \cup A} \models \phi\).

Therefore, if we can model check the property \(\phi\) using the reduced theory \(\hat{R}/\hat{S}\), we are then guaranteed that \(\phi\) holds in \(R\). See Section \([6.2.2]\) for an example.

### 6.2 Case Studies

We present five case studies showing how the \(R/S\) and \(\hat{R}/\hat{S}\) constructions can be achieved in practice for real applications, leading to massive reductions in the state space. All the experiments have been performed with the Maude LTL model checker running on an Intel machine with a 2.6GHz processor and 4GB of memory running Linux. These case studies fit into two categories: (1) pipeline examples, and (2) distributed protocols.

#### 6.2.1 Pipelines

**Mother of Pipelines (MOP)**

Proving correctness of microarchitectural processor pipelines (MA) with respect to their instruction set architecture (ISA) amounts to establishing a simulation relation between the behaviors of MA and ISA. Execution of any instruction can be seen as a sequence of smaller actions, and the observation that the mini-steps can be understood at an abstract level, without mentioning any concrete MA. Examples of mini-steps are fetching an instruction, getting an operand from the register file, having an operand forwarded by a previous instruction in the pipeline, writing a result to the register file, and retiring. MOP \([\text{KJO06}]\), an intermediate specification between ISA and MA, describes the execution of each instruction as a sequence of mini-steps. By design, our highly non-deterministic intermediate specification admits a broad range of implementations. This approach allows one to separate the implementation-independent proof obligations that relate ISA to MOP from those that rely upon the details of the MA. This can potentially amortize some of the proof effort over several different designs. The concept of parcels, formalizing partially-executed instructions, is used for a thorough treatment of mini-steps.

A parcel is a record with the following fields:
The state of ISA contains four elements: (1) program counter pc, (2) register file rf, (3) memory mem, and (4) instruction memory imem. State of MOP contains three additional elements: (5) a queue of parcels pq, and (6,7) two integers head and tail that define its front and back ends.

It can be shown [KJO06] that execution of any instruction at the ISA level is translated into one of the sequences defined by the pattern below, and hence MOP simulates ISA.

```
fetch ; decode ; (data1 rf || (data1 rf ; data2 rf)) ; (result || mem addr || (branch taken ; branch target)) ; [load || store] ; (next pc branch || next pc nonbranch) ; pc update ; retire
```

We specified the MOP as a rewrite theory $R_{MOP}$ by specifying each of the above steps (e.g., fetch) as a rewrite rule. One can show that these set of rules are locally confluent [KJO06]. Except for the fetch rule, the rest of the rules are terminating. The fetch rule is not terminating because it can keep fetching new instructions. The fetch rule is also locally strongly coherent with respect to the rest of the rules. Therefore, $R_{MOP}/S$, where $S$ contains all the rules except fetch, is stuttering bisimilar to $R_{MOP}$. It is also easy to see that $R_{MOP}/S$ is bisimilar to $R_{ISA}$ (each fetch rule is equivalent of execution of one step in ISA) and therefore our technique can show the same result from [KJO06] that $R_{MOP}$ is stuttering bisimilar to $R_{ISA}$. See Appendix 9.4 for the Maude specification of MOP.

We have experimented with MOP using the following simple program where you can see both the C code and the assembly code:

```c
sub r3, r3, r3 // clear r3.
addi r4, r3, 3 // set loop bounds in r4.
beqz r3, L2 // start the loop in a weird way.
L3:
    lw r5, r6, 0 // load array value into r5.
    addi r6, r6, 4 // increment array pointer.
```
add r3, r3, r5  // add r3 and r5, increasing sum.
subi r4, r4, 1  // decrement loop counter (r4).

L2:
bnez r4, L2  // loop control branch.
(done)

for(i = 0; i < 4; i++){
    sum += *array_1;
    array_1++;
}

We run this program on two versions MOP specification: (1) with all the rules (as appears above),
and (2) with everything but the `fetch` rule turned into an equation. Table 6.1 shows the perfor-
mance number for these experiments where \( N \) is the number of iterations of the loop in the program.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Time</th>
<th>Space</th>
<th>Time (reduced)</th>
<th>Space (reduced)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64.4</td>
<td>21264</td>
<td>0.08s</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>0.2s</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>0.3s</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>0.5s</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>0.7s</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 6.1: MOP Performance Results.

**Distributed Pipeline**

This example implements a pipeline with a number of stages ([ \( N \) ]) and buffers between the stages
([ \( N | NL \) ]). There is a first buffer [ 0 | data list ] which contains the data in the form of a list of
natural numbers. The first buffer keeps feeding data from the list to the first stage ([ 0 ]). Each
stage processes the data (here in the form of computing the function \( StageId \times 10^{StageId} + data \))
and passes the processed result to the next buffer. When the first buffer’s list is finished, it changes
its form from [ 0 | nil ] to { 0 | nil }. This marks the fact that no more data is entering the pipeline.
Other stages and buffers subsequently realize this and change their form from square brackets to
curly brackets to indicate this. The property that we would like to check for this system is that
eventually the last stage will realize that the computation is over. Here are the rules that describe
this system:

\[
\begin{align*}
\text{s}_1 & : \quad [N || MNL ][N] \rightarrow \ [N || NL ][N, M] \\
\text{t}_1 & : \quad \{N\}[s(N) || NL] \rightarrow \ \{s(N) || NL\} \\
\text{s}_2 & : \quad [N, M][s(N) || NL] \rightarrow \ [N][s(N) || NL(N \times (10^N) + M)] \\
\text{s}_3 & : \quad \{N || nil\}[N] \rightarrow \ \{N\} \\
\text{s}_4 & : \quad \{N || MNL\}[N] \rightarrow \ \{N || NL\}[N, M]
\end{align*}
\]

The set \( S = \{s_1, s_2, s_3, s_4\} \) can be shown to be confluent and terminating modulo associativity and commutativity, and \( \{t_1\} \) is coherent with respect to \( S \) plus associativity and commutativity axioms. Table 6.2 shows the results of model checking the above property for the original system and then the reduced version with respect to set \( S \).

### 6.2.2 Distributed Protocols

In this section, we look at four different distributed protocols specified as rewrite theories and how our reduction technique can help verify them.

**Leader Election Protocol**

We consider the simple case where the network is a ring consisting of \( n \) nodes, numbered from 1 to \( n \) in the clockwise direction. We want to investigate the LCR algorithm to select a leader. The informal description of this algorithm is as follows [Lyn96]:

Each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own the process declares itself the leader.

We can specify a rewrite theory modeling this algorithm by means of objects and messages, where the distributed state is a multiset of objects and messages built by an associative and commutative multiset union operator “\( \cup \)”: 

\[
\begin{align*}
\text{s}_0 & : \quad \langle I \rangle \rightarrow \ [I] (I \rightarrow I + 1 \mod N) \\
\text{s}_1 & : \quad [I] (J \rightarrow I) \rightarrow \ [I] \quad \text{if } J < I \\
\text{s}_2 & : \quad [I] (J \rightarrow I) \rightarrow \ [I](J \rightarrow I + 1 \mod N) \quad \text{if } J > I \\
\text{t} & : \quad [I] (I \rightarrow I) \rightarrow \ \text{Leader}(I)
\end{align*}
\]
where $N$ is the number of processes on the ring and $\langle I \rangle$ is the initial state of process $I$. In the first phase (rewrite rule $s_0$), each process $I$ sends its identifier to its neighbor and changes its format $[I]$, so that this is done only once. As soon as a process $I$ receives its own identifier through the ring, the computation is over; it removes all the object and the message and outputs Leader($I$). The messages are of the general form $(I \to J)$, where $J$ is the identifier of the receiver and $I$ is the integer content of the message.

The set of rules $S = \{s_0, s_1, s_2\}$ can be shown to be confluent and terminating modulo associativity and commutativity. Let us assume that the property that we are interested in is that eventually some process will be elected as leader. This is expressed by means of a single atomic predicate, $p$, that is true in any state containing Leader($I$). The rules in $S$ are $p$-invisible, and $t$ is coherent with respect to $S$ modulo the associativity and commutativity axioms. Therefore, by Theorem 42, we can use the stuttering bisimilar reduction $R/S$ to model check our property. Note that reducing $R$ with the rewrite rule $s_0$ above (which can easily be shown to be confluent and terminating) collapses an $N$-dimensional cube (generated by rule $s_0$) into a path of length $N$, meaning that the number of states in $R/\{s_0\}$ is reduced from $2^N$ to $N$, and the number of paths reduces from $2^N$ to 1. Table 6.2 shows the performance evaluation of model checking this problem before and after reduction using the Maude LTL model checker.

### Distributed Spanning Tree

A spanning tree of an undirected graph $G = (V, E)$ is a tree (i.e., a connected acyclic graph) that consists entirely of undirected edges and contains every vertex of $G$. The distributed spanning tree problem tries to find a spanning tree for a given set of network nodes $V$ that are connected by $E$. The asynchronous algorithm from [Lyn96] solves this as follows:

There is a distinct node $r$ that is initially marked and acts as the root. A marked node $v$ asynchronously sends a message to each of its neighbors once and for all. An unmarked node $v$ nondeterministically chooses one of the nodes who have sent it a message as its parent in the spanning tree, becomes marked, and discards all the other messages.

One possible way of specifying the above algorithm is by the following rewrite rules:

$$
\begin{align*}
    s_1 : & \quad [N | P, M \ NL] \quad \to \quad [N | P, NL](M \leftarrow N) \\
    t_1 : & \quad [N | \text{none}, NL](N \leftarrow M) \quad \to \quad [N | M, NL] \\
    s_2 : & \quad [N | M, NL](N \leftarrow K) \quad \to \quad [N | M, NL] \\
    s_3 : & \quad [N | \text{root}, NL](N \leftarrow K) \quad \to \quad [N | \text{root}, NL]
\end{align*}
$$

70
where the state is represented as a multiset (modulo associativity, commutativity, and identity) of nodes and messages. Each node is of the form \([N \mid P, L]\), where \(N\) is its unique identifier, \(P\) is its parent node (initially \texttt{none}), and \(L\) is the list of its neighbors (their identifiers to be exact). Variable \(M\) is of type integer which denotes a known parent and consequently cannot be \texttt{none} or \texttt{root}. The node with “root” as its parent is the root of the spanning tree. Let us assume that the property of interest is “to eventually reach a state in which every node has a parent”. This property can be expressed using a single atomic predicate, \(p\), that is false if there is a node with “none” as the parent. One can easily check that the set of rules \(S = \{s_1, s_2, s_3\}\) is \(p\)-invisible, confluent, terminating modulo associativity, commutativity, and identity, and \(t_1\) is coherent with respect to \(S\) modulo the same axioms. Since there are no equations (excluding the associativity, commutativity, and identity axioms) in the theory, one can turn these rules into equations and gain a huge reduction in the state space for model checking. Table 6.2 shows the performance evaluation of model checking this problem before and after this \(R/S\) reduction using the Maude LTL model checker.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Number of Nodes</th>
<th>Time</th>
<th>Space</th>
<th>Time (reduced)</th>
<th>Space (reduced)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader Election</td>
<td>10</td>
<td>3.6s</td>
<td>27633</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>2.7m</td>
<td>506037</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>19.3m</td>
<td>1329885</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Spanning Tree</td>
<td>3</td>
<td>0.02s</td>
<td>417</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10.2s</td>
<td>120183</td>
<td>0.01s</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>–</td>
<td>–</td>
<td>0.17s</td>
<td>625</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>–</td>
<td>–</td>
<td>0.5s</td>
<td>1296</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>–</td>
<td>–</td>
<td>110.22s</td>
<td>117649</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>–</td>
<td>–</td>
<td>99m</td>
<td>2097152</td>
</tr>
<tr>
<td>Client-Server</td>
<td>6</td>
<td>4.0s</td>
<td>125248</td>
<td>0.01s</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>81.4s</td>
<td>1753600</td>
<td>0.01s</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>–</td>
<td>–</td>
<td>0.01s</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>–</td>
<td>–</td>
<td>1.8s</td>
<td>32768</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>–</td>
<td>–</td>
<td>5.3m</td>
<td>1048576</td>
</tr>
<tr>
<td>Pipeline</td>
<td>15</td>
<td>5.4s</td>
<td>29961</td>
<td>0s</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>21.6s</td>
<td>118721</td>
<td>0s</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>117.5s</td>
<td>283401</td>
<td>0s</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>10.6m</td>
<td>579081</td>
<td>0s</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>–</td>
<td>–</td>
<td>0.2s</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>–</td>
<td>–</td>
<td>0.9s</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>–</td>
<td>–</td>
<td>2.2s</td>
<td>301</td>
</tr>
</tbody>
</table>

Table 6.2: Performance Results.
A Distributed Client-Server System

Consider a system consisting of several clients and one server. The server has a log (a list) for incoming requests. The clients send a message to the server to request a service. When the server receives a request message, it sends the relevant client a message containing the requested material, and adds an entry to its log \((B)\) to keep track of this communication. The following set of rewrite rules model a simple version of this system:

\[
\begin{align*}
    s_1 : \quad & [N \mid M] & \rightarrow & \{N \mid M\} (\text{server} \leftarrow (N, M)) \\
    s_2 : \quad & (\text{server} \leftarrow (N, M)) [\text{server} \mid B] & \rightarrow & [\text{server} \mid B (N, M)] (N \leftarrow \text{serv}(M)) \\
    t_1 : \quad & (N \leftarrow \text{serv}(M)) \{N \mid M\} & \rightarrow & \{N\}
\end{align*}
\]

where the state is a multiset (modulo associativity, commutativity, and identity of multiset union operator “\(\_\_\)”) of a server, clients, and messages. The server is indicated by identifier \text{server}. Clients each have an integer identifier \(N\) and another integer index \(M\) indicating the service they require from the server. Each client sends a message including its identifier and the index of the service to the server. The server replies back and logs the communication in its local list \(B\).

Assume that the property of interest is “a client that requires a service will eventually receive it”. This property can be expressed by a set \(P\) of two atomic predicates, of which one indicates the requirement of the service and the other indicates the receipt. The set \(\{s_1, s_2\}\) is \(P\)-invisible and a very good candidate for \(S\), but because of the list nature of the buffer, these rules are not confluent. For the property of interest, it does not matter in what order the messages are buffered; but since the resulting buffer is different, confluence does not hold. If one assumes a lexicographical ordering on the buffer (pairwise comparison of the pairs \((M, N)\)), then we can add the following rule which always sorts the buffer

\[
s_3 : \quad [\text{server} \mid B (N, M) (N', M') B'] \rightarrow [\text{server} \mid B (N', M') (N, M) B'] \text{ if } (N' > N) \lor ((N = N') \land (M' > M))
\]

and makes the set \(\widehat{S} = \{s_1, s_2, s_3\}\) confluent and terminating. \(\widehat{S}\) is also invisible, and \(t_1\) is coherent with respect to \(\widehat{S}\) modulo axioms. Therefore, one can reduce this theory to a theory of the form \(\widehat{R} / \widehat{S}\). Table 6.2 shows the performance evaluation of model checking this problem before and after reduction using the Maude LTL model Checker.

### 6.3 Discharging Proof Obligations

Typically, formal verification efforts using state space reduction techniques involve two separate tasks: (i) model checking the desired properties in the reduced model; and (ii) discharging proof obligations ensuring that the proposed reduction is indeed a correct reduction of the original system.
We discuss here the proof obligations that must be verified to ensure the correctness of an S-reduction \( R/S \), and ways in which the discharging of such obligations can be assisted by formal tools. For \( R/S \) to be a correct reduction of \( R \) the following proof obligations must be discharged:

1. the rules \( S \) must be proved \( P \)-invisible;

2. \( S \cup E_0 \) must be shown ground confluent and terminating modulo \( A \); and

3. the rules in \( R \setminus S \) must be proved locally strongly ground coherent with respect to the equations \( S \cup E_0 \) modulo \( A \).

Proving (1) is an inductive theorem proving task. Specifically, it amounts to proving that each state predicate \( p \in P \) and also its negation \( \neg p \) are both invariants for the rewrite theory \( (\Sigma, E_0 \cup A, S) \). This can be reduced to proving a series of first-order formulas that must be shown to hold inductively in the equational specification \( (\Sigma, E_0 \cup A) \); that is, to be satisfied in the initial model \( T_{\Sigma/E_0\cup A} \). Proofs can be assisted by any first-order inductive theorem prover. For Maude specifications Maude’s ITP \cite{CDEM98} can be used. The proof obligations for this task become considerably easier if the rules in \( S \) are topmost, that is, if all rewriting happens at the top of a term. Many rewrite theories whose state is a set or multiset of objects and messages, such as those in the case studies presented in this work, can be transformed into bisimilar topmost rewrite theories.

Proving (2) can be done mechanically using standard termination and confluence checking tools that support reasoning modulo axioms \( A \) such as associativity and commutativity, and can in some cases handle conditional rules. Tools of this kind include, for example, CiME \cite{CM96} (for both tasks) AProVE \cite{GTSKF04} (for termination), and for Maude specifications the Maude Termination Tool (MTT) \cite{DLM04} and the Maude Church-Rosser Checker \cite{CDEM98}.

There is a discussion on proving (3) in \cite{Vir02}. For most combinations of associativity, commutativity and identity axioms in \( A \) this task can be checked algorithmically when the rules are linear and unconditional. To the best of our knowledge the only tool available is Maude’s Coherence Checker \cite{Dur00}, which currently can only reason modulo commutativity axioms but it will soon be extended to handle associative and commutative axioms.

We now discuss briefly the proof obligations for the \( \hat{R}/\hat{S} \) reductions. To begin with, the same proof obligations (1)–(3) must be discharged, but now for \( \hat{R}/\hat{S} \) instead of \( R/S \). But that still leaves open the task of coming up with the rules \( \hat{S} \) in the first place. Two approaches are possible for this. On the one hand, as done in the case study of Section \ref{sec:6.2.2} one can use insight about the given specification to find a suitable \( \hat{S} \). On the other, it is also possible to automatically search for such a set \( \hat{S} \) by performing Knuth-Bendix (KB) completion modulo \( A \) on the equations \( E_0 \cup S \) using any KB completion tool (modulo \( A \)) such as, for example, CiME \cite{CM96}.
6.4 Related Work and Conclusions

Broadly speaking, our work is related to all other state space reduction and abstraction techniques (see for example [KP00, SS99, LGS+95, CGL94, CU98, DGG97, GW91, Val90, Pel94, ABH+97, KLM+98, FG05]). We discuss below several approaches that are most closely related to our own.

Several partial order reduction (POR) techniques achieve a reduction to a representative subset of all states while preserving various types of bisimilarity. Some of these techniques [GW91, Val90, Pel94, ABH+97, KLM+98, FG05] exploit the notion of invisibility, an idea that is generalized here to arbitrary rewrite theories. A first main difference with the POR approach is that POR techniques are typically dynamic (all except ABH+97, KLM+98), in the sense that the reduction is performed on-the-fly during the model checking and requires substantial changes to the underlying model checking algorithm (see [FM06b, God97] for an exception to this); by contrast, our technique is a static method, since we generate the reduced rewrite theory and then model check it. Furthermore, it does not require any changes in the model checker. A second important difference is in the different levels of generality: POR techniques typically assume a conventional concurrent language with processes and consider invisible process transitions, whereas our approach is much more general: it does not rely on these assumptions, and applies to arbitrary rewrite theories.

Our method has also some similarities with a reduction technique presented in [BvdP02]. However, the settings are quite different, because [BvdP02] works in the framework of process algebras, whereas our technique works for arbitrary rewrite theories. Furthermore, the notion of invisibility used in [BvdP02] is not based on a certain set of predicates. Instead, in our case the invisibility depends on what state predicates are involved in the property that we want to model check. Also, the notion of confluence used in [BvdP02] is completely different from ours: we use the standard term-rewriting notion. The notion of coherence used in this work has some similarities with notion of weak confluence in [Yin00] if one views the rules in \( S \) as \( \tau \)-transitions. Moreover, their approach is dynamic, while ours is static. The symbolic prioritization in [BvdP02] is relevant to our work in two senses: (1) it is static, and (2) it is giving priority to some transitions over the rest, while we also in some sense give priority to some rules over the rest.

Our reduction technique is also closely related to other notions of abstraction and simulation used for reduction purposes in rewriting logic. In the case of equational abstractions [MPMO03] one begins with a rewrite theory \( R = (\Sigma, E_0 \cup A, R) \) and adds extra equations \( G \) to it to obtain an abstract theory \( R/G = (\Sigma, E_0 \cup G \cup A, R) \), so that we have a rewrite theory inclusion \( R \subseteq R/G \). This technique is generalized in [MOMP04] to much more general rewrite theory morphisms \( H : \mathcal{R} \rightarrow \mathcal{R}' \) that need not be theory inclusions, give rise to simulations, and can be used for model checking purposes when \( \mathcal{R}' \) is more abstract than \( \mathcal{R} \). Our proposed technique is different from those in [MPMO03] and [MOMP04]. In our case the relationship between \( \mathcal{R} \) and \( \mathcal{R}/S \) cannot be
understood as a theory *morphism*: it is only a theory *transformation*. This means that we now have a new state space reduction technique for rewrite theories that nicely complements those proposed in [MPMO03, MOMP04].

Our technique makes essential use of Viry’s notion of coherence [Vir02] in rewrite theories. But we use the notion in precisely the *opposite way* than in Viry’s work. The original purpose of coherence is to make a rewrite theory $\mathcal{R} = (\Sigma, E_0 \cup A, R)$ executable by *turning the equations $E_0$ into rules*. Strong coherence then guarantees that $\mathcal{R}$ and the resulting theory $(\Sigma, A, E_0 \cup R)$ are semantically equivalent. We do somehow the opposite: beginning with a rewrite theory $\mathcal{R} = (\Sigma, E_0 \cup A, R)$ we select a subset of rules $S \subseteq R$ and *turn those rules into equations* to obtain our reduced theory $\mathcal{R}/S = (\Sigma, E_0 \cup S \cup A, R \setminus S)$. We then check strong coherence of $\mathcal{R}/S$ for executability and stuttering bisimilarity purposes.

We can summarize our contributions as follows: we have presented a general method to reduce the state space of a concurrent system specified as a rewrite theory $\mathcal{R}$ by selecting a set $S$ of $P$-invisible transition rewrite rules that, when turned into equations, yield a reduced theory $\mathcal{R}/S$. We have shown that if $\mathcal{R}/S$ satisfies reasonable executability assumptions it is stuttering bisimilar to $\mathcal{R}$ and therefore satisfies the same $CTL^*_X$ formulas under this bisimilarity. Several case studies presented show that $\mathcal{R}/S$ can have a drastically smaller state space in practice, making it feasible to model check properties for $\mathcal{R}$ by using $\mathcal{R}/S$ instead. We have also presented a method to obtain reductions of this kind using extra invisible rules not present in the original theory $\mathcal{R}$. The proof obligations that must be discharged to guarantee the correctness of our proposed reductions have also been discussed. Discharging them involves reasonable proof tasks that for the most part can be supported by existing formal tools.

This work is part of a broader effort to develop state space reduction techniques of wide applicability for concurrent systems specified as rewrite theories. In this sense, it complements earlier efforts to develop reduction techniques of this kind for rewrite theories [MPMO03, MOMP04, FM06b]. It is however a new technique, different from earlier ones. In future work we plan to further develop the ideas presented here in two opposite directions. In a more general direction, we plan to investigate *weaker conditions* under which invisible transitions $S$ can be used to reduce the state space. In a more specific direction, we plan to further advance the ideas initiated in Section ?? on applying these techniques to *distributed object systems*, to exploit the more specific nature of those systems to obtain even more drastic reductions. Two other aspects that need to be further developed are: (i) weakening the conditions for the $S$ rules to gain more reduction, and (ii) developing a broader experimental base of case studies.
Chapter 7

Formal Semantics and Analysis of Concurrent Programs

7.1 Introduction

Rewriting logic can be fruitfully used as a unifying framework for defining concurrent programming languages. The formal specification of a concurrent programming language can then be used as the basis for formal analysis tools. In this chapter, we discuss the techniques to develop these formal semantics and the analysis tools. Here, we focus on Java’s bytecode, but our methodology is general and can be applied also to the Java source code level and to many other languages. Some of the ideas presented in this chapter appeared in preliminary form in [FMR04, FCMR04].

The JavaFAN (Java Formal Analyzer) tool specifies the semantics of all JVM bytecode instructions as a Maude module specifying a rewrite theory $T_{JVM} = (\Sigma_{JVM}, E_{JVM}, R_{JVM})$, where $(\Sigma_{JVM}, E_{JVM})$ is an equational theory giving an algebraic semantics with semantic equations $E_{JVM}$ to the deterministic JVM instructions, whereas $R_{JVM}$ is a set of rewrite rules, with concurrent transition semantics, specifying the behavior of all concurrent JVM instructions. The three kinds of formal analysis currently supported in JavaFAN are: (1) symbolic simulation, where the theory $T_{JVM}$ is executed in Maude as a JVM interpreter supporting fair execution and allowing some input values to be symbolic; (2) breadth-first search, where the entire, possibly infinite, state space of a program is explored starting from its initial state using Maude’s search command to find safety property violations; and (3) model checking, where if a program’s set of reachable states is finite, linear time temporal logic (LTL) properties are verified using Maude’s LTL model checker.

A remarkable fact is that, as we explain in Section 7.3 even though $T_{JVM}$ gives indeed a mathematical semantics to the JVM, it becomes the basis of a formal analysis tool whose performance is competitive and in some cases surpasses that of other Java analysis tools. The reasons for this are twofold. On the one hand, Maude [CDE+03] is a high-performance logical engine, achieving millions of rewrites per second on real applications, efficiently supporting search, and performing LTL model checking with performance similar to that of SPIN [Hol97]. On the other, the algebraic specification of system states, as well as the equations $E_{JVM}$ and rules $R_{JVM}$, have been optimized for performance through several techniques explained in Section 7.2.6, including keeping only the dynamic parts of the state explicitly in the state representation, and making most equations and
rules unconditional. In this regard, rewriting logic’s distinction between the equations \( E_{JVM} \) and the rules \( R_{JVM} \) has a crucial performance impact in drastically reducing the state space size. The point is that rewriting with the rules \( R_{JVM} \) takes place modulo the equations \( E_{JVM} \), and therefore only the rules \( R_{JVM} \) affect state space size. Our experience in specifying the JVM in rewriting logic is that we gain the best benefits from the algebraic semantics (equations) and the SOS [Plo81] (Rules) paradigms in a combined way, while being able to distinguish between deterministic and concurrent features in a way not possible in either SOS or algebraic semantics.

**Related Work.** The different approaches to formal analysis for Java can be classified as focusing on either sequential or concurrent programs. Our work falls in the second category. More specifically, it belongs to a family of approaches that use a formal executable specification of the concurrent semantics of the JVM as a basis for formal reasoning. Two other approaches in precisely this category are one based on the ACL2 logic and theorem prover [KMM00], and another based on a formal JVM semantics and reasoning based on Abstract State Machines (ASM) [SSB01]. Our approach seems complementary to both of these, in the sense that it provides new formal analysis capabilities, namely search and LTL model checking. The ACL2 work is in a sense more powerful, since it uses an inductive theorem prover, but this greater power requires greater expertise and effort. In another sense, the ACL2 work is more limited. Because the ACL2 logic is purely functional and deterministic, concurrency has to be treated in an indirect way, by assuming a scheduler. By contrast, the rewriting logic semantics is fully concurrent and allows exhaustive analysis of all the concurrent executions of a JVM program.

Outside the range of approaches based on executable formal specification, but somewhat close in the form of analysis, is NASA’s Java Path Finder (JPF) [BHPV00 HP00], which is an explicit state model-checker for Java bytecode based on a modified version of a C implementation of a JVM. Preliminary rough comparisons of JavaFAN and JPF\(^1\) are encouraging, in the sense that we can analyze the same types of JVM programs of the same or even larger size. Other related work includes [PV98], which proposes an algorithm that takes the bytecode for a method and generates a temporal logic formula that holds iff the bytecode is safe; an off-the-shelf model checker can then be used to determine the validity of the formula. Among the formal techniques for sequential Java programs, some related approaches include the work on defensive JVM [Coh97], and the collective effort around the JML specification language and verification tools for sequential Java, e.g. [LLP+00 vdBJ01].

Another approach to define analysis tools for Java is based on language translators, generating simpler language code from Java programs and then analyzing them later. Bandera [CDH+00] extracts abstract models from Java programs, specified in different formalisms, such as PROMELA, which can be further analyzed with specialized tools such as SPIN. JCAT [DIS99] also translates Java into PROMELA. [PSSD00] presents an analysis tool which translates Java bytecode into C++

---

\(^1\) Authors thank Willem Visser for examples and valuable information about JPF.
code representing an executable version of a model checker. While the translation-based approaches can benefit from abstraction techniques being integrated into the generated code, they inevitably lead to natural worries regarding the correctness of the translations. Unnecessary overhead seems to be also generated, at least in the case of [PSSD00]; for example, exactly the same Remote Agent Java code that can be analyzed in 0.3 second in JavaFAN [FCMR] takes more than 2 seconds even on the most optimized version of the tool in [PSSD00].

### 7.2 Rewriting Logic Semantics of the JVM

We use Maude to specify the operational semantics of the JVM bytecode. This includes all bytecode instructions, defined in about 2000 lines of Maude code, including 373 equations and 43 rewrite rules. All the Java language features are supported. Garbage collection and many of the Java built-in libraries are not supported as part of the semantics. The formal semantics of each instruction is defined based on the informal description of JVM in [Ven99]. Besides the fact that we do not view the built-in libraries as part of the language semantics, supporting the built-in libraries is a non-trivial issue, specially since most of the Java libraries are in fact implemented in C. There is a solution for this problem which requires Maude calling the original libraries whenever needed. Although this may work as remedy for interpretation of programs, it will make a correct formal analysis of them virtually impossible. Since the emphasis of this work is on formal analysis tools based on semantic definitions, we do not see this solution as a suitable one.

Section 7.2.2 explains the operational semantics of the deterministic part of the JVM given by the 373 equations in $E_{JVM}$, and Section 7.2.3 discusses the semantics of the concurrent part of JVM specified by the 43 rewrite rules in $R_{JVM}$.

#### 7.2.1 Algebraic Representation of the JVM State

Here, we describe the representation of states in our model. Our major design goal has been to reduce the size and the number of system states to improve the performance of the formal analysis. The reduction in size has been achieved through separating the static and dynamic aspects of the program, maintaining only the dynamic part in the system’s state. To reduce the number of states, we keep the number of rewrite rules in the specification to a minimum. A detailed discussion on these optimizations is given in Section 7.2.6.

The JVM has four basic components: (1) the class space, (2) the thread space, (3) the heap, and

---

2 There are 373 equations that directly specify semantics of instructions. There are 420 other equations to perform auxiliary computations.
(4) the state transition machine, updating the internal state at each step.

In our model, no specific entity plays the role of the state transition system, and the strict separation of the classes, threads, and objects no longer exists. Instead, the state of the JVM is represented as a multiset of objects and messages\(^3\) in Maude \[CDE+03\]. Rewrites (with rewrite rules and equations) model the changes in the state of the JVM.

**Elements of the multiset.**

Objects in the multiset fall into four categories:

1. Maude objects which represent Java objects,
2. Maude objects which represent Java threads,
3. Maude objects which represent Java classes, and
4. auxiliary Maude objects used mostly for definitional purposes.

Below, we discuss each in detail.

**Java Objects** are modeled by objects, which are record-like data structures having an object identifier \(O\), a class (in this case \(\text{JavaObject}\)), and a set of attribute-value pairs. Objects of class \(\text{JavaObject}\) contain the following attributes.

\[< O: \text{JavaObject} | \mathit{Addr}:\text{HeapAddress}, \mathit{FieldValues}:\text{FieldValues}, \mathit{CName}:\text{ClassName}, \mathit{Lock}:\text{Lock} >\]

The \(\mathit{Addr}\) attribute refers to the heap address at which the object is stored. Physical heap addresses are employed only because they are used in the bytecode to refer to objects. The \(\mathit{FieldValues}\) attribute contains all instance fields and their values. Note that a single field may have more than one value, depending on its appearance in more than one class in the hierarchy of super classes of the Java class from which the object is instantiated. The sort \(\mathit{FieldValues}\) is a list of pairs, with each pair consisting of a class name and a list. The latter list by itself consists of pairs of field names and field values. Therefore, based on the current class of the object, we can extract the right value for a desired field. The \(\mathit{CName}\) attribute holds the name of the object’s class. The \(\mathit{Lock}\) attribute holds the lock associated with the object.

\(^3\)Messages are used as a method to define the semantics in our model. One can use a somewhat different approach which does not include any messages.
**Java Threads** are modeled by objects with the following attributes.

\[
\langle T : \text{JavaThread} \mid \text{callStack}: \text{CallStack}, \text{Status}: \text{CallStackStat}, \text{ORef}: \text{HeapAddress} \rangle
\]

The `callStack` attribute models the runtime stack of threads in Java. It is a stack of `frames`, where each `frame` models the activation record of a method call. Therefore, at any time, the top frame corresponds to the activation record of the method currently being executed. A `frame` is a tuple defined as follows.

\[
\text{op \{...\} : Int \text{Inst} \text{LabeledPgm} \text{LocalVars} \text{OperandStack} \text{SyncFlag} \\
\text{ClassName} \rightarrow \text{Frame}}
\]

The first component is an integer representing the program counter. The second component is the next instruction of the thread to be executed. The third component is a complete list of the instructions of the method, together with their corresponding offsets. The fourth component is the list of the current values of the local variables of the method. The fifth component contains the current operand stack, which carries instruction arguments and results. The sixth component is a flag indicating whether the call of the current method has locked the corresponding class (`SLOCKED`) or the corresponding object (`LOCKED`) or nothing at all (`UNLOCKED`). The last component represents the class from which the method has been invoked.

The `Status` attribute is a flag indicating the scheduling status of the thread: `scheduled` when the thread is ready to execute the next instruction, `waiting`, or `exception`. Examples of threads with `waiting` status include a thread waiting for the completion of a communication it has started in order to get the code of the method being invoked. The `exception` status is used to indicate that the top frame has thrown an exception, and the thread is not in the normal execution mode anymore. There is another attribute that a thread may have if an exception is raised which will indicate the type of exception raised. The `ORef` attribute contains the address of the object to which the thread is associated.

**Java Classes.** Each class is divided into its `static` and `dynamic` parts (see Section [7.2.6](#)), represented by `JavaClassS` and `JavaClassD` objects respectively. These objects contain the following attributes.

\[
\langle C : \text{JavaClassS} \mid \text{SupClass}:\text{ClassName}, \text{StaticFields}:\text{FlatFNL}, \text{Fields}:\text{FlatFNL}, \text{FieldsAR}:\text{ARList}, \text{Methods}:\text{MethodList} \rangle \\
\langle C' : \text{JavaClassD} \mid \text{ConstPool}:\text{ConstantPool}, \text{Lock}:\text{Lock}, \text{StaticFieldValues}:\text{FieldPairList} \rangle
\]

The `SupClass` attribute contains the name of the immediate superclass of the class represented. The attribute `StaticFields` is a list of pairs, each pair consisting of a class name together with the
list of static field names of that class. The classes in the first components of the pairs in this list are exactly the class represented by this object along with all its ancestor superclasses. These lists are compiled in a preprocessing phase. The **Fields** attribute, that keeps track of the instance fields, has exactly the same structure as **StaticFields**. The **FieldsAR** attribute keeps track of field attributes such as access permissions (including **public**, **private**, **protected**) and attributes (including **volatile**, **final**, **transient**). The **ConstPool** attribute models the constant pool in the Java class file. In our model the constant pool is an indexed list containing information about methods, classes, and fields. Bytecode instructions refer to these indices, that the threads use to extract (from the constant pool) the required information to execute the instructions. By looking at the *i*th entry of the constant pool, we get a **FieldInfo**, which contains a field name and the name of the class the field belongs to, or a **MethodInfo**, which contains the method name, the name of the class the method belongs to, and the number of arguments of the method, or a **ClassInfo**, which only contains a class name. Examples of instructions which refer to the constant pool include:

- **new #5**, which creates a new object of the class whose name can be found in the 5th element of the constant pool, or

- **invokevirtual #3**, which invokes a method whose information (name, class, and number of arguments) can be found at the 3rd entry of the constant pool.

The **Methods** attribute contains a list of tuples, each representing a method. The structure of the tuple is as follows:

```
op (..........) : MethodName MethodFormals MethodSync LabeledPgm Int ExceptionTable -> Method
```

The tuple components respectively represent the method name, a list of types of formal arguments of the method, a flag indicating whether or not the method is synchronized, the code of the method, the number of local variables of the method, and an exception table to guide the execution in case of particular exception risings. The **StaticFieldValues** attribute is exactly the same as **FieldValues** already discussed for **JavaObject**, except that this list refers to the values of static fields (which are stored inside the class) as opposed to the values of instance fields (which are stored inside the object). The **Lock** attribute holds the lock associated with the class.

**Auxiliary Objects:** Several objects in the multiset do not belong to any of the above categories. They have been added for definitional/implementation purposes. Examples include:

1. An **object collecting the outputs** of the threads. This object contains a list of values. When
a thread prints a value, it appends this value to the end of this list. Input is assumed to be
hardwired in the Java program at the moment.

2. A heap manager, that maintains the last address being used on the heap. We do not model
garbage collection at the moment, but a modification of the heap manager can add garbage
collection to our current JVM definition.

3. A thread name manager, that is used to generate new thread names.

4. There are several Java built-in classes that had to be apriori defined. The support for in-
put/output, creating new threads, and wait/notify facilities are among the most important
ones. All of these built-in classes have been created separately and are added as part of the
initial multiset.

7.2.2 Equational Semantics of Deterministic JVM Instructions

If a bytecode instruction can be executed locally in the thread, meaning that no interaction with the
outside environment is needed, that instruction’s semantics can be specified using only equations.
The equations specifying the semantics of all these deterministic bytecode instructions form the
equational $E_{JVM}$ part of the JVM’s rewrite theory. In this section we present some examples of how
deterministic bytecode instructions are modeled in our system. The complete Maude representation
and a collection of examples can be found in [FCMR].

iadd

This instruction is executed locally in the thread, and therefore, is modeled by the equation shown
in Figure 7.1. Values I and J on top of the operand stack are popped, and the sum $I + J$ is
pushed. The program counter is moved forward by the size of the iadd instruction to reach the
beginning offset of the next instruction. The current instruction (which was iadd before) is also
changed to be the next instruction in the current method code. Nothing else is changed in the
thread. ATS (of type AttributeSet) captures the rest of the state of the state of the object that
is not relevant to this particular instruction. Many of the bytecode instructions are typed. In this
example, by defining I and J to be integer variables, we support dynamic type checking as well.
Several dynamic checks of this kind are supported.
Invokevirtual

This instruction is used to invoke a method from an object. It is among the most complicated bytecode instructions and its specification includes several equations and rewrite rules. The equation in Figure 7.2 is the first part of \texttt{invokevirtual} semantics. One thread, one Java object, and one Java class are involved. When the thread reaches the \texttt{invokevirtual} instruction, by looking at the reference on top of the operand stack (\texttt{REF(K)}), it figures out from what object it has to call the method. The method information (see Section 7.2.1) will be extracted from the constant pool. The class \texttt{ClassName} needs to be involved, since the constant pool is stored inside this class. The class (\texttt{ClName}) is the current\footnote{Note that this can change dynamically.} class of the object \texttt{O}, therefore the code of the desired method should be extracted from the constant part of it. The thread will send a message asking for the code of the method, sending all the information to uniquely specify it. The last entity before the condition is a message. This message is consumed later and the desired method is sent back to the thread through another message. The thread receives the message, and that is when the invocation is completed. If the method being invoked is a \texttt{synchronized} method, the thread has to acquire a lock before the invocation is completed. This then has to be done using a rewrite rule (see Section 7.2.6).

7.2.3 Rewriting Semantics of Concurrent JVM Instructions

The semantics of those bytecode instructions that involve interaction with the outside environment is defined using rewrite rules, thus allowing us to explore all the possible concurrent executions of a program. In this section we present the semantics of several concurrent bytecode instructions.
monitorenter

This instruction (Figure 7.3) is used to acquire a lock. This makes a change in the shared space between threads, and so has to be specified by a rewrite rule. One Java object and one Java thread are involved. The thread executing monitorenter acquires the lock of the object whose reference is on top of the operand stack (REF(K)). The heap address of the object (K) is matched with this reference, and the lock of the object is changed to indicate that the object is now locked once by the thread T (note that a thread can lock or unlock an object several times). See section 7.2.4 for a more detailed discussion on locking and unlocking procedures.

getfield

This is a more complex instruction modeled by the rewrite rule in Figure 7.4. One thread and two Java classes are involved in this rule. The I operand is an index to the constant pool referring to the field information ([I, {ClName, FieldName}]), namely, field’s name and its corresponding class name. The Java class ClassName is needed to extract the constant pool. The Java object O is identified by matching its heap address K with the reference REF(K) on top of the operand stack. On the right hand side of the rule, the thread proceeds to the next instruction, and the value of the indicated field of object O is placed on top of the operand stack of the thread (FV[ClName, FieldName] # OperandStack).
7.2.4 Synchronization

We support three means of thread synchronization: (1) synchronized sections, (2) synchronized methods, and (3) wait/notifyAll methods. In this section we explain how these means of synchronization are modeled. The first two are the only methods intrinsic to the JVM which are handled in the virtual machine itself, and the only one we had to implement as a part of the complete specification of the JVM. wait/notifyAll methods are built-in Java library methods that we have added to be able to study more interesting examples.

The synchronized sections in Java are translated into sections surrounded by the monitorenter (see Figure 7.3) and monitorexit bytecode instructions. During the execution of both, an object reference is expected to be on top of the operand stack whose corresponding lock is acquired and released, respectively. Each Java object is modeled by a Maude object that includes a Lock attribute. This attribute has a tuple structure of the following form:

\[
\text{op Lock : OidList Oid Int -> Lock}.
\]

The first component is a list of identifiers of all threads that have waited on this object. This corresponds to wait and notifyAll methods (see below). The second component shows which thread currently owns the lock of this object (NoThread if none). The third component is a counter that shows how many times the owner of the lock has acquired the lock, since each lock can be acquired several times by the same owner.

When a thread encounters the monitorenter instruction, it checks whether the lock of the corresponding object is free. If so, the lock is changed to belong to this thread, and the thread can proceed to the critical section. It is also possible that the lock is not free, but has been acquired by the same thread before. In this case, only the counter is increased by one. When the thread finishes the execution of the critical section and reaches the monitorexit instruction, it simply decreases the counter by one. If the counter becomes zero, the lock is marked as free.

The synchronized methods are modeled in a very similar way. The difference is that, when the method is synchronized, monitorenter and monitorexit are replaced by method invocation and return, respectively. These methods are modeled through different rewrite rules, since different bytecode instructions are used for them.

Adding support (with little effort) for the wait and notifyAll methods of the Java built-in class Object is an interesting problem that we have solved. Similar to synchronization primitives, wait and notifyAll are called expecting an object reference on top of the operand stack. The thread (calling these methods) should already own the lock of the object on top of the operand stack. When wait is called, the thread releases the lock of the corresponding object, which it must own,
and goes to sleep. It will not continue unless notified by another thread. The lock of the object is marked as free, the identifier of the current thread is added to the list of threads waiting on this object (the first component of the lock), and the integer indicating the number of times the thread had locked the corresponding object is stored locally in the sleeping thread, so that it can be recalled when the thread wakes up.

When notifyAll is called, a (broadcast) message is created containing the list of all threads which have waited on the corresponding object up to that point. This message will then be consumed by all the threads in this list. Each thread that consumes the message will try to wake up. In order to continue their execution, all these threads have to compete to acquire the lock on the specific object, to follow the rest of their executions inside the synchronized section. After the lock becomes available, one thread nondeterministically\(^5\) acquires it.

### 7.2.5 Exception Handling

If an exception is raised, whether it is the result of some illegal action or an intended programmed one, the status of the thread executing the code is changed to exception. The execution of the thread is stopped in exception status. The thread looks at the exception table of the top frame (the one that raised the exception in the first place). If there are instructions to deal with this exception, meaning a new address to restart the execution from, the instructions are applied and normal execution is resumed. If not, the top frame is popped and the same procedure is performed in the new top frame. If none of the frames in the stack of the thread have an entry in their exception table to deal with the raised exception, then the execution of the thread finishes.

Besides all the exceptions raised during the execution of each thread as the result of something unexpected happening, there is an instruction athrow in the bytecode which allows exceptions to be used as a programming tool. Figure 7.5 specifies the equational semantics of the athrow instruction, which, being a local action to a thread, is modeled using an equation. \(\text{REF}(I)\) is a reference to a throwable object, based on which the type of the exception thrown is decided. The Status of the thread is changed to exception, and a new attribute Exception, of the type of the class of the throwable object, appears in the thread.

\(^5\)In our model, but in general various implementations of the JVM use a variety of algorithms to choose the thread. By not committing to any specific deterministic choice approach, our formal analysis can discover subtle violations that may appear in some JVM implementations, but may not show up in others.
7.2.6 Optimizations

Below, we discuss two major optimizations we have applied to decrease the size and number of system states, as well as the size of the state space.

Size of the State. In order to keep the state of the system small, we only maintain the dynamic part of the Java classes inside the system state. Every attribute of Java threads and Java objects can potentially change during the execution, but Java classes contain attributes that remain constant all along, namely, the methods, inheritance information, and field names. This, potentially huge amount of information, does not have to be carried along in the state of the JVM. The attributes of each class are grouped into dynamic and static attributes. The former group appears in the multiset, and the latter group is kept outside the multiset, in a Maude constant defined equationally and accessed through auxiliary operations.

Rules vs. Equations. Using equations for all deterministic computations, and rules only for concurrent ones leads to great savings in state space size. The key idea is that the only two cases in which a thread interacts with (possibly changes) the outside environment are shared memory accesses and acquiring locks. Examples of the former include the semantics of the instruction `getfield` (see Section 7.2.3), where a rule has been used. As an example for the latter case, we refer the reader to the semantics of the `monitorenter` instruction (see Section 7.2.3). Since only the 40 rules in $R_{\text{JVM}}$ contribute to the size of the state space, which is basically a graph with states as nodes and rewrite transitions as edges, we obtain a much smaller state space than if all the deterministic bytecode instructions had been specified as rules, in which case 340 rules would be used.

7.3 Formal Analysis

Using the underlying fair rewriting, search and model checking features of Maude, JavaFAN can be used to formally analyze Java programs in bytecode format. The Maude’s specification of the JVM
can be used as an interpreter to simulate fair JVM computations by rewriting. *Breadth-first search analysis* is a semi-decision procedure that can be used to explore all the concurrent computations of a program looking for safety violations characterized by a pattern and a condition. Infinite state programs can be analyzed this way. For finite state programs it is also possible to perform explicit-state model checking of properties specified in linear temporal logic (LTL).

### 7.3.1 Simulation

Our Maude specification provides executable semantics for the JVM, which can be used to execute Java programs in bytecode format. This simulator can also be used to execute programs with symbolic inputs. Maude’s `rewrite` command provides fair rewriting with respect to objects, and since all Java threads are defined as objects in the specification, no thread ever starves, although no specific scheduling algorithm is imposed. This assumption of fairness (with respect to threads) coincides with real models of JVM with a built-in scheduler, since scheduling algorithms also take the fairness into account. This fairness assumption does not mean that a deadlock is avoided; a *deadlock* in our model is a state in which no more rewrites are possible. The fair rewriting helps us avoid the situations in which a thread stuck in a loop is being executed forever, while other threads that can also be executed are starving.

To facilitate user interaction, the JVM semantics specification is integrated within the JavaFAN tool, that accepts standard bytecode as its input. The user can use `javac` (or any Java compiler) to generate the bytecode. She can then execute the bytecode in JavaFAN, being totally unaware of Maude. We use `javap` as the disassembler on the class files along with another disassembler `jreversepro` to extract the constant pool information that `javap` does not provide.

### 7.3.2 Breadth-first Search

Using the simulator (Section 7.3.1), one can explore only one possible trace (modeled as sequence of rewrites) of the Java program being executed. Maude’s `search` command allows exhaustively exploring all possible traces of a Java program. The breadth-first nature of the `search` command gives us a semi-decision procedure to find errors corresponding to violation of invariants even in infinite state spaces, being limited only by the available memory. Below, we discuss a number of case studies for this kind of analysis.
Remote Agent.

The Remote Agent (RA) is an AI-based spacecraft controller that has been developed at NASA Ames Research Center and has been part of the software component of NASA’s Deep Space 1 shuttle. On Tuesday, May 18th, 1999, Deep Space 1’s software deadlocked 96 million kilometers away from the Earth and consequently had to be manually interrupted and restarted from the ground station. The blocking was due to a missing critical section in the RA that had led to a data-race between two concurrent threads, which further caused a deadlock [HLP+00, HLP01]. This real-life example shows that even quite experienced programmers can miss data-race errors in their programs. Moreover, these errors are so subtle that they often cannot be exposed by intensive testing procedures, such as NASA’s, where more than 80% of a project’s resources go into testing. This justifies formal analysis techniques like the ones presented in this work which could have caught that error.

The RA consists of three components: a Planner that generates plans from mission goals; an Executive that executes the plans; and a Recovery system that monitors RA’s status. The Executive contains features of a multithreaded operating system, and the Planner and Executive exchange messages in an interactive manner. Hence, this system is highly vulnerable to multithreading errors. Events and tasks are two major components (see [FCMR] for the code). In order to catch the events that occur while tasks are executing, each event has an associated event counter that is increased whenever the event is signaled. A task then only calls wait_for_event in case this counter has not changed, hence, there have been no events since it was last restarted from a call of wait_for_event.

The error in this code results from the unprotected access to the variable count of the class Event. When the value of event1.count is read to check the condition, it can change before the related action is taken, and this can lead to a possible deadlock. This example has been extensively studied in [HLP+00, HLP01]. Using the search capability of our system, we found the deadlock in the same faulty copy in 0.3 seconds. This should be compared with the tool in [PSSD00], which finds it in more than 2 seconds in its most optimized version.

The Thread Game.

The Thread Game [Moo] is a simple multithreaded program which shows the possible data races between two threads accessing a common variable (see [FCMR] for the code). Each thread reads the value of the static variable c twice and writes the sum of the two values back to c. Note that these two readings may or may not coincide. An interesting question is what values can c possibly hold during the infinite execution of the program. Theoretically, it can be proved that all natural

---

[6] All the performance results given in this section are in seconds on a 2.4GHz PC.
numbers can be reached \[\text{Moo}\].

We can use Maude's search command to address this question for each specific value of \(N\). The search command can find one or all existing solutions (sequences) that lead to get the value \(N\). We have tried numbers up to 1000, where for all of them a solution is found in a reasonable amount time (Table 7.1).

<table>
<thead>
<tr>
<th>(N)</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(s)</td>
<td>7.2</td>
<td>17.1</td>
<td>41.3</td>
<td>104</td>
<td>4.5m</td>
<td>10.1m</td>
</tr>
</tbody>
</table>

Table 7.1: Thread Game Times.

### 7.3.3 Model Checking

Maude's model checker is explicit-state and supports Linear Temporal Logic. This general-purpose rewriting logic model checker can be directly used on the Maude specification of JVM’s concurrent semantics. This way, we obtain a model checking procedure for JVM programs for free. The user has to specify the property in a simple specification language that we devised for this purpose. Figure 7.6 demonstrates this language. The tool JavaFAN automatically translates the predicates written in this language into the equivalent atomic propositions in Maude (the formula naturally remains the same), which are then used to model check the LTL property. We illustrate this kind of model checking analysis by the following examples. All the performance figures given indicate the model checking performance \textit{without} the use of partial order reduction optimizations. We postpone the discussion of partial order reduction techniques for the model checking of JVM programs to Chapter 8.

Property ::= Atoms : Formula (property)
Formula ::= formula: LTL-Formula(Atoms) (currently LTL property)
Atoms ::= Atom ; Atoms \mid Atom (list of atomic predicates)
Atom ::= atom name : CAcc \mid atom name : OAcc (atomic predicate)
CAcc ::= . \mid BExpList (static field access)
OAcc ::= @ \mid BExpList (object field access)
BExpList ::= BExp \mid BExp and BExpList (List of boolean expressions)
BExp ::= field name op integer (one field boolean expression)
\quad op \in \{<,\leq,>,\geq,=,!=\}

Figure 7.6: Property Specification Language
<table>
<thead>
<tr>
<th>Tests</th>
<th>Times(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(4)</td>
<td>0.64</td>
</tr>
<tr>
<td>DP(5)</td>
<td>4.5</td>
</tr>
<tr>
<td>DP(6)</td>
<td>33.3</td>
</tr>
<tr>
<td>DP(7)</td>
<td>4.4m</td>
</tr>
<tr>
<td>DP(8)</td>
<td>13.7m</td>
</tr>
<tr>
<td>DP(9)</td>
<td>803.2m</td>
</tr>
<tr>
<td>DF(4)</td>
<td>21.5</td>
</tr>
<tr>
<td>DF(5)</td>
<td>3.2m</td>
</tr>
<tr>
<td>DF(6)</td>
<td>23.9m</td>
</tr>
<tr>
<td>DF(7)</td>
<td>686.4m</td>
</tr>
</tbody>
</table>

Table 7.2: Dining Philosophers Times.

**Dining Philosophers.**

See [FCMR] for the version of the dining philosophers problem that we have used in our experiments (DP). The property that we have model checked is whether the first philosopher can eventually dine. The formula that we model checked is $\Diamond \text{Dine}$. The model checker generates counterexamples, in this case a sequence of states that lead to a possible deadlock. The sequence shows a situation in which each philosopher has acquired one fork and is waiting for the other fork. Currently, we can detect the deadlock for up to 9 philosophers (Table 7.2). We also model checked a slightly modified version of the same program which avoids deadlock (DF). In this case, we can prove the program deadlock-free when there are up to 7 philosophers. This compares favorably with JPF [BHPV00, HP00] which for the same program could not deal with 4 philosophers at the time this performance comparison was made.

**2-stage Pipeline**

This example implements a pipeline computation (see [FCMR] for the code), where each pipeline stage executes as a separate thread. Stages interact through connector objects that provide methods for adding and taking data. The property we have model checked for this program is related to the proper shutdown of pipelined computation, namely, “the eventual shutdown of a pipeline stage in response to a call to stop on the pipeline’s input connector”. The LTL formula for the property is $\Box (c\text{1stop} \rightarrow \Diamond (\neg \text{stage1return}))$. There are two static fields (of the same name belonging to the class CheckPoints) that turn from zero to one when the relevant event is observed. The above property can be specified using our property specification language as follows:

```plaintext
atom c1stop : CheckPoints.(c1stop == 1);
```
<table>
<thead>
<tr>
<th>Tests</th>
<th>Times(s)</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>RW(1,1)</td>
<td>1.5</td>
<td>262</td>
</tr>
<tr>
<td>RW(2,1)</td>
<td>28.3</td>
<td>2900</td>
</tr>
<tr>
<td>RW(1,2)</td>
<td>36</td>
<td>3859</td>
</tr>
<tr>
<td>RW(2,2)</td>
<td>15.0m</td>
<td>70623</td>
</tr>
</tbody>
</table>

Table 7.3: Readers/Writers Performance.

JavaFAN model checks the property and returns true in 17 minutes (no partial order reduction was used). This compares favorably with the model checker in [PSSD00], which without using the partial order reduction performs the task in more than 100 minutes.

Readers/Writers

This example presents the classic readers/writers implemented in Java, where the correctness criteria demands that readers and writers cannot be both present in the critical section. Here is the property in our specification language:

atom state1return : CheckPoints.(stage1return == 1) :
formula : [] ( c1stop => <> stage1return)

This property is modeled by means of two predicates Readers and Writers, which are respectively true if (any) readers, respectively writers are present in the critical section. Table 7.3 shows the results of the model checking this property, where the two parameter in RW(i, j) respectively show the number of readers and writers in the example.

7.4 Lessons Learned and Future Work

We have presented JavaFAN, explained its design, its rewriting logic semantic basis, and its Maude implementation. We have also illustrated JavaFAN’s formal analysis capabilities and its performance on several case studies (except for the use of partial order reduction which is discussed in Chapter 8). The main lessons learned are that, using a rewriting logic semantics and a high-performance logical engine as a basis to build software analysis tools for conventional concurrent
programs has the following advantages: (1) it is *cost-effective* in terms of amount of work needed to develop such tools: a few months for a single person, in our experience, to develop the entire JVM specification; (2) it provides a *generic* technology that can be applied to many different languages and, furthermore, the program analysis tools for each language come essentially *for free* from the underlying logical engine; and (3) it has *competitive performance* compared to state-of-the-art software analysis tools tailored to a specific language.

As always, there is work ahead. On the one hand, in collaboration with Feng Chen support for the Java source code level has also been added to JavaFAN and we plan to gain more experience and further optimize the tool at both levels. On the other hand, a number of state space reduction techniques can be used to further tame the state-space explosion problem and make JavaFAN more scalable. Some of these techniques are discussed in Chapter 8 and others are work in progress.
Chapter 8

State Space Reduction for Programming Languages

This chapter focuses on incorporating generic partial order reduction techniques in semantic-based formal analysis tools. We discuss in detail a technique to develop language-independent partial order reduction tools. Similar techniques can be used to exploit other reduction techniques, such as transaction-based reduction, which is work in progress. This chapter extends the ideas first published in [FM06b, FM05].

8.1 Language-independent Partial Order Reduction Tools

Although the theoretical foundations of POR are very general and can be applied to many different languages, at present POR-enabled software model checkers are typically language-specific: they typically only work for programs in a particular language such as Java, C, Promela, and so on. One exception to this common situation is the Verisoft tool [God97]. However, Verisoft is applied in practice to a limited family of languages. The question this work raises and presents an affirmative answer to is: can POR-enabled software model checking tools become language-independent in the strong sense of being generic, that is, being applicable not just to a few, but to an unlimited class of languages satisfying some very basic abstract requirements?

Our method is based on a theory transformation $\mathcal{R}_L \mapsto \mathcal{R}_{L+POR}$ in which the original rewrite theory $\mathcal{R}_L$ specifying the semantics of the language $L$ is transformed into a stuttering equivalent rewrite theory $\mathcal{R}_{L+POR}$ that accomplishes the desired partial order reduction when used for model checking a given program. This theory transformation approach means that no changes to the underlying model checker are needed to achieve the desired partial order reduction, which is one of the reasons why developing a POR-enabled LTL model checker for a language $L$ using our method requires such little effort. Besides its generality and short development time, our method has two additional advantages:

1. Flexible Partial Order Heuristic Algorithm. The heuristic algorithm can be specified using a few equations. Although our basic version of the heuristic can in principle work for any programming language, additional optimizations, based on specific knowledge of the given programming language
or of the types of programs to be verified, can make the POR reduction considerably more efficient. The tool builder can easily customize the heuristic algorithm, which compares favorably with having to change the source code of a model checker.

2. Flexible Dependence Relation. Although a basic dependence relation can generally hold for a certain programming language, additional knowledge of the types of programs that one needs to verify can result in removing some dependencies; for example, Java supports shared memory in general, so we have to assume that memory read/write pairs are generally interdependent; but if the programs being verified do not use the shared memory at all, we can remove this dependency for such programs. Having the dependence relation as an explicit parameter of the partial order reduction module not only contributes to the generality of the method, but also gives the tool builder the advantage of specializing it, based on the type of input programs.

Our generic theory transformation assumes a simple basic interface of functionality in the language L. This allows a first phase of automatic transformation of the theory \( \mathcal{R}_L \). But this can be followed by a second language-specific customization phase supporting features 1–2 above. This can be easily accomplished by adding or customizing a few equations in this second phase, so that detailed knowledge of L’s semantics can be used to optimize the reduction; for example, by optimizing the heuristic algorithm and/or by defining a more precise dependence relation using static analysis techniques.

Besides developing its theoretical foundations and establishing its correctness, the practical usefulness of a generic method like the one we propose should be evaluated experimentally. Therefore, we have developed a prototype tool in Maude that, given an original semantics of a language L specified as a rewrite theory \( \mathcal{R}_L \), performs the theory transformation \( \mathcal{R}_L \mapsto \mathcal{R}_L^{+\text{POR}} \) and can be used to model check LTL properties of programs in L using Maude’s generic LTL model checker. We have applied this prototype to the rewriting semantics of the Java bytecode, Java source code, a simple Promela-like language, and Maude; and have evaluated the performance of our POR methods for these languages using several benchmarks. The goal of this prototype and experimentation is a proof-of-concept one. Therefore, we have not incorporated a number of well-known optimizations that a mature tool should support. Nevertheless, our experiments indicate that, even without such optimizations, substantial gains in time and space can be obtained using our POR method.
8.2 Partial Order Reduction for Language Definitions

8.2.1 Some Assumptions

In order to devise a general partial order reduction module for semantic definitions of concurrent programming languages, we have to make some basic assumptions about these semantic definitions. These assumptions are quite reasonable and do not limit in practice the class of semantic definitions that we can deal with. They simply specify a standard interface between the semantic definition module and the partial order reduction module. We can enumerate these assumptions as follows:

1. In each program there are entities equivalent to threads which can be uniquely identified by a thread identifier. The computation is performed as the combination of local computations inside individual threads, and communication between these threads through any possible discipline such as shared memory, synchronous or asynchronous message passing, and so on.

2. In any computation step (transition) a single thread is always involved. In other words, threads are the entities that carry out the computations in the system.

8.2.2 The Theory Transformation

The rewrite theory $R_L = (\Sigma_L, E_L, R_L)$ specifying the semantics of a concurrent programming language $L$ is transformed in two steps into the semantically equivalent theory $R_{L+POR} = (\Sigma_{L+POR}, E_{L+POR}, R_{L+POR})$ that is equipped with partial order reduction capabilities.

The Marked-State Theory. The objective of the first step of this transformation is to change the original theory $R_L$ in order to facilitate the addition of the partial order reduction module. In the transformed theory $\hat{R}_L = (\hat{\Sigma}_L, \hat{E}_L, \hat{R}_L)$:

1. the rewrite rules of $R_L$ are changed syntactically to only allow one-step rewrites, and

2. the structure of the states of $R$ is enriched to allow a specific thread to be marked as enabled. Rewrite rules are then modified to only allow the threads that are marked enabled to make a transition.

This way, when the POR heuristic decides on an ample set, the corresponding threads can be marked as enabled, and this causes only the ample transitions to be explored next. Here we give a detailed construction of $\hat{R}_L$ and show that $R_L$ and $\hat{R}_L$ are one-step bisimilar.
We assume that $R_L$ is coherent \[Vir92\] and that all rules in $R_L$ are of the form $l(u(t)) \rightarrow r(u'(t))$, where terms $l$ and $r$ are of sort $State$, and where the subterms $u(t)$ and $u'(t)$ are thread expressions of sort $Thread$, and $t$ is variable ranging over thread identifiers of sort $Tid$. Note that, based on the assumptions we made (see Section \[8.2.1\]), there is going to be exactly one such thread expression $u(t)$ on either side of a rule. We also assume that the equations in $E_L$ are thread-preserving, that is, in any two state expressions equated by $E_L$ both must have the same number of thread expressions and there is a bijective correspondence between such thread expressions preserving their thread identifiers.

We define $\widehat{\Sigma}_L$ by adding fresh new sorts: $MState$ and $MThread$. A new constructor $enabled : Thread \rightarrow MThread$ is introduced for the sort $MThread$ to instrument threads with this additional flag. This allows us to mark them as enabled or not for the next execution step. The use of the sort $Thread$ in all state constructors is everywhere replaced by the sort $MThread$. We also add two unary operators $\{.,[,] : State \rightarrow MState$ for purposes of forcing one-step rewrites as explained below. The equations in $\widehat{E}_L$ are systematically derived from those in $E_L$ by replacing in each equation in $E_L$ each occurrence of a thread expression $u(t)$ by the expression $enabled(u(t), b_t)$, where $b_t$ is a fresh new variable of sort $Bool$ depending on $t$. For every rewrite rule $l(u(t)) \rightarrow r(u'(t))$ if $C$ in $R_L$, the corresponding rewrite rule in $\widehat{R}_L$ is then of the form $\{Ct(l(enabled(u(t),true)))] \rightarrow [Ct(r(enabled(u'(t),true))))]$ if $\widehat{C}$, where $Ct(.)$ is the context expression for the application of the rule in case $r$ does not rewrite the entire state but only a state fragment and where $\widehat{C}$ is the conjunction of equations obtained from $C$ by changing each equation in $C$ containing thread expressions as done in the definition of $\widehat{E}_L$, and leaving all other equations untouched. The use of the operators $\{.,[,]$ in the rules in $\widehat{R}_L$ means that in $\widehat{R}_L$ only one-step rewrites are possible, since the operator $[,]$ in the right-hand side blocks the application of any further rules. This feature is added to allow a more controlled execution of the semantic rules under the guidance of the POR module, as explained later.

As an example of the above transformation, consider the following rewrite rule specifying the semantics of the `monitorenter` instruction of Java bytecode:

```plaintext
r1 < T: JavaThread | callStack:([PC, monitorenter, Pgm, ..., (REF(K) #
OperandStack), ...] CallStack), ... > < O : JavaObject | Addr: K,
..., Lock: Lock(OIL, NoThread, 0) > => < T: JavaThread | callStack:
([PC + 2, Pgm(PC + 2), Pgm, ..., OperandStack, ...] CallStack),
... > < O: JavaObject | Addr: K, ...., Lock: Lock(OIL, T, 1) > .
```

the transformed rewrite rule has the following form:

\[\text{If the rule } r \text{ rewrites the global state of the computation, the context } Ct(.) \text{ is empty, i.e., } Ct(l(\bar{u})) = l(\bar{u}). \text{ We do however allow language specifications in which a rule } r \text{ can be local to some fragment of the state. In this second case, it is important to make explicit a pattern } Ct(.) \text{ for the context in which the rule is applied.}\]
The key point about the transformation $R_L \mapsto \hat{R}_L$ is that if we have a one-step rewrite $u \rightarrow v$ with $R_L$, then we have also a corresponding one-step rewrite $\pi(u) \rightarrow \pi(v)$ with $\hat{R}_L$; and conversely, if we have a one-step rewrite $u' \rightarrow v'$ with $R_L$, then we can find $u \in \pi^{-1}(u') \ v \in \pi^{-1}(v')$ such that we have a one-step rewrite $u \rightarrow v$ with $\hat{R}_L$. The following proposition formalizes this statement:

**Proposition 45** The surjective projection $\pi$ mapping terms of sort MState to terms of sort State defined by: (1) erasing the operators $\{\_\} \ [\_]$, and (2) erasing the enabled operators, the corresponding flags and the context expression defines a one-step bisimulation between the corresponding rewrite theories.

**Proof.** (Sketch). Since we have assumed that $R_L$ is coherent [Vir92], which is the usual requirement for executability of a rewrite theory, the equations $E_L$ are Church-Rosser and terminating, and we can assume that the rules in $R_L$ are only applied to terms in $E_L$-canonical form. The first key observation is that the map $\pi$ defines also a one-step bisimulation between the one-step equational rewriting relation $\rightarrow^1_{E_L}$ and $\rightarrow^1_{\hat{E}_L}$. This can be shown by structural induction on the structure of rewrite proofs, using the fact that the equations in $\hat{E}_L$ preserve all flags. It is then easy to show that $\hat{E}_L$ is also terminating, and that a term $\hat{u}$ is in $\hat{E}_L$-canonical form iff $\pi(\hat{u})$ is in $E_L$-canonical form. Showing that $\hat{E}_L$ is Church-Rosser then follows using the just-established one-step bisimulation at the equational rewriting level, the fact that $E_L$ is Church-Rosser, and by observing that the equations in $\hat{E}_L$ are thread-preserving (because those in $E_L$ are) and flag-preserving by construction. In a similar way we can then show that the rules in $\hat{R}_L$ are coherent with respect to the equations $\hat{E}_L$.

To show the one-step bisimulation at the level of the rewrite rules $\hat{R}_L$ and $R_L$, we can now assume terms in $\hat{E}_L$-canonical form (resp. $E_L$-canonical form). The fact that a one-step rewrite $\hat{u} \rightarrow_{\hat{R}_L} \hat{u}'$ induces a one-step rewrite $\pi(\hat{u}) \rightarrow_{R_L} \pi(\hat{u}')$ follows easily from the definition of $\hat{R}_L$ and of the function $\pi$. Conversely, if we have a one-step rewrite of state terms $u \rightarrow_{R_L} u'$, say with a rule $r \in R_L$, we can always choose a term $\hat{u} = \{v\} \in \pi^{-1}(u)$, where $v$ is the term obtained from $u$ by marking the flags in all threads of $u$ as true. Then, by the definition of $\hat{R}_L$, the rule $\hat{r}$ applies to $\text{widehat} \{v\}$ (note that if $r$ and $\hat{r}$ are conditional, then, by the bisimulation at the equational level, the $r$ condition holds for $u$ iff the $\hat{r}$ condition holds for $\{v\}$) and we have a one-step rewrite $\hat{u} \rightarrow_{\hat{R}_L} \hat{u}'$ with $\pi(\hat{u}') = u'$.

98
The Partial Order Reduction Theory. In the second step, the theory $\hat{R}_L = (\hat{\Sigma}_L, \hat{E}_L, \hat{R}_L)$ is transformed into $R_{L+POR} = (\Sigma_{L+POR}, E_{L+POR}, R_{L+POR})$, which adds to $\hat{R}_L$ the partial order reduction module. Components of the transformed theory are defined based on the components of $\hat{R}_L$ as follows:

- $\Sigma_{L+POR} = \hat{\Sigma}_L \cup \Sigma_{POR} \cup \Sigma_{AUX}$, that is, the signature $\hat{\Sigma}_L$ is extended with the signature $\Sigma_{POR}$ of operators used in implementing the partial order heuristic algorithm, plus the signature of auxiliary operators $\Sigma_{AUX}$ that are used for implementation purposes.

- $E_{L+POR} = \hat{E}_L \cup E_{POR} \cup E_{AUX}$, that is, the set of equations $\hat{E}_L$ is extended with the equations $E_{POR}$ which specify the partial order heuristic algorithm, plus the equations $E_{AUX}$ which define the auxiliary operators.

- $R_{L+POR} = \hat{R}_L \cup \{r_{\text{step}}\}$. In the case of the rewrite rules, only one new rewrite rule is added. We label this rule as $\text{step}$. It is the only rule applicable to the new state, and therefore the only rule which will determine the transitions of the system at a given state.

The New State. There is a new fresh sort $PState$, as part of $\Sigma_{POR}$, representing the new state of the system. A new sort $StateInfoSet$ also belongs to $\Sigma_{POR}$, capturing all the information necessary for the reduction algorithm (see Section 8.2.3). A new constructor operator $\{\_\_\} : MState \rightarrow PState$ is introduced for the new state. Therefore, a state in $R_{POR}$ is a pair $\{s|I\}$, where $s$ is a state in $\hat{R}_L$, and $I$ is a term containing information necessary for the reduction algorithm.

The New Rule (\text{step}). A single new conditional rule $r_{\text{step}}$ in $R_{L+POR}$ simulates one step rewrites of the original system:

$$\text{step} : \{s|I\} \rightarrow [s'|I] \text{ if } s \rightarrow s' \land s \neq s'$$

where $s$ and $s'$ are variables of sort $MState$, and the operators $\{\_\_\}$ and $\{\_|\_\}$ are state constructors for the sort $PState$ and are frozen operators $[\text{CDE}^+03]$, that is, no rewriting is allowed below these operators. $I$ is a variable of sort $StateInfoSet$. By using this single rewrite rule, only one rewrite at a time can happen, which changes the given state to one of its successor states. Since the resulting state is in $\{\_|\_\}$ format, no rewrite rule is applicable to it anymore, until it is changed to the $\{\_\}$ format. This is the point at which the partial order heuristic algorithm is applied, using an equation that completes the effect of the above rule:

$$[s \mid I] = \{\text{state(MarkAmple}(s, I)) \mid \text{stateInfo(MarkAmple}(s, I))\}. \quad (*)$$
The partial order reduction is applied at state \( s \), using the information in \( I \), by means of a single operation \textit{MarkAmples}. This operation takes a pair of elements of sorts \textit{MState} and \textit{StateInfoSet} as an input, and returns a pair of the same sort. The \textit{MarkAmples} operation computes the ample set for the current state and returns the state with the ample transitions marked as specified by the POR algorithm. It also returns an updated version of \textit{StateInfoSet} (see the POR algorithm part of Section 8.2.3). In the next section, we discuss in detail how the \textit{MarkAmples} operations is specified.

8.2.3 The Partial Order Reduction Module

This module performs two main tasks: (1) extracting the set of enabled transitions at a given state, and (2) finding an ample subset of these transitions.

First, we have to define a transition in this context. Having the rewriting semantics \((\Sigma_L, E_L, R_L)\) of a concurrent programming language \( L \), one can view the initial state of the system (a program and its inputs) as a \( \Sigma_L \)-term \( t \) being rewritten by the equations \( E_L \) and the rewrite rules \( R_L \) of the specification.

In a state transition system, a given state \( s \) has a set of immediate successor states \( \{s_1, s_2, \ldots, s_k\} \), and each pair \((s, s_i)\) is an enabled transition from state \( s \). In the rewriting semantics, state \( s \) is a term, and the set of enabled transitions leading to successor states can be represented as a set of pairs \((r_i, p_j)\), where \( r_i \in R_L \) and \( p_j \) is a position in term \( s \). In other words, if a certain rule \( r_i : l(\vec{u}) \rightarrow r(\vec{v}) \) is enabled at a position \( p_j \) in term \( s \), then we have a transition from \( s \) to its successor \( s[l(\vec{u})\backslash r(\vec{v})] \).

In general, a position \( p \) can be any position in the term \( \text{tree} \). However, in our special case of semantics of concurrent programming languages together with the general assumptions discussed in Section 8.2.1 a thread identifier will uniquely specify a position, since we have assumed that a single thread is involved in each rewrite. Therefore, a pair \((t_i, r_j)\) consisting of a thread identifier \( t_i \) together with an applicable rule \( r_j \) uniquely characterizes a transition. This gives us a considerable practical advantage: because when the algorithm decides on an ample subset of the transitions, it suffices to mark the corresponding threads as enabled (see Section 8.2.1), which makes it unnecessary for all the unmarked threads (transitions) to be explored. Note that in the transformed theory, although the only rule applied to the state of the system is the \textit{step} rule, in fact an application of \textit{step} always simulates some rewrite rule \( r_i \) from the original system, and it is that rule that we consider in the above pair.
Extracting Enabled Transitions

As discussed above, a transition is a pair \((t_i, r_i)\) of a thread identifier and a rewrite rule. We can add a third component \(I_k\) to this tuple, which includes all the information about context (i.e., names of variables, functions, locks, ...). This information can later help resolving some dependencies between the transitions, which may result in fewer dependencies and possibly in a better reduction.

At a given state \(s\), we have to find all pairs \((t_i, r_j : l(\vec{u}) \rightarrow r(\vec{v}))\) where the rewrite rule \(r_j\) is enabled for the term \(s\) at the position associated with the thread \(t_i\). In other words, we have to go over all the rewrite rules \(r_j \in R_L\) and find all the positions at which \(r_j\) can be applied to the term \(s\). To do this, we generate a new set of equations, based on the rewrite rules in \(R_L\), with exactly one equation per rule in the following manner. Let us assume that a rewrite rule \(r \in \hat{R}_L\) is of the following general form:

\[
r : \{l(u(t))\} \Rightarrow [r(u'(t))] \text{ if } C
\]

where \(u(t)\) and \(u'(t)\) are subterms of sort \(Thread\), \(t\) is a variable of sort \(Tid\), and \(C\) is the rule’s condition. The corresponding equation for \(r\) is then:

\[
\langle T_e, l(u(t)) \rangle = \langle T_e \cup \{< t, r, I >\}, l(u(t)) \rangle \text{ if } C \land T_e \cup \{< t, r, I >\} \neq T_e
\]

where \(T_e\) is a set that accumulates enabled transitions. Note that rewrite rules in \(\hat{R}_L\) are already modified to capture the context in which the corresponding original rule of \(R_L\) would have been applied. Starting from the pair \(< \emptyset, t_s >\), by applying all equations of the above form, we will converge to the pair \(< T_e, t_s >\), where \(T_e\) is the set of all enabled transitions.

Since the context information \(I\) depends on the specific programming language \(L\) and on the way the semantics of \(L\) is defined, the \(I\) component has to be left as a null constant when these equations are generated automatically based on the rules. However, a tool builder familiar with the language semantics can customize these equations to include whatever context information may be useful later. In our experience with several rewriting semantics for different programming languages, there are relatively few rewrite rules in the semantic definitions (that is, \(E_L\) is much bigger than \(R_L\)), so this process is rather quick and easy.

Computing the Ample Set

**Dependence Relation.** The definition of a dependence relation between the transitions is required for computing the ample sets. The dependence relation is represented by the operator \(Dependence: Transition Transition \rightarrow Bool\). Clearly, the dependence relation is different for different programming languages. Some common dependence properties can be shared by many
**Figure 8.1: Two Partial Order Reduction Heuristics.**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Take a transition $t$ from $T_{e,s}$.</td>
</tr>
<tr>
<td>2</td>
<td>Let $T_a = tr(t)$.</td>
</tr>
<tr>
<td>3</td>
<td>If $C'_1(T_a)$ and $C_2(T_a, P)$ and $C_3(T_a)$.</td>
</tr>
<tr>
<td>4</td>
<td><strong>then</strong> mark thread of $t$ as ample. quit.</td>
</tr>
<tr>
<td>5</td>
<td><strong>else</strong> go to step 1.</td>
</tr>
<tr>
<td>6</td>
<td>Mark all threads as ample.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Take a transition $t$ from $T_{e,s}$.</td>
</tr>
<tr>
<td>2</td>
<td>Let $T_a = tr(t)$.</td>
</tr>
<tr>
<td>3</td>
<td>Let $S = \mu_{\text{CD},T_{e,s}}(T_a)$.</td>
</tr>
<tr>
<td>4</td>
<td>If $C'_1(S)$ and $C_2(S, P)$ and $C_3(S)$.</td>
</tr>
<tr>
<td>5</td>
<td><strong>then</strong> mark thread of $t$ as ample. quit.</td>
</tr>
<tr>
<td>6</td>
<td><strong>else</strong> go to step 1.</td>
</tr>
<tr>
<td>7</td>
<td>Mark all threads as ample.</td>
</tr>
</tbody>
</table>

These procedures are called at each state (see Section 8.2.2) to compute the ample set at that state. The algorithm on the left is a simpler version, which only considers ample sets including transitions of

programming languages, such as: “all the transitions in a single thread are interdependent”, which is expressed by the following equation:

$$\text{Dependence}(<t, r, I>, <t, r', I'>) = \text{true}$$

where $t$ is a variable ranging over thread identifiers, $r$ and $r'$ are variables ranging over rule names, and $I$ and $I'$ are variables ranging over context information.

In order to have the best possible reduction, the language specifier/tool builder should supply the definition of the dependence relation for the given language as a set of additional equations. The dependence relation can often be defined through a few equations, even for complicated languages. See Section 8.4 for the definition of the dependence relation for the Java bytecode. Note that, in general, since the dependence relation is defined by a set of equations (that can potentially be conditional) we can naturally support the case of conditional dependence as in [DHP04, KP92].

**The Heuristic Algorithm.** Since the core of the heuristic algorithm can be specified using a few equations, we have specified two different heuristics. Many additional optimizations for these heuristics and also other heuristics can likewise be specified with little effort (see Section 8.5), but they are beyond the scope of this work. Figure 8.1 shows both algorithms. Functions $C'_1$, $C_2$, and $C_3$ check the three conditions discussed in the next section, returning **true** or **false**. These procedures are called at each state (see Section 8.2.2) to compute the ample set at that state. The algorithm on the left is a simpler version, which only considers ample sets including transitions of
a single thread. The algorithm on the right extends the former to consider ample sets that can include transitions of more than one thread, which can result in a better reduction. If we have \( n \) threads, and at some point no single thread can be a candidate for ample, we may be able to find a subset of threads that can satisfy the conditions as a whole. To do so, we use the transitive closure of the dependence relation \( D \) defined on the set \( T \) of transitions as follows:

\[
D : T^2 \to \{\text{true}, \text{false}\} \quad S, T, U \subseteq T, t \in T
\]

\[
tr : T \to \mathcal{P}(T)
\]

\[
c_{D,S} : \mathcal{P}(T) \to \mathcal{P}(T)
\]

\[
\mu_{c_{D,S}} : \mathcal{P}(T) \to \mathcal{P}(T)
\]

where \( c_{D,S}(T) \) computes all the transitions of \( S \) which are immediately dependent on transitions in \( T \). Since \( S \) is a finite set of transitions, \( c_{D,S} \) is monotonic; if we reapply \( c_{D,S} \) repeatedly, we eventually reach a set \( T \) (a fixpoint) where \( c_{D,S}(T) = T \). The function \( \mu_{c_{D,S}} \) represents this fixpoint. The set \( \mu_{c_{D,T_e}}(t) \) is a good candidate for an ample set, since we know that at least no transition outside the set \( \mu_{c_{D,T_e}}(t) \) is dependent on anything inside it. A good method to find the best ample set is to sort the sets \( \mu_{c_{D,T_e}}(t) \), for all \( t \in T_e \) based on their cardinality, and then start checking the conditions, beginning with the smallest one. This way, if we verify all the conditions for a candidate set, we are sure that it is the smallest possible ample set, and we are done.

Checking The Conditions.

The most involved part of the partial order reduction algorithm is checking the conditions in [CGP01]. Conditions \( C2 \) and \( C3 \) are exactly the same as in [CGP01]. Condition \( C'1 \) is a stronger version (see the proof of Theorem [46]) of condition \( C1 \) from [CGP01] (since the original \( C1 \) from the POR theory is not locally verifiable) and very similar to the variation of it in the heuristic proposed in [CGP01]. Since the algorithm always works on nonempty sets, we are left to check three out of the four conditions. Here, we describe how the conditions are checked for a candidate set of transitions (ample set). The special case of a single transition as a candidate (as in [CGP01]) follows from this easily.

\( T_e \) represents the set of all enabled transitions in the current state. Note that, as argued before, the notions of transition and of enabled thread are equivalent in our framework, so we often switch between the two.

\( C'1: \) if transition set \( T \subset T_e \) is a an ample set, then no thread in \( T_e - T \) should have a transition in the future that is dependent on \( t \). To compute future transitions of a thread
$t_i \in T_e - T$, a conservative flow-insensitive context-insensitive static analysis of the code is performed. This kind of static analysis can be done locally, and is different for different programming languages. Therefore, the language specifier/tool builder needs to provide it. In the definition of the algorithm we assume that there is an operation $ThreadTransitions$ which takes the thread identifier and the current state of the system and returns all the future transitions of the thread in the form of a set of tuples (transition format) through a purely static analysis of the code of the input program which usually offers an overestimation of the actual set. Having the future transitions of all the threads in $T_e - T$, condition $C'1$ can then be easily checked by using the dependence relation.

C2: ample transitions should be invisible if the state is not fully expanded. This condition is the simplest of the three to verify. The set of propositions used in the desired property is given as an input. The check just has to go over this set, element by element, and check whether each proposition has the same truth value in state $s$ and in its successor state with respect to all transitions in the ample candidate set.

C3: Cycle-closeness Condition. This condition ensures that no transition is enabled over a cycle in the state transition graph and is never taken in the ample set. This condition can be easily checked when the partial order reduction algorithm is embedded in a model checker, since the stack of states being explored is available. In our case, we use exactly the same method, but we simulate part of that stack as part of the state. The second component of the new system state, $StateInfoSet$ takes care of this. Whenever in a state $s$ there is a transition $t$ outside the ample set, the pair $(t, s)$ will be stored in the $StateInfoSet$ component. As soon as a transition is taken in some future step, the pair is removed from the $StateInfoSet$. If a pair $(t, s)$ is still there when we revisit $s$, we know that we are closing a cycle, so we must take the transition.

8.2.4 Correctness of the Theory Transformation

The correctness of our theory transformation can be now stated as the following theorem:

**Theorem 46** Assuming that a set $AP$ of atomic state predicates has already been added to $\mathcal{R}_L$ by means of a set of equational definitions, the Kripke structures associated to the rewrite theories $\mathcal{R}_L$ (with State as its sort of states) and to $\mathcal{R}_{L+POR}$ (with $PorState$ as its sort of states) are stuttering bisimilar.

*Proof.* (Sketch). The stuttering bisimulation we are after is a relation between terms of sort $porState$ in $\mathcal{R}_{L+POR}$ and terms of sort $State$ in $\mathcal{R}_L$. The bisimulation relation is defined by a surjective function $\pi'$ which: (i) erases the $\{ | \}$ and $\lfloor | \rfloor$ operators and discards the $StateInfo$
components; and (ii) applies the \( \pi \) function defined in Section 8.2.2. We now have to show that both \( \pi' \) and its inverse relation \( \pi'^{-1} \) are stuttering simulations. But in fact, \( \pi' \) defines an ordinary simulation (therefore a trivial case of a stuttering simulation) from \( R_{L+POR} \) to \( R_L \), since any one-step application of the step rule requires a one-step rewrite with \( \hat{R}_L \) of the corresponding \( Mstate \) components, that is, of the first erasing (i) above; and by Proposition 1 (see Section 8.2.2) \( \hat{R}_L \) is one-step bisimilar to \( R_L \) with \( \pi \), which is the second erasing (ii) above.

To prove that \( \pi'^{-1} \) is a stuttering simulation, we rely on Theorem 12 of [CGP00], which states that for every path in the original system \( R_L \), there is a stuttering equivalent path in the system \( R_{L+POR} \) reduced with respect ample sets which satisfy conditions \( C_1 \), \( C_2 \), and \( C_3 \) (Condition \( C_0 \) is implicit in our case).

Note that the conditions \( C_2 \) and \( C_3 \) used in this paper (for both heuristics) are exactly the same as the corresponding conditions in [CGP00]. Condition \( C'_1 \) is a stronger version of condition \( C_1 \) from [CGP00], meaning that \( C'_1 \Rightarrow C_1 \). [CGP00] defines \( C_1 \) as follows:

**C1:** along every path in the full state graph that starts at \( s \), the following condition holds: a transition that is dependent on a transition in \( \text{ample}(s) \) cannot be executed without a transition in \( \text{ample}(s) \) occurring first.

Our condition \( C'_1 \) strengthens this condition in the sense that it says that a transition that is dependent on a transition in \( \text{ample}(s) \) cannot be executed along any path starting at \( s \) following a transition outside \( \text{ample}(s) \) at all. Those starting at \( s \) and following one of the \( \text{ample}(s) \) transitions clearly satisfy \( C_1 \).

### 8.3 Methodology: POR for JavaFAN

This section includes a short report on how the technique presented in this chapter has been used to add POR capabilities to the JavaFAN tool for analysis of Java programs at both source code and bytecode levels.

#### 8.3.1 A Case Study for the Java Language

The purpose is to illustrate the methodology in a step by step manner, and since the Java specification [Che] has not been written by the author, emphasize more that this technique works for any language specification under minimal assumptions.

The Java specifications [Che] satisfies the two assumptions from Section 8.2.1 threads have unique
identifiers, and each thread has at most one action enabled at each moment which is the action on top of its continuation. Let us show how the theory transformation is done in the following steps:

- **Marked-State Theory.** There are only four rewrite rules in the Java specification, which are all of the form:

  \[ t(T\text{State } tid)env(ENV) \rightarrow t(T\text{State}' tid)env(ENV') \]

  in which \( T\text{State} \) captures the state of a thread and \( env \) encompasses the environment of that thread. Since the left hand sides of the rules do not capture the whole state of the system, we introduce a new variable that captures the rest of the state of the system, let us call it \( Java\text{State} \), and transform each rule of the above form into a rule of the following form:

  \[ \{ t(T\text{State } tid)env(ENV) \ Java\text{State} \} \rightarrow [ t(T\text{State}' tid)env(ENV') \ Java\text{State} ] \]

  We also need a mechanism to mark some threads as enabled and keep the others disabled. We can easily do that in this case by introducing a copy of the operator \( t(\_ ) \), let us call it \( dt(\_ ) \) that represents the disabled thread. Since the operator \( t(\_ ) \) is used as a constructor (wrapper) for a thread in the left-hand side of all the rewrite rules in the Java specification, threads that are represented by \( dt(\_ ) \) will not be rewritten hence these threads are disabled.

- **Customizing the POR module.** There are three tasks to customize the POR module for the Java language:

  - **Extracting Rules.** Four equations in this section, one per rule, can extract all the enable transitions at each state. Consider the rule

    \[ t(T\text{State } tid)env(ENV) \xrightarrow{\_ } t(T\text{State}' tid)env(ENV') \]

    the corresponding equation is

    \[ \langle TL , t(T\text{State } tid)env(ENV) \ Java\text{State} \rangle = \langle < tid, l, info(T\text{State}, ENV) > TL , \]

    \[ t(T\text{State } tid)env(ENV) \ Java\text{State} \rangle \]

    where \( TL \) is a list of transitions, and \( info \) is a function that extracts the relevant information (variable names, lock names, etc) from the state.

  - **Thread Transitions.** The future transitions of a thread are given by an operation \( Thread\text{Transitions} \) that is treated as a black box in the POR module. The provider of the specification is asked to specify this operator. A conservative static analysis can provide this information; the more precise the analysis, the better potential for reduction by the
POR module. In the case of Java, a very simple conservative static analysis module was used to predict the future transitions of each thread. This analysis would simply return all possible transitions (meaning the four possible rules) if there is any ambiguity as the result of an object dereferencing that may not be resolved statically.

- **Dependence Relation.** Equations of the form

\[
\text{Dependence}(\langle \text{tid}, l, I \rangle, \langle \text{tid}', l', I' \rangle) = \text{true}
\]

define the dependence relation. In case of Java, since there are only four rules, there are four options for the labels \(l\) and \(l'\). The dependence relation can then be customized for different examples to achieve better reductions. An example of a slightly relaxed (compare to the default one for Java) which is used to obtain the reductions in Table 8.4 is for example to make the memory accesses of threads that are not communicating at all independent, so that the model checker does not have to consider interleavings of those irrelevant accesses.

### 8.3.2 The JVM POR Unit

Here, we briefly discuss how the language-dependent parts are defined in Maude for the Java bytecode semantics. The same methodology and steps from the previous section is followed, but in order not to repeat ourselves, we only discuss the language-specific parts at the bytecode level.

**Extracting Transitions.** There are 16 equations, corresponding to the 43 rewrite rules in the semantics of the Java bytecode, which extract all the enabled transitions from a given state. Here is an example of one of these equations which corresponds to the rule for the `monitorenter` instruction (see Figure 7.3):

\[
\text{ceq} << S, \langle T: \text{JavaThread | callStack:([PC, monitorenter, ..., (REF(K) \# OperandStack), ...]) CallStack}, .. > < O: \text{JavaObject|Addr:K, ...}, \text{Lock:Lock(OIL, NoThread, 0) > Ct >> =} << S \{'\text{MONITORENTER, T, noInfo}'}, \langle T: \text{JavaThread | callStack: ([PC, monitorenter, ..., OperandStack, ...]) CallStack}, \text{Status: scheduled, ...} > < O: \text{JavaObject | Addr: K, ...}, \text{Lock:Lock(OIL, NoThread, 0) > Ct >> if S \{'\text{MONITORENTER, T, noInfo}'} /=/ S .}
\]

where \(S\) is the enabled transitions set. The equation says that if in the current state (containing a thread \(T\), an object \(O\), and a context \(Ct\) which captures the rest of the JVM state that is a multiset), \(T\) is ready to execute a `monitorenter` (lock) instruction, and \(O\) is not locked by any other thread, it means that the tuple \{'\text{MONITORENTER, T, noInfo}'\} is an enabled transition, and it is added to the set \(S\) if it is not already in it.
**Dependence Relation.** The dependence relation for Java bytecode is defined based on the following facts: (1) two accesses to the same location are dependent if at least one of them is a write. This is defined through a few equations to cover the access to the instance fields as well as static fields; (2) two lock operations accessing the same lock are dependent. This is defined through a few equations to cover synchronized method calls, the \texttt{monitorenter} instruction, as well as the \texttt{notifyAll} built-in method of Java.

As an example of equations defining the dependence relation we have:

\begin{align*}
&\text{eq Dependence}\{(T, 'PutField, I}, \{T', 'GetField, I')\} = \text{true}. \\
&\text{eq Dependence}\{(T, 'InvokeStatic, C}, \{T', 'InvokeStatic, C) = \text{true}. \\
\end{align*}

which specify that a read and a write to an instance field (first line) are always dependent, and (second line) two synchronized static method calls are dependent if they are locking the same class, $C$. Table \ref{table:optimizations} shows optimizations (in addition to traditional POR) gained by customizing the dependence relation for the dining philosophers example. By considering the fact this example does not use shared memory, we can remove the dependence between the rules modeling memory accesses, which gives us substantial additional time/space reductions.

**Thread Transitions.** As mentioned at the end of Section \ref{section:thread-transitions} to check condition $C'1$, the operation \textit{ThreadTransitions}, which conservatively computes the set of future transitions of a thread, has to be specified by the user. In the case of Java bytecode the idea is to start from the current point in $t_i$ and add all the future instructions (transition steps) of the current method executing, and upon a method call, add in all the instructions (transitions) of the code of that method as well (avoiding repetition). This is conservative, in the sense that in the cases where more than one method can be the potential resolution of a call site, all of them are considered, and also in transitions such as reading/writing a field of an object where the object cannot be resolved until the point of execution, conservatively all possible objects will be considered.

**8.4 Applications of the Method and Experiments**

We have implemented the theory transformation for our generic POR reduction method in a Maude prototype and have used it to build POR units for Java source code and bytecode\footnote{see \url{http://javafan.cs.uiuc.edu} for the Maude code of these units.} for a Promela-like language, and for Maude object-oriented modules. In this section we illustrate how the method was used to formally analyze programs in all these languages, based on their rewriting semantics. We also present some performance figures showing that our generic partial order module,
when instantiated for each language, can result in drastic reductions in the state space of programs in the above languages. All experiments were executed on a PC running Linux with a 2.4GHz processor and 4GB of memory.

<table>
<thead>
<tr>
<th></th>
<th>Promela–SPIN</th>
<th>Maude–Ours</th>
<th>Promela–Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Reduction</td>
<td>States</td>
<td>703</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>0.4s</td>
<td>0.2s</td>
</tr>
<tr>
<td>Reduction</td>
<td>States</td>
<td>114</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>0.06s</td>
<td>0.01s</td>
</tr>
</tbody>
</table>

Table 8.1: Time and Space Reduction Comparisons for Sieve of Eratosthenes.

Table 8.1 presents the reduction comparison of a sieve of Eratosthenes (prime generating program) modeled in the Promela language and in Maude. The first column shows the result for the Promela program model checked with SPIN. The second column contains the result for the same model written in the Maude language, together with our POR unit. The third column shows the result of the same Promela program when it is model checked using the rewriting logic semantics of Promela together with our POR unit in Maude. The overhead of interpreting Promela in this case results in a substantially larger number of states. However, after the POR module is added, the number of states is comparable, and the time only increases by a factor of 5 with respect to SPIN.

Table 8.2 shows the results of time/space reduction for a deadlock-free version of dining philosophers with different number of philosophers in the JVM language. Entries left empty indicate that we could not model check the example on our platform.

Table 8.3 illustrates a dining philosophers program (5 philosophers in JVM) model checked in JavaFAN, where two versions of the dependency relation are compared. In the “basic” version, the dependency relation is the general version (presented in Section 8.4) that holds for all Java programs. The “NotShared” version lifts the dependencies of read/write memory accesses, since we know that the dining philosophers code does not use any shared memory and works merely

<table>
<thead>
<tr>
<th>JVM Program</th>
<th>Reduction</th>
<th>Time</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(5)</td>
<td>No</td>
<td>25.1s</td>
<td>56,212</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>7.3s</td>
<td>3,033</td>
</tr>
<tr>
<td>DP(6)</td>
<td>No</td>
<td>146.2s</td>
<td>623,644</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>30.0s</td>
<td>22,822</td>
</tr>
<tr>
<td>DP(7)</td>
<td>No</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>5m</td>
<td>168,565</td>
</tr>
<tr>
<td>DP(8)</td>
<td>No</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>66m</td>
<td>1,412,908</td>
</tr>
</tbody>
</table>

Table 8.2: Dining Philosophers in the JVM.
based on locks. As shown in Table 8.3, a simple change like this (which means commenting out a few equations in the definition of the dependency relation) can result in a considerably better performance. Table 8.4 shows the same reduction for two Java programs which (because of the nature of the program) do not show substantial reductions under classical POR reduction, but which with this “NotShared” variant can be substantially optimized.

Table 8.5 shows the state-space reduction obtained for various JVM programs when the partial order reduction module is used. The JavaFAN tool reduces the number of states substantially by itself, since it uses the rewrite rules to model only the concurrent parts of Java (see [FMR04] for details). But, the partial order reduction can still add a substantial reduction to that. PL

Table 8.4: Java POR Unit Performance.

is a two stage pipeline, DP is a deadlock-free version of the dining philosophers, RA is NASA’s remote agent benchmark, and SE is a distributed sieve of Eratosthenes. All programs in these experiments, as well as the semantic definitions of the JVM and the Promela-like language and their POR-transformations by our method are available in [Far].

Table 8.5: Partial Order Reduction Results for Various JVM Programs.
8.5 Related Work and Conclusions

8.5.1 Related Work

There are two well-known approaches to attack the state-explosion problem while model checking. The first approach consists of partial order reduction (POR) methods introduced by Peled in [Pe93]. The generic method proposed in this work fits within this approach. Several different variations [God96, God97, Val90, ABH+97, FG05, CGMP99, KLM+98] of the POR approach have been introduced since.

A first class of POR methods—including the stubborn sets method of [Val90], the persistent sets method of [GW91], and the ample sets method of [Pel94]—are based on modifying the search algorithm and applying the reduction dynamically. [FG05] takes the matter even further, and dynamically tracks the interactions between threads based on initially exploring an arbitrary interleaving of them. Details of the reduction heuristic are orthogonal to our method; although we propose two different heuristics in this work, many other heuristics can be implemented with little effort. A second class of POR methods such as the one in [KLM+98] use a static approach, in which all partial order reduction information is computed statically, and then an already reduced model is generated to be model checked.

In the dynamic methods, one has to alter the existing model checker to include the reduction, while static methods suffer from the fact that only a limited amount of information is available at compile time. We believe that our method addresses both problems: it can work with an existing model checker, so it has the advantages of the static methods, but it applies the reduction dynamically and therefore can benefit from the runtime information.

It seems fair to say that current POR-enabled model checkers are mostly language-specific, or, by using for example a static approach such as [KLM+98], achieve only a limited “generality by translation into a common intermediate language”. Tools such as Verisoft [God97] that can monitor and control the execution of programs in different languages for model checking purposes are in practice applied to a limited family of languages and cannot benefit from any optimizations that can potentially be introduced using some sort of static analysis of the program, which is not the case in our method. To the best of our knowledge this work represents the first attempt to develop LTL model checkers with POR capabilities for concurrent languages in a fully generic way using their semantic definitions.

Besides the POR methods, a second state space reduction approach, which could be called transaction-based, consists of more recent techniques that consider various kinds of exclusive access predicates for shared variables specifying some synchronization disciplines [Sto00, FQ03, DHP04]. These
predicates can be used to reduce the search space during the state space explorations. The POR techniques (including the method proposed in this work) are complementary to these other methods. We discussed how our method exploits some ideas from [Sto00] in Section 8.5.1. We strongly believe that the reductions in [FQ03] can be achieved using a very similar method to that presented in this work (see below for more details).

8.5.2 Conclusions

We have presented a general method to make software model checkers with POR capabilities language-independent, so that they can be specialized to any programming language $L$ of interest. Our method is based on a theory transformation of the rewriting logic formal semantics of the given language. The language specialization can be done semi-automatically and with relatively little effort by a tool builder. Language-specific optimizations can also be added, because the heuristic algorithm and the dependence relation are explicit parameters of the theory transformation. Since all POR computations are performed in the transformed theory itself, the method does not require any modifications to the underlying LTL model checker. Our experience evaluating this method in practice for Java, the JVM, a Promela-like language, and Maude itself, indicates that significant state space reductions and time speedups can be gained.

Our method is also generic at the semantic framework level: we have developed it in detail within rewriting logic, but the same idea can be applied within other frameworks, for example SOS. In any such framework, the semantics of $L$ will have a specification $S_L$. We then would obtain the POR capabilities by a suitable theory transformation $S_L \mapsto S_L^{+POR}$.

The current prototype implementation of our method does not support various well-known optimization strategies, but many of these can be incorporated into our framework in a straightforward way. These strategies are often based on assumptions about the structure of the programming language under consideration. Therefore, they belong to the second, language-specific customization phase of our theory transformation, although in some cases they can be applied to entire families of languages. For example, a reduction strategy proposed in [DHP04] for concurrent object oriented software is detecting heap objects that are thread-local to sharpen the dependence relation. All the static/dynamic analysis in [DHP04] that leads to detecting the thread locality is possible in our framework, since we have both the static and dynamic information available. A more extensive experimentation with a broader set of language instantiations and incorporating the above optimizations should be performed in the future.

Another interesting direction for future work is extending our generic method beyond POR to also support what we have called “transaction-based reductions”. Such reductions are complementary to those obtained by POR methods. We conjecture that a similar theory transformation would
allow us to achieve transaction-based reductions in a generic way. The equation (*) in Section 8.2.2 works as a nondeterministic scheduler which in the present method schedules all the threads belonging to the ample set for the next step. In a transaction-based method the role currently played by the MarkAmples operation could instead schedule a single thread $t$, provided $t$ is inside a transaction, and the component $I$ could then be used for the instrumentation predicates.

It has been suggested [FG05] that atomicity-based reduction are orthogonal to partial order reduction techniques. One can use techniques from chapter 4 to first to statically check whether a code block is atomic. If it is, then one can use the scheduling techniques mentioned above to achieve further reductions in the state space of the program. Basically, one first marks all the atomic code blocks in the program statically, and then applies the partial order reduction techniques introduced in this chapter on the marked program considering those blocks as executed atomically. The very modularity of these techniques allows one to add several layers of reduction on top of the original specification, a capability which is rarely present in otherwise.

Furthermore, making the prototype tool more mechanized, and carrying out additional experimentation including more languages and more benchmarks per language are both part of the future plans.
Chapter 9

Conclusions and Future Research Directions

The goal of this research is to investigate programming and reasoning methods which facilitate reliable concurrent software. Towards this goal, we have developed techniques for concurrent software analysis which draw on true concurrency models. With the establishment of the multicore processors and the introduction of language-level thread primitives in languages like Java and C#, plus the advent of distributed web services, concurrency has become commonplace even in application software. The design of concurrent software is notoriously error-prone, due to the nondeterministic interaction among concurrently executing threads which makes the problem of reliability of concurrent software a very relevant problem in today’s research scene.

9.1 Modeling Concurrency

Excellent models of sequential behavior that have evolved during the past thirty do not adequately reflect the nature of a concurrent universe. This has spurred interest in true concurrency, namely, in modeling concurrency in a way that is faithful to all currently understood modes of interaction of system components, particularly those beyond the reach of sequential models. The focus of this work is on using true concurrency models as a basis for software reliability and safety for both static and dynamic analyses.

9.1.1 Static Analysis

We have proposed that causality is an appropriate notion to capture the flow of control in concurrent programs, and consequently, a nice basis for static analyses. Hence, (Mazurkiewicz) Traces and Petri nets, which depict the causal structure of concurrent program in natural way, are good candidates to model concurrent programs. We introduced the control net model [FM06a, FM07], a Petri net-based sound control abstraction for concurrent programs. It captures the control flow in a concurrent program, and gives a translation from programs to Petri nets that explicitly abstracts data and captures the control flow in the program. We believe that it is a good candidate for a
standard notion of control flow graph in the concurrent setting. Defining and solving two important and relevant static analysis problems, namely atomicity [FM06a] and dataflow analyses [FM07], based on the control net model (and its partial order semantics) corroborates the pertinence of causality as the right notion for static analyses of concurrent programs.

9.1.2 Dynamic Analysis

True concurrency model of rewriting logic provides an excellent framework for modeling concurrent systems and languages. We have developed a set of language-independent techniques which exploit the advantages of this model in order to build formal analysis tools for concurrent software systems. Moreover, the framework supports a seamless combination of different techniques used for ensuring software reliability including model checking, theorem proving. [FM06c] demonstrates an instance of such combination. Note that, as already mentioned in Chapter 1, Petri net models and their true concurrency semantics are a special case of rewriting logic models and their true concurrency semantics. Therefore, the true concurrency models we use for both static and dynamic analysis are all encompassed within the common semantic framework of rewriting logic.

9.2 Language-independent Techniques for Software Analysis

We have developed JavaFAN [FMR04, FCMR04], a tool for formally analysis of Java programs (interpretation, LTL model checking, searching for safety violation patterns) at both bytecode and source code level, based on definitions of the Java language and the Java Virtual Machine (JVM), which is our own contribution. This work was the first in a series of contributions to introduce the use of language definitions (more specifically rewriting logic semantics in Maude language) as a basis for generic program analysis tools. This method of developing program analysis tools has several advantages over conventional methods, including generality, relative ease of development, modularity, and potential for formal verification of compilers/interpreters. JavaFAN demonstrates competitive performance compared to other (non-generic) Java analysis tools, which supports the effectiveness of these ideas. These experiments include the discovery of the deadlock bug in NASA’s remote agent component in fraction of a second time.

To improve the performance of these generic tools, we proposed a new approach to build language-independent partial order reduction (POR) [FM06b] in the above framework. In this approach, Getting the POR capabilities does not require making any changes to the underlying model checking algorithms. The generic POR unit can be customized for any language \( L \), under very minimal assumptions, with minimal interface, and with relatively little effort. Our experiments with the several languages including Java, JVM, Promela, and Maude indicate that significant state space
reductions and time speedups can be gained for tools generated this way. In fact, comparisons with POR unit of the SPIN model checker for programs in Promela show that our generic POR unit works as well as (and in some cases slightly better than) the SPIN’s specific POR unit designed for Promela. We plan to work on the advancement of the existing tools by rigorous experimentation on real programs on the one hand, and by investigating additional optimizations sculpted specifically for this framework such as abstraction techniques on the other hand. Moreover, the very modular nature of this framework suggests enormous potential for the application of several orthogonal optimization techniques that can be combined in the same manner.

As a generalization of the above reduction technique, we have introduced a new state space reduction technique based on coherence, in which concurrent systems (not just languages as was before) are formally specified as rewrite theories [FM06c]. The fact that rewriting logic has been shown to be a very general and expressive semantic framework to specify concurrent systems makes our proposed state space reduction technique applicable to a very wide range of concurrent systems. Also, achieving a state space reduction typically requires discharging proof obligations to verify that the reduction is correct. In this regard, the fact that the state space is itself axiomatized by an equational theory \((\Sigma, E)\) makes the tool-assisted discharging of such proof obligations using equational theorem proving techniques and tools much easier than if a non-logical specification formalism had been used instead. Experiments with a variety of distributed algorithms specified and checked for correctness with this approach suggest that this method can introduce substantial space and time reductions. Besides investigating weaker conditions for the reduction to achieve a better one, We plan to develop an mechanized reduction tool in the near future, when more sophisticated confluence/coherence/termination checkers for Maude will become available, specially those which work modulo associativity, commutativity, and identity axioms.

9.3 Notions of Atomicity

Atomicity as a correctness criterion for concurrent programs has been widely studied during the past few years. The notion of atomicity has been inspired by and borrowed from the concept of serializability in database systems. We formulated causal atomicity as a new notion of atomicity based on causality which is provably a slightly weaker (in the logical sense) notion of atomicity than serializability, and that in our view is more appropriate for programming languages, as it captures the way people write concurrent programs. In fact, one can show that Lipton’s transactions are the strongest local property that imply causal atomicity (a global correctness property), which brings all these different notions of atomicity in the literature nicely together. It would be interesting to investigate weaker notions of atomicity (such as purity) in the causal setting that can capture the programming patterns in a more precise way and, consequently, reduce the number of falsely reported errors in programs. These results can be useful in database concurrency control as well.
as in software reliability and program verification.

One can view atomicity in a whole different light as a state space reduction technique to facilitate model checking. If a code block is established as atomic, then it can be treated as one transition during the model checking. One can use techniques similar to those presented in Chapter 8 to incorporate this knowledge, gained from static analysis for atomicity, to add another layer of reduction to the generic POR unit introduced there. It has been argued [FG05] that these two different reductions are orthogonal, so that many programs can benefit from both during the model checking process.

9.4 Static Analysis for Concurrency

Static analysis is a natural solution to overcome the high complexity of software verification and analysis. In fact, one of few commercial successes in the area of software verification is the case of a static analysis suite, which has been incorporated into Microsoft code generation routine and is now used by all programmers there. Checking for causal atomicity [FM06a] was the first problem that we investigated in the context of static analysis of concurrent programs. We introduced efficient algorithms to check causal atomicity by enriching the control net with colors and reducing the problem of checking for causal atomicity to a coverability query on the corresponding colored Petri net. The coverability queries are checked by the PEP tool, which works based on net unfoldings (the partial order semantics for Petri nets), and hence the checks are performed very efficiently. In a similar framework, we formulated the causal concurrent dataflow (CCD) framework [FM07] based on causal flows in a program which is the first (up to our knowledge) definition of concurrent dataflow analyses. This work offers algorithmic solutions to causal flow analyses, when the domain of flow facts is a finite set $D$, by exploring the partially-ordered runs of the program as opposed to its interleaved executions. We came up with provably efficient algorithms for the class of distributive CCD problems which work fast in early experiments. We believe the causal setting and partial order semantics are very appropriate for concurrent static analyses problems, and would like to investigate other static analysis problems from this point of view. As a concrete example, we would like to use the control net and net unfoldings as a basis to develop an abstract interpretation theory for concurrent programs, the same way that the control flow graph are used for this purpose in the sequential setting.

We would like to investigate how incremental and compositional techniques can be used to scale up the algorithms for detecting the causal atomicity, computing the dataflow solutions and other static analyses that one may define within this framework. This can help apply these algorithms to large real programs. Moreover, it can be a big step towards the ultimate of software engineering goal of combining design and verification of software.
References


[AMP00] Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness condi-

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, techniques,

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. Concurrency control
and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc.,

[BHMM02] Christiano Braga, Edward Hermann Haeusler, José Meseguer, and Peter D. Mosses.
Mapping modular SOS to rewriting logic. In Michael Leuschel, editor, 12th Inter-
national Workshop, LOPSTR 2002, Madrid, Spain, volume 2664 of Lecture Notes in


[BKKM02] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting

[BN99] F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University Press,
1999.

[BvdP02] Stefan Blom and Jaco van de Pol. State space reduction by proving confluence. In CAV,


[CC96] A. T. Chamillard and Lori A. Clarke. Improving the accuracy of petri net-based anal-
international symposium on Software testing and analysis, pages 24–38, New York,
NY, USA, 1996. ACM Press.

[CCK+06] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-
passing C programs with recursive calls. In TACAS, volume 3920 of LNCS, pages


Maude Specification of MOP

mod ADDRESS is
    protecting NAT .
    protecting BOOL .

    sort Addr .
    subsort Nat < Addr .
    op noAddr : -> Addr .

    sort PCUPD .
    op spec : -> PCUPD .
    op miscal : -> PCUPD .
    op bot : -> PCUPD .
    op top : -> PCUPD .

    sort WB .
    op bott : -> WB .
    op topp : -> WB .

endm

mod MEMORY is
    protecting ADDRESS .

    sort Word .
    subsort Nat < Word .
    subsort Word < Addr .

    sort Mem .
op noWord : -> Word.

op nulMem : -> Mem.
op [_,_] : Addr Word -> Mem [ctor].
op _-_ : Mem Mem -> Mem [ctor assoc id: nulMem].

op _[_,_] : Mem Addr -> Word.


eq (Mem [N, W Mem'])[N] = W.
eq Mem[N] = noWord [owise].
eq Mem[noAddr] = noWord.

op _[_,<-_,_] : Mem Addr Word -> Mem.

eq (Mem [N, W Mem']) [N <- W'] = Mem [N, W'] Mem'.
eq Mem[N <- W'] = Mem [N, W'] [owise].

endm

mod INST is
    protecting MEMORY.

    sorts Opcode Opcode1 Opcode2 Opcode3 Opcode12.
    subsort Opcode3 < Opcode.
    subsort Opcode1 Opcode2 < Opcode12 < Opcode.
    sort Inst.

    op add : -> Opcode1.
    op sub : -> Opcode1.
    op mul : -> Opcode1.
    op addi : -> Opcode2.
    op subi : -> Opcode2.
    op muti : -> Opcode2.
    op beqz : -> Opcode3.
    op bnez : -> Opcode3.
op j : -> Opcode3.
op ld : -> Opcode.
op st : -> Opcode.
op noOpcode : -> Opcode.

op _[_,_,_,_] : Opcode Addr Addr Addr Word -> Inst [ctor].

var O : Opcode. vars D S1 S2 : Addr. var W : Word.

op _->opc : Inst -> Opcode.
op _->des : Inst -> Addr.
op _->src1 : Inst -> Addr.
op _->src2 : Inst -> Addr.
op _->imm : Inst -> Word.

eq (O[D, S1, S2, W]) ->opc = O.
eq (O[D, S1, S2, W]) ->des = D.
eq (O[D, S1, S2, W]) ->src1 = S1.
eq (O[D, S1, S2, W]) ->src2 = S2.
eq (O[D, S1, S2, W]) ->imm = W.

op noInst : -> Inst.

sort IMem.

op nulIMem : -> IMem.
op [_,_] : Addr Inst -> IMem [ctor].
op _-_ : IMem IMem -> IMem [ctor assoc id: nulIMem].

op _[_,_] : IMem Addr -> Inst.


eq IMem[N] = noInst [owise].
eq IMem[noAddr] = noInst.
mod MOP-STATE is
  protecting INST .
  protecting INT .

sort Parcel .
sort ParcelQueue .

op \{_,_,_,_,_,_,_,_,_,_,_,_,_,_,_\} : Inst Word Bool Addr Opcode Addr

op noParcel : -> Parcel .
op noPCUPD : -> PCUPD .
op emptyParcel : -> Parcel .

op noQueue : -> ParcelQueue [ctor] .
op \[_,_\] : Int Parcel -> ParcelQueue [ctor] .
op \_\_ : ParcelQueue ParcelQueue ->
    ParcelQueue [ctor assoc id: noQueue] .

op \_[_] : ParcelQueue Int -> Parcel .
op head : ParcelQueue -> Int .
op tail : ParcelQueue -> Int .


eq (PQ [I, P] PQ')[I] = P .
eq PQ[I] = noParcel [owise] .

op valid : Int Int Int -> Bool .
eq valid(I, J, K) = if ((I >= J) and (I <= K))
then true else false fi.

var 0 : Opcode .

op _->pcupd : Parcel -> PCUPD .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->pcupd = U .

op _->src1 : Parcel -> Addr .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->src1 = A4 .

op _->src2 : Parcel -> Addr .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->src2 = A5 .

op _->des : Parcel -> Addr .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->des = A3 .

op _->npc : Parcel -> Addr .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->npc = A2 .

op _->data1 : Parcel -> Word .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->data1 = W2 .

op _->data2 : Parcel -> Word .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3, W4,
W5, WB, U}->data2 = W3 .

op _->res : Parcel -> Word .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->res = W4 .

op _->wb : Parcel -> WB .
eq {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3,
W4, W5, WB, U}->wb = WB .
op _->inst : Parcel -> Inst .

op _->opc : Parcel -> Opcode .

op _->pc : Parcel -> Addr .

op _->npc : Parcel -> Addr .

sort mopState .

op \[\_,\_,\_,\_,\_,\_,\_\] : Addr Mem IMem Mem ParcelQueue
Int Int -> mopState [ctor] .

var A : Addr .

op mrw : ParcelQueue Int Addr -> Int .

eq mrw(noQueue, J, A) = -1 .
eq mrw(PQ \[I, P\], J, A) = mrw(PQ, J, A) [owise] .

op alu-reg : Parcel -> Bool .
op alu-imm : Parcel -> Bool .

var O1 : Opcode1 .

eq alu-reg({I1, W1, B, A1, O1, A2, A3, A, A5, W2, W3, W4, W5, WB, U}) = true .
eq alu-reg(P) = false [owise] .
eq alu-imm({I1, W1, B, A1, 01, A2, A3, A, A5, W2, W3, W4,  
     W5, WB, U}) = false .
eq alu-imm(P) = true [owise] .

op alu : Opcode Word Word -> Word .

op fit : ParcelQueue Int -> Bool .
op valid-data-upto : ParcelQueue Int -> Bool .

ceq fit(PQ [I, P] [I + 1, P'], J) = false if ((I >= J) and  
     (P' ->pc /= P ->npc) or (P' ->pc == noAddr)) .
eq fit(PQ, J) = true [owise] .

ceq valid-data-upto(PQ [I, P] PQ', J) = false  
     if ((I > J) and (P ->data1 == noWord)) .
ceq valid-data-upto(PQ [I, {I1, W1, B, A1, 01, A2, A3, A, A5, W2, W3,  
        W4, W5, WB, U}] PQ', J) = false if ((I >= J) and (W3 == noWord)) .
eq valid-data-upto(PQ, J) = true [owise] .

op mrma : ParcelQueue Int -> Int .
ceq mrma(PQ [I, {I1, W1, B, A1, ld, A2, A3, A, A5, W2, W3, W4,  
      W5, WB, U}], J) = I if I >= J .
ceq mrma(PQ [I, {I1, W1, B, A1, st, A2, A3, A, A5, W2, W3, W4,  
      W5, WB, U}], J) = I if I >= J .
eq mrma(noQueue, J) = -1 .
eq mrma(PQ [I, P], J) = mrma(PQ, J) [owise] .

op ma-complete : Parcel -> Bool .
ceq ma-complete({I1, W1, B, A1, ld, A2, A3, A, A5, W2, W3, W4,  
      W5, WB, U}) = true if W4 /= noWord .
ceq ma-complete({I1, W1, B, A1, st, A2, A3, A, A5, W2, W3, W4,  
      W5, WB, U}) = true if WB = topp .
eq ma-complete(P) = false [owise] .

sort Bool .
sort Bool < Bool .
op noBool : -> Bool .
mod MOP-TRANSITIONS is
  protecting MOP-STATE .

  var RF : Mem . vars PQ PQ' : ParcelQueue .
  vars H T J : Int . var P : Parcel .

  var B : Bool . vars A1 A2 A3 A4 A5 : Addr . var WB : WB .
  var U : PCUPD . var O : Opcode .
  var B' : Bool .

  --- fetch

  crl [PC, M, IM, RF, PQ, H, T] =>
  [PC, M, IM, RF,
   PQ [T + 1, {IM[PC], noWord, noBool, PC, noOpcode, noAddr,
   noAddr, noAddr, noAddr, noWord, noWord, noWord, noWord,
   bott, bot}]], H, T + 1 ] if H == T or (PQ[T]) ->pcupd == spec
  or (PQ[T]) ->pcupd == top .

  --- decode j

  crl [PC, M, IM, RF, PQ
  [J, {I1, W1, B, A1, 0, A2, A3, A4, A5, W2, W3, W4, W5, WB, U}] PQ'
  , H, T] => [PC, M, IM, RF, PQ [J, {I1, I1 ->imm, B, A1, I1 ->opc,
  A2, I1 ->des, I1 ->src1, I1 ->src2, W2, W3, W4, W5, WB, U}] PQ',
  H, T] if I1 /= noInst and valid(J, H, T) .

  --- dataRF j

  crl [PC, M, IM, RF, PQ
  [J, {I1, W1, B, A1, 0, A2, A3, A4, A5, noWord, W3, W4, W5, WB, U}] PQ’
  , H, T] => [PC, M, IM, RF, PQ [J, {I1, W1, B, A1, 0, A2, A3,
  A4, A5, RF[A4], W3, W4, W5, WB, U}] PQ’ , H, T] if valid(J, H, T)
and I1 =/= noInst and A4 =/= noAddr and mrw(PQ, H, A4) == -1 .

crl [PC, M, IM, RF, PQ

[J, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, noWord, W4, W5, WB, U}]
PQ', H, T] => [PC, M, IM, RF, PQ [J, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, RF[A5], W4, W5, WB, U}] PQ', H, T] if valid(J, H, T) and I1 =/= noInst and A4 =/= noAddr and mrw(PQ, H, A4) == -1 .

--- data1_forward j

crl [PC, M, IM, RF, PQ

[J, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, noWord, W4, W5, WB, U}]

crl [PC, M, IM, RF, PQ

[J, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, noWord, W4, W5, WB, U}]
PQ', H, T] => [PC, M, IM, RF, PQ [J, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, ((PQ[mrw(PQ, H, A5)]) ->res), W4, W5, WB, U}] PQ', H, T] if valid(J, H, T) and I1 =/= noInst and A5 =/= noAddr and mrw(PQ, H, A5) == -1 .

var O1 : Opcode1 .

var O2 : Opcode2 .

--- result j

crl [PC, M, IM, RF, PQ


crl [PC, M, IM, RF, PQ

\[ W2 \neq \text{noWord} \text{ and } W1 \neq \text{noWord}. \]

--- mem-addr j

\[
\text{crl } [\text{PC, M, IM, RF, PQ} \\
[J, \{I1, W1, B, A1, ld, A2, A3, A4, A5, W2, W3, \text{noWord}, W5, WB, U\}] PQ', H, T] \Rightarrow [\text{PC, M, IM, RF, PQ} [J, \{I1, W1, B, A1, ld, A2, A3, A4, A5, W2, W3, \text{noWord}, W2 + W1, WB, U\}] PQ', H, T] \text{ if valid}(J, H, T) \text{ and } W2 \neq \text{noWord}. \\
\]

\[
\text{crl } [\text{PC, M, IM, RF, PQ} \\
\text{ if valid}(J, H, T) \text{ and } W4 = \text{noWord} \text{ and } W3 \neq \text{noWord}. \\
\]

\[ \text{var O12 : Opcode12.} \]

--- write-back j

\[
\text{crl } [\text{PC, M, IM, RF, PQ} \\
[J, \{I1, W1, B, A1, O12, A2, A3, A4, A5, W2, W3, W4, W5, bott, U\}] PQ', H, T] \Rightarrow [\text{PC, M, IM, RF[A3 <- W4]}, PQ \\
[J, \{I1, W1, B, A1, O12, A2, A3, A4, A5, W2, W3, W4, W5, topp, U\}] PQ', H, T] \text{ if valid}(J, H, T) \text{ and } W4 = \text{noWord} \\
\text{ and mrw}(PQ, H, A3) = -1 \text{ and fit}(PQ [J, \{I1, W1, B, A1, O12, A2, A3, A4, A5, W2, W3, W4, W5, topp, U\}], H) \text{ and valid-data-upto}(PQ, H). \\
\]

\[
\text{crl } [\text{PC, M, IM, RF, PQ} \\
[J, \{I1, W1, B, A1, ld, A2, A3, A4, A5, W2, W3, W4, W5, bott, U\}] PQ', H, T] \Rightarrow [\text{PC, M, IM, RF[A3 <- W4]}, PQ \\
[J, \{I1, W1, B, A1, ld, A2, A3, A4, A5, W2, W3, W4, W5, topp, U\}] PQ', H, T] \text{ if valid}(J, H, T) \text{ and } W4 = \text{noWord} \\
\text{ and mrw}(PQ, H, A3) = -1 \text{ and fit}(PQ [J, \{I1, W1, B, A1, ld, A2, A3, A4, A5, W2, W3, W4, W5, topp, U\}], H) \text{ and valid-data-upto}(PQ, H). \\
\]

--- load j
crl \[PC, M, IM, RF, PQ \]

--- store j

crl \[PC, M, IM, RF, PQ \]

var O3 : Opcode3 .

--- branch-target j

crl \[PC, M, IM, RF, PQ \]

--- branch-taken j

crl \[PC, M, IM, RF, PQ \]

crl \[PC, M, IM, RF, PQ \]
crl [PC, M, IM, RF, PQ


--- next-pc-branch j
crl [PC, M, IM, RF, PQ


and W4 =/= noWord .

--- next-pc-nonbranch j
crl [PC, M, IM, RF, PQ


and (O =/= beqz and O =/= j) .

--- pc-update
crl [PC, M, IM, RF, PQ

[T, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3, W4, W5, WB, U}]

--- speculate
crl [PC, M, IM, RF, PQ

--- prediction-ok

crl [PC, M, IM, RF, PQ

   [J, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3, W4, W5, WB, spec}]
PQ', H, T] => [PC, M, IM, RF, PQ [J, {I1, W1, B, A1, O, A2, A3,
A4, A5, W2, W3, W4, W5, WB, top}] PQ', H, T] if valid(J, H, T)
and valid((J+1), H, T) and A2 =/= noAddr and A2 == (PQ'[1])->pc .

--- squash

crl [PC, M, IM, RF, PQ

   [J, {I1, W1, B, A1, O, A2, A3, A4, A5, W2, W3, W4, W5, WB, spec}]
PQ', H, T] => [PC, M, IM, RF, PQ [J, {I1, W1, B, A1, O, A2, A3,
and valid((J+1), H, T) and A2 =/= noAddr and A2 =/= (PQ'[1])->pc .

--- retire

crl [PC, M, IM, RF, PQ, H, T] => [PC, M, IM, RF, PQ, H + 1, T]
if T >= H and (PQ[H])->pcupd == top and ((PQ[H])->opc =/= beqz
   and (PQ[H]) ->opc =/= j) or ( (PQ[H]) ->wb == topp)) .

endm
Author’s Biography

Azadeh Farzan was born on September 20, 1987 in Tehran, Iran. She graduated from Sharif University of Technology at Tehran in February 2000 with a B.S. degree in computer engineering. She joined the computer science program at the University of Illinois at Urbana-Champaign in August 2000 and completed her Ph.D. in January 2007. Following the completion of her Ph.D., Azadeh will begin working as a post-doctoral researcher at the School of Computing at Carnegie-Mellon University, Pittsburgh, Pennsylvania.