Program Transformations for Verifying Parameterized Systems

A DISSERTATION PRESENTED

BY

ABHIK ROYCHOUDHURY

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

STATE UNIVERSITY OF NEW YORK

AT STONY BROOK

December 2000

Abstract of the Dissertation

Program Transformations for Verifying Parameterized Systems

by
Abhik Roychoudhury

Doctor of Philosphy
in
Computer Science
State University of New York at Stony Brook
2001

Formal verification of reactive concurrent systems is important since many hardware and software components of our computing environment can be modeled as reactive concurrent systems. Algorithmic techniques for verifying concurrent systems such as model checking can be applied to only finite state systems. This dissertation investigates the verification of a common class of infinite state systems, namely parameterized systems. Such systems are parameterized by the number of component processes, for example an n process token ring for any n. Verifying the entire infinite family represented by a parameterized system lies beyond the reach of traditional model checking. On the other hand, deductive techniques to verify infinite state systems often require substantial user guidance.

The goal of this dissertation is to integrate algorithmic and deductive techniques for automating proofs of temporal properties of parameterized systems. Here, the parameterized system to be verified and the temporal property are encoded together as a logic program. The problem of verifying the temporal property is then reduced to the problem of determining equivalence of predicates in this logic program. These predicate equivalences are established by transforming the program such that the semantic equivalence of the predicates can be inferred from the structure of their clauses in the transformed program. For transforming the predicates, we use the well-established unfold/fold transformations of logic programs. Unfolding represents a step of resolution and can be used to evaluate the base case and the finite part of the induction step in an induction proof. Folding and other transformations represent deductive reasoning and can be used to recognize the induction hypothesis. Together these transformations are used to construct induction proofs of temporal properties.

The first part of the dissertation develops new, more powerful unfold/fold transformation rules for this purpose. To ensure preservation of program semantics, existing unfold/fold transformation rules impose severe restrictions on program syntax for a folding step to be applicable. This renders them unsuitable for constructing proofs of temporal properties. This dissertation develops unfold/fold transformation rules where the applicability of any rule is not restricted by program syntax. Instead, book-keeping is performed at every transformation step and this book-keeping is used to restrict the applicability of the transformation rules, thereby ensuring that the transformations preserve program semantics. The application of each transformation rule is fully automated.

The second part of the dissertation presents strategies to guide the application of the transformation rules for verifying parameterized systems. The strategies are used to verify parameterized systems of different control structures including chain, ring, tree and star networks. These strategies allow interleaving of algorithmic and deductive steps in a verification proof. Furthermore, since deductive steps are applied lazily, model checking emerges as a special case of the proof technique. The transformation rules and strategies have been implemented to yield an automatic and programmable first order theorem prover for parameterized systems. Case studies include cache coherence protocols and the Java Meta-Locking protocol from Sun Microsystems. The program transformation based prover has been used to automatically prove various safety properties of these protocols.

To the memory of those who inspired me to pursue research

My late uncle Prof. A.P. Chatterjee

My late grandfather Prof. T.C. Rai Chaudhuri

Contents

List of Tables **ix**

List of Figures **x**

Acknowledgements **xi**

1 Introduction **1**

2 Overview **8**

2.1 Logic Programming Preliminaries 8

2.2 Overall Approach 11

2.2.1 Encoding the Verification Problem 11

2.2.2 Proofs by Program Transformations 13

2.2.3 Unfold/Fold Program Transformations 15

2.3 Related Work 19

2.3.1 Related Work on Logic Program Transformations 19

2.3.2 Related Work on Parameterized System Verification 21

2.4 Notational Conventions 27

3 Transformations for Definite Logic Programs **28**

3.1 Background 28

3.2 A Parameterized Transformation Framework 30

3.2.1 Unfolding and Folding 30

3.2.2 Measure-Consistent Proofs and Total Correctness 32

3.3 Constructing Concrete Unfold/Fold Systems 40

3.3.1 Existing Unfold/fold Systems 41

3.3.2 SCOUT— A New Unfold/Fold System 43

3.4 Goal Replacement 46

3.5 Discussions 49

4 Transformations for Normal Logic Programs 51
4.1 Background 51
4.2 The Transformations 53
4.3 Proof of Correctness 55
4.3.1 Semantic Kernel of a Program 56
4.3.2 Preserving the Semantic Kernel 57
4.4 Discussions 66

5 Proofs by Program Transformations 69
5.1 A Proof System for Predicate Equivalences 70
5.2 Automated Instances of Proof Rules 74
5.3 An Algorithmic Framework for Strategies 80

6 A Prover for Verifying Parameterized Protocols 86
6.1 System and Property Specification 87
6.2 Constructing Concrete Strategies 89
6.2.1 Pruning the Transformation Search Space 89
6.2.2 Controlling Algorithmic Steps 92
6.2.3 Controlling Deductive Steps 95
6.3 Example Liveness and Safety Proofs 99
6.3.1 Liveness in Unidirectional Chains 100
6.3.2 Mutual Exclusion in Token Rings 103

7 Applications and Case Studies 107
7.1 Mutual Exclusion of Java Meta-Lock 108
7.1.1 Modeling the Protocol 108
7.1.2 Verifying Mutual Exclusion 113
7.2 Verifying Cache Coherence Protocols 115
7.2.1 A Simple Example 117
7.2.2 Modeling Broadcasts 118
7.2.3 Extending to Tree Based Protocols 121
7.3 Experimental Results 123

8 Discussions **126**
8.1 Summary of the Dissertation 126
8.2 Future Work 128
8.2.1 Verifying Nested Fixed Point Properties 128
8.2.2 Verifying Parameterized Real-time Systems 129
8.2.3 Explaining Transformation Proof Runs 129
8.3 Concluding Remarks 130

List of Tables

1 Chapters of the dissertation 6
2 Proof System for showing Predicate equivalences 71
3 Statistics of mutual exclusion proof of Java Meta-Lock 115
4 Summary of protocol verification timings 124

List of Figures

1 Dependencies among chapters of the dissertation 7
2 Example: Liveness in an unbounded length shift register 12
3 Fragment of Transformed Program for Shift Register Example 13
4 Example of a correct unfold/fold transformation sequence. 15
5 Example of an incorrect unfold/fold transformation sequence. 15
6 Schema for Unfolding Transformation 16
7 Schema for Folding Transformation 16
8 An unfold/fold transformation sequence of normal logic programs . . 52
9 Program with syntactically equivalent predicates. 75
10 Goal replacements to facilitate other transformations. 81
11 Algorithmic framework for equivalence tableau construction. 82
12 Choosing conditions for Conditional equivalence 98
13 Fragments of Liveness and Safety Proofs of Chains and Rings 101
14 Mutual exclusion in a n-process token ring 104
15 Labeled Transition System of (a) any thread (b) hand-off process . . 112
16 Labeled Transition system of any processor in a simple synchronization based cache coherence protocol 118
17 Transition relation of any process in MESI protocol 120
18 Synchronization actions in a tree network of processes 122

Acknowledgements

I need to thank many people for their direct and indirect help over the years. My sincere acknowledgments to all those whom I mention, and sincere apologies to anybody whom I might have forgotten.

My advisor Prof. I.V. Ramakrishnan introduced me to the area of logic programming and helped me to stay on track with his valuable advice and guidance. He has steadily encouraged me to keep the larger picture in mind while solving a specific research problem. I am also indebted to my co-advisor Prof. C.R. Ramakrishnan. His insistence on taking the time to study a problem thoroughly, and explaining technical content clearly has been of immense help. Thanks are due to Prof. Scott Smolka who introduced me to the area of verification. His comments and suggestions on the research work leading to this dissertation have been extremely helpful. Dr. K. Narayan Kumar has truly been a great friend and co-worker. I have not only enjoyed my numerous technical and non-technical interactions with him, but also learnt a lot from them.

The members of my dissertation committee as well as my previous committees have contributed to this dissertation with their comments and suggestions. I would like to thank Dr. Kedar Namjoshi for taking time out of his schedule and agreeing to be my external examiner. His comments on a comparison of the work with theorem proving have been very helpful. Thanks are due to Prof. David S. Warren and Prof. Rance Cleaveland for their valuable feedback. Moreover, Prof. Warren's keen insight and active grasp of programming language research has been truly inspirational.

Throughout my graduate studies I have got the opportunity to interact with other researchers outside Stony Brook. In fact, discussions with them have helped me a lot and significantly improved the content of this dissertation. This includes Prof. Giorgio Delzanno (Universita di Genova), Prof. Sandro Etalle (Universiteit Maastricht), Prof. Amy Felty (University of Ottawa), Prof. Douglas Howe (Carleton University), Prof. Joxan Jaffar (National University of Singapore), Prof. Somesh Jha (University of

Wisconsin Madison), Prof. Claude Kirchner (LORIA and INRIA Nancy), Dr. R.P. Kurshan (Bell Labs), Prof. Madhavan Mukund (Chennai Mathematical Institute), Dr. Kedar Namjoshi (Bell Labs), Prof. Alberto Pettorossi (Universita di Roma Tor Vergata), Prof. David Sands (Chalmers University of Technology and University of Goteborg), Prof. Rakesh Verma (University of Houston) and Prof. Roland Yap (National University of Singapore).

The help and support of the administrative staff of the department, in particular Betty Knittweis and Kathy Germana, is sincerely acknowledged.

The friends inside and outside the university have been a great source of joy and encouragement. It has been a wonderful experience to spend time with them: Samik Basu, Atrayee Bhattacharjee, Hasan Davulcu, Jihad El-sana, Vivek Nagar, April and Robert Pokorny, Abhra Roy, Jim and Terry Sarver, Bikram Sengupta and Harish Seshadri to name a few. My parents and family have been very supportive and encouraging throughout my graduate studies, including times when I felt down. Their help and support has been pivotal and I cannot thank them enough.

My wife Tulika has been a pillar of strength. It has been very difficult for us at times, since we were both going through the doctoral program simultaneously. Sometimes there were paper deadlines which clashed, but she was never one to complain. The hardest part has been the last few months, when we were both writing our dissertations. In spite of all the difficulties, Tulika has always been accommodative and understanding. Whatever be the odds, her smiling face kept me going.

Finally, I gratefully acknowledge the moral support and inspiration I have received from two members of my family : my late uncle Prof. A.P. Chatterjee and my late grandfather Prof. T.C. Rai Chaudhuri. Over the years, they have moulded and inspired me primarily by setting their own examples. My uncle in particular has been a great source of strength, support and rational thought in my formative years. I humbly dedicate this dissertation to their memory.

This work was partially supported by NSF grants CCR-9711386, CCR-9876242 and EIA-9705998.

Chapter 1

Introduction

Many hardware and software components of our everyday computing environment can be modeled as a *reactive concurrent program*. These include hardware controllers, operating systems, network protocols, and distributed applications *e.g.* air traffic control system. Intuitively, a reactive concurrent program is a collection of nonterminating processes which run concurrently, and communicate with each other as well as an external environment to perform a common task. Proving correctness of such a program involves showing that it displays some desired behavior. Formally proving correctness of such systems has been a topic of intense research for the past two decades, leading to the birth of successful techniques like *model checking* [CES86, LP85, QS82].

Formal verification of reactive programs involves: *(i)* constructing the "specification" *i.e.* the description of the desired behavior(s) of the program, *(ii)* constructing the "implementation" *i.e.* the formal description of the reactive system being verified, and *(iii)* formally proving that the implementation satisfies the specification. There are several formalisms for expressing specification and implementation: temporal logic [MP91], ω automata [Kur94], process calculi such as CCS [Mil89] and CSP [Hoa85]. Essentially all of these formalisms can be viewed as languages for expressing behaviors of transition systems.

Given appropriate formalisms for expressing the specification and implementation, we then need a proof system for establishing that a given implementation satisfies a given specification. A proof system is essentially a collection of proof rules corresponding to the operators of the languages chosen for expressing the specification and the implementation. Given a proof system and a proof obligation (*i.e.* a given implementation and specification), one needs to construct a proof tree by repeated application of the rules to the proof obligation. In general, this proof tree construction

is undecidable [MP91].

However, for *finite state* concurrent programs, this can be achieved algorithmically by searching *the* finite model of the implementation, *i.e.* by searching the states of the finite state transition system represented by the concurrent program. This is the basic idea behind model checking. Model checking [CES86, LP85, QS82] is an *automated* formal verification technique for proving properties of finite state concurrent programs. Here the specification is typically provided as a temporal logic formula. The implementation is often expressed using a process calculus, which is translated to a finite state transition system. Verifying the truth of the temporal formula is accomplished by traversing the states of this transition system based on the structure of the temporal formula. If the formula is true, then the search succeeds; otherwise the search fails and yields a counterexample.

The problem addressed: The applicability of model checking is inherently restricted to finite state concurrent systems. Many of the verification tasks one would like to conduct however deal with infinite state systems. In particular, we often need to verify "parameterized" systems such as an n-bit adder or an n process token ring for any n. Intuitively, a parameterized system is an infinite family of finite state systems parameterized by a recursively defined type *e.g.* $\mathbb{N}$. Thus an n bit adder is a parameterized system, the parameter in question being $n \in \mathbb{N}$, the width of the adder circuit. Verification of distributed algorithms can be naturally cast as verifying parameterized systems, the parameter being the number of processes. For example, consider a distributed algorithm where n users share a resource and follow some protocol to ensure mutually exclusive access. Using model checking, we can verify mutual exclusion for only finite instances of the algorithm, *i.e.* for $n = 3$, $n = 4, \ldots$, but never the entire algorithm for any n. The verification of parameterized systems lies beyond the reach of traditional model checkers: the representations and the model-checking algorithms that manipulate these representations are designed to work on finite state systems and it is not at all trivial (or even possible) to adapt them to verify parameterized systems.

In general, automated verification of parameterized systems has been shown to be undecidable [AK86]. Thus, verification of parameterized systems is often accomplished via *theorem proving* [BM90, OSR92], i.e. mechanically checking the steps of a human proof using a deductive system. Even with substantial help from the deductive system in dispensing routine parts of the proof, such theorem proving efforts require

considerable user guidance. Alternatively, one can identify subclasses of parameterized systems for which verification is decidable [GS92, Nam98]. Using this approach meaningful subclasses have been identified, such as token rings of similar processes [EN95] and classes of parameterized synchronous systems [EN96].

The approach taken in this dissertation: Recall that a parameterized system represents an infinite family parameterized by a recursively defined type. Therefore, it is natural to attempt proving properties of parameterized systems by inducting over this type. In this dissertation, we aim to automate the construction of such induction proofs by restricting the deductive machinery for constructing proofs. We construct an automatic and programmable first order logic prover with limited deductive capability.

The work in this dissertation is part of recent efforts to exploit logic programming technology for developing new tools and techniques to specify and verify concurrent systems. For example, constraint logic programming has been used for the analysis and verification of hybrid systems [UR95, Urb96] and more recently for model checking infinite-state systems [DP99]. In [RRR+97], a memoized logic programming engine is used to develop XMC, an efficient and flexible model checker whose performance is comparable to that of highly optimized model checkers such as Spin [Hol97] and Murφ [Dil96]. Recently, [Del00] used constraint logic programming to construct uniform proofs of safety properties of parameterized cache coherence protocols. Essentially, these techniques aim to use (constraint) logic program evaluation to efficiently construct verification proofs involving state space search (accomplished via resolution) and (possibly) constraint solving. These techniques are in general not suitable for constructing induction proofs arising in the verification of parameterized systems.

This dissertation provides a methodology for constructing such proofs by suitably extending the resolution based evaluation mechanism of logic programs [RKR+00]. In this approach, the parameterized system and the property to be verified is expressed as a logic program. The verification problem is reduced to the problem of determining the equivalence of predicates in this program . The predicate equivalences are then established by transforming them s.t. their semantic equivalence can be inferred from the syntax of their transformed definitions. The proof of semantic equivalence of two transformed predicates p, p' then proceeds automatically by a routine induction on the size of the proofs of ground instances of $p(\overline{X})$ and $p'(\overline{X})$.

For transforming the predicates, we use the well-established *unfold/fold* transformations of logic programs which have been previously used for program optimization [BCD90, LSW96, PPR97] and automated deduction [HS87, KF86, PP99]. The major transformations in such a transformation framework are unfolding, folding and goal replacement. One of these transformations (unfolding) represents an application of resolution. In particular, an application of the unfold transformation represents a single resolution step. Therefore, one can achieve on-the-fly explicit state *algorithmic* model checking by repeated unfolding of the verification proof obligation. In constructing induction proofs, unfold transformations are used to evaluate away the base case and the finite portions of the proof in the induction step of the induction argument. Folding and goal replacement, on the other hand, represent a form of *deductive* reasoning. They are used to simplify the given program so that applications of the induction hypothesis in the induction proof can be recognized.

However, one needs to substantially generalize the existing unfold/fold transformation rules [TS84, KF87, GK94, PP99] for using them in proofs of temporal properties. To ensure correctness, existing transformation systems restrict the applicability of the rules representing deductive steps *e.g.* folding. These restrictions are based on program syntax of the clauses participating in a deductive step, and are typically severe *e.g.* only non recursive clauses may participate in a folding step. However temporal properties contain fixed point operators. These properties are typically encoded as a logic program predicate with multiple recursive clauses *e.g.* a least fixed point property containing disjunctions is encoded using multiple recursive clauses. Therefore, one cannot assume restrictions that are imposed by existing transformation systems on the syntax of clauses encoding a temporal property. This dissertation develops a unfold/fold transformation system [RKRR99a, RKRR99b] where the applicability of the transformation rules is not restricted by program syntax. Instead, book-keeping is performed at every transformation step, and this book-keeping is used to restrict the applicability of the transformation rules, thereby ensuring the correctness of the transformations. Note that each application of these transformation rules is mechanizable.

Contributions of this dissertation: In a broad perspective, the contributions of this dissertation can be summarized as follows.

- First, it shows how logic program evaluation based techniques for verifying finite

state systems can be flexibly extended to yield a program transformation framework for constructing inductive proofs of temporal properties of parameterized systems. Since one of our transformations corresponds to a model checking step and the others correspond to deductive reasoning, model checking emerges as a special case when the deductive steps are applied lazily.

- The program transformation framework presented here allows for tight integration of algorithmic and deductive verification steps in a proof. Note that application of unfolding and folding steps can be arbitrarily interleaved in the verification proof of a parameterized system. This constitutes a tighter integration of model checking computation with deductive reasoning as compared to the integration of model checking as a decision procedure in a theorem prover.

- This dissertation reports the development of a new, more powerful unfold/fold transformation rules which are crucial for constructing such program transformation proofs of temporal properties. The development of these transformation rules also solves the following problem : whether it is possible to ensure correctness of unfold/fold rules without imposing restrictions on program syntax. This issue has been resolved for functional programs in [San96] but has not been addressed for logic programs so far. The applicability of our transformation rules is not restricted by program syntax. The development of these more general transformation rules is of independent interest to the logic programming community as well.

- Finally, the dissertation presents terminating strategies for controlling the application of the transformation rules, thereby leading to the implementation of a programmable and fully automatic prover. These strategies are used to construct liveness and safety proofs of parameterized systems of various "structures" including uni and bi-directional chains, rings and trees of processes. The prover has been used to construct automated proofs of safety properties of cache coherence protocols. As a demonstration of the practical utility of the prover, its successful use in automatically verifying mutual exclusion in the Java Meta-Locking Algorithm (a recently developed algorithm to ensure secure access of Java objects by multiple Java threads) from Sun Microsystems [ADG+99] is also described.

Chapter	Logic Programming	Verification	Theorem Proving
1	√	√	
2	√	√	
3	√		√
4	√		
5	√	√	√
6	√	√	√
7		√	
8	√	√	

Table 1: Chapters of the dissertation

Organization of the Chapters: The rest of this dissertation is organized as follows. The next chapter presents an overview of the general approach taken in this dissertation. It also includes comparison of the work with existing literature, and notational conventions used throughout the dissertation. In chapters 3 and 4, we first develop the theory of unfold/fold transformations which we will require to construct our verification proofs. In particular, chapter 3 presents a new transformation framework for definite (*i.e.* positive) logic programs and chapter 4 then extends it to logic programs with negation. Chapter 5 discusses how we can accomplish theorem proving by proving predicate equivalences in a logic program. These predicate equivalences are shown via logic program transformations. In particular, Chapter 5 presents a proof system for solving predicate equivalences and then presents control strategies for guiding the application of the proof rules. Chapter 6 presents an unfold/fold transformation based prover for verifying parameterized protocols. Applications and case studies performed using the prover appear in Chapter 7. Finally, chapter 8 presents the concluding remarks along with extensions and directions for future research.

Reading the dissertation: This thesis spans two different research areas : logic programming and verification. In addition, the technique proposed for automating induction proofs of predicate equivalences might be of independent interest from the perspective of theorem proving as well. Therefore depending on the interests of the reader, some or all of the chapters may be read in an appropriate sequence. Table 1 summarizes the focus of the different chapters in the dissertation.

The dependencies between the chapters are summarized in Figure 1. A solid arrow

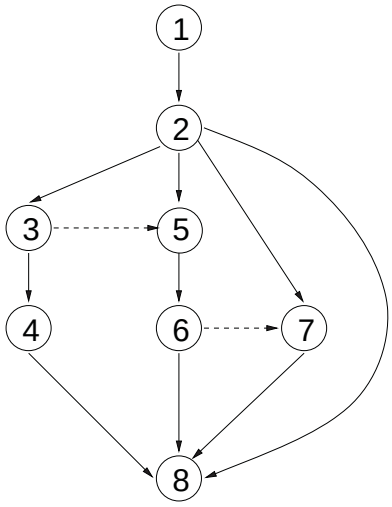


Figure 1: Dependencies among chapters of the dissertation

from i to j indicates that chapter i should be read before chapter j. A dashed arrow from i to j indicates the following. The main results of chapter j can be understood irrespective of the content of chapter i ; however for understanding the technical details (such as proofs) in chapter j, the results of chapter i are helpful.

Chapter 2

Overview

In this chapter, we provide a general overview of the approach embodied in this thesis for verifying parameterized concurrent systems. First, we review some basics of logic programming. Then we provide an informal description of the program transformations and show their application in constructing induction proofs via a simple example. We then discuss the existing literature in program transformations and parameterized system verification, with comparisons to our work. Finally, we conclude with notational conventions followed in the dissertation.

2.1 Logic Programming Preliminaries

We review some basics of logic programming, and present a well understood result on the semantics of logic programs. We assume that the reader is familiar with the notions of terms, atoms, substitutions, unification, most general unifier, clauses and resolution [Das92, Llo93].

Definition 2.1 (Open Atom) *An open atom is an atom of the form* $p(X_1, \ldots, X_n)$ *where* p *is a predicate symbol and* $X_1, \ldots, X_n$ *are distinct variables.*

Definition 2.2 (Ground Atom) *Atoms containing no variables are called ground atoms.*

Similarly, terms containing no variables are called ground terms.

Definition 2.3 (Literal) *A literal is an atom or the negation of an atom. A positive literal is an atom.*

Definition 2.4 (Goal) *A conjunction of literals is called a goal.*

For convenience of notation, we recall the definition of clause below.

Definition 2.5 (Clause) *A clause is a first order logic formula of the form* $\forall \overline{X}(A_1 \vee \ldots A_l \vee \neg B_1 \vee \ldots \vee \neg B_k)$ *where* $A_1, \ldots, A_l, B_1, \ldots, B_k$ *are atoms and* $\overline{X}$ *denotes the variables occurring in* $A_1, \ldots, A_l, B_1, \ldots, B_k$.

In logic programming notation, a clause $\forall \overline{X}(A_1 \vee \ldots A_l \vee \neg B_1 \vee \ldots \vee \neg B_k)$ is often written as $A_1, \ldots, A_l :- B_1, \ldots, B_k$, where $A_1, \ldots, A_l$ is called the *head* of the clause and $B_1, \ldots, B_k$ is called the *body*. The notation $:-$ stands for the logical implication $\Leftarrow$. Note that the commas in the head denote disjunction and the commas in the body denote conjunction.

Definition 2.6 (Horn Clause) *A Horn clause or a definite clause is a clause with only one positive literal,* i.e. *a formula of the form* $\forall \overline{X}(A \vee \neg B_1 \vee \ldots \vee \neg B_k)$ *written as* $A :- B_1, \ldots, B_k$.

Definition 2.7 (Definite Logic Program) *A (definite) logic program is a finite set of Horn clauses.*

Definition 2.8 (Normal Logic Program) *A normal logic program is a finite set of clauses of the form* $A :- L_1, \ldots, L_k$ *where* $L_1, \ldots, L_k$ *are literals.*

Therefore, all definite logic programs are normal logic programs.

Definition 2.9 (Herbrand Universe) *Herbrand universe of a logic program* P*, denoted* $HU(P)$*, is the set of ground terms constructed by using the function symbols*[1] *appearing in* P.

Definition 2.10 (Herbrand Base) *Herbrand base of a logic program* P*, denoted* $HB(P)$*, is the set of ground atoms constructed by using the predicate and function symbols appearing in* P.

Definition 2.11 (Herbrand Interpretation) *Given a logic program* P*, a Herbrand interpretation is an interpretation over the Herbrand Universe* $HU(P)$ *s.t.*

[1] Constants are function symbols with arity 0.

- *If f is an n-ary function symbol appearing in P then f is assigned to an n-ary function $HU(P)^n \rightarrow HU(P)$ which maps $(t_1, \ldots, t_n)$ to $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in HU(P)$.*

Note that the definition of Herbrand interpretation fixes the assignment of function symbols, but does not restrict the assignment of predicate symbols. Thus, we can associate each Herbrand interpretation with a unique subset of the Herbrand base : the set of all ground atoms which are true w.r.t. the interpretation.

Definition 2.12 (Herbrand Model) *Given a logic program P, a Herbrand model for P is a Herbrand interpretation which is a model for the first order formulae represented by P.*

A Herbrand model being a Herbrand interpretation is also simply a subset of the Herbrand Base.

Definition 2.13 (Least Herbrand Model) *Semantics of a definite logic program P, denoted $M(P)$, is the set of atoms in $HB(P)$ which are logical consequences of P. $M(P)$ is called the least Herbrand model since it is the smallest subset of $HB(P)$ which is a Herbrand model.*

For a definite logic program P, the truth of any ground atom $A \in HB(P)$ can be "derived" by repeatedly resolving A with the clauses in program P. This is formally captured via the notion of ground proofs.

Definition 2.14 (Ground Proof) *Let T be a tree, each of whose nodes is labeled with a ground atom. Then T is a ground proof in a definite program P, if every node A in T satisfies the condition : $A :- A_1, ..., A_n$ is a ground instance of a clause in P, where $A_1, ..., A_n$ $(n \geq 0)$ are the children of A in T.*

Using the above definition, we can define the semantics of a definite logic program P as the set of ground atoms in $HB(P)$ which appear in the root of some ground proof in P. The equivalence of the model theoretic and proof theoretic semantics of definite logic programs is summarized by the following theorem. This theorem will be be used in the description of our approach in the next section.

Theorem 2.1 (Semantics of Definite Logic Programs) *Let P be a definite logic program. Then $A \in M(P)$ iff A appears in the root of a ground proof in P.*

The proof of a stronger result, namely the equivalence of the least Herbrand model semantics and SLD resolution based procedural semantics appears in [Das92].

2.2 Overall Approach

In this section, we provide an overview of the proof technique presented in this dissertation. To describe this logic program transformation based technique for verifying parameterized systems, we first describe how the verification problem can be encoded as a logic program.

2.2.1 Encoding the Verification Problem

Recall that a parameterized concurrent system is an infinite family of finite state concurrent systems. Intuitively, a parameterized system can be viewed as a network of an unbounded number of finite state processes which communicate in a specific pattern. These finite state processes constituting the network have a finite number of process types, and their communication pattern is called the *network topology*. For example, an n bit shift register (for any n) is a parameterized system. It represents an unbounded number of finite state processes communicating along a chain. These finite state processes are "similar", each of them representing a single bit. To model a parameterized system as a logic program, the local states of the constituent finite state processes are represented by terms of finite size. The global state of the parameterized system is then represented by a term of unbounded size consisting of these finite terms as sub-terms. The initial states and the transition relation of the parameterized system are then encoded as logic program predicates with such unbounded terms as arguments.

For example, in an n bit shift register (for any n), the local states of the bit process are represented by the terms `0` and `1` (corresponding to the situations where the value stored in the bit is 0 and 1 respectively). A global state of the register is then represented by an unbounded list where each element of the list is `0` or `1`. Now, let us consider an n bit shift register where initially the rightmost bit of the chain contains 1 and all other bits contain 0. The system evolves by passing the 1 leftward. A logic program describing the system is given in Figure 2. The predicate `gen` generates the initial states of an n-process chain for all n. As mentioned above, a global state of the register is represented as an ordered list (a list in Prolog-like notation is of the form `[Head|Tail]`) of zeros and ones. The set of bindings of variable `S` upon evaluation of the query `gen(S)` is { `[1]`, `[0,1]`, `[0,0,1]`, ... }. The predicate `trans` in the program encodes a single transition of the global automaton. The first clause in the definition of `trans` captures the transfer of the 1 from right

```
gen([1]).
gen([0|X]) :- gen(X).
trans([0,1|T], [1,0|T]).
trans([H|T], [H|T1]) :- trans(T, T1).
```

System description

```
thm(X) :- gen(X), live(X).
live(X) :- X = [1|_].
live(X) :- trans(X, Y), live(Y).
```

Property description

Figure 2: Example: Liveness in an unbounded length shift register

to left; the second clause recursively searches the state representation until the first clause can be applied. (i.e., when the 1 is not already in the left-most bit).

So far, we have illustrated how the parameterized system to be verified can be encoded using logic program predicates. The temporal property to be verified can also be encoded as a logic program predicate over global states of the system.[2] For our shift register example, let us consider the following liveness property : eventually the 1 reaches the left most bit. This is encoded by the predicate `live` in Figure 2. The first clause of `live` succeeds for global states where the 1 is already in the left-most bit (a good state). The second (recursive) clause of `live` checks if a good state is reachable after a (finite) sequence of transitions.

Thus, every member of the family satisfies the liveness property if and only if $\forall$ `X` `gen(X)` $\Rightarrow$ `live(X)`. Moreover, this is the case if $\forall$ `X` `thm(X)` $\Leftrightarrow$ `gen(X)`, i.e. if `thm` and `gen` are semantically equivalent. Thus, we have encoded the verification problem as a logic program and reduced the proof obligation to establishing equivalence of program predicates.

[2] [RRR+97] describes how the least fixed point semantics and well founded semantics of logic programs can be utilized to obtain logic program encoding of modal μ-calculus properties. We skip such details here since it is not central to our presentation.

2.2.2 Proofs by Program Transformations

We now illustrate how we can construct induction based proofs arising in parameterized system verification via logic program transformations. Essentially, this is accomplished using the following steps:

1. Encode the temporal property to be verified as well as the parameterized system as a logic program P_0.
2. Convert the verification proof obligation to predicate equivalence proof obligations of the form $P_0 \vdash \mathtt{p} \equiv \mathtt{q}$ ($\mathtt{p}, \mathtt{q}$ are predicates)
3. Construct a transformation sequence $P_0, P_1, \ldots, P_k$ s.t.
   (a) Semantics of P_0 = Semantics of P_k
   (b) from the syntax of P_k we infer $P_k \vdash \mathtt{p} \equiv \mathtt{q}$

In the shift register example, we have encoded the problem of verifying liveness in an n bit shift register as the logic program P_0 in Figure 2. We have reduced the verification proof obligation to establishing the equivalence of `thm` and `gen` predicates in program P_0. We then apply program transformations to program P_0 to obtain a program P_k where `thm` and `gen` are defined as follows:

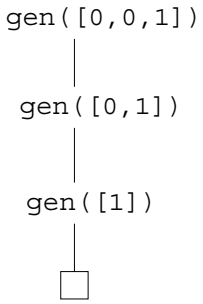
```
gen([1]).                 thm([1]).
gen([0|X]) :- gen(X).     thm([0|X]) :- thm(X).
```

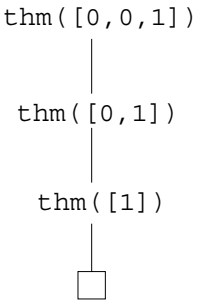
Figure 3: Fragment of Transformed Program for Shift Register Example

Thus, since the transformed definitions of `thm` and `gen` are "isomorphic", their semantic equivalence can be inferred from syntax. In general, we have a sufficient condition called *syntactic equivalence* s.t. if two predicates `p` and `q` are syntactically equivalent in program P_k then `p` and `q` are semantically equivalent in P_k. Furthermore, we ensure that checking syntactic equivalence of two predicates in a given program is decidable. In the shift register example, the transformed definitions of `gen` and `thm` given in Figure 3 are syntactically equivalent. The formal definition of syntactic equivalence is presented in Chapter 5.

The definitions of `gen` and `thm` given above both represent the infinite set $\{[\mathtt{0}^n, \mathtt{1}] \mid n \in \mathbb{N}\}$. For each element `X` in this set, we can therefore construct a

ground proof (refer Definition 2.14) of `thm(X)` and `gen(X)`. For example, a ground proof tree[3] of `gen([0,0,1])` and `thm([0,0,1])` (using the above clauses of `thm` and `gen`) are shown below.





Inferring the equivalence of `thm` and `gen` from the transformed definitions in Figure 3 involves an *induction* on the size of the proof trees of `gen(X)` and `thm(X)` for any ground term `X`. In general, to prove the equivalence of two predicates p, p' of same arity we first transform their definitions to syntactically equivalent forms. Then, the proof of semantic equivalence of two syntactically equivalent predicates p, p' proceeds (by definition of syntactic equivalence) as follows:

- show that for every ground proof of $p(\overline{X})\theta$ (where $\overline{X}$ are variables and θ is any ground substitution of $\overline{X}$) there exists a ground proof of $p'(\overline{X})\theta$. This follows by induction on the size of ground proofs of $p(\overline{X})\theta$.
- show that for every ground proof of $p'(\overline{X})\theta$ (where $\overline{X}$ are variables and θ is any ground substitution of $\overline{X}$) there exists a ground proof of $p(\overline{X})\theta$. This follows by induction on the size of ground proofs of $p'(\overline{X})\theta$.

Thus, transforming `gen` and `thm` to obtain the definitions of Figure 3 and then inferring the equivalence from these transformed definitions amounts to an induction proof of the liveness property. Note that even though we are actually inducting on the size of ground proofs, here this is same as inducting on the process structure of the parameterized system : the length of the shift register. We now discuss how we can transform the definitions of `gen` and `thm` given in Figure 2 to obtain the program in Figure 3. We also show how the application of these transformations aid different steps of the induction proof.

[3] In this particular example, these are the only ground proofs of `gen([0,0,1])` and `thm([0,0,1])`.

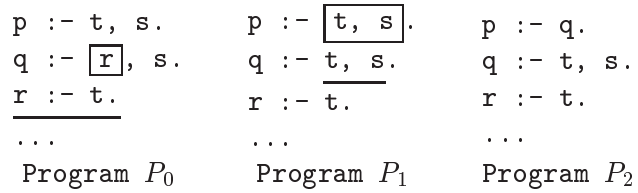


Figure 4: Example of a correct unfold/fold transformation sequence.

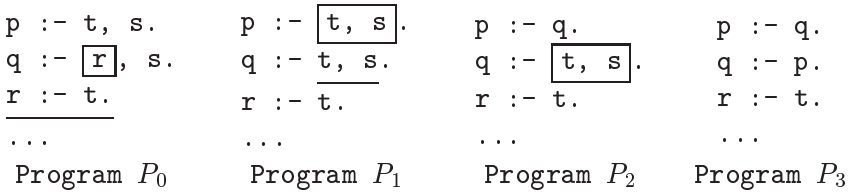


Figure 5: Example of an incorrect unfold/fold transformation sequence.

2.2.3 Unfold/Fold Program Transformations

We transform a logic program P_i to another program P_{i+1} by applying program transformations that include unfolding, folding, goal replacement. For a simple illustration of program transformations, consider Figure 4. There, program P_1 is derived from P_0 by *unfolding* the occurrence of `r` in the definition of `q`. P_2 is derived from P_1 by *folding* `t,s` in the definition of `p` using the definition of `q`.

While unfolding is semantics preserving, indiscriminate folding may introduce circularity, thereby removing finite proof paths. As shown in Figure 5, folding `t,s` in the definition of `q` in P_2 using the definition of `p` in P_0 results in a program

```
p :- q.  q :- p.  r :- t.  ...
```

This removes `p` and `q` from the least Herbrand model. Thus the least Herbrand model of P_3 is different from the least Herbrand model of P_0 and the transformation sequence P_0, P_1, P_2, P_3 is *not* semantics preserving.

We now present the program transformations informally. A formal description appears in the next two chapters. We call a sequence of program $P_0, P_1, \ldots, P_n$ as a program transformation sequence if for all $0 \leq i < n$, program P_{i+1} is obtained from program P_i by applying a program transformation. With each clause C in program P_i of the transformation sequence, we associate a pair of integer counters that bound

$$
\begin{array}{ll}
B : -\mathcal{G}, \boxed{\mathrm{A}}, \mathcal{G}'. & \\
A_1 : -\mathcal{F}_1. & \\
A_2 : -\mathcal{F}_2. & \longrightarrow \\
\vdots & \\
A_n : -\mathcal{F}_n. &
\end{array}
\begin{array}{l}
B\sigma_1 : -(\mathcal{G}, \boxed{\mathcal{F}_1,} \mathcal{G}')\sigma_1. \\
B\sigma_2 : -(\mathcal{G}, \boxed{\mathcal{F}_2,} \mathcal{G}')\sigma_2. \\
\vdots \\
B\sigma_n : -(\mathcal{G}, \boxed{\mathcal{F}_n,} \mathcal{G}')\sigma_n. \\
A_1 : -\mathcal{F}_1. \\
\vdots \\
A_n : -\mathcal{F}_n.
\end{array}
$$

Figure 6: Schema for Unfolding Transformation

$$
\begin{array}{ll}
 & A_1 : -\mathcal{F}'_1. \\
P_j\text{:} & \vdots \\
 & A_n : -\mathcal{F}'_n.
\end{array}
$$

$$
P_i\text{:}\quad
\begin{array}{l}
B : -\mathcal{G}, \boxed{\mathcal{F}_1,} \mathcal{G}'. \\
B : -\mathcal{G}, \boxed{\mathcal{F}_2,} \mathcal{G}'. \\
\vdots \\
B : -\mathcal{G}, \boxed{\mathcal{F}_n,} \mathcal{G}'.
\end{array}
\longrightarrow B : -\mathcal{G}, \boxed{\mathrm{A}}, \mathcal{G}'.
$$

Figure 7: Schema for Folding Transformation

the size of a shortest proof of any ground atom A derived using C in program P_i relative to the size of a shortest proof of A in P_0. Thus the counters keep track of potential reductions in proof lengths. Conditions on counters are then used to ensure that every step of a program transformation sequence is semantics preserving.

Unfolding of an atom A in the body of a clause in P_i is shown in Figure 6. The conditions for applying the transformation are : (i) $A_1, \ldots, A_n$ are the only clause heads in P_i which unify with A, and (ii) σ_j is the mgu of A and A_j for all $1 \leq j \leq n$. Note that these conditions are taken directly from resolution, which means that unfolding is essentially a resolution step.

Folding replaces an occurrence of the body of a clause with its head. The clause where the replacement takes place is called the *folded* clause and the clauses used to perform the replacement are called the *folder* clauses. The folding schema is illustrated in in Figure 7, where the clauses of B are the folded clauses, and the clauses of A are the folder clauses. The folder clauses may come from some earlier

program $P_j (j \leq i)$ in the transformation sequence. The conditions for applying the transformation are[4]: (i) $\mathcal{F}_l$ is an instance of $\mathcal{F}'_l$ with substitution σ_l for all $1 \leq l \leq n$ (ii) there is an atom A such that $\forall 1 \leq l \leq n\ A_l\sigma_l = A$ and the folder clauses are the only clauses in P_j whose heads unify with A.

Goal replacement replaces an atom B in a clause $A :- \mathcal{F}, B, \mathcal{G}$ in program P_i with a semantically equivalent atom B' to obtain the clause $A :- \mathcal{F}, B', \mathcal{G}$. Note that such a replacement can change lengths of proofs of A arbitrarily. To obtain the counters associated with the new clause we conservatively estimate the changes in proof lengths.

Example: Liveness Property in Shift Register Recall the logic program of Figure 2 which formulates a liveness property about shift registers, namely, that the signal `1` eventually reaches the left-most bit in any arbitrary length register. To establish the liveness property, we have to prove `thm(X)` $\equiv$ `gen(X)`.

First we unfold `gen(X)` in the definition clause of `thm` to obtain:

```
thm([1]) :- live([1]).
thm([0|X]) :- gen(X), live([0|X]).
```

Note that this unfolding step corresponds to uncovering of the induction schema. By the process of unification associated with unfolding, we discover the schema to induct on in the induction proof of `thm` $\equiv$ `gen`.

We then repeatedly unfold `live([1])` in the first clause to obtain:

```
thm([1]).
thm([0|X]) :- gen(X), live([0|X]).
```

These steps correspond to establishing the base case of the induction proof. Note that `live([1])` denotes the proof obligation that the liveness property holds for a one-bit register. The unfolding steps required to establish the truth of `live([1])` exactly correspond to top-down algorithmic model checking.

We now repeatedly unfold `live([0|X])` in the second clause of `thm` to obtain:

```
thm([1]).
thm([0|X]) :- gen(X), X = [1|_].
thm([0|X]) :- gen(X), trans(X,Y), live([0|Y]).
```

[4]In addition, certain other conditions need to be imposed including conditions on the counters of the folder and folded clauses; we do not mention them here.

These unfolding steps correspond to the finite part of the induction step. Note that `gen(X), live([0|X])` denotes the liveness property being true in the $n+1$ bit register. By unfolding `live([0|X])` we perform the finite part of the induction step after which the induction hypothesis (of the liveness property being true in n bit register) can be applied.

We now apply goal replacement, replacing `live([0|Y])` by `live(Y)` in the third clause of `thm`.

```
thm([1]).
thm([0|X]) :- gen(X), X = [1|_].
thm([0|X]) :- gen(X), trans(X,Y), live(Y).
```

Note that one of the pre-conditions for applying this goal replacement step is showing the equivalence of `live([0|Y])` and `live(Y)`. We can again show this by transforming them to syntactically equivalent definitions. In other words, the equivalence proof `live([0|Y])` $\equiv$ `live(Y)` appears as a subproof of `thm(X)` $\equiv$ `gen(X)`.

We now fold the last two clauses of `thm` using the definition of `live` as folder. We obtain:

```
thm([1]).
thm([0|X]) :- gen(X), live(X).
```

Finally, folding the second clause of `thm` using the original definition of `thm` as folder we obtain the following program:

```
thm([1]).
thm([0|X]) :- thm(X).
gen([1]).
gen([0|X]) :- gen(X).
trans([0,1|T], [1,0|T]).
trans([H|T], [H|T1]) :- trans(T, T1).
live(X) :- X = [1|_].
live(X) :- trans(X, Y), live(Y).
```

The definitions of `gen` and `thm` in the above program are syntactically equivalent. Recall that inferring the equivalence of `thm` and `gen` from the above definitions proceeds by induction on the size of proof trees of ground instances of `thm(X)` and `gen(X)`. Application of the goal replacement and the folding steps enables us to recognize the induction hypothesis in this induction proof.

Note that the transformation sequence constructed in the above example involves a folding step using the clauses of `live` as the folder. The predicate `live` is a least

fixed point temporal property encoded using multiple clauses one of which is recursive. Since, most interesting temporal properties contain fixed point operators, their logic program encoding will contain multiple recursive clauses. Therefore, existing logic program transformation systems [TS84, GK94, PP99] are not suitable for constructing proofs of temporal properties via program transformations. This is because they impose restrictions on the syntax of the clauses participating in a folding step to ensure correctness. Such restrictions are relaxed in the transformation rules developed in the next two chapters.

2.3 Related Work

This thesis is related to two different areas of research : logic program transformations and parameterized system verification. Hence it is appropriate to compare the work developed in this thesis to the relevant literature of both of these areas. We do so in this section.

2.3.1 Related Work on Logic Program Transformations

Transformations for definite programs: An unfold/fold transformation system for definite logic programs was first described in a seminal paper by Tamaki and Sato [TS84]. Subsequently, a number of unfold/fold transformation systems were developed. Kanamori and Fujita [KF87] proposed a transformation system that was based on maintaining counters to guide folding. Maher described a system that permits only reversible folding [Mah87], *i.e.* folding whose effect can be *reversed* by an unfolding step. The basic Tamaki-Sato system itself was extended in several directions (e.g., to handle folding with multiple clauses [GK94], negation [AD95, Sek91, Sek93]) and applied to practical problems (e.g., [BCD90, BB93, PPR97]). An excellent survey of research on this topic over the past decade appears in [PP98].

If the folding transformation is reversible, then since its effect can be undone by an unfolding, any partially correct unfold/fold transformation sequence is also totally correct. However, for reversibility, folding at step i of the transformation can *only* use the clauses in P_i. Therefore reversibility is a *restrictive* condition that seriously limits the power of unfold/fold systems by disallowing many correct folding transformations. Hence almost all research on unfold/fold transformations have focused on constructing systems that permit irreversible folding. In such systems folding at step i can use

clauses that are not in P_i. For example, in the original and extended Tamaki-Sato systems [TS84, TS86a] folding always uses clauses in P_0 whereas in the Kanamori-Fujita system [KF87] the clauses can come from any P_j ($j \leq i$). But ensuring total correctness of irreversible transformation sequences is difficult. In order to ensure that folding is still totally correct, these systems permit folding using only clauses with certain (syntactic) properties. For instance, the original Tamaki-Sato system permits folding using a single clause only (*conjunctive* folding) and this clause is required to be non-recursive. In [GK94] the above system was extended to allow folding with multiple clauses (*disjunctive* folding) but all the clauses are required to be be non-recursive. Kanamori and Fujita [KF87] as well Tamaki and Sato in a later paper [TS86a] gave two different approaches for conjunctive folding using recursive clauses.

Transformations for normal programs: For normal logic programs, existing unfold/fold transformation systems can be broadly classified based on the semantics preserved by the transformations. Gardner and Shepherdson [GS91] presented unfold/fold transformations which preserve SLDNF derivations and Clark's completion semantics. On the other hand, a host of other authors have investigated the correctness of unfold/fold transformations w.r.t. model theoretic semantics which are constructively defined *e.g.* perfect model and well-founded model semantics. These works include [Mah93, PP00, Sek91] which study transformations for stratified programs preserving perfect model semantics, and [AD95, Sek93] which present transformations for general logic programs preserving well-founded model and other semantics. However, these works are either extensions of Maher's reversible transformation system or the original Tamaki-Sato system. In particular, [Mah93] extends Maher's reversible transformation system to stratified logic programs, and [AD95, PP00, Sek91, Sek93] present extensions of the Tamaki-Sato style irreversible transformation system to normal logic programs.

Logic program transformations for constructing proofs: Note that unfold/fold logic program transformations have been primarily used for program synthesis, specialization and optimization (see for example [BCD90, BB93, LSW96, PPR97]). However, relatively little work has been done on using these transformations for constructing *proofs*. As discussed earlier in this section, unfold/fold transformations can be used to construct induction proofs of program properties.

In such induction proofs, unfolding accomplishes the base case and the finite part of the induction step, and folding roughly corresponds to application of induction hypothesis. This observation has been exploited in [HS87, KF86, PP99, PP00] to perform inductive theorem proving of program properties. All of these works employ more restricted Tamaki-Sato style unfold/fold transformations, which are not suitable in general for constructing induction proofs of temporal properties. This is because temporal properties are typically encoded using multiple recursive clauses. Our work relaxes restrictions on the applicability of the transformation rules thereby enabling their use in proving temporal properties. Furthermore, since our work is more specifically focused towards concurrent system verification, we investigate strategies to guide the transformations for verifying parameterized concurrent systems.

Salient features of our work: In [RKRR99b] we proposed a transformation framework for definite logic programs which generalized the above systems by permitting folding using multiple recursive clauses. This work is described in Chapter 3. In [RKRR99a], we have extended our transformation framework to normal logic programs. This extension is described in Chapter 4 of this thesis. Our transformation rules are more general than the existing unfold/fold transformation systems both for definite and normal logic programs. These rules preserve program semantics without imposing unnecessary restrictions on program syntax.

The key difference between our transformations and the ones proposed in existing literature lies in the book-keeping we perform while constructing a transformation sequence. Our transformation rules annotate every clause of the program P_i and update these annotations while transforming P_i to P_{i+1}. The applicability of a transformation to program P_i is decided based solely on the annotations of the clauses in $P_0, \ldots, P_i$, and not on program syntax. For efficiency purposes, we can easily restrict our transformations s.t. the applicability of a transformation in P_i is decided based on the annotations of the clauses in P_i and P_0. In the next chapter, we present a parameterized framework for unfold/fold transformations of definite logic programs. The framework is in fact parameterized by the "domain" of the annotations or book-keeping of the program clauses, *e.g.* the annotations may be integers.

2.3.2 Related Work on Parameterized System Verification

Regarding related work in the verification area, a plethora of techniques have been proposed during the past decade for verifying parameterized systems. We review

them below, with comparisons to our work.

Developing Decision Procedures for Subclasses: Techniques in this class [GS92, EN95, EN96, ID99] reduce the problem of verifying a parameterized system to the verification of an "equivalent" finite-state system. These techniques consider specific classes of parameterized systems and temporal properties, and present fully automated verification techniques for these classes. Typically, this is achieved by establishing a "cutoff number", *i.e.* the property holds for every member of the parameterized system, if it holds for members up to the cutoff number. The only human reasoning involved is in establishing this cutoff number. Once this is established for a specific class of systems and formula, the verification problem reduces to model checking. The classes of parameterized systems for which the verification problem has been thus reduced include: (a) systems with a single control process and arbitrary number of user processes [GS92, EN96] (b) unidirectional rings with arbitrary number of processes which communicate by passing a token [EN95], and (c) systems formed by composing an arbitrary number of identical finite state processes in parallel [ID99]. Systems covered by classes (a) and (c) commonly occur in bus-based parameterized protocols. Some of these techniques [EN95] establish a stuttering bisimulation equivalence (refer [BCG89]) between a parameterized family with arbitrary number of processes and the member of the family with the cutoff number. Others [ID99] establish only a simulation between the parameterized family and its member with the cutoff number.

Network Invariants: The network invariant approach [WL89, KM95, CGJ97, LHR97] is applicable to parameterized systems consisting of an arbitrary number of copies of identical finite state processes (or finite state processes drawn from some finite set) that are composed in parallel. This approach relies on the existence of a pre-order relation $\preceq$ over processes, s.t. if P, P' are processes and φ is a temporal property then $P \preceq P' \wedge P' \models \varphi \Rightarrow P \models \varphi$. Furthermore, the parallel composition operator of processes is assumed to be monotonic w.r.t. $\preceq$. Under these assumptions, the network invariant approach constructs a finite state process I s.t. $P_i \preceq I$ for any member P_i of the parameterized family being verified. Verifying a temporal property φ for a parameterized family then involves:

1. synthesizing a network invariant I
2. checking $P_i \preceq I$ for any member P_i of the family

3. checking $I \models \varphi$

Step 1 involves substantial human ingenuity, and efforts have been made to partially automate this step. Step 2 proceeds by a routine induction on the "structure" of the parameterized family. Such induction proofs are amenable to automated inductive theorem proving techniques including the one developed in this thesis. Finally, step 3 is accomplished by model checking. Clearly, the main difficulty in using this approach is the synthesis of the network invariant, and the need to refine the invariant if $I \not\models \varphi$. [LHR97] tackles the second problem by viewing network invariant synthesis as least fixed point construction (which may not terminate) and using human assisted fixed point approximation heuristics such as widening (refer [CC92]) to compute the fixed point. Refining of the widening operator then yields a refined invariant. [CGJ97] tackles the problem of automating invariant construction by considering only those invariants which can be gleaned from the "base case" of the inductive definition of the parameterized family. Our proof technique is geared to automate proofs of temporal properties of parameterized systems where: *(i)* the proof proceeds by (nested) induction *(ii)* each of the induction arguments proceed without strengthening of induction hypothesis. There is no restriction on the existence of a network invariant which can be gleaned from the syntax of the parameterized family definition.

Rich Language Model Checking: This approach [KMM+97, EN98, EFM99, Del00] aims to *finitely* represent the state space and transition relation of the entire infinite family of finite-state systems comprising a given parameterized system. Clearly, finite representation of the global states of a parameterized system requires a richer language (than the direct term representation). For this purpose, [KMM+97] uses regular and tree-regular languages to represent global states of linear and tree networks of processes respectively. [EN98] constructs a covering graph which (if the construction terminates) can be used to decide safety and liveness properties of parameterized systems. The nodes of the covering graph represent a set of reachable states but the exact choice of the representation is not specified. [EFM99, Del00] use equality and inequality *constraints* on the counts of the local states (*i.e.* states of the constituent finite state processes in a network) to represent sets of global states of the network. In short, this approach requires a finite representation of infinite sets of states, and employs state space traversal techniques (just like model checking) over this finite representation to construct a proof for the parameterized network. Since, the proof is constructed by state space traversal and/or methods similar to constraint

solving, these proofs are uniform proofs, rather than induction proofs. However, an induction proof over the direct term representation can (in certain cases) be achieved by a uniform proof over this finite representation. In the technique proposed in this dissertation, a global state of a parameterized network is represented by an unbounded term. The state space, transition relation as well as the property to be verified are still represented finitely by means of (recursively defined) logic program predicates. These predicates are then transformed to yield an induction proof over the term representation of global states. In comparison to the rich language model checking approach, our technique is not restricted to specific classes of networks based on the choice of the rich language. For example, using our technique we constructed proofs of temporal properties of cache coherence and broadcast protocols (which can be handled by the count based constraint representation of [EFM99, Del00]) as well proofs of properties of tree networks (which cannot be handled by the count based constraint representation).

Verification via Theorem Proving: The work closest to ours involves the use of inductive theorem provers for verifying parameterized systems. Some of the theorem provers which have been used for verification are: Nqthm and ACL2 [BM90, KMM00], Oyster-CLAM [B$^+$90], PVS [OSR92] and RRL [KZ95]. Nqthm and ACL2 are provers based on first-order logic, RRL and Oyster-CLAM are based on term rewriting, and PVS is based on a higher order logic. These provers have been successfully employed to construct verification proofs (including verification of complete microprocessors) by using substantial human assistance. Combining decision procedures like model checking with theorem provers is thus important for reducing the human assistance required. Rajan et al. [RSS95] have incorporated a finite-state model checker for the modal mu-calculus as a decision procedure within the PVS theorem prover [OSR92]. Inductive proofs can be established by the prover via calls to the model checker to verify finite subparts. Graf and Saidi [GS96] show how a custom-built specification-deduction system can be combined with PVS to formalize and carry out model checking of invariant properties using deduction. The key difference between our approach and these is that we enhance model checking with deductive capabilities, rather than implementing model checking as a decision procedure in a deductive system. Moreover, deductive steps are deployed only on demand and hence do not affect the efficacy of the algorithmic model checking. In particular, the underlying evaluation mechanism for model checking in XMC is essentially unfolding, and we have enhanced this

mechanism with fold, and goal-replacement transformations. These transformations complement the power of model checking with the ability to do lightweight deduction.

Comparison with Theorem Provers: Our program transformation based proof technique can be viewed as a lightweight inductive theorem proving technique for automating nested induction proofs (where each induction argument proceeds without strengthening of hypothesis). We now compare our proof technique with inductive techniques employed in theorem provers. Since our logic program transformations essentially correspond to first order reasoning, we compare with a first order theorem prover (ACL2) to concretely highlight the differences.

Note that in our approach, the induction schema as well as the lemmas to be used in the inductive proof are implicit in the logic program itself. The program transformations make them explicit. Therefore, the proof constructed by our approach has lesser case splits (and hence is more succinct) than the proof constructed by a theorem prover. To concretely understand this point, let us consider the following example. The predicate `gen` [5] generates strings of the form 0^* while predicate `bad` captures strings containing at least one occurrence of `1`. The predicate `badgen` captures all strings for which both `bad` and `gen` hold. We want to prove $\forall$`X` $\neg$`badgen(X)` *i.e.* the predicate equivalence `badgen` $\equiv$ `false`.

```
gen([]).
gen([0|X]) :- gen(X).
bad([1|X]).
bad([H|X]) :- bad(X).
badgen(X) :- bad(X), gen(X).
```

Our unfold/fold transformation based proof technique will first unfold `bad(X)` using the clauses of `bad` to yield

```
badgen([1|X]) :- gen([1|X]).
badgen([H|X]) :- bad(X), gen([H|X]).
```

By unfolding `bad(X)`, the induction schema has been uncovered (which is implicit in the definition of `bad`). The two clauses of `badgen` given above correspond to the *two cases* of this induction schema. Now by unfolding the occurrences of `gen` in these clauses we obtain the following program. These steps correspond to showing the base case and evaluating the finite part of the induction step.

[5] The empty list `[]` denotes the empty string ϵ.

```
badgen([0|X]) :- bad(X), gen(X).
```

Finally, we fold using the original definition of `badgen`. The least fixed point semantics of logic programs allows us to conclude that `badgen` $\equiv$ `false`.

```
badgen([0|X]) :- badgen(X).
```

In contrast if we input this problem to the ACL2 theorem prover [KMM00] it attempts to prove by an induction schema obtained from `bad`, and generates the following *four cases* for showing $\forall$`X` $\neg$`badgen(X)`. Note that our proof technique also derived the induction schema from `bad` but considered only *two cases*.

1. `X` $=$ `[]`
2. `X` $=$ `[H|T]` $\wedge$ `H` $=$ `1`
3. `X` $=$ `[H|T]` $\wedge$ `H` $\neq$ `1` $\wedge$ $\neg$ `bad(T)`
4. `X` $=$ `[H|T]` $\wedge$ `H` $\neq$ `1` $\wedge$ $\neg$ `gen(T)`

The first case above is never considered by our proof technique since we discover the induction schema by the process of *unification* inbuilt into unfolding. The absence of types in logic programs enables us to encode `bad` for only non-null lists, and since we obtain the induction schema from this encoding we never consider the first case. The second case above corresponds to the case obtained from the first clause of `bad` in our proof. Finally, the third and fourth cases are combined into a single case in our proof. In fact in our proof, the folding transformation prevents this unnecessary case split and recognizes the induction hypothesis, all in one step.

In summary, our program transformation based proof technique is useful for constructing induction proofs where the induction schema as well as the requisite lemmas are implicitly encoded in the logic program itself. Our syntax based transformations *(i)* make the schema explicit *(ii)* reason about the different cases of the schema by uncovering the requisite lemmas. Furthermore, our transformations do so by (a) not considering spurious cases in the schema, and (b) combining some of the cases of the schema into a single case. Both of these advantages of our proof technique have been illustrated through the example given above.

As a side remark note that we have encoded `badgen` as `badgen(X) :- bad(X), gen(X)` instead of `badgen(X) :- gen(X), bad(X)`. This is because ACL2 chooses to induct on a scheme obtained from `bad` in this problem. By encoding `badgen` as

`badgen(X) :- bad(X), gen(X)` we can give a point by point comparison with ACL2 since then both proof techniques are inducting on the same schema. In the alternative encoding also our unfold/fold based proof technique generates only two cases in the induction schema (which are obtained from the definition of `gen`).

2.4 Notational Conventions

As discussed before, we assume that the reader is familiar with the notions of terms, atoms, substitutions, unification, most general unifier (*mgu*), clauses and resolution [Das92, Llo93]. Following logic programming conventions, we use names beginning with an uppercase letter for variables. Names beginning with a lowercase letter are used for predicates and function symbols. Recall that a term having no variables is called a *ground* term. *Atoms* are terms with a predicate symbol at the root. A literal is an atom or the negation of an atom. A goal is a conjunctions of literals. Therefore, in definite programs, a goal is a conjunction of atoms. Atoms whose subterms are distinct variables (i.e., atoms of the form $p(X_1, \ldots, X_n)$, where p is a predicate symbol of arity n) are called *open atoms*. We call a sequence of programs $P_0, P_1, \ldots, P_n$ as a *program transformation sequence* if P_{i+1} is obtained from P_i by applying a program transformation rule.

We use the following conventions (possibly with primes, subscripts and superscripts): p, q for predicate symbols; X, Y for variables; t, s for terms; $\overline{X}, \overline{Y}$ for sequences of variables; $\overline{t}, \overline{s}$ for sequences of terms; A, B for atoms; L, K for literals; $\mathcal{F}, \mathcal{G}$ for goals; σ, θ for substitutions; C, D for clauses; P for a logic program which is a set of clauses. A program transformation sequence is often denoted by Γ for brevity. The notation $\mathcal{E}$ is used to denote a predicate equivalence proof obligation of the form $\Gamma \vdash p \equiv q$. It represents the obligation that predicates p and q are semantically equivalent in each of the programs in the sequence Γ.

A Horn clause C is written as $A \text{ :- } B_1, B_2, \ldots, B_n$. A, the consequent, is called the *head* of C and the antecedent $B_1, B_2, \ldots, B_n$ the *body* of C. Note that we can write Horn clauses as $A \text{ :- } \mathcal{G}$. Semantics of a definite (*i.e.* positive) logic program P is given by its least Herbrand model, denoted as $M(P)$. Semantics of a normal logic program (*i.e.* a logic program with negative literals in the bodies of its clauses) is given by its well-founded model [GRS91], stable model [GL88] etc. Throughout the dissertation we use CTL, a branching time temporal logic to represent temporal properties. For a detailed discussion on CTL refer [CGP99, Eme90].

Chapter 3

Transformations for Definite Logic Programs

3.1 Background

Some of the most extensively studied transformation systems for definite logic programs are the so called *unfold/fold* transformation systems. At a high level unfold and fold transformations can be viewed as follows. Definite logic programs consist of definitions of the form $A :- \phi$ where A is an atom and ϕ is a positive boolean formula over atoms. Unfolding replaces an occurrence of A in a program with ϕ while folding replaces an occurrence of ϕ with A. Folding is called *reversible* if its effects can be undone by an unfolding, and *irreversible* otherwise.

Correctness proofs for unfold/fold transformations consider *transformation sequences* of the form $P_0, P_1, \ldots,$ where P_0 is an initial program and P_{i+1} is obtained from P_i by applying an unfolding or folding transformation. The proofs usually show that all programs in the transformation sequence have the same least Herbrand model. It is easy to verify that transforming P_i to P_{i+1} using unfolding or folding is *partially correct*, i.e., the least model of P_{i+1} is a subset of that of P_i. It is also easy to show, by induction on the structure of the proof trees, that unfolding transformation is *totally* correct, i.e., it *preserves* the least model. However, indiscriminate folding may introduce circularity in definitions, thereby replacing finite proof paths with infinite ones. Indiscriminate folding may not preserve program semantics (see Figure 5 in Chapter 2.2.3 for illustration).

The crucial problem in proving correctness of any sequence of unfold/fold transformations lies in the potential circularities introduced by indiscriminate folding. These circularities remove existing finite proof paths, thereby altering the least model of the logic program. To avoid this problem, existing transformation systems ([TS84, TS86a, KF87, GK94] to name a few) impose syntactic restrictions on the clauses participating in a folding step. In this chapter, we present a transformation system which relaxes these restrictions, and prove its total correctness.

To generalize in this direction one needs to first understand the strengths and limitations of the above systems. The key observation is that, although the restrictions needed to determine permissible foldings appear radically different in the different systems, there is a striking similarity in how the transformations are proved correct. Essentially, these systems associate some *measure* with different program elements, namely, atoms and clauses to determine whether folding is permissible in that step (e.g., "foldable" flag in [TS84], descent levels/strata numbers in [TS86a], and counters in [KF87]). Moreover, they ensure that each transformation step maintains an invariant relating proofs in the derived program to the various measures (e.g., the notions of rank-consistency in [KF87, TS84], weight-consistency in [GK94] and μ-completeness in [TS86a]). This raises another interesting question: can we exploit the similarities in the correctness proofs of irreversible unfold/fold systems to develop an abstract framework. Such a framework will specify the obligations that must be satisfied to ensure total correctness and hence can simplify construction of unfold/fold systems to the extent that one is relieved of the burden of giving correctness proofs. We propose such a framework in this chapter.

Summary of Results In this chapter, we develop a general transformation framework for definite logic programs parameterized by certain abstract measures by suitably abstracting and extending the measures used in [GK94, KF87, TS84, TS86a] (see Section 3.2). We relax the invariants needed in the proofs to permit *approximation* of measure values. This is the key idea that enables us to fold using multiple recursive clauses, *i.e.* relax the restrictions on clause syntax. We prove the correctness of transformations in the framework based only on the properties of the abstract measures. We show that various existing unfold/fold transformation systems can be derived from the framework by instantiating these abstract measures (see Section 3.3). We also show how the framework can be extended to include the Goal Replacement transformation (see Section 3.4).

The parameterized framework presented in this chapter is useful for understanding the strengths and limitations of existing transformation systems. It also enables the construction of new unfold/fold systems. As evidence we describe SCOUT (Strata and COunter based Unfold/fold Transformations), a new transformation system that permits disjunctive folding using recursive clauses. The development of SCOUT was based on two crucial observations made possible by the framework. First, when instantiating the framework to obtain the Kanamori-Fujita system, it is easy to see that the counters (the measure used in their system) may come from any linearly ordered set; this permits us to incorporate stratification into the counters to obtain a system that generalizes the extended Tamaki-Sato system [TS86a] as well as the Kanamori-Fujita system. Secondly, the framework enables us to maintain approximate counters; we can hence generalize the combination of the Kanamori-Fujita and the extended Tamaki-Sato systems to fold using multiple recursive clauses.

3.2 A Parameterized Transformation Framework

We now describe our parameterized unfold/fold transformation framework and illustrate the abstractions by drawing analogies to the Kanamori-Fujita system [KF87]. We will use the following symbols (possibly with primes and subscripts): P to denote a definite logic program; $M(P)$ its least Herbrand model; C and D for clauses; A, B to denote atoms and literals and σ for mgu.

3.2.1 Unfolding and Folding

The unfolding and folding rules are defined as follows:

Transformation 3.1 (Unfolding) Let C be a clause in P_i and A an atom in the body of C. Let $C_1, \ldots, C_m$ be the clauses in P_i whose heads are unifiable with A with most general unifier $\sigma_1, \ldots, \sigma_m$. Let C'_j be the clause that is obtained by replacing $A\sigma_j$ by the body of $C_j\sigma_j$ in $C\sigma_j$ $(1 \leq j \leq m)$. Assign $(P_i - \{C\}) \cup \{C'_1, \ldots, C'_m\}$ to P_{i+1}. □

Transformation 3.2 (Folding) Let $\{C_1, \ldots, C_m\} \subseteq P_i$ where C_l denotes the clause $A :- A_{l,1}, \ldots, A_{l,n_l}, A'_1, \ldots, A'_n$, and $\{D_1, \ldots, D_m\} \subseteq P_j$ $(j \leq i)$ where D_l is the clause $B_l :- B_{l,1}, \ldots, B_{l,n_l}$. Further, let:
1. $\forall 1 \leq l \leq m\ \exists \sigma_l\ \forall 1 \leq k \leq n_l\ A_{l,k} = B_{l,k}\sigma_l$

2. $B_1\sigma_1 = B_2\sigma_2 = \cdots = B_m\sigma_m = B$
3. $D_1, \ldots, D_m$ are the only clauses in P_j whose heads are unifiable with B.
4. $\forall 1 \leq l \leq m$, σ_l substitutes the internal variables[1] of D_l to distinct variables which do not appear in $\{A, B, A'_1, \ldots A'_n\}$.

Then $P_{i+1} := (P_i - \{C_1, \ldots, C_m\}) \cup \{C'\}$ where $C' \equiv A :- B, A'_1, \ldots, A'_n$. □

$D_1, \ldots, D_m$ are the *folder* clauses, $C_1, \ldots, C_m$ are the *folded* clauses, and B is the *folder* atom. A folding step is *conjunctive* whenever both the folder and folded clauses are singleton sets and is *disjunctive* otherwise. Note that in the latter step a set of folded clauses is *simultaneously* replaced by a single clause using a set of folder clauses. We say that $P_0, P_1, \ldots, P_n$ is an unfold/fold transformation sequence if the program P_{i+1} is obtained from P_i $(i \geq 0)$ by application of an unfold or a fold rule. Partial correctness of an unfold/fold transformation sequence (Theorem 3.1) now follows straightforwardly by induction on the structure of ground proofs (refer Definition 2.14, page 10).

Theorem 3.1 (Partial Correctness) *Let $P_0, P_1, \ldots, P_i$ be a program transformation sequence where $M(P_j) = M(P_0)$ for all $0 \leq j \leq i$. If P_{i+1} is obtained from P_i by applying either unfolding or folding, then $M(P_{i+1}) \subseteq M(P_i)$.* □

Proof: This is established by showing that a proof T of any ground atom $A \in M(P_{i+1})$, has a corresponding proof T' of A in P_i. This can be proved by induction on the structure of T. Let $C = (A :- A_1, \ldots, A_n)$ be the clause applied at the root of T. There are three cases:

1. $C \in P_i$
2. C is obtained by unfolding
3. C is obtained by folding

For case 1, the result follows by induction hypothesis. For case 2, let $C \in P_{i+1}$ be obtained by unfolding clause $C' \in P_i$ using clause $D \in P_i$. Without loss of generality, there exist ground instances of C' and D of the form $A :- B, A_{k+1}, \ldots, A_n$ and $B :- A_1, \ldots, A_k$. The proof T' of A can be then constructed by applying clause C' at the root, and then clause D. The existence of ground proofs of $A_1, \ldots, A_n$ in P_i follows by induction hypothesis. For case 3, let $C \in P_{i+1}$ be obtained by folding

[1] Variables appearing in the body of a clause, but not its head

$C' \in P_i$ using $D \in P_j (j \leq i)$ as folder. Let A_1 be the folder atom in clause C, *i.e.* the atom introduced by folding. Since $M(P_j) = M(P_i)$ and $A_1 \in M(P_i)$ (by induction hypothesis) therefore $A_1 \in M(P_j)$. Thus, A_1 has a ground proof T_1 in P_j. By condition 3 of the folding transformation, the clause applied at the root of T_1 *must* be one of the folder clauses. Let this folder clause be D and let the corresponding folded clause be $C' \in P_i$. Then, without loss of generality, C' and D have ground instances of the form $A :- A_{1,1}, \ldots, A_{1,l}, A_2, \ldots, A_n$ and $A_1 :- A_{1,1}, \ldots, A_{1,l}$ respectively. Since $A_{1,1}, \ldots, A_{1,l} \in M(P_j)$ therefore $A_{1,1}, \ldots, A_{1,l} \in M(P_i)$. Thus, $A_{1,1}, \ldots, A_{1,l}$ have ground proofs in P_i. Also, $A_2, \ldots, A_n$ have ground proofs in P_i by induction hypothesis. Thus, we can construct a ground proof of A in P_i by applying clause C' at the root. This completes the proof. □

3.2.2 Measure-Consistent Proofs and Total Correctness

Total correctness of an unfold/fold transformation sequence is established by inducting on some well-founded order to construct a proof in P_{i+1} for any atom A in $M(P_i)$. To see the subtleties in showing total correctness, consider transforming P_i to P_{i+1} using a conjunctive folding step. To construct a proof of A (the head of the folded clause) in P_{i+1}, we need a proof of B (the folder atom) in P_{i+1}. But the existence of such a proof can be established (by induction hypothesis) only if B is less than A in the well-founded order on which we are inducting. Note that if the folder clause is picked from P_j, $j < i$, we cannot use simple well-founded orders like size of proof trees in P_i, since proof of B in P_i can be larger in size than the proof of A in P_i. Here we develop an abstract formulation of certain well-founded orders (which we call *measures*) on which we can induct to establish total correctness.

It is worth noting that we do not attempt to translate every proof of A in P_i to a proof of A in P_{i+1}. Instead, following [KF87, TS84, TS86a] we consider a "special proof" called *strongly measure-consistent proof* (see Definition 3.5) of A in P_i and construct a proof of A in P_{i+1}. The induction proof for establishing total correctness is completed by showing that the proof of A in P_{i+1} thus constructed is itself strongly measure consistent.

Recall that irreversible folding steps need to be constrained in order to preserve the semantics. In order to enforce these constraints, we maintain some book-keeping information as we perform the transformations, formalized using the following notions of *Measure structure*, *Atom measure*, and *Clause measure*.

Definition 3.1 (Measure Structure) *A Measure Structure is defined as a 4-tuple* $\mu = \langle \mathcal{M}, \oplus, \prec, \mathcal{W} \rangle$ *where* $\langle \mathcal{M}, \oplus \rangle$ *is a commutative group with* $\mathbf{0} \in \mathcal{M}$ *as its identity element,* $\prec$ *is a linear order on* $\mathcal{M}$*,* $\oplus$ *is monotone w.r.t.* $\prec$*, and* $\mathcal{W}$ *is a subset of* $\{x \in \mathcal{M} \mid \mathbf{0} \preceq x\}$*, over which* $\prec$ *is well-founded.*

We will refer to $\mathcal{M}$, the first component of the measure structure, as the *measure space*. We let $\preceq$ denote $\prec$ or $=$. Moreover, we use $\ominus$ to denote the inverse operation of the group $\langle \mathcal{M}, \oplus \rangle$. We also use $\ominus$ as a binary operator, $a \ominus b$ meaning $a \oplus (\ominus b)$ (where $(\ominus b)$ is the inverse of b). The Kanamori-Fujita system [KF87] keeps track of integer counters. Thus the measure structure is $\langle \mathbb{Z}, +, <, \mathbb{N} \rangle$, where $\mathbb{Z}$ and $\mathbb{N}$ are the set of integers and natural numbers respectively, $+$ denotes integer addition, and $<$ is the arithmetic comparison operator.

Definition 3.2 (Atom Measure) *An atom measure* α *of a program* P *w.r.t. a measure structure* μ *is a partial function from the Herbrand base of* P *to* $\mathcal{W}$ *such that it is total on the least Herbrand model of* P*. For our purposes, it suffices to use the same atom measure for each program in a transformation sequence.*

In the Kanamori-Fujita system, the atom measure of any P_i in the transformation sequence is the number of nodes in the shortest proof tree of A in the initial program P_0. The proof of total correctness for folding will induct on the atom measure, relating the atom measure of A (the head of the folded clauses) with the atom measure of B (the folder atom).

Definition 3.3 (Clause Measure) *A clause measure* $(\gamma_{lo}, \gamma_{hi})$ *of a program* P *w.r.t. a measure structure* μ *is a pair of total functions from clauses of* P *to* $\mathcal{M}$ *such that* $\forall C \in P \ \gamma_{lo}(C) \preceq \gamma_{hi}(C)$*.*

In the Kanamori-Fujita system, γ_{lo} and γ_{hi} are the same and map each clause to its corresponding counter value. However, as we will see later, to allow disjunctive folding we will need the two distinct functions γ_{lo} and γ_{hi}. Henceforth, we denote the clause measure of a program P_i by $(\gamma^i_{lo}, \gamma^i_{hi})$. We will now develop the idea of "special proofs" mentioned earlier. Recall that the notion of a (ground) proof is formally stated in Definition 2.14. Consider transforming P_i to P_{i+1} by a folding step (see figure below). C and D are the folded and folder clauses respectively and $j < i$.

.....	.....	.....
$D:$ $\mathtt{q} :\!- \mathtt{q_1}, ..., \mathtt{q_k}$	$C:$ $\mathtt{p} :\!- \mathtt{q_1}, ..., \mathtt{q_k}, \mathtt{q_{k+1}}, ..., \mathtt{q_n}$	$C':$ $\mathtt{p} :\!- \mathtt{q}, \mathtt{q_{k+1}}, ..., \mathtt{q_n}$
.....	.....	.....
Program P_j	Program P_i	Program P_{i+1}

In order to show that $\mathtt{p} \in M(P_i) \Rightarrow \mathtt{p} \in M(P_{i+1})$ by induction on $\prec$, we would like to show that $\alpha(\mathtt{q}) \prec \alpha(\mathtt{p})$. The atoms $\mathtt{p}$ and $\mathtt{q}$ are related by what is shared between the bodies of the clauses C and D. Hence we attempt to relate their measures via the measures of bodies of C and D. Suppose D satisfies : (i) $\alpha(\mathtt{q}) \preceq \sum_{1 \leq i \leq k} \alpha(\mathtt{q_i})$, then we can relate $\alpha(\mathtt{q})$ to the sum of the measures of the body atoms of the folded clause C (since $k \leq n$). Further if C satisfies : (ii) $\alpha(\mathtt{p}) \succeq \sum_{1 \leq i \leq n} \alpha(\mathtt{q_i})$, then we can establish that $\alpha(\mathtt{q}) \preceq \alpha(\mathtt{p})$. If either (i) or (ii) is a strict relationship then we can establish that $\alpha(\mathtt{q}) \prec \alpha(\mathtt{p})$. Relations (i) and (ii) form the basis for the notions of *weak* and *strong measure consistency.*

Definition 3.4 (Weakly Measure Consistent Proof) *A ground proof T in program P_i is weakly measure consistent w.r.t. atom measure α and clause measure $(\gamma^i_{lo}, \gamma^i_{hi})$ if every ground instance $A :- A_1, ..., A_n$ of a clause $C \in P_i$ used in T satisfies $\alpha(A) \preceq \gamma^i_{hi}(C) \oplus \sum_{1 \leq l \leq n} \alpha(A_l)$.*

Definition 3.5 (Strongly Measure Consistent Proof) *A ground proof T in program P_i is strongly measure consistent w.r.t. atom measure α and clause measure $(\gamma^i_{lo}, \gamma^i_{hi})$ if every ground instance $A :- A_1, ..., A_n$ of a clause $C \in P_i$ used in T satisfies $\forall 1 \leq l \leq n\ \alpha(A_l) \prec \alpha(A)$ and $\alpha(A) \succeq \gamma^i_{lo}(C) \oplus \sum_{1 \leq l \leq n} \alpha(A_l)$*

Definition 3.6 (Measure Consistent Proof) *A ground proof T in program P_i is said to be measure consistent w.r.t. atom measure α and clause measure $(\gamma^i_{lo}, \gamma^i_{hi})$, if it is strongly and weakly measure consistent w.r.t. α and $(\gamma^i_{lo}, \gamma^i_{hi})$.*

We point out that our abstract notion of measure consistency relaxes the concrete notion of rank consistency of [KF87]. While rank consistency of [KF87] imposes a strict equality constraint on $\alpha(A)$, measure consistency only *bounds it from above and below.* As we will show later, this facilitates maintenance of approximate information. This is the central idea that permits us to do disjunctive folding using recursive clauses. For proving total correctness, we need :

Definition 3.7 (Measure consistent Program) *A program P is measure consistent w.r.t. atom measure α and clause measure $(\gamma_{lo}, \gamma_{hi})$, if for all $A \in M(P)$, we have : (1) All ground proofs of A in P are weakly measure consistent w.r.t. α and $(\gamma_{lo}, \gamma_{hi})$ (2) A has a ground proof in P which is strongly measure consistent w.r.t. α and $(\gamma_{lo}, \gamma_{hi})$*

We are now ready to define the abstract conditions on folding and constraints on how the clause measures are to be updated after an unfold/fold step. For each clause C obtained by applying an unfold/fold transformation on program P_i, we derive a lower bound on $\gamma_{hi}^{i+1}(C)$ and an upper bound on $\gamma_{lo}^{i+1}(C)$, denoted by $GLB^{i+1}(C)$ and $LUB^{i+1}(C)$ respectively. We will see later that the conditions on when the rules become applicable, as well as these bounds will be based on the requirements of the proof of total correctness.

We assume that for any atom A (not necessarily ground), $\alpha_{min}(A)$ denotes a lower bound on the measure of any provable ground instantiation of A *i.e.* $\forall\theta\ \alpha_{min}(A) \preceq \alpha(A\theta)$. We use α_{min} in the folding condition of rule 3.4 below.

Transformation 3.3 (Measure Preserving Unfolding) Let P_{i+1} be obtained from P_i by an unfolding transformation as described in Rule 3.1. Then, $\forall 1 \leq j \leq m$

$$\gamma_{lo}^{i+1}(C'_j) \preceq GLB^{i+1}(C'_j) = \gamma_{lo}^{i}(C) \oplus \gamma_{lo}^{i}(C_j) \quad (1)$$

$$\gamma_{hi}^{i+1}(C'_j) \succeq LUB^{i+1}(C'_j) = \gamma_{hi}^{i}(C) \oplus \gamma_{hi}^{i}(C_j) \quad (2)$$

The clause measure of all other clauses in P_{i+1} are inherited from P_i. □

Transformation 3.4 (Measure Preserving Folding) Let P_{i+1} be obtained from P_i by a folding transformation as described in Rule 3.2, s.t. $\forall 1 \leq l \leq m$ we have $\gamma_{hi}^{j}(D_l) \prec \gamma_{lo}^{i}(C_l) \oplus \sum_{1\leq k\leq n} \alpha_{min}(A'_k)$. Then,

$$\gamma_{lo}^{i+1}(C') \preceq GLB^{i+1}(C') = \min_{1\leq l\leq m} (\ \gamma_{lo}^{i}(C_l) \ominus \gamma_{hi}^{j}(D_l)\ ) \quad (3)$$

$$\gamma_{hi}^{i+1}(C') \succeq LUB^{i+1}(C') = \max_{1\leq l\leq m} (\gamma_{hi}^{i}(C_l) \ominus \gamma_{lo}^{j}(D_l)) \quad (4)$$

and the clause measure of all other clauses in P_{i+1} are inherited from P_i. □

Intuitively, if the clause measure of C_l "exceeds" the clause measure of D_l then we can fold C_l using D_l. It should be noted that the above rules do not prescribe *unique* values for upper and lower clause measures for the clauses generated by the transformations. Instead, they only specify bounds of these values; the values themselves are chosen only when instantiating the framework to a concrete system.

Observe from the definition of atom measures that we can always assign $\mathbf{0}$ to α_{min}. However, by setting a more accurate estimate of α_{min}, we can allow more folding steps. As an example, consider any conjunctive folding step where the folded clause $C \in P_i$ has more body atoms than the folder clause $D \in P_j$, and $\gamma_{lo}^{i}(C) = \gamma_{hi}^{j}(D)$. Such a folding step will not be allowed if $\forall A\ \alpha_{min}(A) = \mathbf{0}$.

The Need for Approximate Clause Measures : In the Kanamori-Fujita system, a counter (corresponding to our clause measure) is associated with every clause. Roughly speaking, the counter associated with a clause $C \in P_i$ where $C \equiv A :- A_1, \ldots, A_n$ indicates the number of interior nodes in the smallest proof tree in P_0 that derives $A_1, \ldots, A_n$ from A. Thus, it is the amount saved (in terms of proof tree size, compared to the smallest proof in P_0) whenever C is used in a proof in P_i. The folding rule is applicable provided the savings accrued in the folded clause is more than that in the folder clause.

To see why a single counter is inadequate for disjunctive folding, consider the following example:

C_1: `p :- r, t.` (x_1)	
C_2: `p :- s, t.` (x_2)	C': `p :- q, t.` $(?)$
C_3: `q :- r.` (x_3)	C_3: `q :- r.` (x_3)
C_4: `q :- s.` (x_4)	C_4: `q :- s.` (x_4)
Program P_i	Program P_{i+1}

P_{i+1} is obtained from P_i by folding $\{C_3, C_4\}$ into $\{C_1, C_2\}$. Now, the savings due to C' in a proof of P_{i+1} depends on whether C_3 or C_4 is used to resolve `q` in that proof. Since this information is unknown at transformation time, we can only keep approximate information about savings. In our framework we choose to approximate the savings by the closed interval $[\gamma_{lo}, \gamma_{hi}]$.

We now have the necessary machinery for establishing total correctness of a sequence of unfold/fold transformations.

Lemma 3.2 (Preserving Weak Measure Consistency) *Let $P_0, \ldots, P_i$ be a transformation sequence of measure consistent programs such that $M(P_0) = M(P_j)$ for all $0 \leq j \leq i$. Let P_{i+1} be obtained from P_i by applying measure-preserving unfolding or measure-preserving folding. Then, all ground proofs of P_{i+1} are weakly measure consistent.*

Proof : We will use $M(P_{i+1}) \subseteq M(P_i)$, a result which was independently proved in theorem 3.1. The proof proceeds by induction on size of ground proofs in P_{i+1}. Let T be a ground proof of some ground atom A in P_{i+1}, and let $A :- A_1, \ldots, A_n$ (where $n \geq 0$) be the ground instance of a clause $C \in P_{i+1}$ that is used at the root of the proof T. Then the proofs of $A_1, \ldots, A_n$ in T are weakly measure consistent by induction hypothesis. Hence, it suffices to show that, $\alpha(A) \preceq \gamma_{hi}^{i+1}(C) \oplus \sum_{1 \leq l \leq n} \alpha(A_l)$.

Case 1: C was inherited from P_i
Since $M(P_{i+1}) \subseteq M(P_i)$, hence $A_1, \ldots, A_n$ are provable in P_i. Therefore, the ground clause $A :- A_1, \ldots, A_n$ is used at the root of a ground proof in P_i. Since P_i is measure consistent, the result follows.

Case 2: C was obtained by *unfolding*
Let $A_1, \ldots, A_k$ be the instances of the body atoms of C which were introduced through unfolding. By the definition of the unfolding transformation, then there must be clauses C' and C'' in P_i with ground instances $A :- B, A_{k+1}, \ldots, A_n$ and $B :- A_1, \ldots, A_k$ respectively with $\gamma_{hi}^{i+1}(C) \succeq \gamma_{hi}^{i}(C') \oplus \gamma_{hi}^{i}(C'')$. Again, $A_1, \ldots, A_k, A_{k+1}, \ldots, A_n$ are provable in P_i (as $M(P_{i+1}) \subseteq M(P_i)$). Hence, the above mentioned ground instances of C' and C'' are ground clauses used at the root of some proof in P_i. As P_i is a measure consistent program, we have :

$$\alpha(A) \preceq \gamma_{hi}^{i}(C') \oplus \alpha(B) \oplus \sum_{k+1 \leq l \leq n} \alpha(A_l)$$

$$\alpha(B) \preceq \gamma_{hi}^{i}(C'') \oplus \sum_{1 \leq l \leq k} \alpha(A_l)$$

The result now follows by combining these two inequations.

Case 3: C was obtained by *folding*
Let A_1 be the instance of the folder atom (*i.e.* the atom corresponding to the head of the folder clauses) in C, and let $P_j (j \leq i)$ be the program from which folder clauses were picked. We have $M(P_i) = M(P_j) = M(P_0)$, and hence $M(P_{i+1}) \subseteq M(P_j)$. Thus, $A_1 \in M(P_j)$. Since P_j is a measure consistent program, A_1 must have a strongly measure consistent proof T'_{A_1} in P_j. Let the clause used at the root of this proof be D' and let the ground instance of D' used at the root of T'_{A_1} be $A_1 :- A_{1,1}, \ldots, A_{1,k}$. Then, by the strong measure consistency of T'_{A_1}

$$\alpha(A_1) \succeq \gamma_{lo}^{j}(D') \oplus \sum_{1 \leq l \leq k} \alpha(A_{1,l})$$

But, D' must be a folder clause by definition of folding. Hence, there must be a clause C' in P_i with a ground instance $A :- A_{1,1}, \ldots, A_{1,k}, A_2, \ldots, A_n$ (this is the folded clause corresponding to D'). Now, $A_2, \ldots, A_n$ are provable in P_i (since $M(P_{i+1}) \subseteq M(P_i)$), and also $A_{1,1}, \ldots, A_{1,k}$ are provable in P_i (since $M(P_j) = M(P_i)$). Therefore, the above mentioned ground instance of C' is used at the root of a weakly measure

consistent proof of A in P_i (since program P_i is measure consistent). Hence

$$\begin{aligned}\alpha(A) &\preceq \gamma^i_{hi}(C') \oplus \sum_{1\le l\le k} \alpha(A_{1,l}) \oplus \sum_{2\le l\le n} \alpha(A_l) \\ &\preceq \gamma^i_{hi}(C') \ominus \gamma^j_{lo}(D') \oplus \alpha(A_1) \oplus \sum_{2\le l\le n} \alpha(A_l) \\ &\preceq \gamma^i_{hi}(C') \ominus \gamma^j_{lo}(D') \oplus \sum_{1\le l\le n} \alpha(A_l)\end{aligned}$$

Since D' and C' are folder and folded clauses and C is the clause obtained by folding therefore $\gamma^{i+1}_{hi}(C) \succeq \gamma^i_{hi}(C') \ominus \gamma^j_{lo}(D')$, and hence

$$\alpha(A) \preceq \gamma^{i+1}_{hi}(C) \oplus \textstyle\sum_{1\le l\le n} \alpha(A_l)$$

Thus, we have established that any arbitrary ground proof T in P_{i+1} is weakly measure consistent. □

We now prove our main correctness result : any interleaved application of measure preserving unfolding or folding transformations does not alter program semantics.

Theorem 3.3 (Total Correctness) *Let $P_0, P_1, \ldots, P_i$ be a transformation sequence of measure consistent programs such that $M(P_0) = M(P_j)$ for all $0 \le j \le i$. Let P_{i+1} be obtained from P_i by applying measure-preserving unfolding or measure-preserving folding. Then,* (i) *$M(P_{i+1}) = M(P_i)$ and* (ii) *P_{i+1} is a measure-consistent program.*

Proof: By theorem 3.1, we have $M(P_{i+1}) \subseteq M(P_i)$, and by lemma 3.2 we know that all ground proofs of P_{i+1} are weakly measure consistent. Hence it is sufficient to prove that (1) $M(P_i) \subseteq M(P_{i+1})$ and (2) $\forall A \in M(P_{i+1})$, A has a strongly measure consistent proof in P_{i+1}.

Consider any ground atom $A \in M(P_i)$. Since P_i is measure consistent, A has a strongly measure consistent proof T in P_i. We now construct a strongly measure consistent proof T' of A in P_{i+1}. Construction of T' proceeds by induction on atom measures. Let C be a clause used at the root of T. Let $A :- A_1, ..., A_n$ (where $n \ge 0$) be the ground instantiation of C at the root of T. Since T is strongly measure consistent $\alpha(A_i) \prec \alpha(A)$, for all $1 \le i \le n$. Hence, we have strongly measure consistent proofs $T'_1, ..., T'_n$ of $A_1, ..., A_n$ in P_{i+1}. We construct T' by considering the following cases:

Case 1: C is *inherited* from P_i into P_{i+1}

T' is constructed with $A :- A_1, ..., A_n$ at its root and $T'_1, ..., T'_n$ as its children. This proof T' is strongly measure consistent.

Case 2: C is *unfolded*.

Let A_1 be the atom in the body of C which is unfolded. Let the clause used to resolve A_1 in T be C_1 and the ground instance of C_1 used be $A_1 :- A_{1,1}, ..., A_{1,l_1}$. By definition of unfolding, $A :- A_{1,1}, ..., A_{1,l_1}, A_2, ..., A_n$ is a ground instance of a clause C'_1 in P_{i+1} with $\gamma^{i+1}_{lo}(C'_1) \preceq \gamma^i_{lo}(C) \oplus \gamma^i_{lo}(C_1)$. Also, $\alpha(A_{1,j}) \prec \alpha(A_1) \prec \alpha(A)$, for all $1 \leq j \leq l_1$. Thus, we have strongly measure consistent proofs $T'_{1,1}, ..., T'_{1,l_1}$ of $A_{1,1}, ..., A_{1,l_1}$ in P_{i+1}. The proof T' is now constructed by applying $A :- A_{1,1}, ..., A_{1,l_1}, A_2, ..., A_n$ at the root, and putting $T'_{1,1}, ..., T'_{1,l_1}, T'_2, ..., T'_n$ as the children. Since T is strongly measure consistent,

$$\begin{array}{ll} & \alpha(A) \succeq \gamma^i_{lo}(C) \oplus \sum_{1\leq j\leq n} \alpha(A_j) \text{ and } \alpha(A_1) \succeq \gamma^i_{lo}(C_1) \oplus \sum_{1\leq j\leq l_1} \alpha(A_{1,j}) \\ \Longrightarrow & (\alpha(A) \oplus \alpha(A_1)) \succeq \gamma^i_{lo}(C) \oplus \gamma^i_{lo}(C_1) \oplus \sum_{1\leq j\leq n} \alpha(A_j) \oplus \sum_{1\leq j\leq l_1} \alpha(A_{1,j}) \\ \Longrightarrow & \alpha(A) \succeq \gamma^{i+1}_{lo}(C'_1) \oplus \sum_{2\leq j\leq n} \alpha(A_j) \oplus \sum_{1\leq j\leq l_1} \alpha(A_{1,j}) \end{array}$$

Hence, T' is a strongly measure consistent proof in P_{i+1}.

Case 3: C is *folded*.

Let C (potentially with other clauses) be folded, using folder clauses from P_j, $j \leq i$, to clause C' in P_{i+1}. Assume that $A_1, ..., A_k$ are the instances of the folded atoms in C. Then, C' has a ground instance of the form $A :- B, A_{k+1}, ..., A_n$ where $B :- A_1, ..., A_k$ is a ground instance of a folder clause $D \in P_j$.[2] Since $M(P_i) = M(P_j)$ and $A_1, ..., A_k$ are provable in P_i they must also be provable in P_j. Moreover, since $D \in P_j$, $B \in M(P_j) = M(P_i)$. Since P_j is measure consistent, $\alpha(B) \preceq \gamma^j_{hi}(D) \oplus \sum_{1\leq l\leq k} \alpha(A_l)$.

Now, by the strong measure consistency of T,

$$\begin{array}{rcll} \alpha(A) & \succeq & \gamma^i_{lo}(C) \oplus \sum_{1\leq l\leq k} \alpha(A_l) \oplus \sum_{k+1\leq l\leq n} \alpha(A_l) & \\ & \succeq & \gamma^i_{lo}(C) \oplus (\alpha(B) \ominus \gamma^j_{hi}(D)) \oplus \sum_{k+1\leq l\leq n} \alpha(A_l) & \cdots \quad \cdots (*) \\ & \succeq & (\gamma^i_{lo}(C) \ominus \gamma^j_{hi}(D)) \oplus \alpha(B) \oplus \sum_{k+1\leq l\leq n} \alpha_{min}(A_l) & \\ & \succ & \alpha(B) \text{ (by condition of measure preserving folding)} & \end{array}$$

Now, by induction hypothesis, B has a strongly measure consistent proof T'_B in P_{i+1}. We construct T', the proof of A in P_{i+1}, with $A :- B, A_{k+1}, ..., A_n$ at its root, and $T'_B, T'_{k+1}, ..., T'_n$ as its children. To show that T' is strongly measure consistent, note

[2] Recall that in the folding transformation, all clauses in P_j whose head is unifiable with B are folder clauses.

that $\gamma_{lo}^{i+1}(C') \preceq (\gamma_{lo}^{i}(C) \ominus \gamma_{hi}^{j}(D))$ according to the definition of measure preserving folding, as C and D are folded and folder clauses. Combining this with (*) we get,

$$\alpha(A) \succeq \gamma_{lo}^{i+1}(C') \oplus \alpha(B) \oplus \textstyle\sum_{k+1 \leq l \leq n} \alpha(A_l)$$

This completes the proof. □

Assigning tighter clause measures The measure preserving unfolding and folding transformations of Rules 3.3, 3.4 provide constraints on the clause measures in P_{i+1}. Note that by applying measure preserving unfolding/folding to program P_i we can generate a clause C which is already in P_i, but with new clause measures. Instead of assigning the clause measures as prescribed by Rules 3.3 and 3.4 (computed via addition/subtraction), we can assign tighter measures as follows. Formally, let $unfold(C')$ be the set of clauses generated by measure preserving unfolding of $C' \in P_i$ and let there exist a clause C s.t. $C \in unfold(C') \wedge C \in P_i - \{C'\}$. Clearly, then $C \in P_{i+1}$. However, the question is how do we assign $(\gamma_{lo}^{i+1}(C), \gamma_{hi}^{i+1}(C))$, the clause measures of C in P_{i+1}. Similarly, by measure preserving folding of $\{C_1, \ldots, C_m\} \subseteq P_i$, we can generate a clause $C \in P_i - \{C_1, \ldots, C_m\}$. Again, we need to assign $(\gamma_{lo}^{i+1}(C), \gamma_{hi}^{i+1}(C))$. Let the clause measures of C computed by the unfold/fold transformation be $(\gamma'_{lo}, \gamma'_{hi})$. We can then set $\gamma_{lo}^{i+1}(C) = \min(\gamma'_{lo}, \gamma_{lo}^{i}(C))$ and $\gamma_{hi}^{i+1}(C) = \min(\gamma'_{hi}, \gamma_{hi}^{i}(C))$ without affecting the measure consistency of P_{i+1}. For the purposes of measure consistency, note that we could have chosen $\gamma_{hi}^{i+1}(C) = \max(\gamma'_{hi}, \gamma_{hi}^{i}(C))$. Taking the minimum, which also preserves measure consistency, gives us a tighter bound. This also ensures that when we restrict ourselves to conjunctive folding, the lower and higher measures of any clause in program P_i (appearing in some transformation sequence of measure consistent programs $P_0, P_1, ...$) are identical.

3.3 Constructing Concrete Unfold/Fold Systems

To construct a concrete unfold/fold transformation system from our abstract framework, the following parameters need to be instantiated :

1. a measure structure μ;
2. atom measure α and α_{min};

3. clause measure $(\gamma^0_{lo}, \gamma^0_{hi})$ for clauses in the initial program P_0 such that P_0 is measure consistent; and

4. functions to compute the clause measure of new clauses obtained by the transformations such that they satisfy the constraints imposed by equations (1) through (4) (refer Rules 3.3 and 3.4).

Note that there are *no further* proof obligations. Once the above four elements are defined, total correctness of the transformation system is *guaranteed* by the framework.

3.3.1 Existing Unfold/fold Systems

We first show how our framework can be instantiated to obtain the Kanamori-Fujita and the extended Tamaki-Sato systems. To the best of our knowledge, these are the only two existing systems that allow folding using recursive clauses. However in both of these systems folding is conjunctive.

The Kanamori-Fujita System [KF87]: This system can be obtained as an instance of our framework as follows:

1. $\mu = \langle \mathbb{Z}, +, <, \mathbb{N} \rangle$. This measure structure corresponds to the use of integer counters in [KF87].

2. $\alpha(A)$ = number of nodes in the smallest proof of A in P_0, and for any atom A, $\alpha_{min}(A) = 1$. Thus, $\alpha(A)$ denotes the *rank* of A described in [KF87].

3. $\forall C \in P_0 \ \gamma^0_{lo}(C) = \gamma^0_{hi}(C) = 1$. Since all clause measures are 1, it follows immediately from the definition of atom measures that the smallest proofs of any ground goal G are strongly measure consistent, and all proofs in P_0 are weakly measure consistent. Hence P_0 is measure consistent.

4. $\forall C \in P_{i+1} - P_i$ (i.e., new clauses in P_{i+1}), $\gamma^{i+1}_{lo}(C) = GLB^{i+1}(C)$ and $\gamma^{i+1}_{hi}(C) = LUB^{i+1}(C)$. Under the given measure structure, it is immediate that the above definition is identical to the computation on counters in [KF87].

Furthermore, the measure preserving folding rule (Rule 3.4) is applied only when both folder and folded clauses are singleton sets. It is easy to see a one-to-one correspondence between the conditions on unfold/fold transformations of the above instantiation and the Kanamori-Fujita system.

The Extended Tamaki-Sato System [TS86a]: In this system all the predicate symbols are partitioned into n strata. In the initial program a predicate from stratum j is defined using only predicates from strata $\leq j$. We can obtain this system as an instance of our framework as follows:

1. $\mu = \langle \mathbb{Z}^n, \oplus, \prec, \mathbb{N}^n \rangle$ where $\oplus$ denotes coordinate-wise integer addition of n-tuples of integers, and $\prec$ denotes the lexicographic $<$ order over n-tuples of integers. The n-tuples in the measure structure will correspond to the n strata of the original program.

2. $\alpha(A) = \min(\{w(T) \mid T \text{ is a proof of } A \text{ in } P_0\})$, where $w(T)$ is the *weight* of the proof T defined as an n-tuple $\langle w_1, \ldots, w_n \rangle$ such that $\forall 1 \leq j \leq n$, w_j is the number of nodes of predicates from stratum j in T. $\alpha(A)$ corresponds to the notion of *weight-tuple measure* of A defined in [TS86a].

   For any atom A, $\alpha_{min}(A) = \mathbf{0} = \langle 0, \ldots, 0 \rangle$.

3. $\forall C \in P_0$, $\gamma^0_{lo}(C) = \gamma^0_{hi}(C) = \langle w_1, \ldots, w_n \rangle$, where $C \equiv A :- A_1, \ldots, A_n$ and for $1 \leq j \leq n$, $w_j = 1$ if the predicate symbol of A is from stratum j, and 0 otherwise.

   For any $A \in M(P_0)$, the proof T that defines $\alpha(A)$ (item 2 above) is strongly measure consistent. Weak measure consistency of ground proofs in P_0 is established by induction on their size.

4. $\forall C \in P_{i+1} - P_i$, $\gamma^{i+1}_{hi}(C) = LUB^{i+1}(C)$ and $\gamma^{i+1}_{lo}(C) = approx(GLB^{i+1}(C))$. The function *approx* reduces a measure as follows. Let $u = \langle u_1, \ldots, u_n \rangle$ and k_{min} be the smallest index k such that $u_k > 0$. Then $approx(u) = \langle u'_1, \ldots, u'_n \rangle$ where $u'_{k_{min}} = 1$ and is 0 elsewhere.

As in the Kanamori-Fujita system, here also the measure preserving folding rule is applied only when both folder and folded clauses are singleton sets.

To establish the correspondence between the above instantiation and the extended Tamaki-Sato system, recall that the latter associates a descent level with each clause of every program in a transformation sequence. If a clause C in P_i has the descent level k, then with the above instantiation, $\gamma^i_{lo}(C) = \langle l_1, \ldots, l_n \rangle$ where $l_k = 1$ and 0 elsewhere; i.e. the only non-zero entry in its lower clause measure appears in the k^{th} position. Thus our lower clause measure precisely captures the information that is kept track of by the extended Tamaki-Sato system.

Assigning Measure Structures and Clause Measures Observe that our framework does not prescribe exact values to the clause measures. Instead it bounds the clause measures from above and below. So an important aspect of our instantiation involves assigning values to the clause measures that satisfy these constraints. From an abstract point of view, the Kanamori-Fujita system uses a relatively coarse measure space ($\mathbb{Z}$) but within this space it maintains accurate clause measures (integer counters). Our instantiation reflects this by not relaxing the bounds while updating the clause measures (see step 4 of the instantiation). On the other hand, the extended Tamaki-Sato system uses a more fine-grained measure space ($\mathbb{Z}^n$). But this measure space is not completely utilized since clause measures are the descent level of clauses, which can be simply represented by an integer. Therefore in step 4 of our instantiation we accordingly loosened the bound. As far as the Gergatsoulis-Katzouraki [GK94] and original Tamaki-Sato systems [TS84] are concerned, first note that they do not permit folding using recursive clauses. These systems use coarse measure spaces. Moreover they do not even fully utilize these measure spaces as is evident from the lesser amount of book keeping performed by them. By choosing a coarse measure structure and relaxing the bounds along lines similar to the extended Tamaki-Sato system we have been able to instantiate these two systems as well. The measure structure chosen is $\langle \mathbb{Z}^2, \oplus, \prec, \mathbb{N}^2 \rangle$ where $\prec$ is the lexicographic ordering among 2-tuples, and $\oplus$ is co-ordinatewise addition operation among 2-tuples. Since these systems partition the predicate symbols into "old" and "new" predicates, the choice of a measure structure with two strata is obvious.

3.3.2 SCOUT— A New Unfold/Fold System

We now construct SCOUT (Strata and COunter based Unfold/fold Transformations), an unfold/fold transformation system for definite logic programs that allows disjunctive folding using recursive clauses. It incorporates the notion of strata from the extended Tamaki-Sato system into the counters of the Kanamori-Fujita system. Thus with every clause it maintains a *pair* of stratified counters as the clause measure. The instantiation is as follows. We assume that the predicate symbols appearing in the initial program P_0 are partitioned into n strata, as in the extended Tamaki-Sato system.

1. $\mu = \langle \mathbb{Z}^n, \oplus, \prec, \mathbb{N}^n \rangle$ where $\oplus$ denotes coordinate-wise integer addition of n-tuples of integers, and $\prec$ denotes the lexicographic $<$ order over n-tuples of integers.

2. $\alpha(A)$ is defined exactly as in the instantiation of the extended Tamaki-Sato system above. For any atom A we set $\alpha_{min}(A) = \langle w_1, \ldots, w_n \rangle$ where $w_j = 1$ if A is from stratum j and 0 elsewhere.

3. Clause measure of clauses in P_0 is defined exactly as in the instantiation of the extended Tamaki-Sato system above. Therefore the proofs of measure consistency are also identical.

4. $\forall C \in P_{i+1} - P_i$, $\gamma_{lo}^{i+1}(C) = GLB^{i+1}(C)$ and $\gamma_{hi}^{i+1}(C) = LUB^{i+1}(C)$.

SCOUT provides a solution to two important (and orthogonal) problems that have thus far remained open: folding using clauses that have disjunctions as well as recursion, and combining the stratification-based (extended) Tamaki-Sato system with the counter-based Kanamori-Fujita system thereby obtaining a single system that *strictly subsumes* either of them even when restricted to conjunctive folding. A formal proof of this claim follows. The reader unfamiliar with logic programming may proceed to the next section of the chapter.

Kanamori-Fujita system [KF87] The system reported in [KF87] is special case of SCOUT where folding is conjunctive and all the predicate symbols of the initial program are placed in a single stratum.

Extended Tamaki-Sato system [TS86a] For proving that SCOUT covers any transformation sequence $P_0, P_1, P_2, \ldots$ which is allowed by the fold/unfold system of [TS86a], we define the invariants given below. Recall that in [TS86a] each clause in any P_i is associated with a strata number, also called the descent level. Also, since [TS86a] handles only conjunctive folding, any fold/unfold transformation sequence of [TS86a], *if executable in SCOUT*, will always produce clauses with counters of the form (γ, γ); in other words, the two counters of any clause will always be equal.

We now consider the following invariants :

- $J1(P_i) \equiv$ Any fold/unfold transformation in P_i which is allowed by the extended Tamaki-Sato system [TS86a] is allowed by SCOUT (with n strata).

- $J2(P_i) \equiv$ Let C be any clause in program P_i with strata number (*i.e.* descent level in the terminology of [TS86a]) j. Then, in SCOUT (with n strata), $\gamma_{lo}^{i}(C) = \gamma_{hi}^{i}(C) = \langle \gamma_1, \ldots, \gamma_n \rangle$ where $\gamma_j > 0 \wedge (\forall 1 \leq k < j \ \gamma_k = 0)$

To prove that any unfold/fold transformation sequence covered by [TS86a] is also covered by SCOUT, it is sufficient to prove that $J1(P_i)$ is an invariant.

Theorem 3.4 *Let $P_0, P_1, P_2, \ldots$ be an unfold/fold transformation sequence of the extended Tamaki-Sato system [TS86a]. Then, $\forall i \geq 0$. $J1(P_i) \wedge J2(P_i)$*

Proof : By induction on i. For the *base case*, $J1(P_0)$ is trivially true by the definition of the fold/unfold transformations in [TS86a] and SCOUT. Also, if a clause C in P_0 has descent level j, then $\gamma^o_{lo}(C) = \gamma^o_{hi}(C) = \langle \gamma_1, \ldots, \gamma_n \rangle$ where $\gamma_j = 1$ and $\gamma_l = 0$ when $l \neq j$. Clearly then $J2(P_0)$ is also true. The *induction hypothesis* is $\forall i \leq m$ $J1(P_i) \wedge J2(P_i)$. We now show that $J1(P_{m+1}) \wedge J2(P_{m+1})$ holds.

First we prove $J2(P_{m+1})$. Let C be any clause in P_{m+1}. We show that the property mentioned in $J2$ is true for C.

Case 1: C is inherited from P_m

The result holds since $J2(P_m)$ is true by induction hypothesis.

Case 2: C is obtained by unfolding C' using C''

Since, $\forall i \leq m$. $J1(P_i)$, the sequence $P_0, P_1, \ldots, P_m, P_{m+1}$ can be constructed using SCOUT. Then, $\gamma^{m+1}_{lo}(C) = \gamma^{m+1}_{hi}(C) = \gamma^m_{lo}(C') \oplus \gamma^m_{lo}(C'') = \gamma^m_{hi}(C') \oplus \gamma^m_{hi}(C'')$. Also let the descent level of C, C' and C'' be k,k' and k'' respectively. Then, by [TS86a], $k = min(k', k'')$. By the induction hypothesis, the property in $J2$ is true for both C' and C''. Hence if $\gamma^m_{lo}(C') = \gamma^m_{hi}(C') = \langle \gamma'_1, \ldots, \gamma'_n \rangle$ and $\gamma^m_{lo}(C'') = \gamma^m_{hi}(C'') = \langle \gamma''_1, \ldots, \gamma''_n \rangle$, then $\gamma'_1 = \cdots = \gamma'_{k-1} = 0$, $\gamma''_1 = \cdots = \gamma''_{k-1} = 0$. Also since k is the minimum of k' and k'', we have either $\gamma'_k = 0 \wedge \gamma''_k > 0$, or $\gamma'_k > 0 \wedge \gamma''_k = 0$ Now, $\gamma^{m+1}_{lo}(C) = \gamma^{m+1}_{hi}(C) = \gamma^m_{lo}(C') \oplus \gamma^m_{lo}(C'') = \langle \gamma_1, \ldots, \gamma_n \rangle$ where $\forall 1 \leq l \leq n$ $\gamma_l = \gamma'_l + \gamma''_l$. Hence $\gamma_1 = \cdots = \gamma_{k-1} = 0$ and $\gamma_k > 0$. Thus the property in $J2$ holds for C.

Case 3: C is obtained by folding C' using D'

Since $\forall i \leq m$. $J1(P_i)$, the transformation sequence $P_0, P_1, \ldots, P_m, P_{m+1}$ can be constructed using SCOUT. Let C' and D' have descent levels k and l respectively. Then the descent level of C is also k and $k < l$. But $D' \in P_0$, so $\gamma^0_{lo}(D') = \gamma^0_{hi}(D') = \langle \delta'_1, \ldots, \delta'_n \rangle$ where $\delta'_l = 1$ and $\delta'_j = 0$ when $j \neq l$. Let $\gamma^m_{lo}(C') = \gamma^m_{hi}(C') = \langle \gamma'_1, \ldots, \gamma'_n \rangle$. As the property in $J2$ is true for C', we have $\gamma'_1 = \cdots = \gamma'_{k-1} = 0$ and $\gamma'_k > 0$. Now, $\gamma^{m+1}_{lo}(C) = \gamma^{m+1}_{hi}(C) = \gamma^m_{lo}(C') \ominus \gamma^0_{hi}(D') = \langle \gamma_1, \ldots, \gamma_n \rangle$ where $\forall 1 \leq j \leq n$ $\gamma_j = \gamma'_j - \delta_j$.

Since $k < l$, therefore $\delta'_1 = \cdots = \delta'_k = 0$. Thus, $\gamma_1 = \cdots = \gamma_{k-1} = 0$ and $\gamma_k = \gamma'_k > 0$. Hence the property in $J2$ holds for C.

We now show that $J2(P_{m+1}) \Rightarrow J1(P_{m+1})$. Since the unfolding transformation is independent of any condition on the stratified counter (or descent level) in SCOUT or [TS86a], therefore any unfolding allowed by [TS86a] in P_{m+1} is also allowed by SCOUT. For folding, let $C \in P_{m+1}$ be folded using the folder $D \in P_0$ in the system of [TS86a]. Let the descent levels of C and D be be k and l respectively. Then, $k < l$ (by [TS86a]) and the property of $J2$ is true for both C and D (since $J2(P_{m+1})$ holds). So, if $\gamma^{m+1}_{lo}(C) = \gamma^{m+1}_{hi}(C) = \langle \gamma_1, \ldots, \gamma_n \rangle$ and $\gamma^0_{lo}(D) = \gamma^0_{hi}(D) = \langle \delta_1, \ldots, \delta_n \rangle$ we have $\gamma_1 = ... = \gamma_{k-1} = 0$, $\gamma_k > 0$, $\delta_1 = ... = \delta_{l-1} = 0$. As $k < l$, this means $\delta_1 = ... = \delta_k = 0$. Clearly then $\gamma^{m+1}_{lo}(C)$ is lexicographically greater than $\gamma^0_{hi}(D)$. Hence C can be folded using D as folder in SCOUT. This completes the proof. □

Thus, we have proved that SCOUT allows all unfold/fold transformation sequences allowed by [TS86a]. To prove that it is *strictly more powerful*, we need to give an example transformation sequence which is allowed by SCOUT, but not by [TS86a]. Again, any example requiring disjunctive folding serves this purpose. Hence we conclude that SCOUT is strictly more powerful than [TS86a].

It is interesting to note that by simple inspection of the instantiations, one can see that when the number of strata is 1 and only conjunctive folding is permitted, SCOUT collapses to the Kanamori-Fujita system. Collapsing SCOUT to other existing unfold/fold systems by varying the number of strata and extending the parameters (e.g. measure structure) remains an interesting open problem.

3.4 Goal Replacement

Augmenting an unfold/fold transformation system with the goal replacement rule makes it more powerful. In this section we incorporate goal replacement to our parameterized framework. Goal replacement allows semantically equivalent conjunctions of atoms to be freely interchanged. We formally define it below. For a conjunction of atoms $A_1, ..., A_n$, we use the notation $vars(A_1, ..., A_n)$ to denote the set of variables in $A_1, ..., A_n$.

Transformation 3.5 (Goal Replacement) Let C be a clause $A :- A_1, \ldots, A_k, G$ in P_i, and G' be an atom such that $vars(G) = vars(G') \subseteq vars(A, A_1, ..., A_k)$. Suppose for all ground instantiation θ of G, G' we have $G\theta \in M(P_i) \Leftrightarrow G'\theta \in M(P_i)$. Then $P_{i+1} := (P_i - \{C\}) \cup \{C'\}$ where $C' \equiv A :- A_1, \ldots, A_k, G'$. □

Note that although we replace a single atom G by another atom G' (where G and G' do not contain any internal variables), we can replace conjunctions of atoms using a sequence of folding, goal replacement and unfolding transformations. The above transformation is partially correct, a claim that we formally prove below.

Theorem 3.5 *Let program P_{i+1} be obtained from program P_i by applying goal replacement as described in rule 3.5. Then, $M(P_{i+1}) \subseteq M(P_i)$.*

Proof : We take any ground proof T of some $B \in M(P_{i+1})$ and construct a ground proof T' of B in P_i, thereby proving $M(P_{i+1}) \subseteq M(P_i)$. This proof proceeds by induction on size of ground proofs in P_{i+1}. The base case is obvious because unit clauses are not manipulated by goal replacement. For the induction step, if the clause used at the root of T is not the replacing clause C', then the proof follows from induction hypothesis. Let the clause used at the root of T be a ground instance of C' and let the ground instance used be $A\theta$:– $A_1\theta, \ldots A_k\theta, G'\theta$. Then, $A_1\theta$,...,$A_k\theta, G'\theta$ have ground proofs $T'_1, \ldots, T'_k, T'_{G'\theta}$ in P_i by induction hypothesis. Then, by rule 3.5, there exists a ground proof $T'_{G\theta}$ of $G\theta$ in P_i. Now T', the ground proof of $A\theta$ in P_i, is constructed with the ground clause $A\theta$:– $A_1\theta, \ldots, A_k\theta, G\theta$ at the root and $T'_1, \ldots, T'_k, T'_{G\theta}$ as its children. □

We can ensure that an application of goal replacement to a measure consistent program is totally correct. But then we also need to ensure that the resulting program P_{i+1} is measure consistent. If this is guaranteed, then even if goal replacement is interleaved with irreversible folding total correctness will be preserved. Formally,

Transformation 3.6 (Measure Preserving Goal Replacement) Suppose program P_{i+1} is obtained from program P_i by applying the goal replacement transformation as described in Rule 3.5. Let there exist $\delta, \delta' \in \mathcal{M}$ (where measure structure is $\mu = \langle \mathcal{M}, \oplus, \prec, \mathcal{W} \rangle$) such that for all ground instantiation θ of G, G', we have: *(i)* $\delta \preceq \alpha(G\theta) \ominus \alpha(G'\theta) \preceq \delta'$ *(ii)* $\gamma^i_{lo}(C) \oplus \delta \oplus \sum_{1 \leq p \leq k} \alpha_{min}(A_p) \succ \mathbf{0}$. Then

$$\gamma^{i+1}_{lo}(C') \preceq GLB^{i+1}(C') = \gamma^i_{lo}(C) \oplus \delta \quad (5)$$

$$\gamma^{i+1}_{hi}(C') \succeq LUB^{i+1}(C') = \gamma^i_{hi}(C) \oplus \delta' \quad (6)$$

The clause measures of the other clauses of P_{i+1} are inherited from P_i. □

We now present a formal proof of total correctness and preservation of measure consistency of the above rule.

Theorem 3.6 *Let P_{i+1} be derived from P_i by applying measure preserving goal replacement as described in rule 3.6. If P_i is measure consistent, then $M(P_i) = M(P_{i+1})$ and P_{i+1} is also measure consistent.*

Proof: Since measure preserving goal replacement is a special case of the goal replacement transformation in rule 3.5, we have $M(P_{i+1}) \subseteq M(P_i)$ by partial correctness of rule 3.5. Therefore it is sufficient to prove that : (1) all ground proofs of P_{i+1} are weakly measure consistent (2) $M(P_i) \subseteq M(P_{i+1})$ (3) $\forall B \in M(P_{i+1})$ there exists a strongly measure consistent proof of B in P_{i+1}. We prove proof obligation (1) separately. Proof obligations (2) and (3) are proved by showing that : $\forall B \in M(P_i)$ there exists a strongly measure consistent proof of B in P_{i+1}. This is sufficient since we know $M(P_{i+1}) \subseteq M(P_i)$.

First, we prove that all ground proofs of P_{i+1} are weakly measure consistent. The proof proceeds by induction on the size of ground proofs in P_{i+1}. Let T be a ground proof of a ground atom B in P_{i+1}. If the clause used at the root of T is not the new clause C', then the proof follows by induction hypothesis and the measure consistency of P_i. If the clause used at the root of T is C', then let the ground instance of C' used at the root of T be $A\theta :- A_1\theta, \ldots, A_k\theta, G'\theta$. By induction hypothesis, the proofs of $A_1\theta, \ldots, A_k\theta, G'\theta$ in T are weakly measure consistent. It suffices to show that $\alpha(A) \preceq \gamma_{hi}^{i+1}(C') \oplus \sum_{1 \leq l \leq k} \alpha(A_l\theta) \oplus \alpha(G'\theta)$ Now, $G'\theta \in M(P_{i+1}) \Rightarrow G'\theta \in M(P_i)$. Hence by rule 3.5 we have $G\theta \in M(P_i)$. Also, $\forall 1 \leq l \leq k$ $A_l\theta \in M(P_i)$ (as $M(P_{i+1} \subseteq M(P_i))$. Then, $A\theta :- A_1\theta, \ldots A_k\theta, G\theta$ is a ground instantiation of C which appears at the root of some ground proof in P_i. Since P_i is measure consistent we have

$$\begin{aligned} \alpha(A) &\preceq \gamma_{hi}^{i}(C) \oplus \textstyle\sum_{1 \leq l \leq k} \alpha(A_l\theta) \oplus \alpha(G\theta) \\ &\preceq \gamma_{hi}^{i}(C) \oplus \textstyle\sum_{1 \leq l \leq k} \alpha(A_l\theta) \oplus (\ \alpha(G'\theta) \oplus \delta'\ ) \\ &\preceq \gamma_{hi}^{i+1}(C') \oplus \textstyle\sum_{1 \leq l \leq k} \alpha(A_l\theta) \oplus \alpha(G'\theta) \end{aligned}$$

Now, we prove that $\forall B \in M(P_i)$ there is a strongly measure consistent proof of B in P_{i+1}. Since P_i is measure consistent, it suffices to translate a strongly measure consistent proof T of B in P_i to a strongly measure consistent proof T' of B in P_{i+1} for all $B \in M(P_i)$. We do this translation by induction on the atom measures. If the clause used at the root of T is not C (where C is the clause in P_i that is replaced) then the proof follows from the definition of strong measure consistency and induction hypothesis. Let C be the clause used at the root of T (a strongly measure consistent proof of A in P_i) and let $A\theta :- A_1\theta, \ldots, A_k\theta, G\theta$ be the ground instance of C used. Then, by strong measure consistency of T, $\alpha(A_l\theta) \prec \alpha(A\theta)$ for

all $1 \leq l \leq k$. By induction hypothesis, we then have strongly measure consistent ground proofs $T'_1, \ldots, T'_k$ of $A_1\theta, \ldots, A_k\theta$ in P_{i+1}. Also, by strong measure consistency of T

$$\begin{aligned}
\alpha(A) &\succeq \gamma^i_{lo}(C) \oplus \textstyle\sum_{1\leq l\leq k} \alpha(A_l\theta) \oplus \alpha(G\theta) \\
&\succeq \gamma^i_{lo}(C) \oplus \textstyle\sum_{1\leq l\leq k} \alpha(A_l\theta) \oplus (\ \alpha(G'\theta) \oplus \delta\ ) \quad \cdots\cdots (*) \\
&\succeq (\ \gamma^i_{lo}(C) \oplus \textstyle\sum_{1\leq l\leq k} \alpha_{min}(A_l\theta) \oplus \delta\ ) \oplus \alpha(G'\theta) \\
&\succ \alpha(G'\theta) \text{ (By condition (ii) of rule 3.6)}
\end{aligned}$$

Then, by induction hypothesis, $G'\theta$ has a proof $T'_{G'\theta}$ in P_{i+1}. The ground proof T' is constructed with $A\theta :- A_1\theta, \ldots, A_k\theta, G'\theta$ at the root (this is a ground instance of C', the new clause in P_{i+1}) and $T'_1, \ldots, T'_k, T'_{G'\theta}$ as its children. To show that this proof T' is measure consistent, note that $\gamma^{i+1}_{lo}(C') \preceq \gamma^i_{lo}(C) \oplus \delta$. Combining this with (*), we get

$$\alpha(A) \succeq \gamma^{i+1}_{lo}(C') \oplus \sum_{1\leq l\leq k} \alpha(A_l\theta) \oplus \alpha(G'\theta)$$

This completes the proof. □

Observe that, similar to the goal replacement transformation in [KF87, TS84, TS86a] the conditions under which rule 3.6 may be applied are not testable at transformation time. For testability we need to (1) determine whether G and G' are semantically equivalent, and (2) estimate δ and δ' such that the clause measures of P_{i+1} can be computed. Semantic equivalence is undecidable in general and can be conservatively approximated using program analysis. Our notion of syntactic equivalence (refer Definition 5.3, Page 75) addresses this issue. Estimates for δ and δ' can be obtained by Integer Linear Programming techniques (refer Theorem 5.5, Page 78). These two steps define a testable goal replacement rule.

Note that the goal replacement rule presented here can be added to any arbitrary unfold/fold system instantiated in our framework. More importantly, this is done by simply manipulating the measures; the proof of correctness of the augmented transformation system follow immediately from the correctness proof of our framework.

3.5 Discussions

The development of a parameterized framework for unfold/fold transformations has several important implications. It enables us to compare existing transformation systems and modify them without redoing the correctness proofs (e.g., extending measures for goal replacement in Section 3.4). It also facilitates the development of

new transformation systems. For instance, we derived SCOUT which permits folding using multiple recursive clauses. Such a transformation system is particularly important for verifying parameterized concurrent systems (such as a n-process token ring for arbitrary n) using logic program evaluation and deduction. The utility of the transformation system presented in this chapter is outlined by its use in constructing induction proofs of safety properties of nontrivial distributed protocols (see Chapter 6).

In the next chapter, we extend the work reported here to obtain generalized unfold/fold transformation systems for normal logic programs. Aravindan and Dung [AD95] developed an approach to parameterize the correctness proofs of the original Tamaki-Sato system with respect to various semantics based on the notion of *semantic kernels*. Incorporating the idea of semantic kernel into our framework yields a framework that is parameterized with respect to the measure structures as well as semantics.

In future, it would be interesting to study whether we can develop similar parameterized unfold/fold transformation frameworks for other programming paradigms such as functional and concurrent constraint programming languages [EGM98, San96] as well as process algebraic specification languages (*e.g.* CCS) [FS98].

Chapter 4

Transformations for Normal Logic Programs

4.1 Background

Normal logic programs consist of definitions of the form $A :- \phi$ where A is an atom and ϕ is a boolean formula over atoms. Unfolding replaces an occurrence of A in a program with ϕ while folding replaces an occurrence of ϕ with A. Recall that folding is called *reversible* if its effects can be undone by an unfolding, and *irreversible* otherwise. As in definite logic programs, unfold/fold transformation systems are proved correct by showing that all programs in any transformation sequence $P_0, P_1, \ldots, P_n$ are equivalent under a suitable semantics, such as the well-founded model semantics for normal logic programs.

As an illustration of unfolding/folding, consider the sequence of normal logic programs in figure 8. In the figure, P_1 is derived from P_0 by unfolding the occurrence of `q(X)` in the first clause of P_0. Program P_2 is derived from P_1 by folding the literal `q(Y)` in the body of the second clause of `p/1` into `p(Y)` by using clause `p(X) :- q(X)` in P_0 as folder.

Existing unfold/fold transformation systems for normal logic programs [AD95, Mah93, PP00, Sek91, Sek93] are either extensions of Maher's reversible transformation system [Mah87] or the original Tamaki-Sato system [TS84]. Even for definite logic programs, irreversible transformations of programs were, until recently, restricted to either folding using non-recursive clauses (see [GK94]) or a single recursive clause (see [KF87, TS86a]). In the last chapter, we proposed a transformation framework

for definite logic programs which generalized the above systems by permitting folding using multiple recursive clauses. Here, we extend these transformations to normal logic programs.

Overview of the results: The main result of this chapter (which was also reported in [RKRR99a]) is an unfold/fold transformation system that performs folding in the presence of recursion, disjunction as well as negation. The transformations of the last chapter associate *measures* with program clauses to determine the applicability of fold and unfold transformations. In this chapter, we extend this scheme to accommodate negative literals. We show that this extension is sufficient to guarantee that the resulting transformation system preserves a variety of semantics for normal logic programs, such as the well-founded model, stable model, partial stable model, and stable theory semantics. Central to this proof is the result due to Dung and Kanchanasut [DK89] that preserving the *semantic kernel* of a program is sufficient to guarantee the preservation of the different semantics for negation listed above. However, in contrast to [AD95] where this idea was used to prove the correctness of Tamaki-Sato style transformations, we present a two-step proof which explicitly uses the operational counterpart of semantic kernels (see Section 4.3).

In the first step of our proof, we show that the transformations preserve positive ground derivations, which are derivations of the form $A \rightsquigarrow \neg B_1, \neg B_2, \ldots, \neg B_n$ such that there is a proof tree rooted at A with leaves labeled $\neg B_1$ through $\neg B_n$ (apart from `true`). We then show that preserving positive ground derivations is equivalent to preserving the semantic kernel of the program. Thus positive ground derivations are

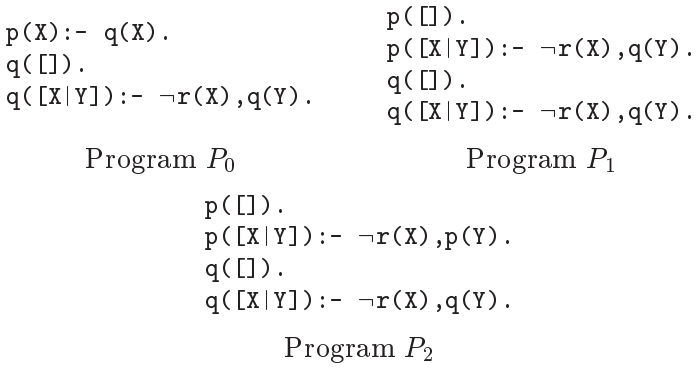


Figure 8: An unfold/fold transformation sequence of normal logic programs

the operational analogues of semantic kernels. This proof suggests that we can treat the negative literals in a program as atoms of new predicates defined in a different (external) module. The correctness of the transformation system is assured as long as the transformations respect module boundaries (see Section 4.4). This indicates how a transformation system originally designed for definite logic programs can be adapted for normal logic programs.

4.2 The Transformations

Below we present our *unfold* and *fold* transformations for normal logic programs. The unfold transformation, following the transformations in [AD95, Mah93, Sek91, Sek93], does not unfold a negative literal. As a result, the negative literals may participate in a transformation, but are by themselves unchanged. The folding transformation is also almost identical to the folding rule for definite logic programs.

For simplicity of exposition, instead of presenting an abstract transformation framework (such as the one given in the last chapter) we now present a concrete transformation system. A general transformation framework for normal logic programs can be easily constructed from this system by using the results of the last chapter.

Thus, in any transformation sequence $P_0, P_1, \ldots, P_n$ we annotate each clause C in program P_i with a pair $(\gamma^i_{lo}(C), \gamma^i_{hi}(C))$ where $\gamma^i_{lo}(C), \gamma^i_{hi}(C) \in \mathbb{Z}$ and $\gamma^i_{lo}(C) \leq \gamma^i_{hi}(C)$. Thus, γ^i_{lo} and γ^i_{hi} are functions from the set of clauses in program P_i to the set of integers $\mathbb{Z}$. The transformation rules dictate the construction of γ^{i+1}_{lo} and γ^{i+1}_{hi} from γ^i_{lo} and γ^i_{hi}. We assume that for any clause C in the initial program P_0, $\gamma^0_{lo}(C) = \gamma^0_{hi}(C) = 1$. Recall that intuitively, $\gamma^i_{lo}(C)$ and $\gamma^i_{hi}(C)$ for a clause C are analogous to the Kanamori-Fujita-style counters [KF87]; the separation of *hi* and *lo* permits us to store estimates of the counter values which is crucial for allowing disjunctive folding.

Transformation 4.1 (Unfolding) Let C be a clause in P_i and A a positive literal in the body of C. Let $C_1, ..., C_m$ be the clauses in P_i whose heads are unifiable with A with most general unifiers $\sigma_1, ..., \sigma_m$. Let C'_j be the clause that is obtained by replacing $A\sigma_j$ by the body of $C_j\sigma_j$ in $C\sigma_j$ $(1 \leq j \leq m)$.
Then, assign $P_{i+1} := (P_i - \{C\}) \cup \{C'_1, ..., C'_m\}$. Set

$$\begin{aligned} \gamma^{i+1}_{lo}(C'_j) = &\quad \gamma^i_{lo}(C) + \gamma^i_{lo}(C_j) \\ \gamma^{i+1}_{hi}(C'_j) = &\quad \gamma^i_{hi}(C) + \gamma^i_{hi}(C_j) \end{aligned}$$

The measures of all other clauses in P_{i+1} are inherited from P_i. □

Transformation 4.2 (Folding) Let $\{C_1, ..., C_m\} \subseteq P_i$ and $\{D_1, ..., D_m\} \subseteq P_j$ ($j \leq i$) where C_l denotes the clause $A :- L_{l,1}, \ldots, L_{l,n_l}, L'_1, \ldots, L'_n$ and D_l denotes the clause $B_l :- K_{l,1}, \ldots, K_{l,n_l}$. Also, let
1. $\forall 1 \leq l \leq m\ \exists \sigma_l\ \forall 1 \leq k \leq n_l\ L_{l,k} = K_{l,k}\sigma_l$, where σ_l is a substitution.
2. $B_1\sigma_1 = B_2\sigma_2 = ... = B_m\sigma_m = B$
3. $D_1, ..., D_m$ are the only clauses in P_j whose heads are unifiable with B.
4. $\forall 1 \leq l \leq m\ \sigma_l$ substitutes the internal variables of D_l to distinct variables which do not appear in $\{A, B, L'_1, ..., L'_n\}$.
5. $\forall 1 \leq l \leq m\ \gamma^j_{hi}(D_l) < \gamma^i_{lo}(C_l)+$ Number of positive literals in the sequence $L'_1, \ldots, L'_n$.
Then, assign $P_{i+1} := (P_i - \{C_1, ..., C_m\}) \cup \{C'\}$ where $C' \equiv A :- B, L'_1, ..., L'_n$. Set

$$\begin{aligned} \gamma^{i+1}_{lo}(C') = &\ min_{1 \leq l \leq m}(\gamma^i_{lo}(C_l) - \gamma^j_{hi}(D_l)) \\ \gamma^{i+1}_{hi}(C') = &\ max_{1 \leq l \leq m}(\gamma^i_{hi}(C_l) - \gamma^j_{lo}(D_l)) \end{aligned}$$

The annotations of all other clauses in P_{i+1} are inherited from P_i. □

Note that the first four conditions of the folding rule are essential for any correct application of folding, including reversible folding. Finally, the fifth condition of the rule governs sound applications of folding following a series of unfoldings. Our folding rule can use multiple recursive clauses as folder unlike existing irreversible unfold/fold transformation systems for normal logic programs. These systems restrict the folding rule to use a single non-recursive clause from P_0 as the folder.

An Example: The following example (derived from [GK94]) illustrates the use of our unfold/fold transformation system for normal logic programs.

```
C1 : in_pos(X,L) :- in_odd(X,L), ¬ even(X).   (1,1)
C2 : in_pos(X,L) :- in_even(X,L), ¬ odd(X).   (1,1)
C3 : in_odd(X,[X|L]).                         (1,1)
C4 : in_odd(X,[Y,Z|L]) :- in_odd(X,L).        (1,1)
C5 : in_even(X,[Y,X|L]).                      (1,1)
C6 : in_even(X,[Y,Z|L]) :- in_even(X,L).      (1,1)
```

In the above program, `in_odd(X,L)` (`in_even(X,L)`) is true if `X` appears in an odd (even) position in list `L`. Thus, `in_pos(X,L)` is true if `X` is in an odd (even) position

in list `L`, and `X` is not an even (odd) number. The `odd/1` and `even/1` predicates are encoded in the usual way and are not shown.
Unfolding `in_odd(X,L)` in C_1 we get: clauses:

```
C7 : in_pos(X,[X|L]) :- ¬ even(X).                (2,2)
C8 : in_pos(X,[Y,Z|L]) :- in_odd(X,L), ¬ even(X). (2,2)
C2 : in_pos(X,L) :- in_even(X,L), ¬ odd(X).       (1,1)
```

Now, unfolding `in_even(X,L)` in C_2 yields the following:

```
C7 : in_pos(X,[X|L]) :- ¬ even(X).                (2,2)
C8 : in_pos(X,[Y,Z|L]) :- in_odd(X,L), ¬ even(X). (2,2)
C9 : in_pos(X,[Y,X|L]) :- ¬ odd(X).               (2,2)
C10 : in_pos(X,[Y,Z|L]) :- in_even(X,L), ¬ odd(X). (2,2)
```

Finally, we fold clauses $\{C_8, C_{10}\}$ using the clauses $\{C_1, C_2\}$ from the initial program as the folder to obtain the following definition of `in_pos/1`.

```
C7 : in_pos(X,[X|L]) :- ¬ even(X).      (2,2)
C9 : in_pos(X,[Y,X|L]) :- ¬ odd(X).     (2,2)
C11 : in_pos(X,[Y,Z|L]) :- in_pos(X,L). (1,1)
```

Note that the final step is an irreversible folding in presence of negation that uses multiple clauses as the folder. Such a folding step is neither allowed in Tamaki-Sato style transformation systems for normal logic programs [AD95, PP00, Sek91, Sek93] nor in reversible transformation systems [Mah93].

Remark: We can maintain more elaborate book-keeping information than integer counters, thereby deriving more expressive unfold/fold systems for normal logic programs. For instance, as in the SCOUT system (presented in the last chapter), we can make the counters range over use a tuple of integers, and obtain a system that is strictly more powerful than the existing Tamaki-Sato-style systems [TS84, TS86a, KF87, Sek91, Sek93, GK94, AD95]. The construction parallels that of the SCOUT system.

4.3 Proof of Correctness

In this section, we show that our unfold/fold transformation system is correct with respect to various semantics of normal logic programs. This proof proceeds in three

steps. First, we introduce the notion of positive ground derivations and show that it is preserved by the transformations. Secondly, we show that preserving positive ground derivations is equivalent to preserving semantic kernel [DK89]. Finally, following [AD95], preserving semantic kernel implies that the transformation system is correct with respect to various semantics for normal logic programs including well-founded model, stable model, partial stable model, and stable theory semantics. We begin with a review of semantic kernels.

4.3.1 Semantic Kernel of a Program

Definition 4.1 (Quasi Interpretation) [DK89, AD95] *A quasi interpretation of a normal logic program P is a set of ground clauses of the form*

$$A :- \neg B_1, \ldots, \neg B_n \ (n \geq 0)$$

where $A, B_1, \ldots, B_n$ are ground atoms in the Herbrand Base of P.

Quasi interpretations form the universe over which semantic kernels are defined. For a given normal logic program P, the set of all quasi interpretations of P (denoted $QI(P)$) forms a complete partial order with a least element (the empty set ϕ) with respect to the set inclusion relation $\subseteq$.

Definition 4.2 *Given a normal logic program P, let $Gnd(P)$ denote the set of all possible ground instantiations of all clauses of P. The function S_P on quasi interpretations of P is defined as*

$$S_P : QI(P) \rightarrow QI(P)$$
$$S_P(I) = \{\mathcal{R}(C, D_1, \ldots, D_m) \mid C \in Gnd(P) \wedge D_i \in I,\ 1 \leq i \leq m\}$$

where, if $D_i (1 \leq i \leq m)$ are ground clauses

$$A_i :- \ \neg B_{i,1}, \ldots, \neg B_{i,n_i} \ (n_i \geq 0)$$

and $A_1, \ldots, A_m (m \geq 0)$ are the only positive literals appearing in the body of ground clause C, then $\mathcal{R}(C, D_1, \ldots, D_m)$ is the clause obtained by resolving the positive body literals $A_1, \ldots, A_m$ in C using clauses $D_1, \ldots, D_m$ respectively. □

If P is a definite program, then the function S_P is identical to the immediate logical consequence operator T_P [Llo93]. The semantic kernel of the program P is defined in terms of S_P as:

Definition 4.3 (Semantic Kernel) [DK89, AD95] *The semantic kernel of a normal logic program P, denoted by $SK(P)$, is the least fixed point of the function S_P, i.e.,*

$$SK(P) = \bigcup_{n\in\omega} SK^n(P) \text{ where } SK^0(P) = \phi \text{ and } SK^{n+1}(P) = S_P(SK^n(P))$$

Example : Consider the following normal logic program P:

```
p :- ¬ q, r.
r :- ¬ r.
```

The semantic kernel of P will be computed as follows.

$SK^0(P) = \{\}$.
$SK^1(P) = S_P(SK^0(P)) = \{$ (`r :- ¬ r`) $\}$
$SK^2(P) = S_P(SK^1(P)) = \{$ (`r :- ¬ r`), (`p :- ¬ q, ¬ r`) $\}$
$SK^3(P) = S_P(SK^2(P)) = SK^2(P)$
Therefore, $SK(P) = \{$ (`r :- ¬ r`), (`p :- ¬ q, ¬ r`) $\}$

The following theorem from [AD95] formally states the equivalence of P and $SK(P)$ with respect to various semantics of normal logic programs.

Theorem 4.1 [Aravindan and Dung] *Let P be a normal logic program and $SK(P)$ be its semantic kernel. Then :*
(1) If P is a definite logic program, then P and $SK(P)$ have the same least Herbrand Model.
(2) If P is a stratified program, then P and $SK(P)$ have the same perfect model semantics.
(3) P and $SK(P)$ have the same well-founded model.
(4) P and $SK(P)$ have the same stable model(s).
(5) P and $SK(P)$ have the same set of partial stable models.
(6) P and $SK(P)$ have the same stable theory semantics.

4.3.2 Preserving the Semantic Kernel

We now show that in any transformation sequence $P_0, P_1, \ldots, P_n$ where P_{i+1} is obtained from P_i by applying unfolding (rule 4.1) or folding (rule 4.2), the semantic kernel is preserved, i.e., $SK(P_0) = SK(P_1) = \ldots = SK(P_n)$. To do so, we introduce the following notion of a positive ground derivation:

Definition 4.4 (Positive ground derivation) *A positive ground derivation of a literal in a normal logic program P is a tree T such that: (1) each internal node of T is labeled with a ground atom (2) each leaf node of T is labeled with a negative ground literal or the special symbol* `true`*, and (3) for any internal node A of T, $A :- L_1, \ldots, L_n$ must be a ground instance of a clause in program P where $L_1, \ldots, L_n$ are the children of A in T.*

Thus, consider any positive ground derivation T in program P. Let the root of T be labeled with the ground literal L and let $\mathcal{N}$ be the *sequence of negative literals*[1] derived in T, i.e., $\mathcal{N}$ is formed by appending the negative literals appearing in the leaf nodes of T from left to right. Then we say that L derives $\mathcal{N}$ in P, and denote such derivations by $L \rightsquigarrow_P \mathcal{N}$ (and $L \rightsquigarrow \mathcal{N}$ if P is obvious from the context). We overload this notation, often denoting existence of such derivations also by $L \rightsquigarrow_P \mathcal{N}$. Note that if L is a ground negative literal, there is only one positive ground derivation for L in any program, namely the *empty* derivation $L \rightsquigarrow L$. We now define:

Definition 4.5 (Weight of a derivation) *Let $L \rightsquigarrow_P \mathcal{N}$ be a positive ground derivation. The number of internal nodes in this derivation (*i.e. *the number of nodes labeled with a ground positive literal) is called the weight of the derivation.*

Definition 4.6 (Weight of a pair) *Let $P_0, P_1, \ldots, P_n$ be a transformation sequence of normal logic programs. Let L be a ground literal, $\mathcal{N}$ be a (possibly empty) sequence of ground negative literals s.t. $L \rightsquigarrow_{P_0} \mathcal{N}$. Then, the weight of $(L, \mathcal{N})$, denoted by $w(L, \mathcal{N})$, is the minimum of the weights of positive ground derivations of the form $L \rightsquigarrow_{P_0} \mathcal{N}$.*

Note that for *any* program P_i in the transformation sequence, the weight $w(L, \mathcal{N})$ is defined as the weight of the smallest derivation $L \rightsquigarrow_{P_0} \mathcal{N}$.

Definition 4.7 *Let $P_0, P_1, ..., P_n$ be a transformation sequence of normal logic programs. A positive ground derivation $L \rightsquigarrow_{P_i} \mathcal{N}$ is said to be weakly weight-consistent if for every ground instance $A :- L_1, ..., L_k$ of a clause C used in this derivation, we have $w(A, \mathcal{N}_A) \leq \gamma^i_{hi}(C) + \sum_{1 \leq l \leq k} w(L_l, \mathcal{N}_l)$ where $\mathcal{N}_A, \mathcal{N}_1, ..., \mathcal{N}_k$ are the sequence of negative literals derived from the literals $A, L_1, ..., L_k$ respectively in this derivation.*

[1] For simplicity of exposition we have considered $\mathcal{N}$ as a sequence. Note that our correctness proof holds even if $\mathcal{N}$ is regarded as a multi-set. This means that our unfold/fold transformations can be interleaved with useful transformations like *literal re-ordering*.

Definition 4.8 *Let $P_0, P_1, ..., P_n$ be a transformation sequence of normal logic programs. A positive ground derivation $L \rightsquigarrow_{P_i} \mathcal{N}$ is said to be strongly weight-consistent if for every ground instance $A :- L_1, ..., L_k$ of a clause C used in this derivation:*

- $w(A, \mathcal{N}_A) \geq \gamma^i_{lo}(C) + \sum_{1 \leq l \leq k} w(L_l, \mathcal{N}_l)$
- $\forall 1 \leq l \leq k \; w(A, \mathcal{N}_A) > w(L_l, \mathcal{N}_l)$

where $\mathcal{N}_A, \mathcal{N}_1, ..., \mathcal{N}_k$ are the negative literal sequences derived from the literals $A, L_1, ..., L_k$ respectively in this derivation.

Definition 4.9 *Let $P_0, P_1, \ldots, P_n$ be a transformation sequence of normal logic programs. Then, program P_i is said to be weight consistent if*

- *for any pair $(L, \mathcal{N})$, whenever L derives $\mathcal{N}$ in P_i, there is a strongly weight consistent positive ground derivation $L \rightsquigarrow_{P_i} \mathcal{N}$.*
- *every positive ground derivation in P_i is weakly weight consistent.*

Using the above definitions, we now state certain invariants which always hold after the application of any unfold/fold transformation.

- $I1(P_i) \equiv \forall L \forall \mathcal{N} \; (L \rightsquigarrow_{P_0} \mathcal{N} \Leftrightarrow L \rightsquigarrow_{P_i} \mathcal{N})$.
- $I2(P_i) \equiv$ P_i is a weight consistent program

We now show that these invariants are maintained after every unfolding and folding step. This allows us to claim that the set of positive ground derivations of P_0 is identical to the set of positive ground derivations of program P_i.

Lemma 4.2 *If $(\forall j \leq i \; I1(P_j))$ holds, then $\forall L \forall \mathcal{N} \; (\; L \rightsquigarrow_{P_{i+1}} \mathcal{N} \Rightarrow L \rightsquigarrow_{P_i} \mathcal{N} \;)$*

Proof: We consider a positive ground derivation $Dr \equiv L \rightsquigarrow \mathcal{N}$ in P_{i+1}, and construct a derivation $L \rightsquigarrow \mathcal{N}$ in P_i by induction on the weight (*i.e* the number of internal nodes) of Dr. If L is a negative literal, then the result is obvious since then $\mathcal{N} = L$, *i.e.* Dr is empty. Otherwise, let $L :- L_1, ..., L_n$ be the ground instance of a clause $C \in P_{i+1}$ used at the root of Dr. For all $1 \leq l \leq n$ let $\mathcal{N}_l$ be the sequence of negative ground literals derived from L_l in the derivation Dr. Then, by induction hypothesis, we have derivations $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$ for all $1 \leq l \leq n$. We consider three possible cases.

Case 1 : C was inherited from P_i

$L \rightsquigarrow_{P_i} \mathcal{N}$ is constructed by applying $L :- L_1, ..., L_n$ at the root and then resolving $L_1, ..., L_n$ by using the derivations $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$.

Case 2 : C was obtained by *unfolding*

Then, without loss of generality, P_i contains clauses with ground instantiations $L :- B, L_{k+1}, ..., L_n$ and $B :- L_1, ..., L_k$. We construct derivation $L \rightsquigarrow_{P_i} \mathcal{N}$ by first applying the clause $L :- B, L_{k+1}, ..., L_n$ then the clause $B :- L_1, ...L_k$ and then resolving $L_1, ..., L_n$ by using $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$ for all $1 \leq l \leq n$.

Case 3: C was obtained by *folding*

Let L_1 be the atom introduced by folding, and let $P_j (j \leq i)$ be the program from which folder clauses were picked. By induction hypothesis, we have derivations $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$ for all $1 \leq l \leq n$. Then, $L_1 \rightsquigarrow_{P_i} \mathcal{N}_1$ and hence $L_1 \rightsquigarrow_{P_0} \mathcal{N}_1$ (by $I1(P_i)$) and hence $L_1 \rightsquigarrow_{P_j} \mathcal{N}_1$ (by $I1(P_j)$). Let a derivation $L_1 \rightsquigarrow \mathcal{N}_1$ in P_j be called Dr_j and let $L_1 :- L_{1,1}, ..., L_{1,k}$ be the ground clause used at the root of Dr_j. Then, by conditions (3) and (4) of the folding transformation, there must be a ground instantiation of a clause in P_i (one of the folded clauses) of the form $L :- L_{1,1}, ..., L_{1,k}, L_2, ..., L_n$. Also, by $I1(P_j)$, we must have derivations $L_{1,l} \rightsquigarrow_{P_0} \mathcal{N}_{1,l}$ for all $1 \leq l \leq k$, where $\mathcal{N}_{1,l}$ is the sequence of negative literals derived from $L_{1,l}$ in Dr_j. Then, by $I1(P_i)$, we must have derivations $L_{1,l} \rightsquigarrow_{P_i} \mathcal{N}_{1,l}$ for all $1 \leq l \leq k$. We can therefore construct $L \rightsquigarrow_{P_i} \mathcal{N}$ by applying the clause $L :- L_{1,1}, ..., L_{1,k}, L_2, ..., L_n$ at the root and then resolving $L_{1,1}, ..., L_{1,k}, L_2, ..., L_n$ by using $L_{1,l} \rightsquigarrow_{P_i} \mathcal{N}_{1,l}$ ($\forall\ 1 \leq l \leq k$) and $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$ ($\forall\ 2 \leq l \leq n$). □

Lemma 4.3 *Let $P_0, \ldots, P_i, P_{i+1}$ be an unfold/fold transformation sequence s.t. $\forall 0 \leq j \leq i\ I1(P_j) \wedge I2(P_j)$. Then, all positive ground derivations of P_{i+1} are weakly weight consistent.*

Proof: The proof proceeds by induction on the weight (*i.e* number of internal nodes) of positive ground derivations in P_{i+1}. Let $Dr \equiv L \rightsquigarrow \mathcal{N}$ be a derivation in P_{i+1} and let $L :- L_1, \ldots L_n$ be the ground instance of a clause $C \in P_{i+1}$ that is used at the root of Dr. Then, the sub-derivations $L_l \rightsquigarrow_{P_{i+1}} \mathcal{N}_l$ (for all $1 \leq l \leq n$) of Dr are weakly weight consistent by induction hypothesis, where $\mathcal{N}_1, \ldots, \mathcal{N}_n$ are the sequence of negative literals derived from $L_1, \ldots, L_n$ in derivation Dr. Hence it suffices to show that $w(L, \mathcal{N}) \leq \gamma_{hi}^{i+1}(C) + \sum_{1 \leq l \leq n} w(L_l, \mathcal{N}_l)$. We consider three cases.

Case 1 : C was inherited from P_i.

Since $L_l \rightsquigarrow_{P_{i+1}} \mathcal{N}_l$, therefore by lemma 4.2, we have $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$. Thus $L :- L_1, \ldots, L_n$ is used at the root of a positive ground derivation in P_i. Since P_i is a weight consistent program and $\gamma_{hi}^{i+1}(C) = \gamma_{hi}^{i}(C)$, the result follows.

Case 2: C was obtained by unfolding.

Let $L_1, \ldots, L_k$ be the body literals of C which were introduced through unfolding. Then, without loss of generality, P_i contains clauses C and C'' which have ground instantiations $L :- B, L_{k+1}, ..., L_n$ and $B :- L_1, ..., L_k$. Also, by lemma 4.2, we have $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$ (for all $1 \leq l \leq n$). Then, the above mentioned two ground instances of C' and C'' are used in some positive ground derivation of P_i. Since P_i is weight consistent, we have :

$$w(L, \mathcal{N}) \leq \gamma_{hi}^{i}(C') + w(B, \mathcal{N}_B) + \sum_{k+1 \leq l \leq n} w(L_l, \mathcal{N}_l)$$
$$w(B, \mathcal{N}_B) \leq \gamma_{hi}^{i}(C'') + \sum_{1 \leq l \leq k} w(L_l, \mathcal{N}_l)$$

where $\mathcal{N}_B$ is the sequence of negative literals derived by B, *i.e.* $\mathcal{N}_B$ is obtained by appending $\mathcal{N}_1, \ldots, \mathcal{N}_k$. Then combining the above inequalities we have $w(L, \mathcal{N}) \leq \gamma_{hi}^{i+1}(C) + \sum_{1 \leq l \leq n} w(L_l, \mathcal{N}_l)$ since we know that $\gamma_{hi}^{i+1}(C) = \gamma_{hi}^{i}(C') + \gamma_{hi}^{i}(C'')$ by definition of unfolding.

Case 3: C was obtained by folding.

Let L_1 be the atom introduced by folding, and let $P_j (j \leq i)$ be the program from which folder clauses were picked. By lemma 4.2 we have $L_1 \rightsquigarrow_{P_i} \mathcal{N}_1$. Again since $I1(P_i) \wedge I1(P_j)$, therefore $L_1 \rightsquigarrow_{P_j} \mathcal{N}_1$. As P_j is a weight consistent program, therefore there exists a strongly weight consistent derivation $L_1 \rightsquigarrow \mathcal{N}_1$ in P_j. Let this derivation be called Dr_j. Let the clause used at the root of Dr_j be D' and let the ground instance of D' used at the root of Dr_j be $L_1 :- L_{1,1}, \ldots, L_{1,k}$. Then, by definition of strong weight consistency

$$w(L_1, \mathcal{N}_1) \geq \gamma_{lo}^{j}(D') + \sum_{1 \leq l \leq k} w(L_{1,l}, \mathcal{N}_{1,l})$$

where $\mathcal{N}_{1,1}, \ldots, \mathcal{N}_{1,k}$ is the sequence of negative literals derived by $L_{1,1}, \ldots, L_{1,k}$ in Dr_j. But D' must be a folder clause by definition of folding. Hence there must be

a clause C' in P_i with a ground instance $L :- L_{1,1}, \ldots, L_{1,k}, L_2, \ldots, L_n$ (this is the folded clause corresponding to D'). Now, by lemma 4.2 we have $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$ for all $2 \leq l \leq n$. Also since $I1(P_j) \wedge I1(P_i)$ therefore $L_{1,l} \rightsquigarrow_{P_i} \mathcal{N}_{1,l}$ for all $1 \leq l \leq k$. Therefore the ground clause $L :- L_{1,1}, \ldots, L_{1,k}, L_2, \ldots, L_n$ appears at the root of a positive ground derivation in P_i. As P_i is a weight consistent program, this derivation must be weakly weight consistent. Hence

$$\begin{aligned} w(L, \mathcal{N}) &\leq \gamma^i_{hi}(C') + \sum_{1 \leq l \leq k} w(L_{1,l}, \mathcal{N}_{1,l}) + \sum_{2 \leq l \leq n} w(L_l, \mathcal{N}_l) \\ &\leq \gamma^i_{hi}(C') - \gamma^j_{lo}(D') + w(L_1, \mathcal{N}_1) + \sum_{2 \leq l \leq n} w(L_l, \mathcal{N}_l) \\ &\leq \gamma^i_{hi}(C') - \gamma^j_{lo}(D') + \sum_{1 \leq l \leq n} w(L_l, \mathcal{N}_l) \\ &\leq \gamma^{i+1}_{hi}(C) + \sum_{1 \leq l \leq n} w(L_l, \mathcal{N}_l) \end{aligned}$$

This completes the proof. □

We now establish the main theorem concerning the preservation of positive ground derivations in a transformation sequence.

Theorem 4.4 *Let $P_0, P_1, \ldots$ be a sequence of normal logic programs where P_{i+1} is obtained from P_i by applying unfolding (rule 4.1) or folding (rule 4.2). Then we have $\forall i \geq 0$ $I1(P_i) \wedge I2(P_i)$.*

Proof : The proof proceeds by induction on i. For the base case, $I1(P_0)$ holds trivially, and $I2(P_0)$ holds because every positive ground derivation of P_0 is weakly weight consistent, and for any pair $(L, \mathcal{N})$ the smallest positive ground derivation $L \rightsquigarrow_{P_0} \mathcal{N}$ is strongly weight consistent.

For the induction step, we need to show $I1(P_{i+1}) \wedge I2(P_{i+1})$. By Lemma 4.2 we have $L \rightsquigarrow_{P_{i+1}} \mathcal{N} \Rightarrow L \rightsquigarrow_{P_i} \mathcal{N}$, and by Lemma 4.3 we know that all positive ground derivations of P_{i+1} are weakly weight consistent. We need to show that:

1. $L \rightsquigarrow_{P_i} \mathcal{N} \Rightarrow L \rightsquigarrow_{P_{i+1}} \mathcal{N}$, and
2. for any pair $(L, \mathcal{N})$ s.t. $L \rightsquigarrow_{P_{i+1}} \mathcal{N}$, there exists a strongly weight consistent derivation $L \rightsquigarrow \mathcal{N}$ in P_{i+1}.

Thus, it suffices to prove that for any pair $(L, \mathcal{N})$ s.t $L \rightsquigarrow_{P_i} \mathcal{N}$, there exists a strongly weight consistent derivation $L \rightsquigarrow_{P_{i+1}} \mathcal{N}$.

Consider a pair $(L, \mathcal{N})$ such that $L \rightsquigarrow_{P_i} \mathcal{N}$. Since P_i is weight consistent, therefore there exists a strongly weight consistent derivation $L \rightsquigarrow \mathcal{N}$ in P_i. Let this be called Dr. We now need to construct a strongly weight consistent derivation $Dr' \equiv L \rightsquigarrow_{P_{i+1}} \mathcal{N}$. Construction of Dr' proceeds by induction on the *weight of* $(L, \mathcal{N})$ *pairs*.[2]

The base case occurs when L is a negative literal, $\mathcal{N} = L$ and $w(L, \mathcal{N}) = 0$. We then trivially have the same derivation $L \rightsquigarrow \mathcal{N}$ in P_{i+1} as well. Otherwise if L is a positive literal, let C be the clause used at the root of Dr. Let $L :- L_1, \ldots, L_n$ be the ground instantiation of C used at the root of Dr. Since Dr is strongly weight consistent $w(L, \mathcal{N}) > w(L_l, \mathcal{N}_l)$ where $\mathcal{N}_l$ is the sequence of negative literals derived by L_l for all $1 \leq l \leq n$. Hence, we have strongly weight consistent derivations $L_l \rightsquigarrow_{P_{i+1}} \mathcal{N}_l$. We construct Dr' by considering the following cases :

Case 1: C is inherited from P_i to P_{i+1}

Dr' is constructed with the clause $L :- L_1, \ldots, L_n$ at the root and then appending the derivations $L_l \rightsquigarrow_{P_{i+1}} \mathcal{N}_l$ for all $1 \leq l \leq n$. This derivation Dr' is strongly weight consistent.

Case 2: C is unfolded.

Let the L_1 be the positive body literal of C that is unfolded. Let the clause used to resolve L_1 in the derivation Dr be C_1 and the ground instance of C_1 used be $L_1 :- L_{1,1}, \ldots, L_{1,k}$. By definition of unfolding $L :- L_{1,1}, \ldots, L_{1,k}, L_2, \ldots, L_n$ is a ground instance of a clause C_1' in P_{i+1} with $\gamma_{lo}^{i+1}(C_1') = \gamma_{lo}^{i}(C) + \gamma_{lo}^{i}(C_1)$. Also, let $\mathcal{N}_{1,1}, \ldots, \mathcal{N}_{1,k}$ be the sequence of negative literals derived by $L_{1,1}, \ldots, L_{1,k}$ in Dr. Then, by strong weight consistency $w(L_{1,l}, \mathcal{N}_{1,l}) < w(L_1, \mathcal{N}_1) < w(L, \mathcal{N})$ for all $1 \leq l \leq k$. Thus we have strongly weight consistent derivations $L_{1,l} \rightsquigarrow_{P_{i+1}} \mathcal{N}_{1,l}$. We construct Dr' by applying $L :- L_{1,1}, \ldots, L_{1,k}, L_2, \ldots, L_n$ at the root and then appending the strongly weight consistent derivations $L_{1,l} \rightsquigarrow_{P_{i+1}} \mathcal{N}_{1,l}$ (for all $1 \leq l \leq k$) and $L_l \rightsquigarrow_{P_{i+1}} \mathcal{N}_l$ (for all $2 \leq l \leq n$). Since Dr is strongly weight consistent, therefore

$$
\begin{array}{ll}
 & w(L, \mathcal{N}) \geq \gamma_{lo}^{i}(C) + \sum_{1 \leq l \leq n} w(L_l, \mathcal{N}_l) \\
\text{and} & w(L_1, \mathcal{N}_1) \geq \gamma_{lo}^{i}(C_1) + \sum_{1 \leq l \leq k} w(L_{1,l}, \mathcal{N}_{1,l}) \\
\Rightarrow & w(L, \mathcal{N}) \geq \gamma_{lo}^{i+1}(C_1') + \sum_{1 \leq l \leq k} w(L_{1,l}, \mathcal{N}_{1,l}) + \sum_{2 \leq l \leq n} w(L_l, \mathcal{N}_l)
\end{array}
$$

This shows that Dr' is strongly weight consistent.

[2]This construction is similar to the construction of the strongly measure consistent proof T' in the proof of theorem 3.3 in chapter 3.

Case 3: C is folded

Let C (potentially with other clauses) be folded, using folder clause(s) from $P_j (j \leq i)$, to clause C' in P_{i+1}. Assume that $L_1, \ldots, L_k$ are the instances of the body literals of C which are folded. Then, C' must have a ground instance of the form $L :- B, L_{k+1}, \ldots, L_n$, where $B :- L_1, \ldots, L_k$ is a ground instance of a folder clause D in P_j. Since, we have derivations $L_l \rightsquigarrow_{P_i} \mathcal{N}_l$ for all $1 \leq l \leq k$, therefore by $I1(P_i) \wedge I1(P_j)$ there exist derivations $L_l \rightsquigarrow_{P_j} \mathcal{N}_l$. Then, there exists a derivation $B \rightsquigarrow_{P_j} \mathcal{N}_B$ where $\mathcal{N}_B$ is obtained by appending the sequences $\mathcal{N}_1, \ldots, \mathcal{N}_k$. Since P_j is a weight consistent program, this derivation must be weakly weight consistent, and therefore $w(B, \mathcal{N}_B) \leq \gamma^j_{hi}(D) + \sum_{1 \leq l \leq k} w(L_l, \mathcal{N}_l)$. By strong weight consistency of Dr, we have

$$\begin{aligned} w(L, \mathcal{N}) &\geq \gamma^i_{lo}(C) + \sum_{1 \leq l \leq k} w(L_l, \mathcal{N}_l) + \sum_{k+1 \leq l \leq n} w(L_l, \mathcal{N}_l) \\ &\geq \gamma^i_{lo}(C) + w(B, \mathcal{N}_B) - \gamma^j_{hi}(D) + \sum_{k+1 \leq l \leq n} w(L_l, \mathcal{N}_l) \qquad \cdots\cdots (*) \\ &> w(B, \mathcal{N}_B) \text{ (by condition (5) of folding)} \end{aligned}$$

Thus there exists a strongly weight consistent derivation $B \rightsquigarrow_{P_{i+1}} \mathcal{N}_B$. We construct Dr' with $L :- B, L_{k+1}, \ldots, L_n$ at the root and then appending the strongly weight consistent derivations $B \rightsquigarrow_{P_{i+1}} \mathcal{N}_B, L_{k+1} \rightsquigarrow_{P_{i+1}} \mathcal{N}_{k+1}, \ldots, L_n \rightsquigarrow_{P_{i+1}} \mathcal{N}_n$. To show that Dr' is strongly weight consistent, note that $\gamma^{i+1}_{lo}(C') \leq \gamma^i_{lo}(C) - \gamma^j_{hi}(D)$ since C and D are folded and folder clauses. Combining this with (*),

$$w(L, \mathcal{N}) \geq \gamma^{i+1}_{lo}(C') + w(B, \mathcal{N}_B) + \sum_{k+1 \leq l \leq n} w(L_l, \mathcal{N}_l)$$

This completes the proof. □

Thus we have shown that all positive ground derivations are preserved at every step of our transformation. Now we show how our notion of positive ground derivations directly corresponds to the notion of semantic kernel. Intuitively, this connection is clear, since a clause in the semantic kernel of program P is derived by repeatedly resolving the positive body literals of a ground instance of a clause in P until the body contains only negative literals. Formally, we prove that :

Theorem 4.5 *Let P be a normal logic program and $A, B_1, \ldots, B_n (n \geq 0)$ be ground atoms in the Herbrand base of P. Let $\mathcal{N}$ denote the sequence of negative literals $\neg B_1, \ldots, \neg B_n$. Then, A derives $\mathcal{N}$ in P iff $(A :- \mathcal{N}) \in SK(P)$.*

Proof : We prove $A \rightsquigarrow_P \mathcal{N} \Rightarrow (A :- \mathcal{N}) \in SK(P)$ by induction on the weight of the derivation $A \rightsquigarrow_P \mathcal{N}$. The base case occurs when the derivation has a weight of 1 *i.e.* it has only 1 internal node. Then $A :- \mathcal{N}$ must be a ground instance of a clause in P. Then, by definition of S_P, we know $(A :- \mathcal{N}) \in S_P(\phi)$. Hence $(A :- \mathcal{N}) \in SK(P)$. For the induction step, let the property be true for all positive ground derivations of weight $< m$, and consider a derivation Dr of the form $A = G_0, G_1, \ldots, G_n = \mathcal{N}$ with weight m. Then, $A :- G_1$ must be a ground instance of a clause in program P and G_1 must be of the form $\neg B_1, \ldots, \neg B_j, A_1, \ldots, A_k$ $(j \geq 0, k \geq 1)$. Let $\mathcal{N}_1, \ldots, \mathcal{N}_k$ be the (possibly empty) sequence of negative literals derived from $A_1, \ldots, A_k$ in derivation Dr. Then, for all $1 \leq i \leq k$, we have $A_i \rightsquigarrow_P \mathcal{N}_i$ and the positive ground derivation $A_i \rightsquigarrow \mathcal{N}_i$ has weight $< m$. By induction hypothesis $(A_i :- \mathcal{N}_i) \in SK(P)$. Hence, for all $1 \leq i \leq k$, $(A_i :- \mathcal{N}_i) \in SK^n(P)$ for some n. Note that, by definition of sequences $\mathcal{N}_i$, we have $\mathcal{N} = \neg B_1, \ldots, \neg B_n = \neg B_1, \ldots, \neg B_j, \mathcal{N}_1, \ldots, \mathcal{N}_k$. Then, clearly by definition of function S_P we have $(A :- \mathcal{N}) \in S_P(SK^n(P))$, *i.e.* $(A :- \mathcal{N}) \in SK^{n+1}(P)$. Hence $(A :- \mathcal{N}) \in SK(P)$.

Now we prove that $(A :- \mathcal{N}) \in SK(P) \Rightarrow A \rightsquigarrow_P \mathcal{N}$ by fixed-point induction. Recall that $SK(P)$ is the least fixed point of the function $S_P : QI(P) \rightarrow QI(P)$ where $QI(P)$ denotes the set of all quasi-interpretations of P. Also note that $(QI(P), \subseteq)$ is a complete partial order with a least element (the empty set ϕ), where $\subseteq$ is the set inclusion relation. Hence to prove the above result by fixed-point induction we need to prove :

1. $(A :- \mathcal{N}) \in \phi \Rightarrow A \rightsquigarrow_P \mathcal{N}$
2. For any quasi-interpretation I, if for any $(A, \mathcal{N})$ we have $(A :- \mathcal{N}) \in I \Rightarrow A \rightsquigarrow_P \mathcal{N}$, then for any $(A, \mathcal{N})$ we must have $(A :- \mathcal{N}) \in S_P(I) \Rightarrow A \rightsquigarrow_P \mathcal{N}$

The first proof obligation trivially holds. To prove the second proof obligation, consider any ground clause $C' \equiv A :- \mathcal{N}$ in $S_P(I)$. By definition of function S_P (definition 4.2), we have $C' = \mathcal{R}(C, D_1, \ldots, D_m)$. Thus, clause C is of the form $A :- \neg B_1, \ldots, \neg B_j, A_1, \ldots, A_m$ and clause D_i $(1 \leq i \leq m)$ is of the form $A_i :- \mathcal{N}_i$ where $\mathcal{N}_i$ is a finite (possibly empty) sequence of ground negative literals. Also $C \in Gnd(P)$ and $D_i \in I$. Then, by induction hypothesis we have $A_i \rightsquigarrow_P \mathcal{N}_i$ for all $1 \leq i \leq m$. Thus we can construct the derivation $A \rightsquigarrow_P \mathcal{N}$ by applying the clause $C \equiv (A :- \neg B_1, \ldots, \neg B_j, A_1, \ldots, A_m)$ at the root, and then resolving $A_1, \ldots, A_m$ by using $A_i \rightsquigarrow_P \mathcal{N}_i$ (for all $1 \leq i \leq m$). □

We can now prove that the semantic kernel is preserved across any unfold/fold transformation sequence.

Corollary 4.6 (Preservation of Semantic Kernel) *Suppose $P_0, \ldots, P_n$ is a sequence of normal logic programs where P_{i+1} is obtained from P_i by unfolding (Rule 4.1) or folding (Rule 4.2). Then $\forall 0 < i \leq n$ $SK(P_i) = SK(P_0)$.*

Proof: We prove that $SK(P_0) = SK(P_i)$ for any arbitrary i. By Theorem 4.4 we know that $A \rightsquigarrow_{P_0} \mathcal{N} \Leftrightarrow A \rightsquigarrow_{P_i} \mathcal{N}$ for any ground atom A and sequence of ground negative literals $\mathcal{N}$. Then, using Theorem 4.5 we get $(A :\!- \mathcal{N}) \in SK(P_0) \Leftrightarrow (A :\!- \mathcal{N}) \in SK(P_i)$. Thus, $SK(P_0) = SK(P_i)$. □

Following Theorem 4.1 and Corollary 4.6 we have:

Theorem 4.7 (Correctness of Unfolding/Folding) *Let $P_0, \ldots, P_n$ be a sequence of normal logic programs where P_{i+1} is obtained from P_i by an application of unfolding (Rule 4.1) or folding (Rule 4.2). Then, for all $0 < i \leq n$ we have*
(1) If P_0 is a definite logic program, then P_0 and P_i have the same least Herbrand Model.
(2) If P_0 is a stratified program, then P_0 and P_i have the same perfect model semantics.
(3) P_0 and P_i have the same well-founded model.
(4) P_0 and P_i have the same stable model(s).
(5) P_0 and P_i have the same set of partial stable models.
(6) P_0 and P_i have the same stable theory semantics.

4.4 Discussions

Goal Replacement The transformation system presented in this chapter can be extended to incorporate a *goal replacement* rule which allows the replacement of a conjunction of atoms in the body of a clause with another semantically equivalent conjunction of atoms provided certain conditions are satisfied (which ensure preservation of weight consistency). In future, it would be interesting to study how we can perform multiple replacements simultaneously without compromising correctness (as discussed in [BCE96]).

Motivation The motivation of the general unfold/fold transformation system reported here is verification of parameterized concurrent systems. Proving temporal properties of parameterized systems requires reasoning about each of the members of the infinite family, which can be accomplished by induction. We have used generalized unfold/fold transformations of definite logic programs (presented in the last chapter)

to verify liveness and safety properties of parameterized systems. However, temporal properties containing both greatest and least fixed point operators *cannot* be encoded as a definite logic program. A trivial example of such a property is the CTL formula $\mathbf{AG}(in \Rightarrow \mathbf{EF}out)$ which could be described in English as follows: "*always* if an input event occurs then an output event *eventually* occurs". This property contains the *always* operator (defined as a greatest fixed point) and the *eventuality* operator (defined as a least fixed point), and is commonly verified for hardware circuits. To use unfold/fold transformations to construct induction proofs of such temporal properties we need to extend our generalized unfold/fold transformation system to normal logic programs. This indeed has been a motivation for developing the transformation system presented here.

Implications of the Correctness Proof Apart from the transformation system, the details of the underlying correctness proof reveal certain interesting aspects generic to such transformation systems. First of all, our proof exploits a degree of modularity that is inherent in the unfold/fold transformations for logic programs. Consider a modular decomposition of a definite logic program where each predicate is fully defined in a single module. Each module has a set of "local" predicates defined in the current module and a set of "external" predicates used (and not defined) in the current module. It is easy to see that Lemma 4.2, 4.3 and Theorem 4.4 can be modified to show that unfold/fold transformations preserve the set of "*local ground derivations*" of a program. We say that $A \rightsquigarrow B_1, B_2, \ldots, B_n$ is a local ground derivation (analogous to a positive ground derivation), if each B_i contains an external predicate, and there is a proof tree rooted at A whose leaves are labeled with $B_1, \ldots, B_n$ (apart from true). Consequently, transformations of a normal logic program P, can be simulated by an equivalent positive program module Q obtained by replacing negative literals in P with new positive external literals. The newly introduced literals can be appropriately defined in a separate module. Thus any transformation system for definite logic programs that preserves local ground derivations also preserves the semantic kernels of normal logic programs.

Secondly, we showed that positive ground derivations form the operational counterpart to semantic kernels. This result, which makes explicit an idea in the proof of Aravindan and Dung [AD95], enables the correctness proof to be completed by connecting the other two steps: an operational first step, where the measure consistency technique is used to show the preservation of positive ground derivations and the

final model-theoretic step that applies the results of Dung and Kanchanasut [DK89] relating semantic kernels to various semantics for normal logic programs.

Interestingly, by its very nature, the notion of semantic kernel cannot be used in proving operational equivalences such as finite failure and computed answer sets. The important task then is to formulate a suitable operational notion that plays the role of semantic kernel in the correctness proofs with respect to these equivalences.

Chapter 5

Proofs by Program Transformations

In our approach for parameterized system verification, the parameterized system and the temporal property to be verified are encoded as a logic program. The verification proof obligation is reduced to establishing predicate equivalences[1]. In this chapter, we design strategies to guide the application of program transformations for proving predicate equivalences.

Section 5.1 presents a formal description of a tableau based proof system for solving predicate equivalences. Application of the rules in this proof system constructs a proof via program transformations as outlined in Chapter 2. Section 5.2 presents automated instances of each of the proof rules. Section 5.3 contains an algorithmic framework to create strategies for guiding the application of the proof rules. In the next chapter, we will discuss development of concrete strategies by instantiating this algorithmic framework.

Assumptions: The abstract transformation rules presented in the previous chapters can be directly used for constructing proofs. For purposes of clear and concrete presentation, we assume the following for the rest of the dissertation.

- the logic program encoding P_0 of the verification problem for parameterized systems (and the subsequent logic programs $P_1, P_2 \ldots$ obtained by transforming it) is a *definite* logic program.

[1]Note that the proof technique outlined here can be easily adapted to prove implications instead of equivalences.

- The Measure Structure (refer Definition 3.1) for the unfolding, folding, goal replacement transformations is $\langle \mathbb{Z}, +, < \mathbb{N} \rangle$. Thus, the clause measures for any clause in a program P_i of a program transformation sequence $P_0, P_1, \ldots, P_n$ is a pair of integers.

- In any transformation sequence $P_0, P_1, \ldots, P_n$, the clauses of P_0 are annotated with $(1, 1)$.

- $\forall C \in P_{i+1} - P_i$ (i.e. new clauses in P_{i+1}) $\gamma^i_{lo}(C) = GLB^{i+1}(C)$ and $\gamma^i_{hi}(C) = LUB^{i+1}(C)$. In other words, the measures of a clause obtained by unfolding (folding) are calculated via integer addition (subtraction).

- The atom measure $\alpha(A)$ of a ground atom $A \in M(P_0)$ in a transformation sequence P_0,P_1,...,P_n is the number of nodes in the smallest ground proof of A in P_0.

The preservation of least Herbrand Model Semantics of any interleaved application of such transformations directly follows from Theorems 3.3 and 3.6.

5.1 A Proof System for Predicate Equivalences

Formally, the *predicate equivalence problem* is: given a logic program P and a pair of predicates p and p' of the same arity, determine if $P \models p \equiv p'$ *i.e.* whether p and p' are semantically equivalent in P. In other words, we need to determine whether for all ground substitutions θ, $p(\overline{X})\theta \in M(P) \Leftrightarrow p'(\overline{X})\theta \in M(P)$. Recall that $M(P)$ denotes the least Herbrand model of program P.

We develop a tableau-based proof system for establishing predicate equivalence. The proof system presented here can be straightforwardly extended to prove goal equivalences[2] instead of predicate equivalences. Our process is analogous to SLD resolution. Recall that given a goal $\mathcal{G}$ and a definite logic program P, SLD resolution is used to prove whether instances of $\mathcal{G}$ are in $M(P)$. This proof is constructed recursively by deriving new goals via resolution. The truth of $\mathcal{G}$ is then shown by establishing the truth of these new goals. In contrast, each node in our proof tree denotes a pair of predicates (p, p'). To establish their equivalence we must establish that the predicates in the pair represented by each child node are equivalent. Note

[2]Recall that in a definite logic program, a goal is a conjunction of atoms.

Name	Top-down Inference (one step)	Side Condition
(Ax)	$\dfrac{\mathcal{E},\quad \Gamma \vdash p \equiv p',\quad \mathcal{E}'}{\mathcal{E},\quad \mathcal{E}'}$	$p \overset{P_i}{\cong} p'$
(Tx)	$\dfrac{\mathcal{E},\quad \Gamma \vdash p \equiv p',\quad \mathcal{E}'}{\mathcal{E},\quad \Gamma, P_{i+1} \vdash p \equiv p',\quad \mathcal{E}'}$	$M(P_{i+1}) = M(P_i)$
(Gen)	$\dfrac{\mathcal{E},\quad \Gamma \vdash p \equiv p',\quad \mathcal{E}'}{\mathcal{E},\quad \Gamma, P_{i+1} \vdash p \equiv p',\quad P_0 \vdash q \equiv q',\quad \mathcal{E}'}$	$P_0 \models q \equiv q'$ $\Rightarrow M(P_{i+1}) = M(P_i)$

Table 2: Proof System for showing Predicate equivalences

that the predicates in the child node are to be obtained from the syntax of the current definitions of p, p'. We now define:

Definition 5.1 (e-atom) *Let $\Gamma = P_0, P_1, \ldots, P_i$ be a sequence of programs. An e-atom is of the form $\Gamma \vdash p \equiv p'$ where p and p' are predicates of same arity appearing in each of the programs in Γ. It represents the proof obligation $\forall 0 \leq j \leq i\ P_j \models p \equiv p'$ i.e. p, p' are semantically equivalent in each of the programs in Γ.*

We generalize the problem of establishing a single e-atom to that of establishing a sequence of e-atoms. We define an *e-goal* as a (possibly empty) sequence of e-atoms. We will often denote an e-goal by $\mathcal{E}$, possibly with primes and subscripts. Recall that SLD resolution proves a goal by unfolding an atom in the goal. Similarly, we proceed to prove an e-goal by transforming the relevant clauses of an e-atom (*i.e.* the clauses of the predicates appearing in the e-atom) in the e-goal.

The three rules used to construct an equivalence tableau are shown in Table 2. In the description of the proof rules Γ denotes a sequence of programs $P_0, \ldots, P_i$. Given a definite logic program P_0, and a pair of predicates of same arity p, p', we construct a tableau for the proof obligation $P_0 \vdash p \equiv p'$ by repeatedly applying the inference rules in Table 2.

The *axiom elimination rule* (**Ax**) is applicable whenever the equivalence of the predicates p and p' can be established by some automatic mechanism, denoted in the rule by $p \overset{P_i}{\cong} p'$. Thus, $\overset{P_i}{\cong}$ is a decision procedure which infers the equivalence of p, p' in program P_i. Axiom elimination will typically be an application of what we call "syntactic equivalence", a decidable equivalence of predicates based on the syntactic form of the clauses defining them (see Definition 5.3).

The *program transformation rule* (**Tx**) attempts to simplify the program in order to expose the equivalence of predicates (which can then be inferred via an application of **Ax**). The program P_{i+1} is constructed from Γ using a *semantics preserving* program transformation. We use this rule whenever we apply an unfolding, folding, or any other (semantics-preserving) transformation that does not add any equivalence proof obligations.

The *equivalence generation rule* (**Gen**) proves an e-atom $\Gamma \vdash p \equiv p'$ by performing replacements in the clauses of p, p'. In particular, occurrence of some predicate q in the clauses of p, p' is replaced by occurrence of another predicate q'. The guarantee is that if the predicates q, q' are semantically equivalent then the program thus obtained is semantics preserving. This appears as the side condition of the **Gen** rule. The notation $P_0 \models q \equiv q'$ is a shorthand for the following: for all ground substitution θ, $q(\overline{X})\theta) \in M(P_0) \Leftrightarrow q'(\overline{X})\theta) \in M(P_0)$ where $M(P_0)$ is the least Herbrand model of P_0. Note the proof of semantic equivalence of p and p' is being constructed by using the semantic equivalence of q and q'. This allows us to simulate nested induction proofs. Typically, an application of the **Gen** rule corresponds to applying the goal replacement transformation.[3]

The notion of a tableau for a predicate equivalence proof obligation in a definite logic program P_0 is then defined in the usual way.

Definition 5.2 (Equivalence Tableau) *An* equivalence tableau *of an e-goal $\mathcal{E}_0$ is a finite sequence of e-goals $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_n$ where $\mathcal{E}_{i+1}$ is obtained from $\mathcal{E}_i$ by applying one of the rules described in Table 2 and $\mathcal{E}_n$ is empty.*

Now, let P_0 be a definite logic programs and p, p' be predicates of same arity appearing in P_0. Then we use our proof system to construct an equivalence tableau of $\mathcal{E}_0 = (P_0 \vdash p \equiv p')$.

Theorem 5.1 (Soundness of Proof System) *Let $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_n$ be a successful tableau with $\mathcal{E}_0 = (P_0 \vdash p \equiv p')$ for some (definite) logic program $\mathcal{P}_0$. Then $P_0 \models p \equiv p'$ i.e. predicates p and p' are semantically equivalent in the least Herbrand model of P_0.*

Proof: We prove a stronger result. For any successful tableau of an e-goal $\mathcal{E}_0$ if $\Gamma \vdash p \equiv p'$ is an e-atom in $\mathcal{E}_0$ where $\Gamma = P_0, \ldots, P_i$ then $P_i \models p \equiv p'$.

[3]The **Gen** rule does not require $\{p, p'\} \cap \{q, q'\} = \phi$. When we synthesize an algorithmic framework for applying the proof rules we will keep track of the past history of equivalence proof obligations.

The proof for this result is established by induction on the length of the tableau. For the base case, we have a tableau of length 1, which is formed by an application of the **Ax** rule. For such a tableau the result holds trivially since **Ax** is applied only when the semantic equivalence of p, p' can be automatically inferred in P_i. For the induction step, we consider a tableau $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_{k+1}$ of length $k+1$. For all e-atoms of $\mathcal{E}_0$ which are not modified in the step $\mathcal{E}_0 \rightarrow \mathcal{E}_1$, the result follows by induction hypothesis. Let $P_0, \ldots, P_i \vdash p \equiv p'$ be the e-atom in $\mathcal{E}_0$ that is modified.

- **Ax**: If the rule applied to $\mathcal{E}_0$ is **Ax**, then from the side condition of **Ax** we have $P_i \models p \equiv p'$.
- **Tx** : If the rule applied to $\mathcal{E}_0$ is **Tx**, then $P_0, \ldots, P_i, P_{i+1} \vdash p \equiv p'$ is an e-atom in $\mathcal{E}_1$. Since $\mathcal{E}_1, \ldots, \mathcal{E}_{k+1}$ is a successful tableau of $\mathcal{E}_1$, therefore by induction hypothesis $P_{i+1} \models p \equiv p'$. By the side condition of **Tx**, we have $M(P_i) = M(P_{i+1})$ and therefore $P_i \models p \equiv p'$.
- **Gen** : If the rule applied to $\mathcal{E}_0$ is **Gen**, then $P_0, \ldots, P_i, P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$ are e-atoms in $\mathcal{E}_1$. Again $\mathcal{E}_1, \ldots, \mathcal{E}_{k+1}$ is a successful tableau of $\mathcal{E}_1$. By induction hypothesis, we have $P_{i+1} \models p \equiv p'$ and $P_0 \models q \equiv q'$. From the side condition of **Gen** we have $M(P_i) = M(P_{i+1})$ and therefore $P_i \models p \equiv p'$.

Now, if $\mathcal{E}_0, \ldots, \mathcal{E}_n$ is a successful tableau of $\mathcal{E}_0 = P_0 \vdash p \equiv p'$ then $P_0 \models p \equiv p'$. □

From the soundness of the proof system, we can also infer the following property for equivalence tableaus. It shows that for any e-atom $\Gamma \vdash \ldots$ appearing in an equivalence tableau, all programs in Γ are semantically equivalent.

Lemma 5.2 *Let $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_n$ be an equivalence tableau of $\mathcal{E}_0 = P_0 \vdash p \equiv p'$. For every e-atom $(\Gamma \vdash \ldots)$ in the tableau, if $\Gamma = P_0, \ldots, P_i$ then we have $M(P_0) = \ldots = M(P_i)$.*

Proof: We show that the above invariant is maintained in every step of the top-down tableau construction. Initially, the invariant is trivially true for the e-atom $\mathcal{E}_0 = (P_0 \vdash \ldots)$. For any application of **Ax**, since no e-atoms are added, the invariant holds. For any application of **Tx**, we get the e-atom $\Gamma, P_{i+1} \vdash p \equiv p'$ from the e-atom $\Gamma \vdash p \equiv p'$ where $\Gamma = P_0, \ldots, P_i$. We have $M(P_{i+1}) = M(P_i)$ from the side condition of **Tx**. Therefore the invariant must hold for $\Gamma, P_{i+1} \vdash p \equiv p'$ if it holds for $\Gamma \vdash p \equiv p'$. For any application of **Gen**, we obtain the e-atoms $\Gamma, P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$

from the e-atom $\Gamma \vdash p \equiv p'$ where $\Gamma = P_0, \ldots, P_i$. The invariant trivially holds for the e-atom $P_0 \vdash q \equiv q'$. Since $P_0 \vdash q \equiv q'$ is a proof obligation appearing in an equivalence tableau therefore $P_0 \vdash q \equiv q'$ itself has an equivalence tableau. By the soundness of the proof system shown by Theorem 5.1 we have $P_0 \models q \equiv q'$. Then, from the side condition of **Gen** we have $M(P_{i+1}) = M(P_i)$. Again the invariant must hold for $\Gamma, P_{i+1} \vdash p \equiv p'$ if it holds for $\Gamma \vdash p \equiv p'$. □

Note that the proof system given in Table 2 is not complete. There can be no such complete proof system as attested to by the following theorem.

Theorem 5.3 *Determining equivalence of predicates described by logic programs is not recursively enumerable.*

The theorem is easily proved using a reduction described in [AK86]. For a Turing machine M, we construct a program having two predicates, one that describes the natural numbers and the other that identifies an n such that M does not halt within n moves. These predicates are equivalent if and only if M does not halt. The non-halting problem is not recursively enumerable and so the predicate equivalence problem cannot be recursively enumerable.

5.2 Automated Instances of Proof Rules

In this section, we discuss the automation of each application of an **Ax**, **Tx** or **Gen** rule. In the next section we present an algorithmic framework for guiding the application of these rules. The application of the **Tx** and **Gen** rules is achieved by unfolding, folding and goal replacement transformations presented in Chapter 3.

Ax rule: The *axiom elimination rule* (**Ax**) infers the equivalence of two predicates p, p' in a semantics preserving program transformation sequence $\Gamma = P_0, \ldots, P_i$. In the light of Theorem 5.3, any such rule will be incomplete. Therefore, we will construct an effectively checkable sufficient condition for predicate equivalence. We call this sufficient condition as syntactic equivalence. Given a program transformation sequence $\Gamma = P_0, \ldots, P_i$ and two predicates p, p', we apply **Ax** if p, p' are syntactically equivalent in program P_i.

As an illustration, consider the program P (with clauses annotated with integer clause measures) in Figure 9. We can infer that $P \models \texttt{r} \equiv \texttt{s}$ since `r` and `s` have

```
p(X) :- r(X).          (γ1, γ1')    q(X) :- s(X).          (μ1, μ1')
p(X) :- e(X,Y), p(Y).  (γ2, γ2')    q(X) :- e(X,Y), q(Y).  (μ2, μ2')
r(X) :- b(X).          (γ3, γ3')    s(X) :- b(X).          (μ3, μ3')
```

Figure 9: Program with syntactically equivalent predicates.

identical definitions. Using the equivalence of `r` and `s` we can infer that $P \models \mathtt{p} \equiv \mathtt{q}$, since the definitions of `p` and `q` are, in a sense, isomorphic.

We formalize this notion of equivalence in the following definition. The following definition partitions the predicate symbols of a program into equivalence classes. Each predicate is assumed to be assigned a *label*, the partition number of the equivalence class to which it belongs. The labels of all predicates belonging to the same equivalence class is thus the same, and each equivalence class has a unique label.

Definition 5.3 (Syntactic Equivalence) *A syntactic equivalence relation $\overset{P}{\sim}$, is an equivalence relation on the set of predicates of a program P such that for all predicates p, q in P, if $p \overset{P}{\sim} q$ then the following conditions hold:*

1. *p and q have same arity, and*

2. *Let the clauses defining p and q be $\{C_1, \ldots, C_m\}$ and $\{D_1, \ldots, D_n\}$ respectively. Let $\{C'_1, \ldots, C'_m\}$ and $\{D'_1, \ldots, D'_n\}$ be such that C'_l (D'_l) is obtained by replacing every predicate symbol r in C_l (D_l) by s, where s is the label of the equivalence class of r (w.r.t. $\overset{P}{\sim}$). Then there exist two functions $f : \{1, \ldots, m\} \to \{1, \ldots, n\}$ and $g : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that*
   (i) $\forall 1 \leq i \leq m$ C'_i is an instance of $D'_{f(i)}$
   (ii) $\forall 1 \leq j \leq n$ D'_j is an instance of $C'_{g(j)}$.

Note that there is a largest syntactic equivalence relation. It can be computed by starting with all predicates in the same class, and repeatedly splitting the classes that violate properties (1) and (2) until a fixed point is reached. Also note that to show $p \overset{P}{\sim} q$ we need not establish a bijection between the clauses of p, q in P. As shown by the following lemma, the existence of the mapping f ensures that for any ground substitution θ we have $p(\overline{X})\theta \in M(P) \Rightarrow q(\overline{X})\theta \in M(P)$ whereas the mapping g ensures $q(\overline{X})\theta \in M(P) \Rightarrow p(\overline{X})\theta \in M(P)$.

Lemma 5.4 (Syntactic Equivalence ⇒ Semantic Equivalence) *Let $\overset{P}{\sim}$ be the syntactic equivalence relation of the predicates of a program P. For all predicates p, q, if $p \overset{P}{\sim} q$, then $P \models p \equiv q$.*

Proof : Let $p \overset{P}{\sim} q$. Then we show that for any ground proof T of a ground atom $p(\overline{X})\theta$ in program P there is a ground proof T' of $q(\overline{X})\theta$ in program P and vice-versa.

For any ground proof T of $p(\overline{X})\theta$ we can show the existence of a ground proof T' of $q(\overline{X})\theta$ by induction on the size (number of nodes) of T. Let the clause used at the root of T be $C = (p(\ldots) :- B_1, \ldots, B_k)$. Since $p \overset{P}{\sim} q$ therefore q has a clause $C' = (q(\ldots) :- B'_1, \ldots, B'_k)$ and $p_l \overset{P}{\sim} p'_l$ where p_l (p'_l) is the predicate symbol in B_l (B'_l) for all $1 \leq l \leq k$. Let $p_l(\overline{Y})\sigma$ be the ground instantiation of B_l appearing in T. Now, the size of the subproof of $p_l(\overline{Y})\sigma$ in T is clearly less than the size of T. By induction hypothesis there exists a ground proof of $p'_l(\overline{Y})\sigma$. Also $p'_l(\overline{Y})\sigma$ is an instance of $B'_l\theta$ since clause C is an instance of clause C' when all predicates are replaced by their labels. By applying clause C' at the root we can construct a ground proof T' of $q(\overline{X})\theta$.

For any ground proof T' of $q(\overline{X})\theta$ we can show the existence of a ground proof T of $p(\overline{X})\theta$ in a similar fashion. □

Note that given a definite logic program P, and two predicates p, p' in P, the corresponding syntactic equivalence check $p \overset{P}{\sim} p'$ has worst case time complexity $O(Pred(P)^2 * NumPredClause(P)^2 * MaxClause(P))$ where

- $Pred(P)$ = Number of predicate symbols in program P
- $NumPredClauses(P)$ = Maximum number of clauses of any predicate in P
- $MaxClause(P)$ = Maximum size of any clause in P

Given a logic program P, there are at most $Pred(P)^2$ distinct syntactic equivalence checks. Each of these checks can be performed in polynomial time. Since the largest syntactic equivalence relation can be computed as a greatest fixed point, we can define a dual relation $\not\sim$ which can be computed as a least fixed point. Given two predicates p, p' in program P we then check $p \overset{P}{\sim} p'$ by checking $\neg(p \overset{P}{\not\sim} p')$. In checking $p \overset{P}{\not\sim} p'$ we need to show the non-existence of a suitable mapping as required by Definition 5.3 as follows: we find a clause of either predicate which cannot be mapped to any clause of the other predicate under condition 2 of Definition 5.3. Given two clauses C and D of p, p' checking whether one is an instance of the other (as mentioned in condition 2

of Definition 5.3) takes $O(MaxClause(P))$ time. Checking $p \overset{P}{\not\sim} p'$ involves $O(\mid \mathcal{C}(p) \mid * \mid \mathcal{C}(p') \mid)$ instantiation checks *i.e.* $O(NumPredClauses(P)^2)$ instantiation checks, and leads to recursive invocations of $\overset{P}{\not\sim}$ checks. Since at most $O(Pred(P)^2)$ such $\overset{P}{\not\sim}$ checks may be recursively invoked, the total time complexity of checking $p \overset{P}{\not\sim} p'$ (and hence $p \overset{P}{\sim} p'$) is $O(Pred(P)^2 * NumPredClause(P)^2 * MaxClause(P))$.

As a side remark, note that the semantics of definite logic programs is given by its least Herbrand model which can be computed as a least fixed point by memoized logic program evaluation [CW96, TS86b]. This yields a declarative encoding for a logic programming based implementation of the syntactic equivalence check.

Tx rule: The transformation rule **Tx** corresponds to applying a program transformation which does not add any new equivalence proof obligations. Typically an application of this step is either unfolding (Transformation 3.3) or folding (Transformation 3.4), or other standard transformations like generalization and equality introduction, deletion of subsumed clauses and deletion of failed clauses [PP98]. A single application of all these transformations is fully automatable.

Gen rule: The **Gen** rule attempts to prove the e-atom $\Gamma \vdash p \equiv p'$ by proving the e-atoms $\Gamma, P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$ where $\Gamma = P_0, \ldots, P_i$ is a program transformation sequence. It generates a new lemma $P_0 \vdash q \equiv q'$ whose proof is used to ensure that $M(P_i) = M(P_{i+1})$. An application of **Gen** corresponds to an application of the Goal Replacement transformation (Transformation 3.6). Recall that the Goal Replacement transformation obtains program P_{i+1} from program P_i by replacing an atom in a clause $C \in P_i$ with another semantically equivalent atom. Here, we replace an occurrence of q with q' in a clause of p or p' as shown below.

...	...
$C : p(\overline{t})$:– $\mathcal{G}$, $q(\overline{s})$, $\mathcal{G}'$. (γ,γ')	$C' : p(\overline{t})$:– $\mathcal{G}$, $q'(\overline{s})$, $\mathcal{G}'$. ($\gamma + \Delta(q, q')$, $\gamma' + \Delta'(q, q')$)
...	...
Program P_i	Program P_{i+1}

This requires us to show $P_0 \models q \equiv q'$ and therefore we obtain a new proof obligation $P_0 \vdash q \equiv q'$. We prove $P_0 \vdash q \equiv q'$ by constructing a different transformation sequence $P_0, P'_1, \ldots, P'_k$ s.t. $q \overset{P'_k}{\sim} q'$ *i.e.* q, q' are syntactically equivalent in P'_k. Note that since we are replacing q with q' in program P_i, the goal replacement rule requires

$P_i \models q \equiv q'$. However for any e-atom $\Gamma \vdash \ldots$ appearing in a successful tableau, $M(P_0) = \ldots = M(P_i)$ where $\Gamma = P_0, \ldots, P_i$ (refer Lemma 5.2). Thus, $P_0 \models q \equiv q'$ implies $P_i \models q \equiv q'$.

In addition, we must compute the measures of the clause C' obtained by Goal Replacement. For this purpose, the Goal Replacement transformation (refer Transformation 3.6) requires us to compute an estimate of $\alpha(q(\overline{X})\theta - \alpha(q'(\overline{X})\theta)$ for any ground substitution θ. Thus we compute $\Delta(q, q')$ and $\Delta'(q, q')$, the lower and upper bounds of the difference in atom measures, that is

$$\forall \text{ ground substitutions } \theta \quad \Delta(q, q') \leq \alpha(q(\overline{X})\theta) - \alpha(q'(\overline{X})\theta) \leq \Delta'(q, q')$$

$\Delta(q, q')$, $\Delta'(q, q')$ can be computed from the measures of the clauses of q, q' in P'_k.

Recall that an application of the **Gen** rule proves $\Gamma \vdash p \equiv p'$ by proving $\Gamma, P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$. The proof of $P_0 \vdash q \equiv q'$ returns the values $\Delta(q, q')$ and $\Delta'(q, q')$ which are then used in the proof of $\Gamma, P_{i+1} \vdash p \equiv p'$. Therefore, even though our proof system in Table 2 does not impose any order in which the obligations $\Gamma, P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$ are proved, in an automated application of the **Gen** rule we will always prove $P_0 \vdash q \equiv q'$ before $\Gamma, P_{i+1} \vdash p \equiv p'$.

The computation of Δ and Δ' is done by using the following theorem relating the values of Δ, Δ' for syntactically equivalent predicates in a measure consistent program (refer Definition 3.7). Measure consistency of the program is *not* a restriction since our transformations are guaranteed to preserve measure consistency (as shown in Theorems 3.3 and 3.6).

Theorem 5.5 (Values of Δ and Δ') *Let p and q be two distinct predicates in a measure consistent program P (Definition 3.7) such that $p \overset{P}{\sim} q$. Let the clauses of p in P be $C_1, \ldots, C_m$ with clause measures $(\gamma_1, \gamma'_1), \ldots, (\gamma_m, \gamma'_m)$. Let the clauses of q in P be $D_1, \ldots, D_n$ with clause measures $(\mu_1, \mu'_1), \ldots, (\mu_n, \mu'_n)$. Moreover, let each clause C_i be $p(\overline{t}) :- p_{i,1}(\overline{t_{i,1}}), \ldots, p_{i,k_i}(\overline{t_{i,k_1}})$, and D_j be $q(\overline{s}) :- q_{j,1}(\overline{s_{j,1}}), \ldots, q_{j,l_j}(\overline{s_{j,l_j}})$. Finally, let f and g be the mappings (from Definition 5.3) used to show $p \overset{P}{\sim} q$. Then,*

$$\begin{aligned}
\Delta(p,p) &= \Delta(q,q) = \Delta'(p,p) = \Delta'(q,q) = 0 \\
\Delta(p,q) &\leq \min_{1 \leq i \leq m} [(\gamma_i - \mu'_{f(i)}) + \sum_{1 \leq l \leq k_i} \Delta(p_{i,l}, q_{f(i),l})] \\
\Delta'(p,q) &\geq \max_{1 \leq j \leq n} [(\gamma'_{g(j)} - \mu_j) + \sum_{1 \leq l \leq l_j} \Delta'(p_{g(j),l}, q_{j,l})]
\end{aligned}$$

Proof: Consider any ground substitution θ s.t. $p(\overline{X})\theta \in M(P)$. Then, since $p \overset{P}{\sim} q$ therefore $q(\overline{X})\theta \in M(P)$ and $\alpha(p(\overline{X})\theta)$, $\alpha(q(\overline{X})\theta)$ are defined. Since P is measure consistent, therefore $p(\overline{X})\theta$ has a strongly measure consistent proof in P. Let C_i be the clause used at the root of this proof. Thus:

$$\alpha(p(\overline{X})\theta) \geq \gamma_i + \sum_{1 \leq l \leq k_i} \alpha(p_{i,l}(\overline{Y_l})\sigma_{i,l}) \tag{7}$$

where $\sigma_{i,l}$ is the corresponding ground substitution of $p_{i,l}(\overline{Y_l})$. Then, by the definition of syntactic equivalence (refer Lemma 5.4), $q(\overline{X})\theta$ has a ground proof in P with the clause $D_{f(i)}$ at the root where $f : \{1, \ldots, m\} \to \{1, \ldots, n\}$ is the mapping used to show $p \overset{P}{\sim} q$ (refer Definition 5.3). This proof is weakly measure consistent (by the measure consistency of P) and we have:

$$\alpha(q(\overline{X})\theta) \leq \mu_{f(i)} + \sum_{1 \leq l \leq k_i} \alpha(q_{f(i),l}(\overline{Y_l})\sigma_{i,l}) \tag{8}$$

By subtracting inequation 8 from inequation 7 we obtain:

$$\alpha(p(\overline{X})\theta) - \alpha(q(\overline{X})\theta) \geq \gamma_i - \mu_{f(i)} + \sum_{1 \leq l \leq k_i} (\alpha(p_{i,l}(\overline{Y_l})\sigma_{i,l}) - \alpha(q_{f(i),l}(\overline{Y_l})\sigma_{i,l})) \tag{9}$$

Note that for any two syntactically equivalent predicates Δ is a lower bound on the difference in atom measures *i.e.* $\Delta(p_{i,l}, q_{f(i),l}) \leq \alpha(p_{i,l}(\overline{Y_l})\sigma_{i,l}) - \alpha(q_{f(i),l}(\overline{Y_l})\sigma_{i,l})$. Therefore,

$$\alpha(p(\overline{X})\theta) - \alpha(q(\overline{X})\theta) \geq \gamma_i - \mu_{f(i)} + \sum_{1 \leq l \leq k_i} \Delta(p_{i,l}, q_{f(i),l}) \tag{10}$$

The above inequations were obtained by assuming clause C_i is used at the root of the strongly measure consistent proof of $p(\overline{X})\theta$. Since C_i can be any clause of p therefore we have for any ground substitution θ

$$\alpha(p(\overline{X})\theta) - \alpha(q(\overline{X})\theta) \geq min_{1 \leq i \leq m}[(\gamma_i - \mu_{f(i)}) + \sum_{1 \leq l \leq k_i} \Delta(p_{i,l}, q_{f(i),l})] \tag{11}$$

Since $\Delta(p, q) \leq \alpha(p(\overline{X}\theta)) - \alpha(q(\overline{X}\theta))$ for any ground substitution θ, we can set:

$$\Delta(p, q) \leq \min_{1 \leq i \leq m}[(\gamma_i - \mu_{f(i)}) + \sum_{1 \leq l \leq k_i} \Delta(p_{i,l}, q_{f(i),l})] \tag{12}$$

By following similar reasoning, we can obtain the corresponding inequality for $\Delta'(p,q)$. □.

Theorem 5.5 immediately suggests a method to *compute* Δ and Δ'. Observe that the constraints on Δ and Δ' for different pairs of equivalent predicates form a set of linear inequalities. We can therefore use integer linear programming to maximize Δ values and minimize Δ' values to arrive at the tightest bounds on atom measures. If the system of inequalities for Δ (or Δ') is unsolvable, we can set the corresponding Δ (Δ') to $-\infty$ ($+\infty$).

For example, consider again the program in Figure 9. We have: $\Delta(p,q) \leq (\gamma_1 - \mu'_1) + \Delta(r,s)$; $\Delta(p,q) \leq (\gamma_2 - \mu'_2) + \Delta(p,q)$; and $\Delta(r,s) \leq \gamma_3 - \mu'_3$. Note that when $(\gamma_2 - \mu'_2) < 0$ the inequalities on $\Delta(p,q)$ are unsolvable. Otherwise, the non-recursive inequation gives the optimal value. Thus, applying minimization, we get $\Delta(p,q) = (\gamma_1 - \mu'_1) + (\gamma_3 - \mu'_3)$ if $(\gamma_2 - \mu'_2) \geq 0$, and $-\infty$ otherwise.

5.3 An Algorithmic Framework for Strategies

We describe an algorithmic framework for creating strategies to automate the construction of the equivalence tableau of an e-atom. The objective is to: (a) find equivalence proofs that arise in verification with little or no user intervention, and (b) apply deduction rules lazily, i.e. for finite state systems a proof using the strategy is equivalent to algorithmic verification. In the next chapter, we will discuss how this algorithmic framework can be instantiated to derive concrete strategies for guiding the transformations.

Our framework specifies the order in which the different program transformations (corresponding to each tableau rule) will be applied. If multiple transformations of the same kind (e.g., two folding steps) are possible at any point in the proof, the framework itself does not specify which transformations to apply. That is done by a separate selection function (analogous to literal selection in SLD resolution).

The tableau rules and associated transformations are applied in the following order. As would be expected, the axiom elimination rule (**Ax**) is used whenever it is applicable. When the choice is between the **Tx** and **Gen** rules, we choose the former since the default transformation employed by **Tx** is unfolding, *i.e.* resolution. This will ensure that our strategies will perform on-the-fly model checking, a' la XMC [RRR+97] for finite-state systems. For infinite-state systems, however, uncontrolled unfolding may diverge. To create finite unfolding sequences we impose a finiteness

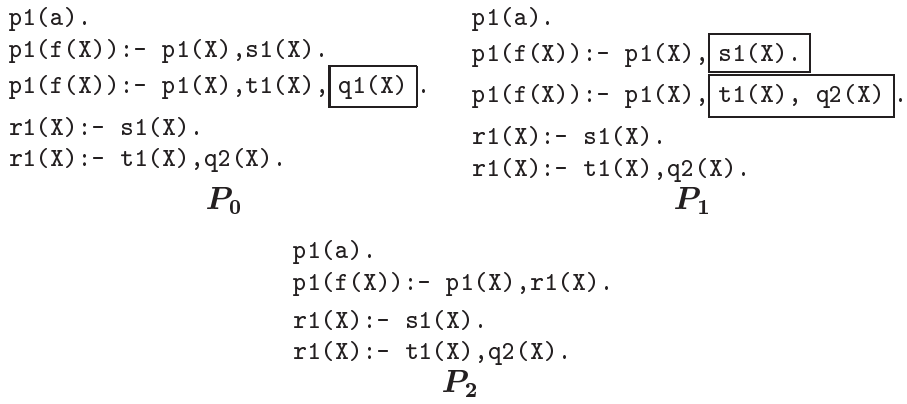


Figure 10: Goal replacements to facilitate other transformations.

condition *FIN* on transformation sequences. We do not give an exact implementation of *FIN* but only a sufficient condition s.t. the resultant unfolding sequences terminate. In the next section, we will discuss the issue of controlling unfolding in more detail.

Definition 5.4 (Finiteness condition) *Given an a-priori fixed constant $k \in \mathbb{N}$, an unfolding program transformation sequence $\Gamma = P_0, \ldots, P_i, \ldots$ satisfies the finiteness condition $FIN(\Gamma, k)$ if for the clause C and atom A selected for unfolding at every P_i: (1) A is distinct modulo variable renaming from any atom B which was selected in unfolding some clause $D \in P_j (j < i)$ where C is obtained by repeated unfolding of D (2) the term depth of each argument of A is $\leq k$.*

Typically, we will assume a suitable choice of k and write the finiteness condition simply as $FIN(\Gamma)$. Condition 1 prohibits infinite unfolding sequences of the form: unfolding `p(X)` using the clause `p(X) :- p(X)` *i.e.* unfolding sequences where the same atom is infinitely unfolded. Condition 2 prohibits infinite unfolding sequences of the form: unfolding `p(X)` using the clause `p(X) :- p(s(X))` *i.e.* where a different atom is unfolded every time, but there are infinitely many atoms to unfold.

If *FIN* prohibits any further unfolding we either apply the folding transformation associated with **Tx** or use the **Gen** rule. Care must be taken, however, when **Gen** is chosen. Recall from the definition of **Gen** (refer Table 2) that $\Gamma, P_{i+1} \vdash p \equiv p'$ implies $\Gamma \vdash p \equiv p'$ only if we can prove a new equivalence $P_0 \vdash q \equiv q'$. In other words, $P_{i+1} \models p \equiv p'$ implies $P_i \models p \equiv p'$ only if $P_0 \models q \equiv q'$. Since **Gen** itself does not specify the goals q and q' in the new equivalence, its application is highly

```
algorithm Prove(p, p': predicates, Γ:prog. seq.)
begin
  let Γ = P_0, ..., P_i
  mark proof_attempt(p, p')
  (* Ax rule *)
  if (p ~^{P_i} p' ∨ proved(p, p')) then
    return (Δ(p, p'), Δ'(p, p')) (* Theorem 5.5, page 78 *)
  else if proof_attempt(p, p') is not marked
    nondeterministic choice
      (* Tx rule *)
        case FIN(⟨Γ, unfold(P_i)⟩): (* Unfolding *)
          return Prove(p, p', ⟨Γ, unfold(P_i)⟩)
        case Folding is possible in P_i:
          return Prove(p, p', ⟨Γ, fold(P_i)⟩)
      (* Gen rule *)
        case Conditional folding is possible in P_i:
          let (G, G') = new_goal_equiv_for_fold(P_i)
          return replace_and_prove(p, p', G, G', Γ)
        case Conditional equivalence is possible in P_i:
          let (G, G') = new_goal_equiv_for_equiv(p, p', P_i)
          return replace_and_prove(p, p', G, G', Γ)
    end choices
  mark proved(p, p')
  record (Δ(p, p'), Δ'(p, p'))
  unmark proof_attempt(p, p')
end
```

Figure 11: Algorithmic framework for equivalence tableau construction.

nondeterministic. We limit the nondeterminism by using **Gen** only to enable **Ax** or **Tx** rules. For instance, consider the transformation sequence in Figure 10. Applying goal replacement in P_0 under the assumption that that $P_0 \models \texttt{q1} \equiv \texttt{q2}$ enables the subsequent folding which transforms P_1 into P_2.

Hence, when no further unfoldings are possible, we apply any possible folding. If no foldings are enabled, we check if there are new goal equivalences that will enable a folding step. We call this a *conditional folding* step. For instance, in program P_0 of Figure 10, equivalence of `q1(X)` and `q2(X)` enables folding. Note that the test for syntactic equivalence is only done on predicates, whereas a goal is a conjunction of atoms. However, we can reduce a goal equivalence check to a predicate equivalence check by introducing new predicate names for the goals. A keen point needs to be noted here. When we introduce new predicate names to a program, clearly the least Herbrand model can never be preserved. As is common in program transformation literature [TS84, GK94], we rectify this apparent anomaly by assuming that all new predicate names introduced are present in the initial program P_0 of a program transformation sequence.

Finally, we look for new goal equivalences, which, if valid, can lead to syntactic equivalence. This is called a *conditional equivalence* step. For instance, suppose in program P_2 (in Figure 10), there are two additional predicates `p2` and `r2` and further assume that `p2` is defined using clauses

```
p2(a).
p2(f(Y)):- p2(Y), r2(Y).
```

Now if `r2` and `r1` are semantically equivalent, we can perform this goal replacement to obtain the program P_3 where `p1` and `p2` are defined as follows. Thus, in P_3 we can conclude that $\texttt{p1} \overset{P_3}{\sim} \texttt{p2}$.

```
p1(a).                          p2(a).
p1(f(X)):- p1(X), r1(X).        p2(f(Y)):- p2(Y), r1(Y).
```

The above intuitions are formalized in Algorithm *Prove* (see Figure 11). Given a program transformation sequence Γ, and a pair of predicates p, p', algorithm *Prove* attempts to prove that $\Gamma \vdash p \equiv p'$. If $\Gamma \vdash p \equiv p'$ is a subproof that needs to be dispensed because of a goal replacement step (*i.e.* an application of **Gen**) the corresponding bounds on atom measures are needed. If *Prove* succeeds in finding a proof, it returns the bounds on the corresponding atom measures.

Algorithm *Prove* searches nondeterministically for a proof: if multiple cases of the nondeterministic choice are enabled, then they will be tried in the order specified in *Prove*. If none of the cases apply, then evaluation fails, and backtracks to the most recent unexplored case. There may also be nondeterminism within a case; for instance, many fold transformations may be applicable at the same time. We again select nondeterministically from this set of applicable transformations. By providing selection functions to pick from these applicable transformations, one can implement a variety of concrete strategies (discussed in the next chapter). Note that Algorithm *Prove* uses two different markings in the process of constructing a proof for $\Gamma \vdash p \equiv p'$. The marking *proved* remembers predicate equivalences which have been already proved. This marking allows us to cache subproofs in a proof. The marking *proof_attempt* keeps track of predicate pairs whose equivalence has not yet been established, but is being attempted by Algorithm *Prove* via transformations. This marking is essential for ensuring termination of the algorithm. The proof of $P_0 \vdash p \equiv p'$ may (via a conditional equivalence step) generate the (sub)-equivalence $P_0 \vdash p \equiv p'$. Algorithm *Prove* deems this proof path as failed and explores the other proof paths.

Algorithm *Prove* uses the following functions. Functions $unfold(P)$ and $fold(P)$ apply unfolding and folding transformations respectively to program P and return a new program. Whenever conditional folding is possible, the function $new\_goal\_equiv\_for\_fold(P)$ finds a pair of goals whose replacement is necessary to do a fold transformation. Similarly, when conditional equivalence is possible, $new\_goal\_equiv\_for\_equiv(p, p', P)$ finds a pair of goals $\mathcal{G}, \mathcal{G}'$ s.t. syntactic equivalence of p and p' can be established after replacing $\mathcal{G}$ with $\mathcal{G}'$ in P.

Finally, function *replace_and_prove* constructs nested proofs for sub-equivalences created by applying the **Gen** rule. Thus, $replace\_and\_prove(p, p', \mathcal{G}, \mathcal{G}', \Gamma)$ performs the following sequence of steps (where $\Gamma = P_0, \ldots, P_i$):

1. first introduces new predicate definitions q and q' for goals $\mathcal{G}$ and $\mathcal{G}'$ respectively (if such definitions do not already exist),

2. proves the equivalence $P_0 \vdash q \equiv q'$ by invoking *Prove*,

3. replaces goal $\mathcal{G}$ by goal $\mathcal{G}'$ in clauses of p or p' in program P_i to obtain program P_{i+1}, and

4. finally invokes *Prove* to dispense the obligation $\Gamma, P_{i+1} \vdash p \equiv p'$. This completes the proof of $\Gamma \vdash p \equiv p'$.

Note that when we replace goal $\mathcal{G}$ with goal $\mathcal{G}'$ in step (3), the goal replacement operation returns the bounds in the difference in atom measures of $\mathcal{G}$, $\mathcal{G}'$. In fact, if the predicates defined by $\mathcal{G}, \mathcal{G}'$ in step (1) of *replace_and_prove* are q, q' respectively, then the goal replacement operation returns $\Delta(q, q')$ and $\Delta'(q, q')$ as defined in Theorem 5.5. We then use these bounds to compute the clause measures of the clause introduced by replacement, and complete the proof of the obligation $\Gamma \vdash p \equiv p'$.

Termination of *Prove* It can be easily verified that only finite unfolding sequences satisfy *FIN*. This is because in any unfolding sequence of clauses $C_1, \ldots, C_n$ where C_{i+1} is obtained from C_i via unfolding, condition 1 ensures that the selected atom each C_i is distinct, and condition 2 ensures that there are only finitely many atoms which can ever be selected for unfolding.

Therefore, the length of each predicate equivalence proof itself is finite (assuming folding always reduces program size which can be ensured). However, a proof for $p \equiv p'$ may require $q \equiv q'$ as a lemma, whose proof in turn may require $r \equiv r'$ as a lemma, and so on. Since the number of distinct equivalences are quadratic in the number of predicate symbols in the program, the number of subproofs is finite if the number of new predicates names introduced is finite. Thus, we have :

Lemma 5.6 *Algorithm Prove (refer Figure 11) terminates provided the number of definitions introduced (*i.e. *new predicate symbols added) is finite.*

Proving Predicate Implications Note that the proof system given in Table 2, the algorithmic framework *Prove* and the strategies to guide the transformations in *Prove* are aimed at proving equivalence of program predicates. Our proof technique can be readily extended to prove predicate implications *i.e.* proof obligations of the form

$$\forall \text{ ground substitutions } \theta \quad p(\overline{X})\theta \in M(P_0) \Rightarrow p'(\overline{X})\theta \in M(P_0)$$

This extension involves (1) relaxing the definition of syntactic equivalence (Definition 5.3) to test for implications only, and (2) generating conditions of the form $q \Rightarrow q'$ by applying conditional folding and conditional equivalence.

Chapter 6

A Prover for Verifying Parameterized Protocols

In the last chapter, we presented a tableau based proof system for proving equivalence of predicates in a logic program. Furthermore, we presented an algorithmic framework *Prove* for guiding the application of the rules in the proof system. However, this algorithmic framework *Prove* is nondeterministic since at each step several transformations may be applicable. Hence it is necessary to develop appropriate selection functions to distill concrete strategies from the algorithmic framework.

In this chapter, we discuss some of the strategies which have been effectively used to verify parameterized protocols of different network topologies. Our objective here is to develop an *automated* prover for verifying parameterized systems. However, to add to the flexibility of the prover, we make it *programmable*. Thus, based on the system and the property to be verified, the user provides some guidance (typically by setting some flags) to the prover at the *beginning* of the proof. The proof attempt then proceeds completely automatically without *any* user interaction. A prototype implementation of this automated and programmable prover has been completed using the XSB tabled logic programming system [XSB00]. XSB closely approximates the ISO standard of Prolog, and has been installed in more than thousand sites over different platforms.

In Section 6.1, we present how the parameterized system and the temporal property are specified in the prover. Section 6.2 discusses the major design decisions taken to instantiate the algorithmic framework *Prove*, thereby obtaining concrete strategies to guide the transformation rules. Section 6.3 applies the strategies to construct example liveness and safety proofs of parameterized networks.

6.1 System and Property Specification

First, we present how the parameterized system to be verified and the temporal property are specified in our prover. Since our prover employs program transformations of definite logic programs, it cannot verify temporal properties containing both greatest and least fixed point operators. This is because the logic program clauses encoding a temporal property with both greatest and least fixed point operators must contain negated atoms. In the following, we outline the use of our prover in verifying safety and liveness properties, temporal properties with a single fixed point operator.

System Specification To use our prover, first the initial states and the transition relation of the parameterized system are specified as two logic program predicates `gen` and `trans`. The global states of the parameterized system are represented by infinite sized terms, and `gen`, `trans` are predicates over these terms. The recursive structure of `gen` and `trans` depends on the topology of the parameterized network being verified. For example, consider a network of similar processes where any process may perform an autonomous action or communicate with any other process. This situation may occur in a bus-based protocol, where each unit attached to the bus may communicate with any other unit connected to the bus. We can model the global state of this parameterized network as an unbounded list of the local states of the individual processes. The transition relation `trans` can then be defined over these global states as follows:

```
trans([H|T], [H1|T1]) :-  ltrans(H, in(Act), H1),
                          trans_rest(T, out(Act), T1).
trans([H|T], [H1|T1]) :-  ltrans(H, out(Act), H1),
                          trans_rest(T, in(Act), T1).
trans([H|T], [H1|T]) :-   ltrans(H, self(Act), H1).
trans([H|T], [H|T1]) :-   trans(T, T1).

trans_rest([S|T], A, [S1|T]) :-  ltrans(S, A, S1).
trans_rest([H|T], A, [H|T1]) :-  trans_rest(T, A, T1).
```

Thus, each process can perform an autonomous action (denoted in the above as `self(A)`) or an input/output action (denoted as `in(A)` and `out(A)` respectively) where matching input and output actions synchronize. The predicate `ltrans` encodes the local transition relation of each process. For the global transition relation `trans`, the last rule recursively searches the global state representation until one of the first

three rules can be applied. The third rule allows any process to make an autonomous action. The first and second rules correspond to the scenario where *any* two processes communicate with each other (possibly via a bus, which is not modeled). In particular, if the processes are $\mathcal{P}_1, \ldots, \mathcal{P}_n$ for some $n \geq 1$, then the possible transitions are:

1. a process $\mathcal{P}_i$ $(1 \leq i \leq n)$ makes an autonomous action. This is simulated by $i-1$ applications of the fourth rule followed by an application of the third rule.

2. a process $\mathcal{P}_i$ $(1 \leq i < n)$ performs an `in(A)` action and another process $\mathcal{P}_j$ $(i < j \leq n)$ performs an `out(A)` action. This is simulated by $i-1$ applications of the fourth rule followed by an application of the first rule. In the body of the first rule, `ltrans` simulates the `in(A)` transition of $\mathcal{P}_i$, and `trans_rest` simulates the `out(A)` transition of a process in $\mathcal{P}_{i+1}, \ldots, \mathcal{P}_n$.

3. a process $\mathcal{P}_i$ $(1 \leq i < n)$ performs an `in(A)` action and another process $\mathcal{P}_j$ $(i < j \leq n)$ performs an `out(A)` action. This is simulated by $i-1$ applications of the fourth rule followed by an application of the second rule. Again, in the body of the second rule, `ltrans` simulates the `out(A)` transition of $\mathcal{P}_i$, and `trans_rest` simulates the `in(A)` transition of a process in $\mathcal{P}_{i+1}, \ldots, \mathcal{P}_n$.

Property Specification and Proof Obligations A liveness property, denoted in CTL as **EF** `good` is encoded as a program predicate `live` defined by the clauses:

```
live(S) :- good(S).
live(S) :- trans(S, T), live(T).
```

The liveness property is proved by showing the predicate implication `gen` $\Rightarrow$ `thm` where `thm` is defined as `thm(X) :- gen(X),live(X).`

A safety property, denoted in CTL as **AG** $\neg$`bad` can be verified in two different ways. We can define predicates `unsafe` and `thm` as:

```
unsafe(S) :- bad(S).
unsafe(S) :- trans(S, T), unsafe(T).

thm(S) :- gen(S), unsafe(S).
```

and prove the implication `thm` $\Rightarrow$ `false`. This constitutes a proof of the safety property since it shows that none of the initial states are `unsafe` *i.e.* a `bad` state is not reachable from any initial state satisfying `gen`.

Alternatively, one can prove a safety property **AG** $\neg$`bad` by proving transition invariance. We prove that (1) a `bad` state is reachable only from a `bad` state, and (2)

none of the initial states satisfying `gen` are `bad`. This constitutes a proof of the safety property, and is shown by establishing the predicate implications (1) `bad_dest` ⇒ `bad_src`, and (2) `bad_start` ⇒ `false` where the predicates `bad_dest`, `bad_src` and `bad_start` are defined as:

```
bad_dest(S, T) :- trans(S, T), bad(T).
bad_src(S, T) :- trans(S, T), bad(S).
bad_start(S) :- gen(X), bad(X).
```

Note that a proof of the safety property by showing `thm` ⇒ false typically involves two nested inductions. The outer one (uncovered by unfolding `gen`) inducts on the process structure of the parameterized network. The inner one (uncovered by unfolding `unsafe`) is a computational induction which shows that a `bad` state is never reachable. The alternative formulation of proving transition invariance simplifies the proof of the safety property. It breaks the proof into two different obligations `bad_dest` ⇒ `bad_src` and `bad_start` ⇒ `false` which are proved by induction on the structure of the global states of the parameterized network. Since none of the initial states are `bad` and a `bad` state is only reachable from a `bad` state, this means that a `bad` state is never reachable from an initial state.

6.2 Constructing Concrete Strategies

Once the parameterized system and the temporal property are specified, the prover attempts to produce a completely automated proof of the corresponding predicate equivalences / implications. We now present concrete strategies (which are employed by the prover) to guide the application of the transformation rules in a proof. As mentioned before, these strategies serve to instantiate our abstract algorithmic framework *Prove* presented in Figure 11 (refer Chapter 5.3).

6.2.1 Pruning the Transformation Search Space

The evaluation of Algorithm *Prove* presented in Figure 11 permits backtracking as in logic program evaluation. When multiple transformations are applicable, the algorithm explores a proof path (based on the selection functions). If this path fails, the algorithm backtracks to the last choice point and explores the next proof path. This can lead to substantial overheads, since we need to store each program in the program transformation sequence for every proof path. Moreover, for any realistic

verification problem, the number of proof paths easily becomes prohibitively large, *e.g.* at each step numerous unfold transformations are enabled each of which leads to a different proof path. For this reason, we enforce the following *design decisions* to prune the search space (of applicable transformations) explored by Algorithm *Prove*.

1. To prove a proof obligation $\Gamma \vdash p \equiv p'$, at every step, we only transform $\mathcal{C}(p)$ and $\mathcal{C}(p')$, the clauses of p, p'.

2. If p (p') is a predicate appearing in the system or property specification, then p (p') is not transformed.

3. No backtracking is allowed in the execution of *Prove*(p, p', Γ) for any two predicates p, p' and program transformation sequence Γ.

4. Delete failed and/or subsumed clauses of p, p' before every deductive step in the proof of $\Gamma \vdash p \equiv p'$.

The motivation and consequences for choosing these design decisions is now given. To prove the equivalence $\Gamma \vdash p \equiv p'$, we need to transform only the clauses *relevant* to p and p'. The set of clauses relevant to a predicate p, denoted by $\mathcal{R}(p)$, is the smallest set such that $C \in \mathcal{R}(p)$ if the head of C unifies either with $p(\overline{X})$, or with some atom $q(\overline{X})$ such that predicate q is in the body of a clause $D \in \mathcal{R}(p)$. It can be shown any proof of $\Gamma \vdash p \equiv p'$ transforms only clauses in $\mathcal{R}(p)$ and $\mathcal{R}(p')$. Note that we choose to transform only clauses in $\mathcal{C}(p)$ and $\mathcal{C}(p')$ at every transformation step. Clearly, $\mathcal{C}(p) \subseteq \mathcal{R}(p)$ for any predicate p. However, note that the transformation of predicates defined using $\mathcal{R}(p) - \mathcal{C}(p)$ can be simulated by spawning conditional equivalences on these predicates. Thus, we still preserve all equivalences that can be established by any strategy implemented using *Prove*. Moreover, this design decision makes our proof search more *goal directed*. It ensures that we transform clauses in $\mathcal{R}(p) - \mathcal{C}(p)$ and $\mathcal{R}(p') - \mathcal{C}(p')$ only when it is required for proving the equivalence of p and p'.

Note that the purpose of applying our unfold/fold transformations to the clauses of a predicate is to expose its underlying recursive structure. For this purpose we transform predicates which are defined using the system and property description. However, the system or property description itself is not transformed in our prover. This is because the system description is assumed to succintly encode the recursive structure of the parameterized family (the network topology and the pattern of interactions in a global transition). Similarly, the property description is assumed to

encode the recursive structure underlying the temporal property being verified. For example suppose the predicate `gen` denotes a system specification (it generates the global states of the parameterized system) and the predicate `prop` denotes a property specification. Then, if we have a new predicate `new` defined as:

```
new(X)  :- gen(X), prop(X).
```

Then, we transform `new` by using the definitions of `gen` and `prop`. However the predicates `gen` and `prop` are not transformed.

The third design decision is prompted by the prohibitive blow-up in the number of proof paths, due to numerous transformations being applicable at every step of any realistic proof. Thus we enforce the following : once a transformation is performed, it is *committed* and we proceed along that proof path trying to perform other applicable transformations. Since we cannot backtrack to another proof path if our chosen proof path fails, we need to engineer sophisticated selection strategies for choosing a transformation from several applicable transformations. These strategies are discussed in the later sections.

The decision to cleanup failed and subsumed clauses is obviously prompted by the need to prevent proliferation of conditional equivalence steps in the proof of $\Gamma \vdash p \equiv p'$. Since most steps in a predicate equivalence proof are algorithmic (unfold steps), the decision to cleanup only before deductive steps prevents unreasonable time overheads. The cleanup of subsumed and failed clauses is accomplished via the application of the following transformations to the clauses of p, p'.

Transformation 6.1 (Deletion of Subsumed Clauses) Let $C \in P_i$ be a clause $A :- B_1, \ldots, B_k$ and $D \in P_i$ be another clause $A\theta :- B_1\theta, \ldots, B_k\theta, B'_1, \ldots, B'_m$ *i.e.* clause D is subsumed by clause C.

Then, $P_{i+1} := P_i - \{D\}$. Set the integer counters of C in P_{i+1} as follows.[1] $\gamma_{lo}^{i+1}(C) = min(\gamma_{lo}^{i}(C), \gamma_{lo}^{i}(D))$ and $\gamma_{hi}^{i+1}(C) = max(\gamma_{hi}^{i}(C), \gamma_{hi}^{i}(D))$. □

Transformation 6.2 (Deletion of failed clauses) Let q be a n-ary predicate symbol in P_i and let $Cl_i(q)$ be the clauses of q in P_i. Suppose for all $C \in Cl_i(q)$, the body of C contains either the symbol `false` or a recursive occurrence of q.

Then, $P_{i+1} := P_i - Cl_i(q) - Occ_i(q)$ where $Occ_i(q)$ is the set of clauses in P_i in which predicate q appears in the body. □

[1] Recall that $\gamma_{lo}^{i}(C)$ and $\gamma_{hi}^{i}(C)$ denote the two integer counters of clause C in program P_i.

Note that deleting subsumed clauses from the clause set of p has time complexity $O(\mid \mathcal{C}(p) \mid^2)$, since the clauses need to be compared with each other for subsumption check. However, deleting failed clauses from the clause set of p has time complexity $O(\mid \mathcal{R}(p) \mid)$. In the proofs of parameterized protocols that we have constructed, the number of clauses of a predicate p increases rapidly due to unfolding *e.g.* $\mid \mathcal{C}(p) \mid = 100$ is a modest estimate. In using the prover, the user has the flexibility of selecting the kind of clauses to be deleted, *e.g.* to avoid time overheads the user may disable subsumed clause deletion and allow only deletion of failed clauses. However, all such user guidance is programmed before a proof attempt to avoid any user interaction during the proof construction.

6.2.2 Controlling Algorithmic Steps

In a predicate equivalence proof of $P_0 \vdash p \equiv p'$, where P_0 is the initial program, most of the transformation steps are algorithmic *i.e.* unfolding steps. Moreover, even when we restrict ourselves to transform only clauses in $\mathcal{C}(p)$ and $\mathcal{C}(p')$, several unfolding steps are enabled in every step of the proof. The criteria considered for choosing unfolding steps are:

- *Termination* : Unfolding sequences must terminate.
- *Selection Order* : In a clause, body atoms are expanded left to right.
- *Convergence* : Unfolding steps should not disable deductive steps which lead to a proof.

Our unfolding strategy guarantees termination and left-to-right selection order. It also includes heuristics for fast convergence. We now discuss the motivations and the heuristics for following each of the above three criteria. For this purpose, we introduce the following notion of *unfolding tree*. This notion formalizes the repeated unfolding of a clause.

Definition 6.1 (Unfolding Tree) *Let P be a program and $C \in P$. An unfolding tree of C is a tree T s.t.*

1. *each node of T is labeled by a clause; the root is labeled by C.*
2. *let $\mathcal{N}$ be an internal node in T which is labeled by a clause C. Then the children of node $\mathcal{N}$ in T are labeled by the clauses obtained by unfolding A in clause C, where A is any body atom of C. The atom A is called the selected atom of $\mathcal{N}$.*

Termination: The finiteness condition *FIN* of unfolding transformation sequences (presented in Definition 5.4) can be used to ensure finiteness of unfolding trees.

Lemma 6.1 *For any clause C in program P, any unfolding tree of C is finite if every unfolding step satisfies FIN (refer Definition 5.4).*

The finiteness condition *FIN* ensures termination of any unfolding transformation sequence. As discussed before, condition (1) of *FIN* ensures that in any path of an unfolding tree no two selected atoms are same (modulo variable renaming). Condition (2) of *FIN* ensures that there are finitely many atoms which may appear as selected atom of a node of the unfolding tree. Thus, *FIN* ensures that every path in an unfolding tree is finite. Furthermore, every node of the unfolding tree is finitely branching, thereby establishing Lemma 6.1.

Now, let us examine the amount of book-keeping needed to ensure termination of a sequence of unfolds. Let $P_0, \ldots, P_n$ be a sequence of unfold steps. Then, for every clause $C \in P_n$ (the current program in the transformation sequence) we need to store the selected atoms of the clauses in $P_0, \ldots, P_{n-1}$ from which C was obtained via repeated unfolding. We store this information efficiently in $O(n)$ space, by augmenting the unfold transformation as in Transformation rule 6.3. Thus, for any clause $C \in P_n$ we need to remember the clauses in $P_0, \ldots, P_{n-1}$ from which it was obtained, as well as the selected atoms of the clauses. Note that this does not require us to store all the programs in the program transformation sequence $P_0, \ldots, P_{n-1}$. Rather we maintain a forest of trees where each node in a tree contains the clause identification number and the selected atom of a clause in P_i $(0 \leq i < n)$.

Transformation 6.3 (Unfolding with Marking) Let $C_1, \ldots, C_m$ be obtained by unfolding body atom A of clause $C \in P_i$ as in Transformation 3.3. Then, mark clause C as the *parent* of clauses $C_1, \ldots, C_m$ and mark A as the *selected atom* of C. □

Clearly, the number of *selected atom* marks is exactly equal to n, the number of unfold steps performed. Also, assuming each unfold step generates $O(1)$ clauses, there are $O(n)$ *parent* marks stored for the clauses in program P_n. Note that we only unfold occurrences of predicates appearing in the system/property description, and the number of clauses of these predicates is typically a small constant. Therefore, the additional book-keeping needed to ensure termination of an unfolding sequence is $O(n)$ where n is the length of the sequence.

Selection Order: Note that the logic program encoding of a verification problem for parameterized systems inherently has a "*producer-consumer*" nature. For example, consider the problem of verifying liveness in an infinite family. The verification proof obligation is reduced to showing equivalence of predicates `gen` and `thm` where:

```
thm(X) :- gen(X), live(X).
```

In this case, we intend to produce instantiation for variable `X` by unfolding the system description (in this case `gen(X)`) and then this instantiation is to be consumed by unfolding the property description (in this case `live(X)`). Similarly, the encoding of the predicates whose equivalence needs to be proved for establishing safety properties, also has a producer-consumer nature. In general, in the proof of a parameterized protocol if p is a predicate being transformed, then any clause of p contains system description predicates *followed* by property description predicates in the body. There is an intended left to right ordering for unfolding the body atoms of the predicates in these predicate equivalence proofs. Therefore, we always unfold the body atoms of a clause from *left to right*.

Convergence Ensuring termination of unfold sequences is *only one* of the considerations in guiding unfolding. In the proof of $P_0 \vdash p \equiv p'$, algorithm *Prove* applies deductive steps like folding and conditional folding subsequent to unfolding. Suppose by guiding unfolding solely based on termination, we produce a transformation sequence $P_0, \ldots, P_i, \ldots, P_n$. However, this might disable certain deductive steps *i.e.* a folding step applicable in P_i which leads to a proof of $P_0 \vdash p \equiv p'$ might be disabled in P_n. In general, to prevent disabling of deductive steps we need to check for applying deductive steps ahead of algorithmic steps. However, this would add theorem proving overheads to model checking. Our goal is to perform zero-overhead theorem proving, where deductive steps are never applied if model checking can complete the verification task. Therefore, we apply the deductive steps on demand (as shown in *Prove*) and identify "redundant" unfolding steps which are likely to diverge.

As mentioned earlier, the logic program encoding of a parameterized system verification problem possesses a "producer-consumer" nature. Therefore occurrences of predicates appearing in the property description are intended to consume variable instantiations rather than produce it. For this purpose, our prover prohibits the unfolding of any open atom[2] of a predicate appearing in the property description.

[2]Atoms of the form $p(X_1, \ldots, X_n)$ where p is a predicate and $X_1, \ldots, X_n$ are variables.

Note that atoms of predicates appearing in the system description when unfolded instantiate variables to terms representing global states of the parameterized family. For example in verification of liveness properties to prove `thm` $\Rightarrow$ `gen`, we first unfold `gen` appearing in the definition clause of `thm` given below.

```
thm(X) :- gen(X), live(X).
```

Suppose by repeatedly unfolding `gen` the variable `X` gets instantiated to a term $\overline{\texttt{t}}$. Now, by unfolding `live`($\overline{\texttt{t}}$) we intend to *test* whether the liveness property holds in global states represented by $\overline{\texttt{t}}$. Therefore, the unfolding of `live`($\overline{\texttt{t}}$) should consume the instantiation $\overline{\texttt{t}}$ rather than instantiating $\overline{\texttt{t}}$ further. However, since each unfolding step involves unification, it is possible to instantiate $\overline{\texttt{t}}$ by unfolding `live`($\overline{\texttt{t}}$). Therefore we prohibit the unfolding of any open atom of a predicate appearing in the property description. Since all arguments of such an atom are variables, any unfolding is likely to instantiate them. As system and property description predicates share variables, such an instantiation will enable further unfolding of the system description predicates (*e.g.* `gen`) without violating the finiteness condition *FIN*. This can disable deductive steps converging to a proof *e.g.* folding of conjunction of `gen` and `live` to `thm`.

An Unfolding Strategy The three criteria of termination, convergence and selection order can now be distilled into the following unfolding strategy. In unfolding a clause C, we select a body atom A of C s.t.

1. A does not violate the finiteness condition *FIN*.
2. A is not an open atom of a predicate appearing in the property description.
3. A is the leftmost body atom of C satisfying (1) and (2).

6.2.3 Controlling Deductive Steps

We now discuss control strategies for guiding the different deductive steps: folding, conditional folding and conditional equivalence. In the proof of $P_0 \vdash p \equiv p'$ suppose we have constructed a transformation sequence $P_0, P_1, \ldots, P_i$ by unfolding the clauses of p and p'. In the following we present strategies for choosing a step from the different applicable deductive steps in P_i (and the subsequent programs formed by repeatedly transforming P_i).

Folding Choosing from applicable fold transformations involves:
(i) selecting folder clauses $\{D_1, \ldots, D_m\} \subseteq P_j$ $(0 \leq j \leq i)$,
(ii) selecting folded clauses $\{C_1, \ldots, C_m\} \subseteq P_i$, and then
(iii) checking whether $\{C_1, \ldots, C_m\}$ is foldable using $\{D_1, \ldots, D_m\}$ as folder.
By suitably restricting the choice of folder and folded clauses we can avoid expensive foldability checks which fail. Moreover, the number of unfold steps in the proofs of predicate equivalence in our parameterized protocols are typically > 100. Thus, we do not store the intermediate programs $P_1, \ldots, P_{i-1}$; only the initial program P_0 and the current program P_i are remembered by the prover. The folder clauses are accordingly restricted to be drawn from P_0. Moreover, since we add new predicate definitions (via conditional folding and conditional equivalence) and all these predicate definitions are in P_0, therefore we choose the folder clauses by simply picking a predicate q from P_0. This choice is further restricted by preferentially picking predicates from P_0 which appear in some equivalence proof obligation. This is because our proof technique is geared towards exposing the underlying recursive structure of the predicates appearing in proof obligations. Thus the clauses of these predicates are likely candidates for folder. Once a predicate q is selected, the clause set of q in P_0 then constitutes our choice of folder.

Once the folder clause is selected, due the blow-up in the number of clauses generated by unfolding, the search for folded clauses is extremely expensive. Therefore, we consider only those clause sets which match with our choice of folder at the propositional level. Note that any clause-set which does not match with the folder at the propositional level does not constitute a valid choice of folded clauses. This can be formally stated as follows.

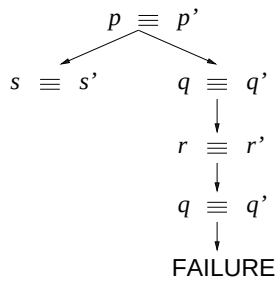
Definition 6.2 (Propositional Approximation of a Clause) *Let P be a definite logic program and $C \in P$ s.t. $C = (p(\ldots) :- p_1(\ldots), \ldots, p_k(\ldots))$. Then the propositional approximation of C denoted $prop(C)$ is the clause $p :- p_1, \ldots, p_k$.*

Now, suppose we are attempting to prove $P_0 \vdash p \equiv p'$ and we have chosen folder clauses $\{D_1, \ldots, D_m\} \subseteq P_0$. Then, we choose a set of clauses $\{C_1, \ldots, C_m\} \subseteq P_i$ as folded clauses s.t. (i) $\{C_1, \ldots, C_m\}$ are drawn from the clause set of p or p', and (ii) $\{prop(C_1), \ldots, prop(C_m)\}$ is foldable using $\{prop(D_1), \ldots, prop(D_m)\}$.

Conditional Folding As far as conditional folding is concerned, note that it is a more expensive deductive step. It involves goal replacement of a goal $\mathcal{G}$ with another

goal $\mathcal{G}'$, proving $\mathcal{G} \equiv \mathcal{G}'$, followed by folding. To restrict its applicability, we need to restrict the goals $\mathcal{G}$, $\mathcal{G}'$, *e.g.* restricting them to be conjunctive and restricting the size of conjunctions. In addition, we need to impose another important restriction on $\mathcal{G}, \mathcal{G}'$. Note that we prove $\mathcal{G} \equiv \mathcal{G}'$ by defining predicates $q(\ldots) :- \mathcal{G}$ and $q'(\ldots) :- \mathcal{G}'$ and then proving $q \equiv q'$. Thus, we do not select goals $\mathcal{G}, \mathcal{G}'$ s.t. for the corresponding predicates q, q', *proof_attempt*(q, q') is marked. Recall that line 2 of algorithm *Prove* marks all unsolved predicate equivalence proof obligations with *proof_attempt*.

For example, suppose we want to show $P_0 \vdash p \equiv p'$ and let the tree of predicate equivalences constructed by *Prove* via transformation is as follows. In other words, proof of $P_0 \vdash p \equiv p'$ invokes $P_0 \vdash s \equiv s'$ (via conditional folding or conditional equivalence). This proof is completed without invoking any further obligations. $P_0 \vdash p \equiv p'$ then invokes $P_0 \vdash q \equiv q'$ as a subproof. This proof may again invoke $P_0 \vdash q \equiv q'$ as a subproof as shown in the figure. However, at this stage *proof_attempt*(q,q') is already marked. This is because all proof obligations appearing in an ancestor node of this predicate equivalence proof tree are marked by *Prove* as *proof_attempt*. These are the proof obligations whose proofs have been attempted but not completed. Invoking such subproofs again result in potential non-termination of the proof search.



Thus, if *proof_attempt*(q, q') is true, and we choose $q \equiv q'$ as a condition to enable a folding step, the nested sub proof of $q \equiv q'$ will immediately fail (line 5 of Algorithm *Prove* fails such proof paths to ensure termination). Since our strategies do not permit backtracking from a failed proof path, we *must* avoid choosing such conditions.

Conditional Equivalence Conditional equivalence is the final step in proving a predicate equivalence proof obligation $p \equiv p'$. It involves invoking new predicate equivalence proof obligations, which if proved, establish the equivalence of p and p'. First we show an example to illustrate its use. Let us suppose that the clauses of predicates `q`, `q'` after unfolding, folding and conditional folding are as shown in Figure 12. We then define new predicates `new1`, `new2`, `new3`, `new1'`, `new2'`, `new3'`

```
C1  q([a|X]) :- a1(X), a2(X).      C1'  q'([a|X]) :- a1'(X), a2'(X).
C2  q([a|X]) :- a3(X).             C2'  q'([b|X]) :- b1'(X), b2'(X).
C3  q([b|X]) :- b1(X), b2(X).      C3'  q'([b|X]) :- b3'(X), b4'(X).
C4  q([c|X]) :- c1(X), c2(X).      C4'  q'([c|X]) :- c1(X), c2(X).
C5  q([d|X]) :- q(X).              C5'  q'([d|X]) :- q'(X).
C6  q([e|X]) :- e1(X), e2(X).      C6'  q'([f|X]) :- f1'(X), f2'(X).

    new1(X) :- a1(X), a2(X).   new1'(X) :- a1'(X), a2'(X).
    new1(X) :- a3(X).          new2'(X) :- b1'(X), b2'(X).
    new2(X) :- b1(X), b2(X).   new2'(X) :- b3'(X), b4'(X).
    new3(X) :- e1(X), e2(X).   new3'(X) :- f1'(X), f2'(X).
```

Figure 12: Choosing conditions for Conditional equivalence

and spawn off the predicate equivalence obligations `new1` $\equiv$ `new1'`, `new2` $\equiv$ `new2'`, `new3` $\equiv$ `false` and `new3'` $\equiv$ `false`. Note that this is sufficient to prove `q` $\equiv$ `q'`. By folding using the definitions of the new predicates the clauses of `q` and `q'` become syntactically equivalent provided these equivalences hold.

Formally, suppose we are proving an obligation $P_0 \vdash p \equiv p'$ and let $P_0, P_1, \ldots, P_n$ be the transformation sequence constructed by applying unfolding, folding and conditional folding to the clauses of p, p'. Then, we compare the clause sets of p, p' in P_n and spawn off a set of equivalence proof obligations which are sufficient to show the syntactic equivalence of p, p'. To explain how these equivalence proof obligations are selected we define:

Definition 6.3 (Matching Partition) *Let p, p' be two predicates of same arity in a program P and let $\{C_1, \ldots, C_m\}$ and $\{C'_1, \ldots, C'_n\}$ be the clauses of p, p' in P. Then $(\langle CS_1, \ldots, CS_k\rangle, \langle CS'_1, \ldots, CS'_k\rangle)$ is a matching partition of predicates p, p' provided*

- $\bigcup_{1\leq i\leq k} CS_i = \{C_1, \ldots, C_m\}$ *and* $\bigcup_{1\leq i\leq k} CS'_i = \{C'_1, \ldots, C'_n\}$
- $\forall 1 \leq i \leq k$, *the heads of all clauses in CS_i (CS'_i) are variants*[3] *of each other.*
- $\forall 1 \leq i, j \leq k$, *if $i \neq j$ then the head of any clause in CS_i (CS'_i) is not a variant of the head of any clause in CS_j (CS'_j).*
- $\forall 1 \leq i \leq k$, *if any clause in CS_i has a head $p(t_1, \ldots, t_n)$ then all clauses in CS'_i have head $p'(t'_1, \ldots, t'_n)$ where $(t'_1, \ldots, t'_n)$ is a variant of $(t_1, \ldots, t_n)$.*

[3]Identical modulo variable renaming

For example, in Figure 12 the matching partition of clause-set of `q` is $\{C_1, C_2\}$, $\{C_3\}$, $\{C_4\}$,$\{C_5\}$, $\{C_6\}$, $\{\}$. The corresponding matching partition for the clause-set of `q'` is $\{C'_1\}$, $\{C'_2, C'_3\}$, $\{C'_4\}$,$\{C'_5\}$, $\{\}$, $\{C'_6\}$. Note that some of the partitions in the matching partition may be empty leading to nested proof obligations of the form $new_i \equiv false$. Using the notion of matching partition of predicates p, p', we can compute the conditions required for showing the equivalence $p \equiv p'$ as follows.

Definition 6.4 (Conditions for showing equivalence) Let $P_0 \vdash p \equiv p'$ be an equivalence proof obligation and let $P_0, \ldots, P_n$ be the transformation sequence constructed after unfolding, folding and conditional folding. Let $\langle CS_1, \ldots, CS_k \rangle$ and $\langle CS'_1, \ldots, CS'_k \rangle$ be the matching partitions of clause-sets of p, p' in P_n. Then, the set of conditions selected for proving $p \equiv p'$ are $q_i \equiv q'_i$ (for all $1 \leq i \leq k$) where:

1. q_i, q'_i are defined by the clauses in CS_i and CS'_i respectively.
2. *proof_attempt*(q_i,q'_i) has not been marked by Algorithm *Prove*.

The above three restrictions ensure that we select conditional equivalences, which if true, are guaranteed to make p, p' syntactically equivalent. In particular, (2) avoids selecting conditions whose equivalence proof attempt is guaranteed to fail. Recall that if *proof_attempt*(q, q') is true then $q \equiv q'$ represents a proof obligation whose proof has been attempted but not completed. As discussed earlier, algorithm *Prove* avoids re-invoking such proof obligations to prevent non-termination. Finally, (1) states that if

$$CS_i = \{p(\ldots) :\!- \mathcal{G}_1, \ldots, p(\ldots) :\!- \mathcal{G}_{l_i}\}$$
$$CS'_i = \{p'(\ldots) :\!- \mathcal{G}'_1, \ldots, p'(\ldots) :\!- \mathcal{G}'_{l'_i}\}$$

then q_i and q'_i are defined by the clauses:

$$\{q_i(\ldots) :\!- \mathcal{G}_1, \ldots, q_i(\ldots) :\!- \mathcal{G}_{l_i}\}$$
$$\{q'_i(\ldots) :\!- \mathcal{G}'_1, \ldots, q'_i(\ldots) :\!- \mathcal{G}'_{l'_i}\}$$

6.3 Example Liveness and Safety Proofs

The use of our strategies for establishing predicate equivalences is now illustrated over two examples of parameterized networks of linear topology : chain, token ring. The

logic program predicates encoding these networks have a linear recursive structure in their clauses. Application of our proof technique to more substantial case studies appear in the next chapter.

6.3.1 Liveness in Unidirectional Chains

Recall the logic program of Figure 2 (page 12) which formulates a liveness property about token-passing chains, namely, that the token eventually reaches the left-most process in any arbitrary length chain. We obtain P_0, the starting point of our transformation sequence, by annotating each clause of the program in Figure 2 with counter values of $(1,1)$. To establish the liveness property, we prove that `thm(X)` $\equiv$ `gen(X)`, by invoking $Prove(\texttt{thm}, \texttt{gen}, \langle P_0 \rangle)$. The proof is illustrated in Figure 13a.

Proof of `thm` $\equiv$ `gen`: Since `thm` $\overset{P_0}{\gamma}$ `gen`, we must transform the predicates. Since `gen` is a predicate appearing in the system description, we do not transform it. As `thm` is a predicate defined using the system and property description, we transform it to make it syntactically equivalent to `gen`. By repeatedly unfolding the definition of `thm` in P_0, we obtain program P_5 where `thm` is defined as:

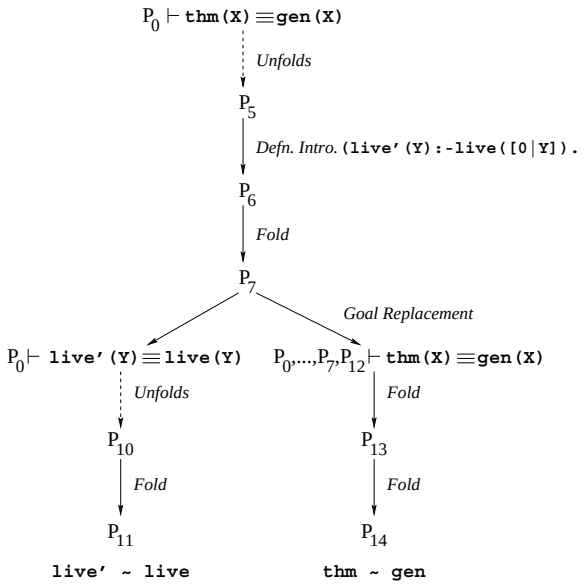
```
thm([1]).                                          (3,3)
thm([0|X]) :- gen(X), X = [1|_].                   (5,5)
thm([0|X]) :- gen(X), trans(X,Y), live([0|Y]).     (4,4)
```

Further unfolding in P_5 is not possible since it involves unfolding an atom which is already unfolded in the sequence $P_0, \ldots, P_5$, thereby risking non-termination. In addition no folding transformation is applicable at this stage. However, if $\forall$`Y live([0|Y])` $\Leftrightarrow$ `live(Y)` we can fold the last two clauses of `thm`. Thus, *conditional folding* is true at P_5, and hence *replace_and_prove* is invoked with $\mathcal{G}$ = `live([0|Y])` and $\mathcal{G}'$ = `live(Y)`. Since `live([0|Y])` is not an open atom, a new name:

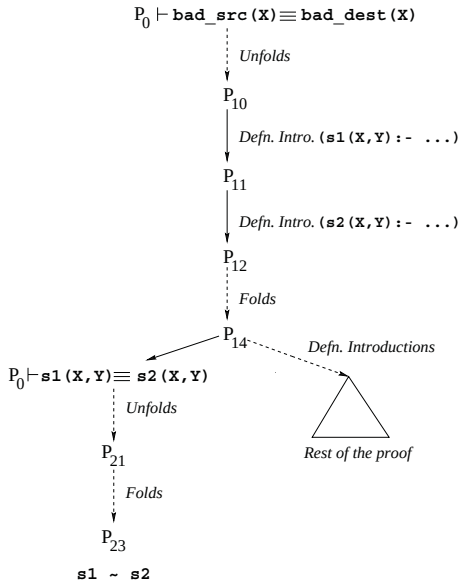
```
live'(Y) :- live([0|Y]).                           (1,1)
```

is added to P_5 to yield P_6. This simply converts the goal equivalence problem of showing $\forall$`Y live([0|Y])` $\Leftrightarrow$ `live(Y)` to a predicate equivalence problem. We fold third clause of `thm` above using the newly introduced clause as folder, obtaining P_7:

```
thm([1]).                                          (3,3)
thm([0|X]) :- gen(X), X = [1|_].                   (5,5)
thm([0|X]) :- gen(X), trans(X,Y), live'(Y).        (3,3)
```



(a) Liveness in unidirectional chains



(b) Mutual exclusion in token rings

Figure 13: Fragments of Liveness and Safety Proofs of Chains and Rings

We then proceed to prove `live'` $\equiv$ `live`. This subproof is shown in the left branch of the tree in Figure 13a). Then using the bounds on atom measures returned by this call, we replace `live'(X)` with `live(X)` in the definition of `thm` in P_7 (right branch in Figure 13a).

Proof of `live'` $\equiv$ `live`: *Prove*(`live'`, `live`, $\langle P_0 \rangle$) performs a series of unfoldings, yielding programs P_8, P_9 and P_{10}. Any further unfolding involves unfolding an atom already unfolded in the sequence P_0, P_8, P_9, P_{10} and risks non-termination. In P_{10}, `live'` is defined by the following clauses:

```
live'([1|Z]).                                   (4,4)
live'(X) :- trans(X,Z), live([0|Z]).            (3,3)
```

Folding is applicable is P_{10}, in the second clause of `live'`, yielding P_{11} with

```
live'([1|Z]).                                   (4,4)
live'(X) :- trans(X,Z), live'(Z).               (2,2)
```

Now, `live'` $\overset{P_{11}}{\sim}$ `live` and hence *Prove*(`live'`, `live`, $\langle P_0 \rangle$) terminates. To compute the bounds in the atom measures we use theorem 5.5 to obtain the constraints:

$$\Delta(\texttt{live'}, \texttt{live}) \leq 4 - 1$$
$$\Delta(\texttt{live'}, \texttt{live}) \leq 2 - 1 + \Delta(\texttt{live'}, \texttt{live})$$
$$\Delta'(\texttt{live'}, \texttt{live}) \geq 4 - 1$$
$$\Delta'(\texttt{live'}, \texttt{live}) \geq 2 - 1 + \Delta'(\texttt{live'}, \texttt{live})$$

The tightest bounds satisfying these constraints are given by:

$$\Delta(\texttt{live'}, \texttt{live}) = 3,\ \Delta'(\texttt{live'}, \texttt{live}) = \infty$$

These bounds are returned by *Prove*.

Resuming proof of `thm` $\equiv$ `gen`: Now *replace_and_prove* replaces `live'(X)` with `live(X)` in the definition of `thm` in P_7, yielding P_{12} with:

```
thm([1]).                                       (3,3)
thm([0|X]) :- gen(X), X = [1|_].                (5,5)
thm([0|X]) :- gen(X), trans(X,Y), live(Y).      (6,∞)
```

We can now fold the last two clauses of `thm` using the definition of `live` in P_0. Note that the folding uses a recursive definition of a predicate with multiple clauses. The program-transformation system developed in Chapter 3 was the first to permit such folding. Thus we obtain P_{13}:

```
thm([1]).                                   (3,3)
thm([0|X]) :- gen(X), live(X).              (4,∞)
```

This completes the conditional folding step (which had invoked *replace_and_prove* and thereby constructed `live'` $\equiv$ `live` as a subproof). We can fold again using the definition of `thm` in P_0, giving P_{14} where `thm` is defined as:

```
thm([1]).                                   (3,3)
thm([0|X]) :- thm(X).                       (3,∞)
```

We now have `thm` $\overset{P_{14}}{\sim}$ `gen`, thereby completing the equivalence proof.

It is interesting to observe in Figure 13a that the unfolding steps that transform P_0 to P_5 and P_7 to P_{10} are interleaved with folding steps. In other words, algorithmic and deductive steps are interleaved in the proof of the equivalence `thm` $\equiv$ `gen`.

6.3.2 Mutual Exclusion in Token Rings

We present a proof for mutual exclusion in a n process token ring for any n. The program P_0 encoding the verification problem is given in Figure 14. The n-process token ring is described by the predicates `gen`, `trans`, `trans1`, `trans2`. As in the case of chains, we represent the global state of a ring as a list of local states, by arbitrarily choosing a process in the ring as the first in the list. Processes with tokens are in local state 1 while processes without tokens are in state 0. `trans` is now divided into two parts: `trans1` which transfers the token to the left neighbor in the list, and `trans2` which transfers the token form the front of the list to the back, thereby completing the ring.

We want to verify that for any n, for any state in a n-process token ring, *always there is at most one token in the ring.* The complement of this property is encoded by the predicate `bad`. We then show transition invariance *i.e.* a `bad` state can be reached only from a `bad` state. This is proved by showing `bad_dest` $\Rightarrow$ `bad_src`; in this example we show a stronger result, namely the equivalence of `bad_dest` and `bad_src`. In addition, if we show that none of the initial states are `bad`, which is established by showing `bad_start` $\equiv$ `false`, then the proof of mutual exclusion is

```
C1:  bad_start(X) :- gen(X), bad(X).                (1,1)
C2:  bad_src(X,Y) :- trans(X,Y),bad(X).             (1,1)
C3:  bad_dest(X,Y) :- trans(X,Y),bad(Y).            (1,1)
C4:  gen([0,1]).                                    (1,1)
C5:  gen([0|X]) :- gen(X).                          (1,1)
C6:  trans(X, Y) :- trans1(X, Y).                   (1,1)
C7:  trans([1|X], [0|Y]) :- trans2(X, Y).           (1,1)
C8:  trans1([0,1|T], [1,0|T]).                      (1,1)
C9:  trans1([H|T], [H|T1]) :- trans1(T, T1).        (1,1)
C10: trans2([0], [1]).                              (1,1)
C11: trans2([H|X], [H|Y]) :- trans2(X, Y).          (1,1)
C12: bad([1|X]) :- one_more_token(X).               (1,1)
C13: bad([_|X]) :- bad(X).                          (1,1)
C14: one_more_token([1|_]).                         (1,1)
C15: one_more_token([_|X]) :- one_more_token(X).    (1,1)
```

Figure 14: Mutual exclusion in a n-process token ring

complete. We now show the proof constructed by algorithm *Prove* to demonstrate these predicate equivalences.

Proof of `bad_start` $\equiv$ `false` C_1 is the only clause defining `bad_start` in P_0. *Prove* first unfolds this clause to obtain the following clauses for `bad_start`. No further unfolding is possible without violating *FIN*.

```
C1': bad_start([0|X]) :- gen(X),bad(X).   (3,3)
```

An unconditional folding step is now applicable. Folding C_1' using C_1 we obtain:

```
C2': bad_start([0|X]) :- bad_start(X).   (2,2)
```

The predicate `bad_start` is now defined by only a self-recursive clause. Therefore, by applying *deletion of failed clauses*, predicate `bad_start` has no defining clauses, and hence `bad_start` $\equiv$ `false`.

Proof of `bad_src` $\equiv$ `bad_dest` We now present the proof of `bad_src(X,Y)` $\equiv$ `bad_dest(X,Y)` (it has been sketched in Figure 13b). Both `bad_dest` and `bad_src` are defined using system and property description predicates and do not themselves appear in the system/property description. Therefore, we transform both of them. First *Prove* performs unfolding of clauses C_2, C_3 (the defining clauses of `bad_src`,

bad_dest) to obtain a program P_{10} where bad_src, bad_dest are defined as follows. The clause measures are not shown since they are not required in the rest of the proof.

C'_3: `bad_src([0,1,1|X], [1,0,1|X]).`
C'_4: `bad_src([0,1,H|T], [1,0,H|T]) :- one_more_token(T).`
C'_5: `bad_src([1|X],[1|Y]) :- trans1(X,Y), one_more_token(X).`
C'_6: `bad_src([H|X],[H|Y]) :- trans1(X,Y), bad(X).`
C'_7: `bad_src([1,1|X],[0,1|Y]) :- trans2(X,Y).`
C'_8: `bad_src([1,H|X],[0,H|Y]) :- trans2(X,Y), one_more_token(X).`
C'_9: `bad_dest([0,1,1|X], [1,0,1|X]).`
C'_{10}: `bad_dest([0,1,H|T], [1,0,H|T]) :- one_more_token(T).`
C'_{11}: `bad_dest([1|X],[1|Y]) :- trans1(X,Y), one_more_token(Y).`
C'_{12}: `bad_dest([H|X],[H|Y]) :- trans1(X,Y), bad(Y).`
C'_{13}: `bad_dest([1,1|X],[0,1|Y]) :- trans2(X,Y), one_more_token(Y).`
C'_{14}: `bad_dest([1,H|X],[0,H|Y]) :- trans2(X,Y), bad(Y).`

No more unfolding is applicable without violating the finiteness condition *FIN* *i.e.* without risking non-termination of unfolding. Also, no unconditional folding or conditional folding step is applicable. At this stage *Prove* performs conditional equivalence steps. By using the notion of matching partition, the following new goal equivalences will be generated (refer Definition 6.4).

`trans1(X,Y), one_more_token(X)` $\equiv$ `trans1(X,Y), one_more_token(Y)`
`trans1(X,Y), bad(X)` $\equiv$ `trans1(X,Y), bad(Y)`
`trans2(X,Y), one_more_token(Y)` $\equiv$ `trans2(X,Y)`
`trans2(X,Y), one_more_token(X)` $\equiv$ `trans2(X,Y), bad(Y)`

For each of these goal equivalences, *replace_and_prove* is invoked. Since these goals are not open atoms, this will introduce the following definitions

D_1: `s1(X,Y) :- trans1(X,Y), one_more_token(X).` $(1,1)$
D_2: `s2(X,Y) :- trans1(X,Y), one_more_token(Y).` $(1,1)$
D_3: `f1'(X,Y) :- trans1(X,Y), bad(X).` $(1,1)$
D_4: `f2'(X,Y) :- trans1(X,Y), bad(Y).` $(1,1)$
D_5: `g2''(X,Y) :- trans2(X,Y), one_more_token(Y).` $(1,1)$
D_6: `g1''(X,Y) :- trans2(X,Y), one_more_token(X).` $(1,1)$
D_7: `f2''(X,Y) :- trans2(X,Y), bad(Y).` $(1,1)$

Prove now performs folding using the above definition clauses as folder. To complete the mutual exclusion proof, we then have to execute the following subproofs:

- *Prove*(`s1(X,Y)`, `s2(X,Y)`, $\langle P_0 \rangle$)
- *Prove*(`f1'(X,Y)`, `f2'(X,Y)`, $\langle P_0 \rangle$)
- *Prove*(`trans2(X,Y)`, `g2''(X,Y)`,$\langle P_0 \rangle$)
- *Prove*(`g1''(X,Y)`, `f2''(X,Y)`, $\langle P_0 \rangle$).

Once these subproofs are established we can infer `bad_src` $\equiv$ `bad_dest`. We now show the construction of the first subproof. The others follow similarly.

Proof of `s1` $\equiv$ `s2` Algorithm *Prove* employs unfolding to get the following clauses for `s1`,`s2`. No further unfolding is possible without violating *FIN*.

D'_1:	`s1([0,1|X], [1,0|X]).`	$(4,4)$
D'_2:	`s1([1|X], [1|Y]) :- trans1(X, Y).`	$(3,3)$
D'_3:	`s1([H|X], [H|Y]) :- trans1(X, Y), one_more_token(X).`	$(3,3)$
D'_5:	`s2([0,1|X], [1,0|X]).`	$(3,3)$
D'_6:	`s2([1|X], [1|Y]) :- trans1(X, Y).`	$(3,3)$
D'_7:	`s2([H|X], [H|Y]) :- trans1(X, Y), one_more_token(Y).`	$(3,3)$

Unconditional folding steps are now applicable. Folding clause D'_3 using clause D_1 as folder we get the following definition of `s1`

D'_1:	`s1([0,1|X], [1,0|X]).`	$(4,4)$
D'_2:	`s1([1|X], [1|Y]) :- trans1(X, Y).`	$(3,3)$
D'_4:	`s1([H|X], [H|Y]) :- s1(X, Y).`	$(2,2)$

Only one more unconditional folding step is applicable. We fold D'_7 using clause D_2 as folder. We get:

D'_5:	`s2([0,1|X], [1,0|X]).`	$(3,3)$
D'_6:	`s2([1|X], [1|Y]) :- trans1(X, Y).`	$(3,3)$
D'_8:	`s2([H|X], [H|Y]) :- s2(X, Y).`	$(2,2)$

We now have `s1` $\sim$ `s2`, thereby completing the proof of `s1` $\equiv$ `s2`.

Chapter 7

Applications and Case Studies

In this chapter, we discuss application of the program transformation based proof technique in automatically proving safety properties of parameterized protocols. First, we present the verification of mutual exclusion in the Java Meta-Locking Algorithm [ADG+99]. The Java Meta-Locking Algorithm is a distributed algorithm recently proposed by Sun Microsystems to ensure mutually exclusive access of shared Java objects by Java threads. A proof of correctness of the algorithm involves proving mutual exclusion in the access of a Java object by arbitrary number of Java threads. Previously, model checking has been used to verify mutual exclusion for different instances of the protocol, obtained by fixing the number of threads [BSW00]. In Section 7.1 we outline the use of the transformation based prover in automatically constructing a proof of mutual exclusion for the entire infinite family.

In Section 7.2 we discuss the use of the transformation based prover in proving data consistency properties of cache coherence protocols. These protocols are used in shared memory multiprocessor systems with local caches. They ensure that multiple cached copies of a shared block of memory posses a consistent data value. A formal correctness proof of such protocols involves proving certain consistency properties as invariants for any run of the protocol with any number of processes. We discuss the use of our prover for proving invariants of cache coherence protocols with any number of processors attached to a single bus. We then present the applicability of the proof technique in verifying cache coherence protocols with tree networks of buses.

Finally, in Section 7.3 we present the experimental results obtained from the verification of these protocols in our transformation based prover.

7.1 Mutual Exclusion of Java Meta-Lock

In recent years, Java has gained popularity as a concurrent object oriented language, and hence substantial research efforts have been directed to efficiently implementing the different language features. One of the features which is commonly used in the execution of most Java programs are the synchronization operations. In Java language, any object can be synchronized upon by different threads via synchronized methods and synchronized statements. Mutual exclusion in the access of an object is ensured since a synchronized method first acquires a lock on the object, executes the method and then releases the lock. To ensure fairness and efficiency in accessing any object, each object maintains some *synchronization data*. Typically this synchronization data is a FIFO queue of the threads requesting the object. Note that to ensure mutually exclusive access of an object, it is necessary to observe a protocol while different threads access this synchronization data. The Java meta-locking algorithm [ADG+99] solves this problem. It is a distributed algorithm which is observed by each thread and any object for accessing the synchronization data of that object. It is a time and space efficient scheme to ensure mutually exclusive access of the synchronization data, thereby ensuring mutually exclusive access of any object. Below, we outline the modeling of the algorithm and its verification in our prover.

7.1.1 Modeling the Protocol

First we informally describe the meta-locking algorithm and then present the formal model used by us for verification. As mentioned above, the meta-locking algorithm is a protocol observed by threads in accessing an object's synchronization data.

Informal description We consider the protocol for a single object and an arbitrary number of threads. The pattern of a synchronization operation executed by a thread is as follows:

1. *Get* the object's meta-lock if no thread is accessing the synchronization data; otherwise wait for a *hand-off*.
2. *Manipulate* the synchronization data.
3. *Release* the meta-lock if no other thread is waiting; otherwise *hand-off* the meta-lock to a waiting thread.

Thus, both the acquisition as well as the release of the meta-lock by a thread may involve contention from other threads. A thread that faces no contention from other threads while acquiring/releasing the meta-lock is said to execute a *fast path*. Otherwise, it executes a *slow path* involving a hand-off from another thread.

For the purposes of describing the meta-locking scheme, an object can be assumed to be either busy (when a thread is modifying its synchronization data) or not busy (otherwise). The synchronization data maintained by the object is a FIFO queue of lock records. Each lock record denotes a thread waiting to access the object. A thread is allowed to access the object once it reached the head of the queue. Since we are only interested in modeling and verifying the meta-locking scheme for accessing this FIFO queue, we do not model the queue itself.

A thread attempts to acquire the meta-lock by performing an atomic swap operation of its thread id with the object's internal data-structures, called the *multi-use word* by the designers of the protocol. If the contents of the multi-use word indicate that the object is not busy then the thread has acquired the meta-lock via a fast path. It now proceeds to manipulate the synchronization data. Otherwise, the thread must wait for a hand-off by the thread currently owning the meta-lock. In this case, threads contending for the meta-lock acquire it in the order in which they executed their swap instructions. Note that at any point of time the object's multi-use word records which thread last tried to acquire the meta-lock. When several threads are contending to acquire the meta-lock, each of them knows its predecessor via the swap operation *i.e.* each thread knows the thread which tried to acquire the meta-lock just before it. The first one to execute the swap operation has no predecessor and acquires the meta-lock.

To release the meta-lock, thread i executes an atomic compare-and-swap operation to compare the current value recorded in the multi-use word with its own identifier. If the two values are the same, then the thread infers that no other thread is waiting to access the meta-lock. It then releases the meta-lock via the fast path, and the object is restored to non-busy state. Otherwise, if other threads are trying to acquire the meta-lock, thread i obtains the id of the thread which last tried to acquire the meta-lock. This however, need not be the thread which tried to acquire the meta-lock immediately after thread i, *i.e.* the successor of thread i. In other words, thread i is ready to hand-off the meta-lock but does not know which thread to wake up. For this reason, the algorithm requires the threads acquiring and releasing the meta-lock

to enter a race. The purpose of this race is to eliminate busy-waiting from the meta-locking algorithm. The race proceeds as follows. Every thread attempting to acquire the meta-lock executes its portion of the hand-off thereby informing its predecessor of its identity. Simultaneously, the thread releasing the meta-lock executes its portion of the hand-off thereby informing its desire to relinquish the meta-lock. Note that this is effectively a race between the releasing thread i and the successor of i (say thread j) where both i and j compete to execute its portion of the hand-off first. Through the race, thread i releases the meta-lock. Also thread j is identified as the successor of thread i, and thread j acquires the meta-lock.

Formal model The formal model of the protocol consists of the parallel composition of an object process, a hand-off process and an arbitrary number of thread processes. Each process either performs an autonomous action or communicates with another process via synchronization. Two processes synchronize as follows : one of them makes an `in(a)` action and the other makes an `out(a)` action where `a` is an action label. Any process makes an autonomous action by performing a `self(a)` action where `a` is an action label.

We model the object process without the synchronization data (the FIFO queue of lock records) since we are only interested in verifying mutually exclusive access of this data. Thus we do not model the synchronization data and its manipulation by the threads. Apart from the synchronization data, note that the meta-locking algorithm *implicitly* maintains another queue : the queue of threads currently contending for the meta-lock. Modeling this queue of contending threads is essential for a faithful modeling of the acquisition and the release of the meta-lock. However, for verifying mutual exclusion we do not need to model the actual contents of the queue *i.e.* the identifiers of the threads currently contending for the meta-lock. Instead we only model the length of the queue, a natural number. This abstraction of modeling the queue of contending threads via the queue length does not allow us to prove properties like : if thread i requested the meta-lock, it will eventually get the meta-lock. However, we can verify global consistency properties which can be inferred from the global state of the protocol without referring to individual threads. In this example we seek to verify one such property, namely mutual exclusion: no two threads ever own the meta-lock together.

The object can therefore modeled as a process with a boolean control variable and a data variable which is a natural number. The control variable records whether

the object is busy, *i.e.* whether any thread currently possesses the meta-lock. The data variable records the number of processes currently waiting for the meta-lock. The states of the object are represented by the infinite set of terms { `not_busy` } ∪ { `busy(N)` | `N` ∈ ℕ }. The state of the object is `not_busy` if no thread owns the meta-lock, and none is waiting for the meta-lock. Otherwise, if a thread owns the meta-lock and n others are waiting for it, the state of the object is $\texttt{busy}(\texttt{s}^n(\texttt{0}))$. The transition relation of the object is described by the following set of logic program clauses ; `obj_trans(S, A, T)` is true if the object makes a transition from state `S` to state `T` on action `A`. Note that the object is an *infinite state system* since it has a data variable which is a natural number. We assume that natural numbers are represented by the terms $\{\texttt{0}, \texttt{s}(\texttt{0}), \texttt{s}(\texttt{s}(\texttt{0})), \ldots\}$ as captured by the predicate `nat` shown below.

```
obj_trans(not_busy, in(get_fast), busy(0)).
obj_trans(busy(0), in(put_fast), not_busy).
obj_trans(busy(X), in(get_slow), busy(s(X))) :- nat(X).
obj_trans(busy(s(X)), in(put_slow), busy(X)) :- nat(X).

nat(0).
nat(s(X)) :- nat(X).
```

On the other hand, the thread and the hand-off process are finite state systems. Figure 15 shows the labeled transition systems of a thread and the hand-off process. The logic program encoding of these relations is simply a set of facts[1].

All the transitions of a thread involve synchronizing with either the object process or the hand-off process. When the object is in `not_busy` state, it synchronizes with a thread in `idle` state using a `get_fast` signal and moves to state `busy(0)`. The thread moves to `owner` state and this corresponds to the acquisition of the meta-lock by the thread via fast path. Note that no other thread is waiting for the meta-lock at this time. Similarly, a thread releases a meta-lock via the fast path by synchronizing with the object using a `put_fast` signal. The object state is then restored to `not_busy`. While the object is meta-locked in a state $\texttt{busy}(\texttt{s}^n(\texttt{0}))$ if another thread tries to acquire the meta-lock via the slow path it synchronizes with the object using a `get_slow` signal. The number of threads waiting for the meta-lock is then incremented and the object moves to state $\texttt{busy}(\texttt{s}^{n+1}(\texttt{0}))$. Similarly, if the object is state $\texttt{busy}(\texttt{s}^n(\texttt{0}))$, and a thread releases the meta-lock via the slow path using a `put_slow`

[1]Logic program clauses with empty body

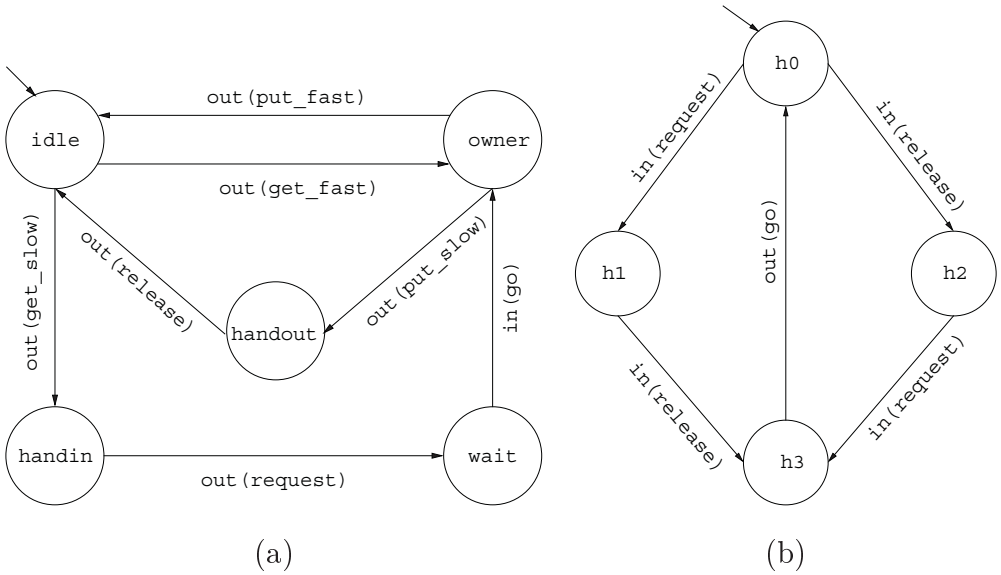


Figure 15: Labeled Transition System of (a) any thread (b) hand-off process

signal, then the number of threads waiting for the meta-lock is decremented and the object moves to state $\texttt{busy}(\texttt{s}^{n-1}(\texttt{0}))$.

The race between the acquiring and the releasing threads is modeled by the hand-off process. The acquiring (releasing) thread synchronizes with the hand-off process using the `request` (`release`) signal. Through this synchronization, the releasing thread moves back to `idle` state since it no longer possesses the meta-lock. The acquiring thread on the other hand waits for a go-ahead from the hand-off process. The hand-off process waits for both the `request` and the `release` signals (which may arrive in any order) after which it synchronizes with the acquiring thread using the `go` signal. Having obtained the go-ahead, the acquiring thread gets the meta-lock and moves to `owner` state.

Having modeled the thread, object and hand-off processes, the parameterized system that we need to verify is a parallel composition (in the sense of Milner's CCS [Mil89]) of one object process, one hand-off process and ≥ 2 thread processes. Initially, the object is `not_busy`, the hand-off process is in state `h0` and all threads are `idle`. This composed system has infiniteness in both data and control. The infiniteness in control is due to infinite number of thread processes whereas the infiniteness in data is due to an infinite domain variable in the object process.

7.1.2 Verifying Mutual Exclusion

We straightforwardly encoded the state representations and the transitions in the formal model of the protocol as a logic program. A global state of the protocol in our logic program encoding is a 3-tuple `(Th, Obj, H)` where `Th` is a list of thread states (of length ≥ 2) , `Obj` is a state of the object process and `H` is a state of the hand-off process. The global transition relation is straightforwardly programmed as a predicate `trans` which allows synchronization between a thread and the hand-off process, or between a thread and the object. We automatically verified mutual exclusion using our prover. Note that this was achieved without performing any manual abstractions of the formal model. However, it requires us to provide a strengthening of the mutual exclusion invariant property.

From the labeled transition system of a thread process in Figure 15 we observe that a thread possesses the meta-lock when it is in state `owner` or `handout`. Thus, we need to prove the invariant **AG** $\neg$`bad` where `bad` is a logic program predicate which is true in all global states where ≥ 2 threads are in `owner` or `handout` states. This corresponds to the *indexed CTL* (refer [BCG89]) formula **AG** $\oplus_i$`owner`$_i \vee$ `handout`$_i$. The program predicate `bad` encodes the proposition $\neg(\oplus_i$`owner`$_i \vee$ `handout`$_i)$.

We prove this invariant by showing transition invariance, *i.e.* by showing that none of the initial states are `bad`, and each transition preserves the invariant. Recall from the last chapter that this is proved by showing `bad_start` $\Rightarrow$ `false` and `bad_dest` $\Rightarrow$ `bad_src` where the predicates `bad_dest`, `bad_src` and `bad_start` are defined as:

```
bad_start(S) :- gen(S), bad(S).
bad_dest(S, T) :- trans(S, T), bad(T).
bad_src(S, T) :- trans(S, T), bad(S).
```

The predicate `trans` encodes the global transition relation of the protocol. The encoding of the predicate `bad` is obtained by strengthening the invariant we need to prove. Note that the mutual exclusion property is expressed only in terms of the local states of the threads. However, the global state of the protocol is not simply the collection of the local states of the threads but also that of the object and the hand-off process. In the protocol description, the local states of the object and the hand-off process also bear an intended meaning in terms of contention for the meta-lock. For example, if the object is in state `not_busy` then no thread possesses the meta-lock. This is an assertion stronger than $\neg$`bad` which only requires ≤ 1 thread to posses the meta-lock. Similarly, once the hand-off process obtains the `release` signal from

the releasing thread (states `h2` and `h3` in Figure 15(b)), no thread must posses the meta-lock, since the next thread has not yet been given a go-ahead. By making these two assertions explicit, we obtain the definition of predicate `bad` given below. The predicates `zero_more`, `one_more` and `two_more` are defined as follows: `zero_more(Th)`, `one_more(Th)` and `two_more(Th)` are true if `Th` is a list of thread states and contains ≥ 0, ≥ 1 and ≥ 2 threads possesing the meta-lock respectively. Thus we prove the safety property **AG** $\neg$`bad` where predicate `bad` is defined below.

```
bad((Th, not_busy, h0)) :- one_more(Th).
bad((Th, not_busy, h1)) :- one_more(Th).
bad((Th, not_busy, h2)) :- zero_more(Th).
bad((Th, not_busy, h3)) :- zero_more(Th).
bad((Th, busy(N), h0)) :- nat(N), two_more(Th).
bad((Th, busy(N), h1)) :- nat(N), two_more(Th).
bad((Th, busy(N), h2)) :- nat(N), one_more(Th).
bad((Th, busy(N), h3)) :- nat(N), one_more(Th).
```

The property **AG** $\neg$`bad` is a strengthening of the mutual exclusion invariant which only forbids ≥ 2 threads to posses the meta-lock irrespective of the object and hand-off states. Note however that the strengthening of the invariant was performed by reasoning *only* about the local states of the object and hand-off processes. We did not need to consider interleavings of interactions between the constituent processes to strengthen the invariant. The strengthening was needed since our proof technique does not support strengthening of induction hypothesis in an induction proof. The mutual exclusion property is not preserved by every transition (even though a state violating mutual exclusion is never reached from the initial state of the protocol). This is because the mutual exclusion property only restricts the local states of the threads without taking into account the intended semantics of the local states of the other processes: object and hand-off. Therefore, we formulated a stronger invariant on the local states of all the constituent processes and proved that it is preserved by every transition step in the protocol by using our prover.

Experimental numbers The proof of the property **AG** $\neg$`bad` in the formal model of the meta-locking algorithm proceeds completely automatically in our program transformation based prover. The timing and other relevant statistics are furnished in Table 3. The proof was conducted on a Sun Ultra-Enterprise workstation with two 336 MHz CPUs and 2 GB of RAM. The memory consumption is lean since in

Total Time Taken:	129.8 secs
Memory consumed:	1.26 MB
# of Unfolding steps:	1981
# of Deductive steps:	311
Nesting depth of proof:	3
# of Predicate Implications proved:	39

Table 3: Statistics of mutual exclusion proof of Java Meta-Lock

the construction of a transformation sequence $P_0, P_1, \ldots, P_i$ we remember only the initial program P_0 and the current program P_i. The relatively large running time is because the unfolding or algorithmic steps have been implemented through meta-programming instead of the underlying abstract machine. Experimental evidence suggests that an implementation of the unfolding search at the abstract machine level should reduce the running time by at least an order of magnitude. Out of the 2292 transformation steps, 1981 are unfolding steps, showing that deductive steps are sparingly employed in the proof. This is because our proof strategy applies deductive steps lazily in a predicate implication proof *i.e.* only when no unfolding steps are applicable. Also, note that our proof technique supports *nested induction* proofs by spawning new predicate implications from a predicate implication proof. In other words, we construct a "proof tree" where each node of the tree is a predicate implication proof obligation. The depth of this tree for the proof of `bad_dest` $\Rightarrow$ `bad_src` in Java meta-lock is 3. The total number of nodes in the proof trees of `bad_dest` $\Rightarrow$ `bad_src` and `bad_start` $\Rightarrow$ `false` is 39. Note that in the proof of mutual exclusion even though only two predicate implications need to be proved (`bad_start` $\Rightarrow$ `false` and `bad_dest` $\Rightarrow$ `bad_src`), the prover proves 39 predicate implications (including these two) due to the nesting of proof obligations.

7.2 Verifying Cache Coherence Protocols

We now discuss the application of our proof technique to verify parameterized cache coherence protocols. Cache coherence protocols are used in shared memory multiprocessor systems where each processor posseses its own private cache. At any point of time, there may be multiple copies of the same memory block in the cache of different processors. Thus, we need to ensure that an updation to a shared memory block

is visible to all the processors. A cache coherence protocol solves this problem. It ensures that multiple cached copies of the same memory block are consistent in their data content. [AB86] examines the basic concepts in the design, implementation and evaluation of various cache coherence protocols.

Cache coherence protocols are parameterized by the number of processors which are often connected to a single bus. Initial efforts to verify cache coherence protocols were based on explicit-state and symbolic model checking. In the recent past, automated verification of parameterized cache coherence protocols has been investigated. Pong and Dubois [PD95] exploit the symmetry of these protocols and the independence of the data consistency properties on the exact number of cached copies to obtain a symbolic state space representation. In particular, they represent a set of global states by keeping track of whether 0, 1 or > 1 caches are in a particular local state. Verification of data consistency properties (safety properties) is achieved by reachability analysis in this symbolic state space. Ip and Dill [ID99] incorporate a similar state representation into the Murφ verification system to automate this abstraction. More recently, count based abstraction of global states *i.e.* keeping track of the exact number of caches in the different local states, has been studied [Del00, EFM99, EN98]. The work closest to ours is [Del00] which represents sets of states by constraints on the number of caches in various local states. A proof of a safety property then proceeds by backward reachablity analysis over sets of states represented by such constraints. The reachablity analysis is achieved by evaluation of a constraint logic program encoding of the symbolic state space of the protocol.

In our approach, we encode the states and the transitions of the protocol directly as a logic program. We encode a global state of the protocol directly as an unbounded list of local cache states; we then define logic program predicates over these lists to represent the initial states and the transition relation of the protocol. The underlying induction machinery of our prover is used to induct on the state representation and automatically produce safety proofs of the parameterized protocol. This is in contrast to [Del00] which represents global states as constraints over counts of local states and then encodes the transitions to show how these constraints are changed with each transition. This is because [Del00] is based on state space traversal (and hence needs to exploit the symmetry of cache coherence protocols to obtain a symbolic state space representation), whereas our proof technique is based on induction on the global state representation.

Since we represent the global states of a cache coherence protocol directly as

unbounded terms, we can readily extend our approach to hiererchical cache coherence protocols as well. These protocols consist of a network of buses which are arranged in the form of a tree. Each node of the tree is a bus to which multiple caches and a cache agent is connected. These protocols are parameterized by the number of buses and the number of caches in each bus. Verification of these protocols via symbolic state space traversal requires a finite representation of the infinite state space. Note that keeping track of the number of caches in different local states (as is done in [EFM99, Del00] for protocols with a single bus) no longer suffices as a finite representation. This is because the tree structure among the different buses is crucial for proving temporal properties and cannot be abstracted away. Our proof technique will represent a global state as an tree of unbounded size and induct on this tree structure to prove safety properties.

7.2.1 A Simple Example

First, we use our proof technique to prove invariants of a simple synchronization based cache coherence protocol adapted from [KM95]. We assume a single shared cache line and do not model the data in the cache line. Each processor is invalid, valid or owner. The processors either perform an autonomous read or write action, or synchronize with another processor using a invalidate, copy or ownership-transfer action. The protocol does not allow broadcasts and restricts the number of valid copies of the data to two, *i.e.* at most two processors posses a valid copy of the data. Thus, invalidation of all existing valid copies of data can be achieved by synchronization. The protocol keeps track whether a processor is a sole owner, or a shared owner. The labeled transition system of any processor is shown in Figure 16. The following shorthands are used for action labels: `rd` denotes read, `wr` denotes write, `cp` denotes copy, `inv` denotes invalidate and `ot` denotes ownership-transfer. For any action label `a`, `in(a)` and `out(a)` synchronize and `self(a)` denotes an autonomous action.

The global state of the protocol can be represented as a list of local states of each processors. The global transition relation of the protocol for any number of processors is defined over these unbounded lists. It can be encoded as the logic program predicate `trans` shown in Section 6.1, page 87. Note that the global transition relation `trans` is defined using `ltrans`, the local transition relation of each individual processor. The local transition relation can be encoded as a set of facts, where each fact corresponds to an arc in the labeled transition system of Figure 16.

To prove data consistency we need to ensure that each processor reads the value

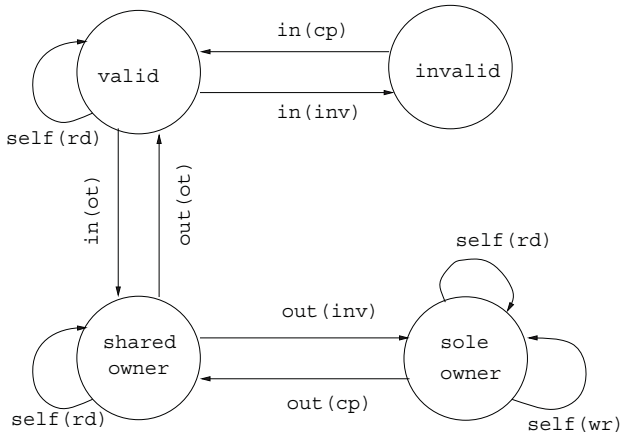


Figure 16: Labeled Transition system of any processor in a simple synchronization based cache coherence protocol

last written. For this purpose we prove the following invariants: *(i)* if one processor is a `sole_owner` all other processors are `invalid`, and *(ii)* there is exactly one owner of the cache line. In particular, for this protocol we prove the CTL properties

AG $\neg(\#\texttt{sole\_owner} + \#\texttt{valid} \geq 2)$
AG $\neg(\#\texttt{shared\_owner} + \#\texttt{sole\_owner} \geq 2)$
AG $\neg(\#\texttt{shared\_owner} + \#\texttt{sole\_owner} = 0)$

where #`shared_owner`, #`sole_owner`, #`valid` denote the number of processors in `shared_owner`, `sole_owner` and `valid` states respectively. Each of these are safety properties is of the form **AG** ¬`bad`. They are proved by encoding `bad` as a program predicate and reducing the proof obligation to predicate implications as descibed in Section 6.1. The proof of these predicate implications proceed automatically in our prover. The timings and the number of transformation steps taken to construct these proofs appear in Table 4. The annotations needed to guide the proof search are provided as described in Section 6.2. This involves marking `bad` and all predicates used in its definition (predicates describing the temporal property) as *consumer i.e.* predicates which only consume bindings via unfolding.

7.2.2 Modeling Broadcasts

Cache coherence protocols often involve broadcast of signals over a bus, say an invalidation signal may be broadcast before a write. We now show how broadcasts can be

modeled as a logic program and apply our proof technique to verify a cache coherence broadcast protcol.

Broadcast protocols [EN98, EFM99] are parameterized systems which are composed of an unbounded number of indistinguishable processes which can perform the following actions: *(i)* an autonomous action, *(ii)* synchronization of two processes on an action label *(ii)* broadcast of an action label by a process to all other processes. The global state of a broadcast protocol can therefore be modeled as a list of local states of the indistinguishable processes. The autonomous and synchronization actions can be modeled as in Section 6.1. Recall that for any action label `a` we denote `self(a)` as an autonomous action and assume that the actions `in(a)` and `out(a)` synchronize.

To model broadcast actions, we define a predicate `trans_bcast` such that `trans_bcast(S, Act, T)` is true if a transition from global state `S` to global state `T` is possible by making a broadcast of action label `Act`. A broadcast transition on action label `Act` is performed by one process making an output action (denoted `b_out(Act)`) and all other processes making the corresponding input action (denoted `b_in(Act)`). This is captured in the predicate `trans_bcast`, where `ltrans` denotes the local transition relation of each process.

```
trans_bcast([H|T], Act, [H1|T1]) :-
              ltrans(H, b_out(Act), H1), trans_all(T, Act, T1).
trans_bcast([H|T], Act, [H1|T1]) :-
              ltrans(H, b_in(Act), H1), trans_bcast(T, Act, T1).

trans_all([], _, []).
trans_all([H|T], Act, [H1|T1]) :-
              ltrans(H, b_in(Act), H1), trans_all(T, Act, T1).
```

As an application of our proof technique for the verification of broadcast protocols, we consider the single bus MESI protocol for cache coherence. Verification of safety properties of this protocol has been previously reported in [Del00, EFM99, EN98, ID99]. The local transition relation for any process is shown in Figure 17. This is encoded as the predicate `ltrans`. Each process is in one of four states:
`m` = modified. The content of the local cache has been modified and the modification has not been updated to main memory.
`e` = exclusive owner. No other process has a valid copy.
`s` = shared owner. The process has read but not write permission on the cache.
`i` = invalid. The content of the local cache is invalid.

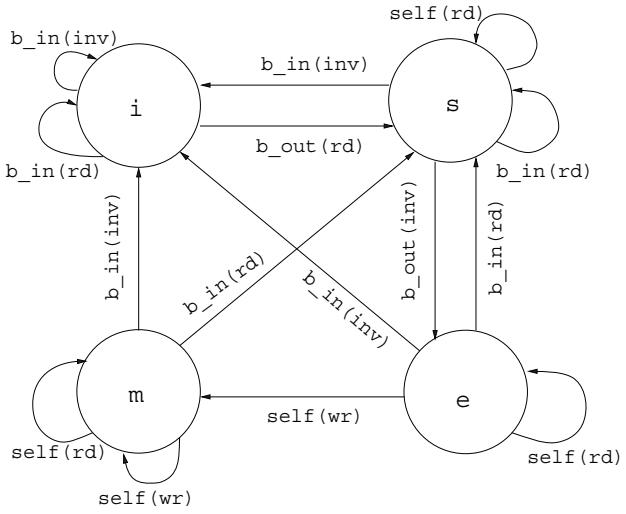


Figure 17: Transition relation of any process in MESI protocol

Initially all processes are in invalid state. Each process can make a local read (denoted `self(rd)`), a local write (denoted `self(wr)`), a broadcast on invalidate (denoted `b_in(inv)` and `b_out(inv)`) or a broadcast on read (denoted `b_in(rd)` and `b_out(rd)`). An invalidate broadcast allows a process to exclusively own and modify the data in the cache; a broadcast on read allows processes in invalid state to acquire valid copies of the data. We used our program transformation based prover to automatically prove the following invariants:

AG $\neg(\ \#\mathtt{m} + \#\mathtt{e} \geq 2\ )$
AG $\neg(\ \#\mathtt{m} + \#\mathtt{e}\ > 0 \wedge \#\mathtt{s} > 0)$

The first property ensures the number of exclusive owners and dirty caches is at most one. The second ensures mutual exclusion among the readers and writers of the data in the caches.

In [EFM99], verification of safety properties of broadcast protocols has been shown to be decidable. It is accomplished by abstracting global states as a vector of counts of the local states and then computing backward reachablity of the unsafe states. The program transformation sequence constructed by our proof technique does not necssarily simulate this backward reachability computation. Hence we cannot give completeness guarantees of our proof technique for verifying invariants of broadcast protocols. On the other hand note that our technique is based on automating induction proofs, and not tied to any particular abstract representation of states. Thus it

is applicable to parameterized systems where abstracting a global state as counts of local states does not suffice, *e.g.* parameterized tree networks.

7.2.3 Extending to Tree Based Protocols

A motivating application domain for verifying parameterized networks of tree topologies is the verification of cache coherence protocols over hierarchically arranged buses. The buses are arranged in the form of a tree. In each bus, an arbitrary number of processors, and a single cache agent is attached. Processor a_i attached to bus a communicates with another processor b_j attached to a bus b by the following sequence of actions. First, a_i communicates its request to the cache agent of bus a. This agent then percolates this request in the tree network. Once it reaches the cache agent of bus b, unit b_j communicates with its agent. The cache coherence protocol described in the IEEE Futurebus+ standard [CGH$^+$95] belongs to this category.

To study the interactions between the different nodes of the tree network in such protocols, we modeled the family of binary trees, where each node in the tree is in one of three local states: `c` (critical), `i` (idle) and `t` (trying). Thus every node in the tree represents a bus. Only one of the buses initially contain a data (which is not modeled). We need to prove mutually exclusive access to the data. The local states of each bus are:

`c` : contains data
`i` : does not contain data and none of the units have issued a request
`t` : does not contain data, trying to acquire it

Thus we need to prove that no two nodes in the tree are in the state `c`. The transition relation allows the following actions:

1. autonomous actions `i` $\rightarrow$ `t`. Any bus can try to acquire data at any point of time (due to request from some attached unit).

2. percolation of `t` (try) states up and down the tree via synchronization of child and parent. This models the percolation of the request for the data.

3. exchange of `c` (critical) states up and down the tree via synchronization of child and parent. This models the transfer of the data from one bus to another.

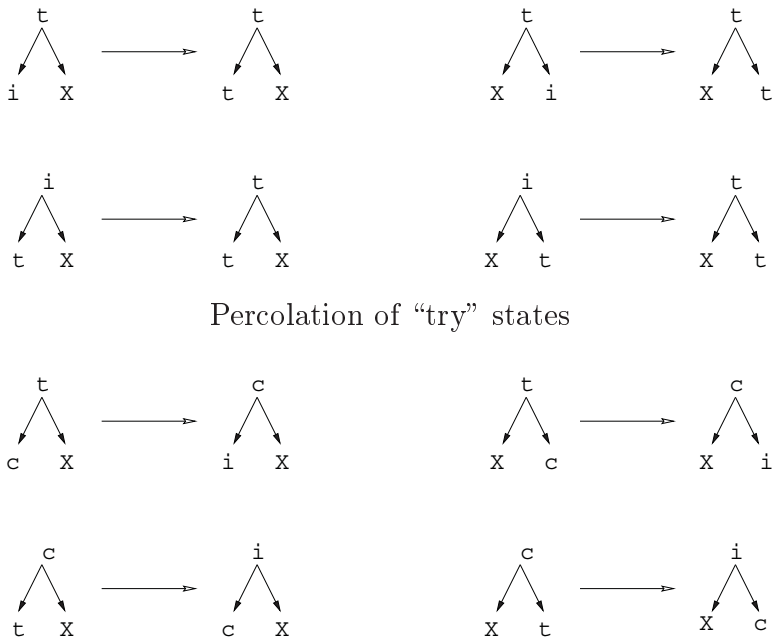


Figure 18: Synchronization actions in a tree network of processes

The synchronization actions in categories (2) and (3) are shown pictorially in Figure 18. For proving data consistency, we need to verify mutual exclusion among buses containing data, *i.e.* no two nodes in the tree are in state `c`.

In our program transformation based prover, we represent a global state of the tree network as a term `f(Root, Left, Right)` where `Root` $\in$ { `c`, `t`, `i` } is the local state of the root node, and `Left` (`Right`) is the term representing the left (right) subtree. The mutual exclusion property is defined using double recursive predicates[2] over these terms. Our prover constructs a completely automated proof of mutual exclusion in the above mentioned model of tree network. The proof is accomplished by showing transition invariance. Note that our strategies for guiding the algorithmic and deductive steps are not derived by using arguments specific to the recursive structure of the program being transformed. Therefore, they are applicable for constructing proofs of tree networks as well.

Interestingly, the prover can exploit the network topology of the problem under

[2]A logic program predicate `p` is double recursive if at least one of the clauses defining `p` contains two recursive occurrences of `p` in its body.

verification to apply heuristics for faster convergence of proof attempts. In particular, note that bad is encoded as a double recursive program predicate and trans encodes the global transition relation of the family. We need to show bad_dest $\Rightarrow$ bad_src where bad_dest and bad_src are defined as follows.

```
bad_dest(S, T)  :- trans(S, T), bad(T).
bad_src(S, T)   :- trans(S, T), bad(S).
```

Now, by unfolding and folding bad_dest and bad_src will be defined by *multiple* clauses, each of the form:

```
bad_dest(f(t̄, L, R), f(s̄, L1, R1))  :- p(L,L1), q(R,R1).
bad_src(f(t̄, L, R) f(s̄, L1, R1))  :- p'(L,L1), q'(R,R1).
```

The local state of the root node before and after a transition are $\overline{t}$ and $\overline{s}$ respectively. The left (right) child before and after a transition are denoted by L and L1 (R and R1) respectively. The meaning of the predicates p and p' can be understood as follows. A clause of bad_dest shown above captures a condition under which the state f($\overline{t}$, L, R) makes a transition to the state f($\overline{s}$, L1, R1), and the destination state f($\overline{s}$, L1, R1) is bad. This condition may be decomposed into sub-conditions on the left and right subtrees L, L1 and R, R1. Note that this is because : (i) in a tree network, the left and right subtree do not communicate directly (ii) bad(f($\overline{s}$, L1, R1)) can be decomposed into independent sub-conditions on L1 and R1. In such a situation, predicate p captures the sub-condition on the left subtree and p' captures the sub-condition on the right subtree. Similar meanings can be assigned to predicates q and q'. Our general technique to compute conditions (refer Definition 6.4) in a conditional equivalence step attempts to prove the condition:

$$\forall\ \mathtt{L}, \mathtt{L1}, \mathtt{R}, \mathtt{R1}\ \mathtt{p(L,L1)} \wedge \mathtt{q(R,R1)} \Rightarrow \mathtt{p'(L,L1)} \wedge \mathtt{q'(R,R1)}$$

Instead, the prover can exploit the knowledge of the network topology and prove the conditions: $\forall$ L, L1 p(L, L1) $\Rightarrow$ p'(L, L1) and $\forall$ R, R1 q(R, R1) $\Rightarrow$ q'(R, R1) thereby converging to a proof of bad_dest $\Rightarrow$ bad_src.

7.3 Experimental Results

This chapter presents the use of our transformation based prover in automatically establishing safety properties of small to medium sized real-life protocols. Table 4 presents a summary of the properties proved along with the time taken, the number

Protocol	Invariant	Time(secs)	# Unfolding	#Deductive
Meta-Lock	#owner + #handout < 2	129.8	1981	311
Simple-cache	#sole_owner + #valid < 2	13.97	418	146
	#shared_owner + #sole_owner < 2	13.53	402	128
	#shared_owner + #sole_owner $\neq 0$	0.81	101	21
Mesi	#m + #e < 2	3.2	325	69
	#m + #e $= 0 \vee$ #s $= 0$	2.9	308	63
Tree-cache	#c < 2	9.9	178	18

Table 4: Summary of protocol verification timings

of unfolding steps performed and the number of deductive steps performed in constructing the proof. All experiments reported here were conducted on a Sun Ultra-Enterprise workstation with two 336 MHz CPUs and 2 GB of RAM. Note that the prototype implementation of the transformation based prover implements both the algorithmic and deductive steps via meta-programming. Even though the deductive steps must be implemented by meta-programming, the proof search accomplished by the algorithmic or unfolding steps can be implemented at the level of the underlying abstract machine. Experimental evidence suggests that this will lead to reduction of the running times by *at least* an order of magnitude.

In the table, we have used the following notational shorthand: $\#s$ denotes the number of processes in local state s. *Meta-lock* denotes the Java meta-locking algorithm from Sun Microsystems [ADG+99] and *Simple-cache* is the simple synchronization based cache coherence protocol described in [KM95]. *Mesi* is a single bus broadcast protocol [EN98, EFM99], while *Tree-cache* is a binary tree network which simulates the interactions between the cache agents in a cache coherence protocol with multiple buses arranged hierarchically.

Note that the number of deductive steps in a proof is consistensly small compared to the number of unfolding steps. This is owing to our proof search strategy which applies algorithmic steps until none are applicable. Furthermore, note that the tree network example consumes larger running time with fewer unfolding and deductive steps as compared to other cache coherence protocols like the Mesi protocol. Due to

its network topology, the state representation in the tree network has a different term structure than the other protocols (where the global states are typically represented as lists). This partially accounts for the increase in the running time. In addition, certain deductive steps (such as conditional equivalence) employ more expensive search heursitics for the tree topology. Finally, the Java meta-locking algorithm represents global states as lists, but involves nested induction over both control and the data of the protocol thereby increasing the number of predicate implication proof obligations. Thus extra proof obligations are incurred due to nested induction on the infinite data domain thereby increasing the time to construct the proof.

Chapter 8

Discussions

This chapter concludes the dissertation. Section 8.1 presents a summary of the technical contributions of the work while Section 8.1 discusses some directions for future research. Concluding remarks appear in Section 8.3.

8.1 Summary of the Dissertation

This thesis investigates the problem of formally verifying parameterized concurrent systems, *i.e.* concurrent systems which are parameterized by the number of constituent processes. Parameterized systems occur widely in computing since most distributed algorithms typically describe a parameterized concurrent system. Distributed algorithms are algorithms designed to work on arbitrary number of interconnected sites and appear widely in telecommunication, process control and information processing applications. [Lyn96] discusses the pervasiveness of distributed algorithms in safety critical applications and the difficulties in formally reasoning about them.

In this dissertation we have presented a logic program transformation framework which unifies algorithmic and deductive verification techniques for formally analyzing parameterized systems. The program transformations are designed to simulate both state space exploration which is accomplished by algorithmic techniques as well as (limited) inductive reasoning. In particular, our framework is geared to automate nested induction proofs where each of the inductive subproofs proceed without strengthening of hypothesis. This gives us a lightweight automated theorem proving framework for proving temporal properties of parameterized systems.

Development of the logic program transformation based proof technique involves

development of *transformation rules* and *strategies*. The dissertation investigates both of these topics at length. First, we develop new unfold/fold transformation rules which can simulate induction proofs of temporal properties. We show how the unfolding transformation simulates a step of state space exploration, whereas folding and goal replacement transformations simulate recognition of induction hypothesis and nesting of induction proofs.

We prove the correctness of any interleaved application of the transformation rules w.r.t. the least Herbrand model semantics of definite logic programs. The transformation rules are formulated in terms of certain abstract *measures* which in general form an algebraic structure. These measures are updated after every transformation step, and decidable checks are imposed on these measures to ensure preservation of program semantics after a sequence of transformation steps. We show that by instantiating the algebraic structure associated with the abstract measures we can obtain transformation rules which subsume (in terms of transformation sequences allowed) existing program transformation systems for definite logic programs. The additional power of our transformation rules is crucial for constructing proofs of temporal properties. Furthermore, we extend our results to obtain similar transformation rules for normal logic programs. A uniform proof of correctness of our transformation rules is given w.r.t. the different semantics of normal logic programs such as well-founded model and stable model semantics.

In terms of transformation strategies, the dissertation presents a framework which tightly integrates algorithmic and deductive steps in a verification proof. Concrete proof search strategies can be obtained by instantiating this framework. The framework applies the deductive transformations (folding and goal replacement) only on demand. Thus, finite state model checking (accomplished via unfolding) emerges as a special case of the proof technique. Moreover, some of the deductive transformations (such as goal replacement) spawn nested subproofs. Thus, our strategies allows arbitrary interleaving of algorithmic and deductive steps in a proof. Note that the logic program transformation based proof technique is applicable to parameterized networks of various interconnection patterns *e.g.* chain, ring, star and tree networks. Moreover, it can even be used to verify infinite families of infinite state systems provided the state space of each instance of the family is countably infinite. The transformation rules and strategies developed in this dissertation have been implemented to yield a prover for parameterized systems. The prover has been used to automatically verify critical properties of real-life distributed algorithms including

the recently developed Java meta-locking algorithm from Sun Microsystems.

8.2 Future Work

In recent years, memoized logic program evaluation has been used to efficiently achieve local model checking of finite state systems [RRR+97, NL00]. This dissertation shows how logic program evaluation can be flexibly extended to unfold/fold transformations thereby yielding a machinery for verifying parameterized systems. Some of the avenues for future work in this direction are now outlined.

8.2.1 Verifying Nested Fixed Point Properties

This thesis presents a transformation based proof technique for proving equivalence of predicates in definite logic programs. However, temporal properties with nested fixed point operators of different kind (such as the CTL property **AGEF** φ) cannot be encoded as a definite logic program. Thus our program transformation rules and strategies need to be extended to normal logic programs. For this purpose, we have developed unfolding and folding transformation rules for normal logic programs (refer Chapter 4). However, note that our proof technique relies on constructing nested induction proofs via goal replacement. Thus we also need to develop a non-trivial goal replacement rule which is correct w.r.t. the model theoretic semantics of normal logic programs. For verifying nested non-alternating fixed point temporal properties, the corresponding logic program encoding is guaranteed to be dynamically stratified [Prz89]. Then it suffices to develop a goal replacement transformation rule for stratified logic programs which preserves the *perfect model semantics* [Prz88].

To develop such a goal replacement rule, we need to extend the technique for proving total correctness given in Chapter 4. In particular, the proofs of temporal properties may require replacement of an atom A with an atom A' where the set of positive ground derivations of A and A' are not equivalent. The notion of positive ground derivation for an atom A in program P is dependent on the stratification of P, and does not directly indicate the truth or falsehood of A in the model of P. Note that this is unlike the notion of ground proofs in definite programs; the existence of a ground proof of an atom A ensures its truth in the least Herbrand model. Since unfolding and folding are syntactic program manipulation steps, we can show that they preserve the set of all positive ground derivations. In the presence of

goal replacement, we can relax the notion of positive ground derivations to consider other proof theoretic notions (such as SLS proof trees [Prz89]) which if preserved ensure the preservation of the perfect model.

8.2.2 Verifying Parameterized Real-time Systems

Our logic program transformation based technique can be used to verify concurrent systems whose state space is countably infinite. The infiniteness of the state space may arise either from control *i.e.* the number of constituent processes is infinite, or from data *i.e.* the concurrent system has a variable with a countably infinite data domain such as integers, or from both.

Real-time systems constitute an interesting class of infinite state concurrent systems where the state space is uncountably infinite. Recently in [DRS00], memoized logic program evaluation has been extended with an external constraint solver to verify real-time systems. In general, one can model a real-time system as a constraint logic program and verify the system by query evaluation of the program [MP00]. Extending unfold/fold transformations to constraint logic programs [EG96] yields a machinery to verify parameterized networks of timed processes. Several issues need to be investigated for evaluating the efficacy of such an approach. Folding of constraint logic programs involves a constraint subsumption check which may increase the nondeterminism in the transformation search space and slow down the proof search. Unfolding of constraint logic programs involve constraint accumulation and hence may result in infinite call sequences. Widening based heuristics are needed to ensure (and accelerate) termination of the unfolding search.

8.2.3 Explaining Transformation Proof Runs

Finite state model checking provides a counterexample to the user whenever a proof attempt fails. Since proof systems for parameterized systems are incomplete, this problem is more involved. This is because a proof attempt may fail due to either the temporal property being false or the inability of the proof system to construct a proof. The problem of navigating and explaining proof attempts has been studied for interactive theorem provers [Coq99]. For our transformation based automated prover, we can *a-posteriori* provide explanation of success/failure of a proof attempt. In particular, we can provide to the user the tree of predicate equivalence proof obligations constructed. Once a node in this tree is selected (by the user), we can provide

snapshots of the transformation sequence constructed which led to the success/failure of the proof attempt of that predicate equivalence.

Recently, we have built a justifier to explain the results of memoized evaluation of queries in logic programs [RRR00]. Subsequent to the query evaluation, we post-process the memo tables thus created to extract evidence of truth/falsehood of the query. The justifier has been hooked to the XMC model checker to explain *both* success and failure of model checking runs. Note that memoized query evaluation essentially amounts to repeated unfolding with a memo based control strategy. Thus the development of the justifier is a first step in the direction of explaining transformation proof runs.

8.3 Concluding Remarks

Formal verification of parameterized systems in real life is often performed via hand proofs or via theorem proving with substantial user interaction. In the recent past, there has been a lot of research activity towards automating parameterized system verification. These include development of meaningful classes for which parameterized verification is decidable, and application of (symbolic) model checking over rich assertional languages. In this dissertation, we have taken an automated theorem proving approach. We have extended the underlying enumeration based evaluation mechanism of a model checker with limited deductive capabilities. Essentially this amounts to integrating lightweight deductive techniques with algorithmic verification.

Several interesting aspects of this integration stand out. First the proof technique thus obtained allows arbitrary interleaving of algorithmic and deductive steps in a proof. Secondly, the integration is not only *tight* but also *extensible* for verification of different flavors of concurrent systems. Our transformation based proof technique is a flexible extension of model checking via logic program evaluation as one of our transformations correspond to logic program evaluation. Furthermore, by extending the underlying programming language to constraint logic programs we can verify (families of) timed systems with the same proof technique. Finally, note that the proof technique supports *zero overhead theorem proving*, often considered as a desirable feature for integrating model checking and deduction [Seg00]. Concurrent systems which can be verified without deductive reasoning (such as finite state and data independent systems) are verified via model checking since the deductive transformations are applied lazily.

In conclusion, we remark that the work in this dissertation gives rise to many questions worth investigating. The relatively immediate or short-term extensions have been sketched in the last section. Longer term research directions include: (a) integration of automated assertion strengthening techniques into our transformation based proof technique and (b) identification of classes of concurrent systems for which the transformation based proof method serves as a decision procedure. To extend our proof technique with assertion strengthening, we need to integrate automated program analysis techniques with our transformation based proof method. To investigate the possibility of synthesizing decision procedures from our unfold/fold based proof technique we need to consider classes of infinite state concurrent systems for which model checking is decidable.

The class of context free processes stands out in this perspective. Context free processes are infinite state sequential processes which allow nonatomic prefixing *i.e.* prefixing of process constants is allowed. To perform local model checking of a context free process $p \circ t$ (where p is a process constant and $\circ$ denotes the prefix operator of CCS) we need to guess the continuation of p [HS93]. Since the number of continuations of a context free process p is potentially infinite, we need (i) a finite representation of this infinite set, and (ii) techniques to reason (in finite time) about all elements of such a finitely represented infinite set. It seems that a logic program can serve as such a finite representation and unfold/fold transformations can serve as the technique to finitely reason about this representation. Based on this intuition, one can investigate whether an unfold/fold program transformation framework can be used to build a sound and complete, local model checker for context free processes.

Bibliography

[AB86] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multi-processor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[AD95] C. Aravindan and P.M. Dung. On the correctness of unfold/fold transformations of normal and extended logic programs. *Journal of Logic Programming*, pages 295–322, 1995.

[ADG+99] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications*, 1999. Technical report available from `http://www.sun.com/research/techrep/1999/abstract-76.html`.

[AH96] R. Alur and T. A. Henzinger, editors. *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[AK86] K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *Information Processing Letters*, 15:307–309, 1986.

[B+90] A. Bundy et al. The oyster-clam system. In *International Conference on Automated Deduction (CADE), LNAI 449*, pages 647–648, 1990.

[BB93] D. Boulanger and M. Bruynooghe. Deriving unfold/fold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, pages 495–521, 1993.

[BCD90] A. Bossi, N. Cocco, and S. Dulli. A method of specializing logic programs. *ACM Transactions on Programming Languages and Systems*, pages 253–302, 1990.

[BCE96] A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. *Journal of Logic and Computation*, 6(1):79–120, February 1996.

[BCG89] M. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computation*, 81(1):13–31, 1989.

[BM90] R.S. Boyer and J.S. Moore. A Theorem Prover for a Computational Logic. In *International Conference on Automated Deduction (CADE), LNAI 449*, pages 1–15, 1990.

[BSW00] S. Basu, S.A. Smolka, and O.R. Ward. Model checking the Java metalocking algorithm. In *IEEE International Conference on the Engineering of Computer Based Systems*. IEEE Press, April 2000.

[CC92] P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *International Symposium on Programming Language Implementation and Logic Programming (PLILP), LNCS 631*, 1992.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[CGH+95] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[CGJ97] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5), 1997.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Coq99] INRIA Rocquencourt, URL : `http://pauillac.inria.fr/coq/doc/main.html`, Paris, France. *The Coq Proof Assistant : Reference Manual*, 1999.

[CW96] W. Chen and D.S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[Das92] Subrata K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.

[Del00] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *International Conference on Computer Aided Verification (CAV), LNCS 1855*, 2000.

[Dil96] D. L. Dill. The Murφ verification system. In Alur and Henzinger [AH96], pages 390–393.

[DK89] P.M. Dung and K. Kanchanasut. A fixpoint approach to declarative semantics of logic programs. *Proceeding of North American Conference on Logic Programming*, 1:604–625, 1989.

[DP99] G. Delzanno and A. Podelski. Model checking in CLP. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume LNCS 1579, pages 74–88. Springer-Verlag, 1999.

[DRS00] X. Du, C.R. Ramakrishnan, and S.A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *International Conference on Real-time Systems (RTSS)*, 2000.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *IEEE Annual Symposium on Logic in Computer Science (LICS)*, pages 352–359, 1999.

[EG96] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1):101–146, 1996.

[EGM98] S. Etalle, M. Gabrielli, and M.C. Meo. Unfold/fold transformations of CCP programs. In *International Conference on Concurrency Theory (CONCUR), LNCS 1466*, 1998.

[Eme90] E.A. Emerson. *Temporal and Modal Logic*, volume B of *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier/North-Holland, 1990.

[EN95] E. Emerson and K.S. Namjoshi. Reasoning about rings. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, pages 85–94, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automated verification of parameterized synchronous systems. In Alur and Henzinger [AH96].

[EN98] E. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite state systems. In *IEEE Annual Symposium on Logic in Computer Science (LICS)*, pages 70–80, 1998.

[FS98] Nicoletta De Francesco and Antonella Santone. A transformation system for concurrent processes. *Acta Informatica*, 35(12):1037–1073, 1998.

[GK94] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In *International Symposium on Programming Language Implementation and Logic Programming (PLILP), LNCS 844*, pages 340–354, 1994.

[GL88] M. Gelfond and V. Lifshitz. The stable model semantics for logic programming. In *International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[GRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Gru97] O. Grumberg, editor. *Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[GS91] P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 565–583. MIT Press, Cambridge, MA, 1991.

[GS92] S. German and A. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.

[GS96] S. Graf and H. Saidi. Verifying invariants using theorem proving. In Alur and Henzinger [AH96].

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HS87] J. Hsiang and M. Srivas. Automatic inductive theorem proving using Prolog. *Theoretical Computer Science*, 54:3–28, 1987.

[HS93] H. Hungar and B. Steffen. Local model checking for context-free processes. In *International Colloqium on Automata, Languages and Programming (ICALP), LNCS 700*, pages 593–605, 1993.

[ID99] C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. *Formal Methods in System Design*, 14(3), May 1999.

[KF86] T. Kanamori and H. Fujita. Formulation of Induction Formulas in Verification of Prolog Programs. In *International Conference on Automated Deduction (CADE)*, pages 281–299, 1986.

[KF87] T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. In *USA-Japan Seminar on Logics of Programs*, 1987.

[KM95] R.P. Kurshan and K. Mcmillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In Grumberg [Gru97], pages 424–435.

[KMM00] M. Kaufmann, P. Manolis, and J.S. Moore. *Computer-Aided Reasoning: An approach.* Kluwer Academic, 2000.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1994.

[KZ95] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *Journal of Computer Mathematics with Applications*, 29(2):91–114, 1995.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, pages 346–357, 1997.

[Llo93] J.W. Lloyd. *Foundations of Logic Programming, Second Edition.* Springer-Verlag, 1993.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, 1985.

[LSW96] M. Leuschel, D. De Schreye, and A. De Waal. A conceptual embedding of folding into partial deduction : Towards a maximal integration. In *Joint International Conference and Symposium on Logic Programming*, pages 319–332, 1996.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc, 1996.

[Mah87] M. J. Maher. Correctness of a logic program transformation system. Technical report, IBM T.J. Watson Research Center, 1987.

[Mah93] M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[MP00] S. Mukhopadhyay and A. Podelski. Model checking for timed logic processes. In *Computational Logic, LNCS 1861*, 2000.

[Nam98] K.S. Namjoshi. *Ameliorating the State Explosion Problem*. PhD thesis, University of Texas at Austin, 1998.

[NL00] U. Nilsson and J. Lubcke. Constraint logic programming for local and symbolic model checking. In *Computational Logic, LNCS 1861*, 2000.

[OSR92] S. Owre, N. Shankar, and J. Rushby. PVS: A Prototype Verification System. In *International Conference on Automated Deduction (CADE)*, 1992.

[PD95] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transacations on Parallel and Distributed Systems*, 6(8):773–787, 1995.

[PP98] A. Pettorossi and M. Proietti. *Transformation of logic programs*, volume 5 of *Handbook of Logic in Artificial Intelligence*, pages 697–787. Oxford University Press, 1998.

[PP99] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2–3):197–230, 1999.

[PP00] A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In *Computational Logic, LNCS 1861*, 2000.

[PPR97] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, pages 414–427, 1997.

[Prz88] T. C. Przymusinski. *On the declarative semantics of stratified deductive databases and logic programs*, pages 193–216. Foundations of deductive databases and logic programming. Morgan Kaufmann, 1988.

[Prz89] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Principles of Database Systems (PODS)*, pages 11–21, 1989.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *International Symposium on Programming, LNCS 137*, 1982.

[RKR+00] A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume LNCS 1785, pages 172–187. Springer-Verlag, 2000.

[RKRR99a] A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. In *Asian Computer Science Conference (ASIAN), LNCS 1742*, 1999.

[RKRR99b] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A parameterized unfold/fold transformation framework for definite logic programs. In *International Conference on Principles and Practice of Declarative Programming (PPDP), LNCS 1702*, pages 396–413, 1999.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In Grumberg [Gru97].

[RRR00] A. Roychoudhury, C.R. Ramakrishnan, and I.V. Ramakrishnan. Justifying proofs using memo tables. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 178–189, 2000.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liége, Belgium, July 1995. Springer-Verlag.

[San96] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, 1996.

[Seg00] Carl Seger. Combining functional programming and hardware verification. In *ACM SIGPLAN International Conference on International Conference on Functional Programming, Invited Talk*, 2000.

[Sek91] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, pages 107–139, 1991.

[Sek93] H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *Journal of Logic Programming*, pages 5–23, 1993.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 127–138, 1984.

[TS86a] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical report, Ibaraki University, Japan, 1986.

[TS86b] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, 1986.

[UR95] L. Urbina and G. Riedewald. Simulation and verification in constraint logic programming. In *2nd European Workshop on Real Time and Hybrid Systems*. Springer-Verlag, 1995.

[Urb96] L. Urbina. Analysis of hybrid systems in CLP(R). In *Constraint Programming (CP'96)*, volume LNCS 1102. Springer-Verlag, 1996.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 66–80, Grenoble, June 1989. Springer-Verlag.

[XSB00] XSB. The XSB logic programming system v2.2, 2000. Available for downloading from `http://xsb.sourceforge.net/`.