STATE REDUCTION METHODS
FOR AUTOMATIC FORMAL VERIFICATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Chung-Wah Norris Ip
December 1996
© Copyright 1996 by Chung-Wah Norris Ip
All Rights Reserved
I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

\[\text{(signed)}\]

David L. Dill  
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

\[\text{(signed)}\]

Jennifer Widom

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

\[\text{(signed)}\]

John C. Mitchell

Approved for the University Committee on Graduate Studies:

\[\text{(stamped)}\]
Dedicated to Tiffany
and my parents
Abstract

Validation of industrial designs is becoming more challenging as technology advances and demand for higher performance increases. One of the most suitable debugging aids is automatic formal verification. Unlike simulation, which tests behaviors under a specific execution, automatic formal verification tests behaviors under all possible executions of a system. Therefore, it is able to detect errors that cannot be reliably repeated using simulation.

However, automatic formal verification is limited by the state explosion problem. The number of states for practical systems is often too large to check exhaustively within the limited time and memory that is available. Existing solutions have widened the range of verifiable systems, but they are either insufficient or hard to use.

This thesis presents several techniques for reducing the number of states that are examined in automatic formal verification. These techniques have been evaluated on high-level descriptions of industrial designs, rather than gate-level descriptions of circuits, because maximum economic advantage of using verification relies on catching the most expensive errors as early as possible.

A major contribution of this thesis is the design of simple extensions to the Murϕ description language, which enable us to convert two existing abstraction strategies into fully automatic algorithms, making these strategies easy to use and safe to apply.

The algorithms rely on two facts about high-level designs: they frequently exhibit structural symmetry, and their behavior is often independent of the exact number of replicated components they contain. A static analysis of a Murϕ description can identify these characteristics, and the verification tool (or the user) can then safely change the description to include appropriate extensions. With the extensions, the
verification tool can automatically remove redundant information in the corresponding state graph, thereby decreasing the number of states necessary to represent the system.

Reductions of more than two orders of magnitude, in both time and memory requirements, have been obtained through the use of these two reduction algorithms. In the cases of two important classes of infinite systems, infinite state graphs can be automatically converted to small finite state graphs.

Another contribution is the design of a new state reduction algorithm, which relies on reversible rules (transitions that do not lose information) in a system description. This new reduction algorithm can be used simultaneously with the other two algorithms, further reducing the time and memory requirements in automatic formal verification.

These techniques, implemented in the Mur\'e verification system, have been applied to many applications, such as cache coherence protocols and distributed algorithms. With these new techniques, complex systems that used to take more than a few hundred megabytes of memory and many hours to verify can now be verified in less than one megabyte of memory and a few minutes.
Acknowledgments

This work would not have been possible without the help of many people. It is impossible to compose an exhaustive list, which would include almost everyone of my friends, the fellows in our research group, faculty members, and researchers that I have met on many occasions.

However, I must give special thanks to my research advisor, Professor David L. Dill. If not for his encouragement when I first arrived at Stanford, I would not have chosen formal verification as my research area. As a constant source of technical feedback and encouragement, he has shown me how to conduct research and to present it. He has also given me the freedom to exercise my creativity. In an ocean of so many interesting problems, he has helped me keep a consistent direction for my research.

I would also like to thank Professor Jennifer Widom for her guidance when I was working with her in the class on concurrent programming; and for her serving in my reading and oral committees. I am also grateful to Professor John C. Mitchell who kindly served in my reading and oral committees; and to Professor Zohar Manna and Professor Simon Wong, who kindly served in my oral committee.

Financial support from the following agencies is gratefully acknowledged: the “Multi-Module Systems” thrust of the Stanford Center for Integrated Systems, Mitsubishi Electronics Research Laboratories, Semiconductor Research Corporation under contract 95-DJ-389, and the Advanced Research Projects Agency through NASA grant NAG-2-891. Sun Microsystems provided computers.
Throughout my years at Stanford, many people have encouraged and helped me through many obstacles. In particular, I would like to thank Andreas Drexler for getting me up to speed with Murϕ, Elizabeth Wolf for careful readings of the manuscript on symmetry, Fong Pong for the discussion on the symbolic state model, Ganesh Gopalakrishnan, Seungjoon Park, Ulrich Stern and Han Yang for suggestions on how the work on repetition constructors should be presented, and Jens Skakkebaek and Lauren Trinh for their comments on an earlier draft of my thesis.

On the other hand, the completion of my Ph.D. program at Stanford also relies on support unrelated to the technical content of my research. My office-mate, Hugh McGuire, and many others, have put a significant effort in creating a friendly and enjoyable atmosphere at our office. My brothers and sisters in the Chinese Christian Fellowship at Stanford and the Lord’s Grace Church, and other friends have made my years at Stanford a very memorable and treasured period in my life. My parents and my wife, Tiffany, have been very supportive; and I am particularly grateful for my wife’s love, care and understanding, especially during busy periods when I have had to spend extra time working on my JAVA compiler.

Last but not least, everything that I have is given by God, and my eternal gratitude goes to Him.

C. Norris Ip
December 1996
# Contents

Abstract vii

Acknowledgments ix

1 Introduction 1

1.1 Motivation and Goal ........................................... 1
1.2 Background and Related Work .................................. 3
  1.2.1 Formal Verification ........................................ 3
  1.2.2 Existing Solutions for the State Explosion Problem ....... 4
  1.2.3 Beyond Finite State Verification .............................. 6
1.3 Summary of the Thesis ........................................... 8

2 The Mur$\varphi$ Verification System 13

2.1 Overview ............................................................. 13
  2.1.1 Description Language ........................................ 14
  2.1.2 Specification .................................................. 18
  2.1.3 State Graph and Verification Algorithm ...................... 19
2.2 Verification of Practical Systems ............................... 22
  2.2.1 Directory-Based Cache Coherence Protocols .................. 22
  2.2.2 Other Practical Systems ...................................... 28
  2.2.3 Verification Results ......................................... 29

3 Verifying Symmetric Systems 33

3.1 Symmetry ............................................................. 33
6.2.2 Restrictions for Abstract Successor Generation ....... 94
6.3 Automatic Abstraction ........................................ 99
  6.3.1 Abstract States ...................................... 99
  6.3.2 Abstract Transitions .................................. 100
  6.3.3 On-the-Fly Abstraction Algorithms ..................... 104
6.4 Heuristics ..................................................... 107
  6.4.1 Checking Maximal States ............................... 109
  6.4.2 Reducing the Number of Non-Maximal States ........... 109
6.5 Combining the Three Reduction Techniques ................. 113
6.6 Implementation and Results .................................. 117

7 Contributions and Future Work ............................... 121
  7.1 Contributions ............................................. 121
  7.2 Future Work ............................................... 123

A Semantic Analysis and Proofs .................................. 125
  A.1 Semantics of the Core Murϕ Language ..................... 125
      A.1.1 Abstract Transition Programs .................... 127
      A.1.2 Formal Semantics .................................. 127
  A.2 Automorphism Induced by Scalarsets ..................... 132
  A.3 Data Saturation Induced by Data Scalarsets ............. 139
  A.4 Repetitive Property Enforced by RepetitiveIDs .......... 147

B Murϕ Descriptions ............................................. 157

Bibliography ...................................................... 177
# List of Tables

2.1 Verification results using the basic algorithm, I .......................... 32  
2.2 Verification results using the basic algorithm, II .......................... 32  
3.1 Performance of the graph isomorphism heuristics .......................... 48  
3.2 Performance of the normalization algorithm .............................. 50  
3.3 Improvement in performance with scalarset ............................... 51  
3.4 Inefficiency in handling symmetry in a message channel ............... 52  
3.5 Improvement in performance with multiset .............................. 53  
4.1 Data saturation ............................................................................. 62  
5.1 Comparison of the two reduction algorithms using reversible rules .......................... 77  
5.2 Improvement on performance with singular reversible rules .......... 79  
5.3 Improvement on performance with both symmetry and reversible rules .... 80  
5.4 Comparison of partial order and reversible rules, I ....................... 83  
5.5 Comparison of partial order and reversible rules, II ..................... 84  
6.1 Performance of the special hashing scheme .................................. 110  
6.2 Performance of the priority search strategies ............................. 112  
6.3 Improvement in performance with repetitiveID .......................... 119  
6.4 Monitoring the progress towards a saturated model .................... 119  

xiv
List of Figures

2.1 A Murφ description of a simple mutual exclusion protocol .......... 15
2.2 A simple on-the-fly algorithm used in Murφ .......................... 21
2.3 The DASH multiprocessor ............................................. 23
2.4 Examples of cache coherence transactions .............................. 26
2.5 An counter-example in the DASH protocol ............................ 31

3.1 Symmetry in a cache coherence protocol ............................. 34
3.2 An example on the use of scalarset .................................. 39
3.3 Permutation on state variables ........................................ 42
3.4 An on-the-fly symmetry reduction algorithm ......................... 44

4.1 An example of data independence and data saturation ............... 59

5.1 Reversible rules in a cache coherence protocol ...................... 64
5.2 State explosion due to reversible rules .............................. 65
5.3 An on-the-fly reduction algorithm for reversible rules .............. 70
5.4 Potential false error report from reduction using reversible rules .. 71
5.5 Singular property in a cache coherence protocol .................... 74
5.6 An on-the-fly reduction algorithm for singular reversible rules ..... 76

6.1 Repetitive property in a cache coherence protocol .................. 89
6.2 Manual translation to use repetition constructors ................... 91
6.3 Automatic generation of abstract states ............................. 95
6.4 Automatic generation of abstract successors ........................ 103
6.5 A simple on-the-fly abstraction algorithm .......................... 105
6.6 An efficient on-the-fly abstraction algorithm ............................. 108
6.7 Hashing structure for efficient comparison of abstract states ......... 110
A.1 Syntax for the Simple Description Language ............................... 126
Chapter 1

Introduction

1.1 Motivation and Goal

Industrial designs are becoming more complicated as technology advances and demand for higher performance increases. Growing complexity causes an increasingly serious challenge of avoiding design errors: even when a designer exercises the utmost care, the mere scale of complexity makes it likely that he or she will fail to anticipate some possible interaction between different components of the system. Furthermore, these errors may exhibit nondeterministic behaviors, and therefore, will not be reliably repeatable, making testing and debugging using simulation difficult. Without effective debugging aids, errors in the designs of these systems may go undetected for a long time.

Locating and correcting design errors may be a time consuming and expensive process. Indeed, a significant portion of development time for a new product is typically spent in locating and correcting errors. The later an error is detected, the harder and the more expensive it is to correct. If ever an error slips into a product delivered to a customer, the consequence can be major financial loss, as was the result of the Pentium floating point division bug [Hof95].
CHAPTER 1. INTRODUCTION

The challenge is therefore to provide an effective debugging aid. Formal methods, with emphasis on formal verification, may be used for this purpose. In particular, an apt candidate for such a debugging aid is automatic formal verification [ZWR+80, BWHM86, Hol87]. In contrast to the manual methods where major human involvement is needed, automatic formal verification methods are easy to use and require little effort. When used properly, they can catch design errors quickly, and provide diagnostic messages in the form of counter-examples.

Automatic formal verification has three components: a specification of the correctness properties to be verified, a formal model of a system, and an automatic procedure to determine if the model correctly preserves the correctness properties. In order to maximize the benefits, high-level descriptions of the designs, rather than gate-level descriptions of circuits, should be used. The high-level models used early in the design process are simpler than the implementation in the later stages, making verification computationally feasible, and the most expensive errors are caught as early as possible, achieving maximum economic advantage.

The major problem with automatic formal verification is that a large amount of memory and time is often required, because the underlying algorithm in these methods usually involves systematic examination of all reachable states of the system to be verified. As the number of reachable states increases rapidly with the size of a system, the basic algorithm, by itself, becomes impractical: the number of states for a system is often too large to check exhaustively within the limited time and memory that is available. This phenomenon is known as the state explosion problem.

Therefore, the goal of this work is to ameliorate the state explosion problem for verification of high-level descriptions of practical systems:

- The number of states to be checked should be significantly reduced.
- The new algorithm should use as little amount of time and memory as possible.
- The new algorithm should be easy to use and highly automatic.
1.2 Background and Related Work

1.2.1 Formal Verification

Formal verification can be traced back to Turing [Tur49], Floyd [Flo67], and Hoare [Hoa69]. They have verified simple deterministic sequential programs, through a manual construction of assertional proofs, without computer assistance. The assertional proof consists of a set of assertions, which specify properties at various locations of the program, and whose correctness is guaranteed by a set of proof rules.

These methods were then applied to concurrent and distributed programs, and extended to verify temporal behaviors. The analysis of concurrent and distributed programs follows a similar framework as the method for sequential programs [OG76a, OG76b, Lam77, AFdR80, LG81]. Temporal logic was introduced as a specification formalism in formal verification by Pnueli [Pnu77] and further developed by Manna and Pnueli [MP81, MP83, MP84], and Owicki and Lamport [OL82]. Instead of using simple assertions as the specification to be verified on a program, the new method used temporal logic to describe dynamic behaviors, that is, the effect of the program over time.

As the complexity of a program increases, the construction of the assertional proofs rapidly becomes tedious and error-prone. Therefore, automated theorem provers were developed to handle most of the tedious application of proof rules, and to make sure that each step in a proof is correct [WOLB84] (for example, HOL [GM93, Gor85], the Boyer-Moore theorem prover ‘nqthm’ [BM79] and PVS [ORS92, ORSvH95]). These techniques provide effective means to guarantee the correctness of safety-critical systems, in which the consequences of an error outweighs the time and the financial investment required to construct the proof.

However, for designs that are not safety critical, the assertional proof and automated theorem proving techniques are too expensive. Furthermore, if a system contains an error, it is impossible to construct a proof, and often, no useful diagnostic information is generated.

On the other hand, automatic formal verification methods, although specialized for finite-state systems, are easy to use, and they provide useful diagnostic messages
if the system fails to observe the required properties. The basic idea in automatic formal verification methods is that of state enumeration. Zafiropulo et al. [ZWR+80] originally proposed the state enumeration method for protocol verification. The states of a finite-state system can be systematically explored by a simple search algorithm to check whether bad states can be reached from the initial states. For example, a bad state can be any state that does not satisfy a user-provided predicate. If a bad state is reached, a path from an initial state to the bad state provides a useful diagnostic message to help the designer to correct the error. Many successful applications of this approach have been reported [BWHM86, Hol87].

Model Checking, based on the state-space exploration technique, is designed to verify temporal properties of a system. Proposed by Queille and Sifakis [QS81], and Clarke and Emerson [CE81], model checking uses the reachability state graph as a Kripke structure, which encodes the set of all possible sequences of states for a system over computation trees.

Tools using state-space exploration and model checking are effective as debugging aids for industrial designs. Examples include SPIN [HP96], COSPAN [HHK96], SMV [CMCHG96], and Murϕ [DDHY92, Dill96]. Because they are fully automatic, minimal user effort and user knowledge about formal verification are required to use these debugging aids.

### 1.2.2 Existing Solutions for the State Explosion Problem

The efficiency of state exploration and model checking methods depends heavily on the size of the reachability state graph. The larger the reachability state graph, the more time and memory it takes to verify a system. The biggest obstacle of these methods is the often unmanageably huge number of reachable states: the state explosion problem.

However, this problem is a PSPACE-hard problem, and there is no universal solution. In order to reduce the state explosion problem, several heuristics and solutions for some narrow classes of problems have been proposed. The most powerful frameworks are abstraction [BBL92, BBG+93, GL93a, GL93b, CGL91, Lon93] and abstract
interpretation [CC77, Cou81, Cou90, CC92, DGG94]. These frameworks rely on the user to provide an appropriate abstraction to remove irrelevant information from a state space, thereby decreasing the number of states necessary to represent a system. An abstraction mapping from an original state to an abstract state is required in both cases, and an abstract interpretation of every operation in the system is required in the abstract interpretation framework. If the abstraction is appropriate, the smaller abstract state graph can be used to verify the properties of the system.

There are also fully automatic solutions, such as symbolic model checking, minimal state partitioning, and partial order methods. Symbolic model checking methods [BCM+90, CBM89, TSL+90] were able to verify a wider range of complex systems. By representing a large set of states in a compact Binary Decision Diagram (BDD) [Bry86], a large state space may be stored in a relatively small piece of memory. Operations on a BDD can execute transitions on a large set of states at the same time. However, efficient use of BDDs often depends on some subtle properties of the systems to be verified. In particular, high-level descriptions of systems represent an application domain that has been particularly difficult for BDD-based methods. In fact, for many high-level descriptions, traditional explicit state enumeration outperforms the methods using BDDs [HD93a, HD93b, Hu95].

Minimal state partitioning [BFH90, BFH+92, Fer93, LY92, ACH+92] takes advantage of the bisimulation equivalence relation in the state graph. A minimal partitioning of the set of possible states (both reachable and unreachable) is generated and used to verify the system. However, its efficiency depends on how well sets of states can be manipulated efficiently, and therefore, faces similar problems with BDD-based methods. Furthermore, the examination of the unreachable states often present other difficulties in applying this method (c.f.[LY92]).

Partial order reduction [Val90, Val91, Val93, God90, GW93, GW94, God95, HP94, Pei94, Pei96] is a fully automatic method, taking advantage of independent transitions in an interleaving model of a system. It has aroused significant interest because of its theoretical intricacy and the good reductions obtained in many systems. However, for many practical systems, the reduction obtained by only partial order reduction is still not sufficient.
Some fully automatic methods are heuristics that do not reduce the number of states examined, but reduce the amount of memory to examine the same number of reachable states. State space caching methods [Hol85, JJ91] use a fixed amount of memory, but they may re-visit the same states many times, thereby increasing the verification time significantly. Supertrace [Hol91a] and hash compaction [WL93, SD95a] methods are partial search techniques that use a small approximated signature for each state, and allow a very small probability of omitting some reachable states. Therefore, these methods may miss errors in a system.

In general, the existing solutions are not sufficient to solve the state explosion problem in practice. While the abstraction and abstract interpretation frameworks are powerful enough to verify a lot of designs, the effort involved is often not economically feasible for industrial uses. Automatic methods using BDDs, partitioning, or partial order are still insufficient, especially for verification of high level descriptions of designs. Heuristics, such as state space caching, supertrace, and hash compaction, may increase verification time or overlook errors; therefore, they should be avoided if better solutions exist.

1.2.3 Beyond Finite State Verification

Although state exploration and model checking methods rely on the fact that the state space is finite, it is often desirable, and sometimes possible, to apply them to the verification of infinite systems.

The problem of verifying an infinite system typically arises during the development of a scalable system. Although most systems are finite-state in nature, they are often designed to be scalable, so that a description gives a family of systems, each member of which has a different size. Typical examples include cache coherence protocols, communication protocols, or hardware controllers. Such systems may have a different data domain (such as the number of bits in a cache line), or a different number of replicated, identical components (such as the number of processors, address line, peripherals or entries in a buffer).
In these cases, it is desirable to verify the entire family of systems, independent of the actual sizes. This is typically achieved by an appropriate abstraction to convert the infinite state space to a finite state space.

**Arbitrary Data Domain**

The class of data-independent protocols with an arbitrary data domain can be abstracted into finite systems, as proposed by Wolper [Wol86]. Because the control flow of a data-independent protocol does not depend on the exact values of the data, a temporal statement involving an infinite data domain can be converted into one with a finite set of data. However, Wolper requires the user to recognize that a protocol is data-independent, and to manually transform the description in order to exploit the data-independence.

Aggarwal, Kurshan and Sabnani also briefly mentioned the application of similar idea to verify an alternative bit protocol [AKS83].

**Arbitrary Number of Components**

The general problem of verifying systems with an arbitrary number of replicated components is known to be undecidable [AK86, GS92]. However, several approaches have been proposed for specific instances of the problem. Some of them use induction over the replicated components and require an invariant process or a network invariant [KMOS94, CG87, CGJ95, WL89]. Coming up with a proper invariant is not easy. Although automatic generation of network invariants for certain classes of systems have been explored, they are very expensive and only apply to a very narrow range of designs [RS93, BSV94, SG87, GS92].

There are also approaches that do not use induction. Shibata et al. [SHTO93] presented an algorithm to verify a simple telecommunication system with limited interaction between the processes. However, the class of problems they can verify is severely restricted. On the other hand, Graf [Gra94] has a more general method based on abstraction, which was applied to a simple distributed cache memory, but it requires substantial manual effort to complete the proof.
CHAPTER 1. INTRODUCTION

One of the most successful approaches is to use an abstraction that ignores the exact number of components. For example, Lubachevsky [Lub84] verified a concurrent program by collapsing all reachable states into a fixed number of 'metastates', in which the number of processes is represented by $N$ with an unspecified value. Dijkstra [Dij85] verified a ring network by representing classes of similar states in regular expressions. Clarke and Grumberg [CG87] verified an alternating-bit protocol by constructing an invariant process that records only the existence of any component in a certain state. Pong et al. [PD95b, PNAD95, Pon95] verified many cache coherence protocols by representing classes of similar states in a set of repetition constructors, recording only whether there are zero, exactly one, one-or-more, or zero-or-more components in a certain state.

While these approaches have successfully verified many designs, there are still many issues to be addressed. First of all, using these methods requires significant effort from the users, because the user has to provide the abstraction mapping, the abstract transition relations, or even the full abstract model. Secondly, it is difficult to determine when the abstraction is appropriate. In some cases, the user may not realize that the abstraction would be useful, and in other cases, the user may use the abstraction inappropriately, and generate incorrect verification results. Furthermore, the abstraction mapping provided by the user may not be the most efficient way to generate the abstract state space; heuristics are useful in many cases, but the user may find it error-prone to incorporate the heuristics manually into the abstraction mapping.

1.3 Summary of the Thesis

This thesis presents several techniques for reducing the number of states that are examined in automatic formal verification. While these techniques are easy to use and highly automatic, they generate significant reductions in both time and memory requirements, and guarantee the correctness of the results.
1.3. SUMMARY OF THE THESIS

Detecting Commonly-found Characteristics

A major contribution of this thesis is the design of simple extensions to the Murϕ description language, which enables us to convert two existing abstraction strategies into fully automatic reduction algorithms, and therefore, make these strategies easy to use and safe to apply.

The language extensions were designed so that whenever a system can be described using the extensions, the results of verification using the reduction algorithms are guaranteed to be correct. For example, a new datatype, called Scalarset, is defined in Chapter 3 to replace an integer subrange that is involved in a restricted set of operations, and its restrictions guarantee certain symmetries to hold on the state graph. Scalarset is easy to use: if a description has a subrange involved only in these restricted operations, the verification tool or the user may convert this subrange into a scalarset. In this case, it is safe to exploit these symmetries in the verification algorithm. On the other hand, if a description has a scalarset with operations outside these restricted operations, the verifier reports an error. In this case, the user should either change the scalarset to a subrange, or change the offending operation to a legal scalarset operation.

Exploiting Commonly-found Characteristics

Three fully automatic state space reduction methods are described in this thesis, two of which are based on existing abstraction strategies [Lub84, AKS83, HJJ84, Sta91, CG87, PD95b, PNAD95]. They rely on two characteristics commonly found in high-level descriptions of designs:

a) **Structural Symmetry:** A lot of designs are symmetric systems. Their behavior does not depend on a particular ordering of the components, or the exact value of the data.

b) **Repetitive Property:** A lot of designs have a set of replicated components. Their behavior does not depend on the exact number of the components.
The main contribution in these two methods is the technique for detecting these properties and the fully automatic algorithms that do not require manual translation of a conventional description into an abstract description.

The third reduction method exploits reversible rules in a system description:

c) **Reversible Rules:** A lot of designs contain transitions that are reversible: there is no information lost during the execution of these transitions, and the original state can be automatically reconstructed from the next state.

This method was developed from scratch. The main contribution is to identify the appropriate properties and to design the new algorithm to take advantage of these properties in automatic formal verification.

With the language extensions and the reduction algorithms, verification can be performed in as little amount of memory and time as possible, with little effort from the users. Given a conventional description of a system with the language extensions, a verifier can automatically verify the system using a reduced state space, using one or more of the three reduction algorithms. Appropriate heuristics are automatically constructed and adapted to the structure of the system.

**Performance of the Reduction Algorithms**

The reduction methods in this thesis can be used to verify a wide range of properties. The reduction using symmetry is sound and complete for LTL and CTL* model checking and deadlock detection. The reduction using reversible rules is sound and complete for stuttering-invariant LTL model checking and deadlock detection (sound but not complete for stuttering invariant ∀CTL* model checking). The reduction using the repetitive property is sound for ∀CTL* model checking; however, because it is an approximation, it may produce false error reports.

The reduction methods are compatible: they can be used at the same time to achieve even greater savings. Future research is needed to ascertain how well these methods can be combined with other reduction methods.

These techniques, implemented in the Murϕ verification system, have been applied to many applications, such as cache coherence protocols and distributed algorithms.
With these new techniques, complex systems that used to take more than a few hundred megabytes of memory and many hours to verify can now be verified in less than one megabyte of memory and a few minutes.

Furthermore, in the cases of two important classes of infinite systems, infinite state graphs can be converted into finite state graphs automatically. Reduction using symmetry can be used to verify data-independent systems with an arbitrary data domain, and reduction using the repetitive property can be used to verify systems with an arbitrary number of replicated components.
Chapter 2

The Murφ Verification System

Chapter Overview

The reduction methods described in this thesis have been implemented in the Murφ verification systems [DDHY92, Dil96]. This chapter summarizes Murφ (without these reductions), and describes several designs that have been verified using Murφ.

2.1 Overview

The Murφ verification system was designed for verification of asynchronous high-level systems. A number of practical problems, such as protocols, synchronization algorithms, and memory consistency models, have been verified using Murφ, by researchers and designers in universities and within industry [PNAD95, YGM+95, DDHY92, SD95b, DPN93, PD95a, Par94, WG94, CRL96].

The Murφ verification system consists of two components: the Murφ description language and the Murφ compiler. The language describes an asynchronous model of the system to be verified, and the compiler compiles the model into a special purpose verifier. This special purpose verifier uses an explicit state enumeration algorithm to check the properties of the system, such as error assertions, invariants, and deadlocks. If the system fails to observe these properties, a counter-example is generated by the verifier.
The Mur\(\varphi\) description language was designed to be the simplest high level language that supports nondeterministic, scalable descriptions. Mur\(\varphi\) meets these particular goals (especially simplicity) better than many hardware and protocol description languages [Bri86, BD87, Ora88, Hol91b, LSU89, TM91]. The Mur\(\varphi\) language is inspired by the Chandy and Misra’s Unity language [CM88]. A Mur\(\varphi\) description of a system is typically at a high level of abstraction. Most features in Mur\(\varphi\) can be found in common programming languages, such as Pascal or C, and high-level data structures, such as arrays and records, are supported. Non-determinism in a system is described in Mur\(\varphi\) by a set of guarded commands, introduced by Dijkstra [Dij76]. The language is also designed to be scalable, meaning that it is easy to change a description to model a larger or smaller system.

The Mur\(\varphi\) compiler generates a C++ program from a description in the Mur\(\varphi\) language. The C++ program is then compiled into a special purpose verifier. The optimizations performed by the C++ compiler improve the efficiency in the execution of the arbitrarily-complex atomic transition rules in the description.

In order to maximize the benefits of verification, Mur\(\varphi\) is designed to be used as a debugging tool in the early design stages. The high-level description used early in the design process is simpler than the implementation in the later stages, making verification computationally feasible, and the most expensive errors are caught as early as possible, achieving maximum economic advantage.

### 2.1.1 Description Language

A Mur\(\varphi\) description consists of constant and type declarations, variable declarations, invariants declarations, and rule declarations. As an example, the Mur\(\varphi\) description of a simple mutual exclusion protocol is shown in Figure 2.1. From this description, Mur\(\varphi\) searches for errors in the corresponding state graph: a state is an assignment of values to the global variables; the initial states are constructed by executing the the startstates; and the states reachable from the initial states are generated by recursively apply the transition rules to the initial states. The verifier checks that the invariants are true in every reachable state.
\begin{verbatim}
const NumProcesses : 2;

type Pid : 1 .. NumProcesses ;

var P : array[Pid] of enum { Critical, NonCritical };

startstate for i : Pid do P[i] := NonCritical; end

ruleset i : Pid do
    rule “Entering Critical Section”
        forall j : Pid do P[j] = NonCritical end \Rightarrow P[i] := Critical; end
    rule “Leaving Critical Section”
        P[i] = Critical \Rightarrow P[i] := NonCritical; end
end

invariant “Mutual Exclusion”
    forall i : Pid; j : Pid do i \neq j \rightarrow (P[i] \neq Critical \lor P[j] \neq Critical) end

the reachability state graph

\begin{tikzpicture}[level distance=1.5cm,
    level 1/.style={sibling distance=3.5cm}]
    \node {0,0}
    child {node {0,1}
        child {node {0,0}}
        child {node {1,0}}
    }
    child {node {1,0}};
\end{tikzpicture}

0 : NonCritical
1 : Critical
\end{verbatim}

Figure 2.1: A Murϕ description of a simple mutual exclusion protocol
Constant and type declarations:

- Integer constants can be declared at the beginning of the description. In the mutual exclusion example, the number of processes is declared to be 2.
- The basic types are Booleans, subranges, enumerations, arrays and records. In the mutual exclusion example, a subrange is used to describe the process-indices, and an enumeration is used to describe the internal state of a process.Booleans, subranges and enumerations in Murϕ are extended to include a special undefined value \( \perp \). A special function \textbf{IsUndefined} can be used to check if a variable has the value \( \perp \). Otherwise, access to a variable with the undefined value causes the verifier to report an error.

Variable declarations:

- The global variables declared are used to describe the states of a system. In the mutual exclusion example, the global variable is an array of two elements, each corresponding to a process in the non-critical section or the critical section.

Invariants:

- Invariants are Boolean expressions that reference the global variables. In the mutual exclusion example, the invariant states that the two processes cannot be in the critical section at the same time.

Rule declarations:

- Murϕ describes the transitions of a system using a set of rules. Each rule is a guarded command, consisting of a condition and an action. In the mutual exclusion example, the following rule is used to describe the transition that a process exits a critical section:

  \[
  \textbf{rule} \quad \text{"Leaving Critical Section"} \\
  P[i] = \text{Critical} \Rightarrow P[i] := \text{NonCritical}; \quad \textbf{end}
  \]
The condition in this rule is $P[i] = \text{Critical}$, which makes sure that the process $i$ is originally in the critical section. The action in this rule is $P[i] := \text{NonCritical}$, which changes the internal state of the process $i$ from critical to noncritical.

In general, a condition is a Boolean expression consisting of constants, global variables, arithmetic operators and Boolean operators. The action is a sequence of statements. The action can also declare local variables before the statements.

- Mur$^2$ describes the start states of a system using special rules called startstate, in which the Boolean condition is by default true. In the mutual exclusion example, the startstate generates an initial state in which both processes are in the non-critical section.

- A collection of similar rules can be instantiated using a ruleset over a finite range. In the mutual exclusion example, the ruleset instantiates two set of transition rules, one for each of the processes.

  A ruleset can be used to model a family of similar rules, each for a different component of the system to be verified, or to model a non-deterministic assignment of values to a local variable in a particular rule.

- Boolean operators include conventional Boolean operators such as negation, conjunction, and disjunction. There are also existential and universal quantifiers over a finite range. An example of a universal quantifier can be found in the condition of the first transition rule, which states that both processes are in the non-critical section.

  Other operators include conventional comparisons (such as equality testing and greater-than testing) and conventional integer arithmetic operators (such as addition and multiplication).

- Statements have sequential semantics: assignments take place in the environment that has been modified by all previous assignments in the same rule.
CHAPTER 2. THE MURφ VERIFICATION SYSTEM

The usual assignment, if statement and switch statement are part of Murφ. There is a restricted for statement that must have compile-time constant loop bounds, and a restricted while statement that reports an error if the number of iterations exceeds a bound specified by the user.

A special undefined value Undefined can be used to assign ⊥ to any basic type. An alias statement introduces an identifier which abbreviates an expression. Procedures and functions in Murφ are essentially "macros" with parameter type-checking.

The Murφ language is well-suited for an asynchronous, interleaving model of concurrency, where atomic steps of individual processes are assumed to happen in sequence, and one process can perform any number of steps between the steps of another process. When two steps are truly concurrent, there will be executions that allow them to happen in either order in the interleaving model. In Murφ, concurrent composition is very easy: to model two processes in parallel, just form a new description using the union of their rules. Coordination among the processes is by shared variables; to model a message passing system, synchronization and buffering must be handled explicitly.

The Murφ language was designed so that the size of a system can be changed by changing a single parameter. A Murφ description of a system with $n$ processes can be written with a declared constant (for example, NumProcesses) of value $n$. Then a subrange Processes : 1..NumProcesses can be declared, and the states of the processes can be stored in an array indexed by Processes. The rules for each process can be instantiated by a ruleset, and Boolean expressions can be written using exists and forall quantifiers. Therefore, a description written in this style can be scaled by changing only the constant declarations.

2.1.2 Specification

Murφ has several features for specifying design errors. Firstly, it automatically detects deadlocks, which are defined to be states with no successors other than themselves.
Secondly, an **Error** statement can appear in the body of a rule (almost always embedded in a conditional). Executing an **Error** statement prints a user-supplied error message and an error trace. This feature is especially useful when some branches of an **If** or **Switch** statements are not intended to be reachable. There is also an **Assert** statement, which is an abbreviation for a conditional error statement. Finally, the user can define *invariants* in a separate part of the Murϕ description. An invariant is a Boolean expression that is desired to be true in every state. When an invariant is violated, a user-supplied error message and an error trace are generated.

Although these specification facilities are limited, many real errors have been detected using these features only. Furthermore, history dependent properties can be checked by adding explicit state variables and statements to monitor the state associated with the property. An extension to liveness and fairness properties was implemented in an earlier release of Murϕ, to check sequential behaviors such as livelocks and starvations. However, the latest release (Version 3.0S) does not contain this extension.

### 2.1.3 State Graph and Verification Algorithm

The verification algorithm in Murϕ explores the state graph described by a Murϕ description, which encodes all possible executions of the system. An *execution* of the system is a finite or infinite sequence of states $q_0, q_1, \ldots$, where $q_0$ is a start state of the description. If $q_i$ is any state in the sequence, $q_{i+1}$ can be obtained by applying some rule whose condition is true in $q_i$ and whose action transforms $q_i$ to $q_{i+1}$. In general, $q_i$ can satisfy the conditions in several rules, so there is more than one execution (nondeterminism). A simulator for Murϕ might choose the rule randomly; a verification tool must cover all the possibilities.

In order to define the verification algorithm in Murϕ more precisely, we need to define some fundamental concepts.

**Definition 2.1 (state graph)** A state graph is a quadruple $A = (Q, Q_0, \Delta, \text{error})$, where $Q$ is a set of states, $Q_0 \subseteq Q$ is a set of initial states, $\text{error} \in Q$ is a unique error state, and $\Delta \subseteq Q \times Q$ is a transition relation with the property that $q = \text{error}$ whenever $(\text{error}, q) \in \Delta$. 


The special error state, \textbf{error}, is used as the next state whenever an invariant is violated or an error statement is executed.

In Mur$\varphi$, a state graph is defined implicitly by a set of \textit{transition rules}, $T$, where each rule maps a state to a successor state. Formally, for all $q_1, q_2 \in Q$, we have $(q_1, q_2) \in \Delta$ if and only if there exists $t \in T$ such that $q_2 = t(q_1)$.

\textbf{Definition 2.2 (successor/predecessor)} If $(q, q') \in \Delta$, then $q'$ is a successor of $q$, and $q$ is a predecessor of $q'$.

\textbf{Definition 2.3 (path)} A finite sequence of states $q_0, \ldots, q_n$ is called a path if $q_0 \in Q_0$ and $q_i$ is a successor of $q_{i-1}$ for all $1 \leq i \leq n$.

\textbf{Definition 2.4 (reachability)} A state $q$ is reachable if there exists a path $q_0, \ldots, q_n$.

\textbf{Definition 2.5 (deadlock state)} A state is a deadlock state if it has no successors other than itself.

We usually denote $(q_1, q_2) \in \Delta$ as $q_1 \rightarrow q_2$, denote $q_2 = t(q_1)$ as $q_1 \xrightarrow{t} q_2$, and denote $q_n = t_n(...t_1(q_0)...)$ as $q_0 \xrightarrow{t_1 \cdots t_n} q_n$.

The algorithm in Mur$\varphi$, as shown in Figure 2.2, checks whether \textbf{error} or a deadlock state is reachable. It is an \textit{on-the-fly} algorithm, which generates and explores new states only when all previous states are known to be error-free. The states are stored in a hash table, so that it can be decided efficiently whether or not a newly-reached state is old (has been examined already) or new (has not been examined already). New states are stored in a queue of \textit{active states} (whose successors still need to be generated). Depending on the organization of this queue, the verifier does a breadth-first search or a depth-first search. Except for the initial states, every state in the hash table has a pointer to a predecessor. Therefore, if a problem is detected during the search, an error trace can be generated. Breadth-first search is used by default, because it produces a shortest error trace for an error.
Simple Algorithm()

begin

\( \text{Reached} = \text{Unexpanded} = \{ q \mid q \in \text{StartState} \} \);

while \( \text{Unexpanded} \neq \emptyset \) do

Remove a state \( q \) from \( \text{Unexpanded} \);

\( \text{Deadlock} = \text{true} \);

for each transition rule \( t \in T \) do

if an error statement is executed in \( t \) on \( q \) then report error; endif;

let \( q' = t(q) \) in

if \( q' \) does not satisfy one of the invariants then report error; endif;

if \( q' \neq q \) then \( \text{Deadlock} = \text{false} \); endif

if \( q' \) is not in \( \text{Reached} \) then put \( q' \) in \( \text{Reached} \) and \( \text{Unexpanded} \); endif;

endlet;

endfor;

if \( \text{Deadlock} \) then report deadlock; endif

endwhile;

end
2.2 Verification of Practical Systems

2.2.1 Directory-Based Cache Coherence Protocols

Two cache coherence protocols are used throughout this thesis for discussions and evaluations on the performance of the new verification algorithms:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DASH-C</td>
<td>The directory-based cache coherence protocol for the DASH multiprocessor [LLG+90].</td>
</tr>
<tr>
<td>ICCP</td>
<td>An industrial directory-based cache coherence protocol described in [DDHY92] (an early version of the cache coherence protocol in Sun S3mp systems).</td>
</tr>
</tbody>
</table>

The concept of directory-based cache coherence was first proposed by Tang [Tan76]. For each memory block, a directory stores the identities of all remote nodes caching that block. Instead of broadcasting to every processing node, the node with the physical memory location can send point-to-point invalidation or update messages to those nodes that are actually caching that block.

These two protocols are used as the main examples in this thesis because they reflect the typical complexity and level of abstraction that the Murφ verifier was designed for. Unlike so-called snooping protocols that are based on broadcast of messages on a bus, directory-based cache coherence protocols do not have a single serialization point for all memory transactions. While this feature is responsible for their scalability, it also makes them more complex.

The remainder of this section summarizes these two protocols and describes how they are modeled in Murφ, so that they can be used to illustrate the reduction methods described in subsequent chapters. The complete Murφ descriptions for the DASH cache coherence protocol can be found in Appendix B. The verification results without using any reduction techniques are also presented.
Figure 2.3: The DASH multiprocessor: The processors are arranged into clusters, which communicate with one another through an interconnection network.

The DASH cache coherence protocol (DASH-C)

DASH is a scalable shared-memory multiprocessor developed at Stanford University. It consists of a collection of processing nodes, communicating through a scalable interconnection network, as shown in Figure 2.3. A key feature of DASH is its directory-based cache coherence protocol.

Each processing node contains several processors with individual caches, in which up-to-date data are cached for fast access from the local processors. Hence, multiple copies of the same data may appear in various places, and a cache coherence protocol is needed to guarantee consistency in a particular memory model.

The DASH protocol relies on point-to-point communication through two separate networks for requests and replies. The aim of separating requests and replies into two networks is to avoid deadlock in the protocol, but it also introduces complex interactions that may arise because of messages received out of order.

This architecture is modeled in Murϕ as two arrays for the processing nodes and the physical memory distributed among the nodes, and two buffers for the communication networks. In order to reduce the complexity of the verification problem, some of the processing nodes are modeled without physical memory, as shown in the following code (some details are omitted):
**CHAPTER 2. THE MURφ VERIFICATION SYSTEM**

```
CONST
HomeCount: 1;  -- number of nodes with memory
RemoteCount: 3;  -- number of nodes without memory
NodeCount: HomeCount + RemoteCount;  -- number of nodes
ChanMax: 2 * ProcCount * HomeCount;  -- buffer size in a single channel

TYPE
Node : 1 .. NodeCount;  -- an integer subrange from 1 to NodeCount
Home : 1 .. HomeCount;  -- an integer subrange from 1 to HomeCount
NodeState : Record ...;
HomeState : Record ...;
Request : Record ...;
Reply : Record ...;  -- details omitted

VAR
Nodes : Array [ Node ] Of ProcState;  -- the nodes
Homes : Array [ Home ] Of HomeState;  -- the memory blocks
ReqNet: Array [ Node ] of Array [ Node ] of Record -- the request network
    Count: 0 .. ChanMax;
    Messages: Array [ 0 .. ChanMax-1 ] of Request;
End;
ReplyNet: Array [ Node ] of Array [ Node ] of Record -- the reply network
    Count: 0 .. ChanMax;
    Messages: Array [ 0 .. ChanMax-1 ] of Reply;
End;

In the physical memory, each memory block can be in one of three states, as indicated by the associated directory entry: (i) uncached-remote, that is, not cached by any remote node; (ii) shared-remote, that is, cached in an unmodified state by one or more remote nodes; or (iii) dirty-remote, that is, cached in a modified state by a single remote node. The directory does not maintain information concerning the processors at the same node. The directory is modeled as:

```
HomeState:
    Record
        Mem: Array [ Address ] of Value;
        Dir: Array [ Address ] of
            Record
                State: enum { Uncached, Shared_Remote, Dirty_Remote };
                SharedCount: 0 .. DirMax;
                Entries: Array [ 0 .. DirMax-1 ] of Node;
            End;
    End;

In each processing node, the cache coherence among the local processors is maintained by a snooping protocol on a local bus. Since the snooping protocol ensures
consistency within the node, each node actually behaves like a single processor with a single cache.

Each processor contains a memory cache and a Remote Access Cache (RAC). A cache block in the memory cache may be in one of three states: invalid, shared, or dirty. The invalid state implies the data is not present in the cache line; the shared state implies the data is valid, but it may be cached by other processors; and the dirty state implies that the data is valid, and no other cache contains a copy of the data. RAC stores the status of outstanding memory requests and remote replies. Therefore, each node is modeled as:

```plaintext
NodeState: Record
  Cache: Array [ Home ] of Array [ Address ] of Record
    State: enum { invalid, shared, dirty };
    Value: value;
  End;
  RAC: Array [ Home ] of Array [ Address ] of Record
    State: RAC_State;
    Value: value;
    InvalidationCount: NodeCount;
  End;
End;
```

There are three basic transactions supported by the DASH cache coherence protocol: read, read-exclusive and write-back, with eight kinds of requests. There is also a direct memory access protocol (DMA read and DMA write) built on top of the basic transactions, with five kinds of requests. Four kinds of replies may be generated by basic and DMA transactions.

**The basic cache transactions for DASH-C**

Two typical cache coherence transactions are shown in Figure 2.4. In the first scenario, Node A issues a request for a read-exclusive copy of a particular memory location. When the request arrives at the home node, the directory is checked. Since the location is not cached by other nodes, the directory controller sends the data, and records the fact that Node A has a read-exclusive copy of the memory location. In the second scenario, after Node A receives the read-exclusive copy, Node B may request for a shared copy. Because Node A has the most up-to-date copy, the memory
Figure 2.4: Examples of cache coherence transactions: In the first transaction, processor 1 requests for an exclusive copy of a memory location, and the memory replies with the appropriate data. In the second transaction, processor 2 requests for a shared copy of the same location, and the memory forwards the request to the owner of the location.

forwards the request to Node A. Node A then sends two replies, one to Node B with the up-to-date data, and one to the home node to confirm the change in status.

Since simultaneous accesses can be made to the same location, the node receiving the forwarded requests may not have the dirty copy anymore, or it may not be able to release the copy. In these cases, negative acknowledgment is sent directly to the requesting node.

In the Murφ description, initiations of requests and receptions of messages are modeled by rulesets, such as:

```plaintext
Ruleset n : Node ; h : Home; a : Address Do
  Rule "Request data"
    ...  
End

Ruleset Dest : Node ; Src : Node Do
  Rule "Process message"
    ... 
End;
```
The first ruleset instantiates the enclosed rule for each distinct node, each physical memory, and each address in a physical memory. The second ruleset nondeterministically chooses a destination node and a source node, so that the enclosed rule can check whether there is a message from the source node to the destination node.

The action in each phase of a transaction is modeled in nested switch statements, such as the following rule, which spontaneously generates a read request:

```RULE "Remote Memory Read Request"
( h != n ) -- the current node does not contain the
-- physical memory of the requested location.
==>
Begin
Switch RAC_State
Case Invalid_RAC:
-- no pending event
Switch Cache_State
Case Invalid:
-- set outstanding event: waiting for read reply
RAC_State := Read_Pending;
-- send request to home cluster
Send_Read_Request_To_Home(h,n,a);
Else
-- other cache supplies data using snoopy protocol
End;
Case WINV:
-- waiting for invalidation: defer request
Assert ( Cache_State = Dirty ) "WINV with non-dirty copy";
Case ...
End;
End;
```

The DMA transactions for DASH-C

Direct memory read and write transactions increase the number of exceptions to the normal flow of operations and the likelihood of an error in the protocol. These transactions are modeled in a similar way to the basic transactions, and Murφ was able to re-discover a bug due to the interaction of basic transactions and DMA transactions: the DASH design team discovered the same bug, but only after extensive simulation of the whole multiprocessor.
An industrial cache coherence protocol (ICCP)

An industrial cache coherence protocol is also included as one of the main examples in this thesis. In this protocol, the messages in the network may arrive in arbitrary order. Therefore, more complicated interactions arise between different transactions, and more synchronization messages are needed to maintain cache coherence. This protocol is included here to illustrate how Murφ can model an unordered network with a ruleset to select an arbitrary message:

```plaintext
VAR
Homes: Array [Home] of HomeState;
Nodes: Array [Node] of ProcState;
Net: Record
  Count: 0..NetMax;
  Ar: Array [0..NetMax-1] of Message;
End;

Ruleset Index: 0..NetMax-1 Do
  Rule
    (Index < Net.Count) -- is it a valid message?
    ==> process_message(Index); -- process the message at slot 'Index'
    remove_message(Index); -- remove the message and move up all
    -- other messages appear after this message
  Endrule;
Endruleset;
```

The reduction algorithms presented in this thesis can take advantage of the unordered network to achieve even better reductions.

### 2.2.2 Other Practical Systems

Many other systems have been verified using Murφ, including abstract algorithms and multiprocessor implementations [PNAD95, YGM+95, DDHY92, SD95b, DPN93, PD95a, Par94, WG94, CRL96]. The verification results of the following systems are also presented in subsequent chapters to illustrate the performance of the new verification algorithms:
2.2. VERIFICATION OF PRACTICAL SYSTEMS

| PETE RSON | Peterson's algorithm for the $n$-process mutual exclusion problem [Pet81]. |
| MCS1     | MCS distributed lock algorithm with atomic compare and swap operation [MCS91]. |
| MCS2     | MCS distributed lock algorithm without atomic compare and swap operation [MCS91]. |
| DASH-L   | A queue-based lock implementation for the Stanford DASH Multiprocessor [LLG+92]. |
| LIST1    | A distributed list protocol (version 1) [Dil95] |
| LIST2    | A distributed list protocol (version 2) [Dil95] |

2.2.3 Verification Results

In order to verify the systems described in the previous sections, several in-line assert statements, in-line error statements, and invariants are specified. The in-line assert and error statements are used to detect any message arriving at a node with an unexpected state, and any inconsistency among the states of different components. Invariants are used to check some global properties, such as:

Invariant "Only a single master copy exists"
Forall n1 : Proc ; n2 : Proc ; h : Home ; a : Address Do
  ! ( n1 ! = n2 & Procs[n1].Cache[h][a].State = Dirty
   & Procs[n2].Cache[h][a].State = Dirty )
Endforall;

Invariant "Adequate invalidations with Read Exclusive request"
Forall n1 : Node ; n2 : Node ; h : Home ; a : Address Do
  ( n1 = n2 ) | ( ( Nodes[n1].RAC[h][a].State = WINV )
   & ( Nodes[n2].Cache[h][a].State = shared ) )
  ->( Exists i : 0..ChanMax-1 Do
     ( i < ReqNet[h][n2].Count
      & ReqNet[h][n2].Messages[i].Mtype = INV )
    End ) )
Endforall;

The first invariant states that there can be at most 1 dirty cache line for each address. The second invariant states that whenever a node is waiting for invalidation
of other shared copies (WINV), all nodes with shared copies of the memory location should have a pending invalidation message (INV).

The Murφ verifier\(^1\) was able to detect an error in the DASH protocol with two remote nodes, after exploring 14,825 states in 63 seconds. The counter-example constructed is shown in Figure 2.5.

Inconsistent cache values were obtained because the DMA write request and the reply with the writable copy were received out-of-order. Node B originally had a shared copy of the data, and then it proceeded to request a read-exclusive copy of the data. As an attempt to minimize the number of invalidate requests, an invalidate request was not sent to node B, since the shared copy will get promoted to a read-exclusive copy anyway. However, this allows Node B to modify and write back the data before processing the DMA write request.

Once the protocol was fixed so that an invalidate request is also sent to Node B, no error was found. Node B can no longer write back the data until it has processed (and ignored) the invalidate request. Since the requests arrive in order, Node B will have processed the DMA write request before it processes the invalidate request. Murφ was able to verify the new protocol with 41848 states in 191s. Murφ also shows that the other systems in the previous sections satisfy the specified invariants, lead to no deadlock situation, and do not violate the error assertions.

The verification results are summarized in Table 2.1. However, when we increase the sizes of the systems modelled in Murφ, the number of states increases rapidly. In order to verify the DASH-C protocol with 3 remote nodes and 3 data values, the DMA operations are disabled. On the other hand, in order to verify the ICCP protocol with 5 processors and 1 data value, the maximum number of outstanding messages in the network is restricted to 7. The verification results for these larger systems are summarized in Table 2.2.

\(^1\)The results presented in this thesis have been obtained on a SPARC 20 Workstation using Murφ Version 2.9S, or an extension of Version 2.9S to include the new reduction algorithms.
1. A remote node A issues a DMA write request.
2. Another remote node B issues a normal read request.
3. Memory receives the read request and sends read data to B.
4. B receives read data and updates the cache line.
5. Memory forwards the DMA write from A to B and updates its own copy.
6. B issues a normal write request.
7. Memory receives the write request and sends a write reply.
8. B receives the write reply and updates the cache line.
10. B issues an explicit writeback request.
11. Memory receives the writeback from B and updates its own copy.
12. B issues a normal read request.
13. Memory receives the read request from B and sends read data to B.
14. B receives read data and updates the cache line.
15. B receives the DMA write request forwarded from memory and updates its copy.
16. Memory and B now have inconsistent cache values.

Figure 2.5: An counter-example in the DASH protocol: Inconsistent cache values were obtained because the DMA write request is delayed.
CHAPTER 2. THE MURφ VERIFICATION SYSTEM

<table>
<thead>
<tr>
<th>system</th>
<th>DASH-C with 2 remote nodes and 2 data values</th>
<th>ICCP with 3 processors and 2 data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>41,848</td>
<td>150,794</td>
</tr>
<tr>
<td>time</td>
<td>191s</td>
<td>96s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>system</th>
<th>PETERSON</th>
<th>MCS1</th>
<th>MCS2</th>
<th>DASH-L</th>
<th>LIST1</th>
<th>LIST2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 procs</td>
<td>4 procs</td>
<td>3 procs</td>
<td>3 clusters</td>
<td>4 nodes</td>
<td>4 nodes</td>
</tr>
<tr>
<td>size</td>
<td>628,868</td>
<td>554,221</td>
<td>3,240,032</td>
<td>55,366</td>
<td>560,185</td>
<td>1,382,001</td>
</tr>
<tr>
<td>time</td>
<td>1,032s</td>
<td>194s</td>
<td>6454s</td>
<td>188s</td>
<td>175s</td>
<td>1,117s</td>
</tr>
</tbody>
</table>

Table 2.1: Verification results using the basic algorithm, I

<table>
<thead>
<tr>
<th>system</th>
<th>DASH-C with 3 remote nodes and 3 data values without DMA operations</th>
<th>ICCP with 5 processors and 1 data values with at most 7 outstanding messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>210,663</td>
<td>2,093,231</td>
</tr>
<tr>
<td>time</td>
<td>1,011s</td>
<td>2,048s</td>
</tr>
</tbody>
</table>

Table 2.2: Verification results using the basic algorithm, II
Chapter 3

Verifying Symmetric Systems

Chapter Overview ¹

This chapter describes how structural symmetry can be exploited in automatic formal verification. In order to use symmetry in a practical verification system, two main challenges must be met: to detect symmetries and to generate a symmetry-reduced state graph efficiently.

These two challenges must be solved without generating the original state graph, otherwise, we would not benefit from the reduction. Indeed, if we already had the original state graph, the symmetry-reduced state graph could be constructed in polynomial time using standard bisimulation minimization algorithm, without even knowing the structural symmetries explicitly.

3.1 Symmetry

To illustrate the concept of symmetries, let’s examine the cache coherence protocols described in Section 2.2. Such protocols consider all processors to be identical, and therefore, all states resulting from permuting the processors are equivalent. For example, the states A and B in Figure 3.1 are equivalent, and have similar behavior. In a manual proof, once we had proved that all states reachable from state A are

¹This chapter is based on materials published in [ID93a, ID93b, ID96a].
Figure 3.1: Symmetry in a cache coherence protocol: A cache coherence protocol considers all processors to be identical. While states A and B have different processors caching the data from the memory, each of them have exactly one processor caching the data, and exactly one processor having an invalid cache line. Their behaviors in the protocol are therefore similar.

error-free, we could argue "by symmetry" that all states reachable from state B are also error-free.

However, if we assign a distinct integer processor-id to represent each processor, as in the description in the previous chapter, we implicitly impose an ordering on the processors. In fact, most of the properties of integers are irrelevant for the processor-ids. It only matters whether two processor-ids are the same; it does not matter whether one is numerically less than the other, or whether they are consecutive. But most verifiers have no way to detect this fact, so they may inspect what is basically the same state many times.

In addition to the processors, there are several other symmetries for the cache coherence protocols: addresses, data values, and memory module-ids. The messages in the unordered network in ICCP also represent another symmetry in the system. Although their numerical properties are likely to be important at some level of abstraction, they are irrelevant for reasoning about the correctness of the protocols at the level of descriptions in the previous chapter.
3.2 Automorphisms Induced by Symmetry

In order to make the idea of symmetry more precise, and to construct the foundation for proving the soundness and completeness of the reduction theorem, this section describes the automorphisms induced by symmetry, and the properties of the corresponding quotient graph.

Structural symmetry in a system always generates a set of non-trivial automorphisms in the state graph of the system. An automorphism can be defined as:

**Definition 3.1 (automorphism)** A graph automorphism on a state graph \( A = (Q, Q_0, \Delta, \text{error}) \) is a bijection \( h : Q \to Q \) with three properties: for every \( q_1 \) and \( q_2 \) in \( Q \), \((q_1, q_2) \in \Delta \) if and only if \((h(q_1), h(q_2)) \in \Delta \); for every \( q \) in \( Q \), \( h(q) \in Q_0 \) if and only if \( q \in Q_0 \); and \( h(\text{error}) = \text{error} \).

Since an automorphism is a bijection, function inversion and composition preserve automorphism. Therefore, a group can be generated from a set of automorphisms.

**Lemma 3.1** For any set of automorphisms \( H \), the closure \( G(H) \) of \( H \cup \{\text{id}\} \) under inverse and composition is a group.

Any automorphism group \( G(H) \) defines a symmetry-equivalence relation \( \approx_H \) on the state, such that \( p \approx_H q \) if and only if \( \exists h \in G(H) : p = h(q) \). This symmetry-equivalence relation is a congruence relation:

**Definition 3.2 (congruence)** A congruence \( \approx \) on a graph is an equivalence relation on \( Q \) such that for all \( q_1, q_2 \in Q \) such that \( q_1 \approx q_2 \), if there exists \( q'_1 \in Q \) such that \((q_1, q'_1) \in \Delta \), then there exists \( q'_2 \in Q \) such that \( q'_1 \approx q'_2 \) and \((q_2, q'_2) \in \Delta \).

**Theorem 3.1** The symmetry-equivalent relation \( \approx_H \) induced by a set of automorphisms \( H \) on a state graph \( A \) is a congruence relation on \( A \).

**Proof.** Because of the properties of a group, \( \approx_H \) is obviously an equivalence relation. In addition, for every \( q_1, q_2 \in Q \) with \( q_1 \approx_H q_2 \), there is an automorphism \( h \) such that \( q_2 = h(q_1) \). By definition, if there exists \( q'_1 \in Q \) such that \((q_1, q'_1) \in \Delta \), then \((h(q_1), h(q'_1)) \in \Delta \), i.e. \((q_2, h(q'_1)) \in \Delta \).

It should be clear from the definition of automorphism that \text{error} is congruent only to itself.
We denote the equivalence class of states congruent to \( q \) as \([q]\), and restate the definition of a congruence using this notation:

**Lemma 3.2** For all \( q_1, q_1' \in Q \), if \((q_1, q_1') \in \Delta\), then for all \( q_2 \in [q_1] \), there exists \( q_2' \in [q_1'] \) such that \((q_2, q_2') \in \Delta\).

Therefore, a quotient graph can be constructed from the original state graph using the symmetry equivalence relation:

**Definition 3.3 (quotient graph)** A quotient graph of a graph \( A = \langle Q, Q_0, \Delta, \text{error} \rangle \) w.r.t. a congruence \( \approx \) is \( A/\approx = \langle Q', Q_0', \Delta', \text{error}' \rangle \), where \( Q' = \{[q] : q \in Q\} \), \( Q_0' = \{[q] : q \in Q_0\} \), \( \text{error}' = [\text{error}] \), and \( \Delta' = \{(p, [q]) : (p, q) \in \Delta\} \).

The quotient graph constructed from the symmetry equivalence relation is called a symmetry-reduced state graph. Because of the following theorem and lemmas, this reduced state graph can be used to verify the desired properties of the system.

**Theorem 3.2 (reachability)** Given a state graph \( A = \langle Q, Q_0, \Delta, \text{error} \rangle \) with a set of automorphisms \( H \) on \( A \), \( q \) is reachable in \( A \) if and only if \([q]\) is reachable in \( A/\approx_H \).

**Proof.**

\[\Rightarrow\] Suppose \( q \) is reachable in \( A \). Then there exists a path \( q_0, \ldots, q_{n-1}, q \) in \( A \).
\([q_0], \ldots, [q_{n-1}], [q]\) is the path to reach \([q]\) in \( A/\approx_H \).

\[\Leftarrow\] If \([q]\) is reachable in \( A/\approx_H \), there exists a path \([q_0], \ldots, [q_{n-1}], [q]\) in \( A/\approx_H \). We show by induction on the length of the path that \( q \) is reachable in \( A \). The base is obvious: every member of \([q_0]\) is reachable, since for all \( h \in H \), \( h(q) \in Q_0 \) if and only if \( q \in Q_0 \). Now consider the path, \([q_0], \ldots, [q_{i-1}], [q_i]\). By the definition of \( \Delta' \), there exist states \( p_{i-1} \in [q_{i-1}] \) and \( p_i \in [q_i] \) such that \((p_{i-1}, p_i) \in \Delta\).

Since \([p_{i-1}] = [q_{i-1}]\), by the induction hypothesis, \( p_{i-1} \) is reachable in \( A \), and there exists a path \( p_0, \ldots, p_{i-1} \). Because \((p_{i-1}, p_i) \in \Delta\), there exists a path \( p_0, \ldots, p_{i-1}, p_i \). But \( p_i \approx_H q_i \), so there exists an automorphism \( h \) such that \( q_i = h(p_i) \). Since \( p_0, \ldots, p_i \) is a path, \( h(p_{i-1}), h(p_i) \) is also a path to \( q_i \) in \( A \), so the state \( h(p_i) = q_i \) is reachable in \( A \).
Corollary 3.1 (error detection) error is reachable in $A$ if and only if $[\text{error}]$ is reachable in $A/\approx_H$.

Corollary 3.2 (deadlock detection) If $A$ has a reachable deadlock state $q$, $A/\approx_H$ also has a reachable deadlock state $[q]$.

Furthermore, because of Theorem 3.2, the same quotient graph can be used for LTL model checking, and because the branching structure is also preserved, it can be used for CTL* model checking [ES96, CEFJ96]. For the definitions of LTL and CTL* model checking, see [Wol87] (c.f. [Lam80, EH83]).

On the other hand, although we have a necessary and sufficient condition for error detection, we only have a necessary condition for deadlock detection. The difficulty is that $[q]$ may be a deadlock in $A/\approx_H$ even though there are transitions between states within $[q]$.

Fortunately, there is an alternative test for deadlocks because, if one of the states in an equivalence class $[q]$ is a deadlock state in the original state graph, all of the states in $[q]$ are deadlock states. This fact is restated as the following lemma:

**Lemma 3.3** For every state $[q]$ in the quotient $A/\approx_H$, if there exists one state in $[q]$ that is not a deadlock state in $A$, no state in $[q]$ is a deadlock state in $A$.

**Proof.** Suppose there is a state $q \in [q]$ that is not a deadlock state in $A$, so there exists $q' \neq q$ such that $(q, q') \in \Delta$. For every $p \in [q]$, there exists an automorphism $h$ such that $p = h(q)$. By definition, $(p, h(q')) \in \Delta$, but $h$ is a bijection, so $p \neq h(q')$. Hence, $p$ cannot be a deadlock state in $A$, either.

This lemma allows us to check whether state $q$ in $A$ is a deadlock by choosing the most convenient member $q'$ of its equivalence class $[q]$ in $A/\approx_H$, and checking whether $q'$ has a successor other than itself in $A$. This check can be done locally, as described in Section 3.5, so deadlock checking does not require inspecting the original graph $A$.

### 3.3 Detecting Symmetry

An important element for symmetry reduction in verification is how to find an appropriate set of automorphisms. Instead of relying on the user to provide the automorphisms, we propose to extract automorphisms automatically from a description
of the system to be verified, so that the symmetry-equivalent relations are guaranteed to be valid.

An appropriate set of automorphisms can be extracted through the use of a new datatype, called *scalarset*, which models full symmetry in a system: the behavior of a program is the same under arbitrary permutations of the elements in a scalarset. Scalarset is a subtype of conventional integer subrange: a scalarset is involved in a subset of the conventional subrange operations. The restrictions, presented later in this section, outlaw non-symmetry operations.

For example, since the mutual exclusion protocol shown in Figure 2.1 never uses a symmetry-breaking operation on the type *Pid*, the subrange corresponding to *Pid* can be converted to a scalarset of size 2, as shown in Figure 3.2. Using the scalarset type has the advantage that the compiler can report to the user when a symmetry-breaking operation has been unintentionally applied. Therefore, a verification tool would not risk unsoundness by exploiting invalid symmetries. Also the more abstract description allowed by using scalarsets may have other advantages; for example, in a more general programming language, a scalarset could be safely refined into many different implementations, such as a subrange type or an enumerated type.

Scalarset enables the verifier to automatically construct the appropriate symmetry-equivalent relation. Without scalarset, it would be necessary for users to write their own procedures to check whether two states are equivalent. Such a procedure is not guaranteed to be sound, and may also be complicated, error-prone, and slow. With scalarset, we are able to automatically generate the correct procedure with appropriate heuristics for fast symmetry equivalence checking.

While scalarset is designed to model full symmetry, in which any permutation of the scalarset values results in an equivalent state, there are also other kinds of symmetries that can occur in a system, such as symmetry in the classical dining philosopher problems (symmetry with respect to rotation), and additional data types could be added to support these symmetric types, as presented in Section 3.8. However, full symmetry is especially important because the savings can approach a factor of \( N! \) for a scalarset of size \( N \) (summarized in Section 3.7), as opposed to a factor of \( N \) for rotation. Full symmetry is also prevalent in the applications of practical importance,
const NumProcesses : 2;

type Pid : scalarset( NumProcesses );

var P : array[ Pid ] of enum { Critical, NonCritical };

startstate for i : Pid do P[i] := NonCritical; end

ruleset i : Pid do
  rule “Entering Critical Section”
    forall j : Pid do P[j] = NonCritical end ⇒ P[i] := Critical; end

  rule “Leaving Critical Section”
    P[i] = Critical ⇒ P[i] := NonCritical; end

end

invariant “Mutual Exclusion”
  forall i : Pid; j : Pid do i ≠ j → (P[i] ≠ Critical ∨ P[j] ≠ Critical) end

Figure 3.2: An example on the use of scalarset: Because the operations for Pid are symmetric, it can be converted from a subrange to a scalarset.
such as cache coherence. Thus, the remaining discussion concentrates on scalarsets and full symmetry.

**Scalarset Restrictions**

The restrictions for scalarset were designed to make sure that, given a program containing a scalarset, any function that permutes the elements of the scalarset consistently throughout a state generates symmetry-equivalent states.

When a new scalarset is declared, the size $n$ is included in the declaration$^2$, and it represents a subrange from 1 to $n$ with the following restrictions:

1. An array with a scalarset index type cannot be indexed by integer constants and expressions other than a scalarset variable of exactly the same type.

2. Scalarset variables cannot be used as operands of arithmetic operators.

3. Scalarset variables cannot be compared with integer constants, and two scalarset variables can only be compared using $\neq$. In such cases, the two variables being compared must be of exactly the same type.

4. For all assignments $d := t$, if $d$ is a scalarset variable, $t$ must be another scalarset variable of exactly the same type.

5. If a scalarset variable is used as the index of a for statement, the body of the statement is restricted so that the result of the execution is independent of the order of the iterations.$^3$

---

$^2$In Chapter 4, scalarsets of arbitrary size are discussed for data-independent systems.

$^3$One sufficient (but not necessary) restriction to obtain this property is that the set of variables written by any iteration be disjoint from the set of variables referenced (read or written) by other iterations.
3.4 Extracting Automorphisms

Given a description with a scalarset, the automorphisms induced by the corresponding symmetry can be constructed from the set of permutations on the value of a scalarset. A permutation on a scalarset is defined as follows:

**Definition 3.4 (permutation on values)** If \( \alpha \) is a scalarset type, a permutation \( \pi_\alpha : (\alpha \cup \{\bot\}) \rightarrow (\alpha \cup \{\bot\}) \) is a bijection such that \( \pi_\alpha(\bot) = \bot \).

In this definition, we use the same symbol \( \alpha \) to present the set of values represented by the scalarset \( \alpha \), excluding the special undefined value \( \bot \).

Extending this notion of permutation, we can obtain symmetry-equivalent states by permuting the scalarset entries of a state:

- When the permutation is applied to a scalarset variable, the value is modified to the corresponding permuted value.
- When an array indexed by a scalarset is permuted, the contents of the array elements are permuted and the elements are rearranged according to the permutation.

For example, as shown in Figure 3.3, \( \pi_\alpha \) permutes a scalarset \( \alpha \) of size 4 so that 0 maps to 0, 1 maps to 2, 2 maps to 1, and 3 maps to 3. When \( \pi_\alpha \) is used to permute a state, \( \pi_\alpha \) will apply the permutation to each element of the array \( P \), and then rearrange the positions of the elements. The variable \( Dir \) is changed from 1 to 2 accordingly. Therefore, all references to the array \( P \) through \( Dir \) still give the corresponding permuted element.

This definition of permutation on a state guarantees that the permutations are automorphisms on the state graph.

**Theorem 3.3 (soundness theorem)** Given a Murϕ description containing a scalarset \( \alpha \), every permutation \( \pi_\alpha \) on the states of the state graph \( A \) derived from the program is an automorphism on \( A \).

---

4When we refer to “applying a permutation to a state,” we are referring to a one-to-one mapping on the elements of a scalarset, not necessarily a permutation of the state variables.
Corollary 3.3 Two states p and q are symmetry-equivalent if there exists a permutation h such that p = h(q).

The proof of Theorem 3.3 is presented in Appendix A.2, which contains the following steps:

1. Define precisely how to permute the scalarset values in a state to generate an equivalent state.

2. Show that equivalent states satisfy the same Boolean expression expressible in the description language.

3. Show that, if a state q' is generated by a transition rule f from a state q, every state equivalent to q' can be generated by another transition rule similar to f from a state equivalent to q.

3.5 On-the-Fly Reduction Algorithm

After extracting the appropriate automorphisms from a description, a canonizer can be automatically generated, and incorporated into the on-the-fly symmetry-reduction algorithm.
Definition 3.5 (canonizer) Given a graph $A = \langle Q, Q_0, \Delta, \text{error} \rangle$ and a set of automorphisms $H$ on $A$, a canonizer $\zeta$ is a function that maps each state $q$ to a unique member of its equivalence class $[q]$ in $A/\approx_H$.

The unique member is called the canonical state. A straightforward algorithm of this canonizer may generate all equivalent states and choose a lexicographical minimum state as the canonical state. Heuristics for faster algorithm are presented in the next section.

The only change to the original verification algorithm is the addition of a canonizer, which is highlighted by an underline in Figure 3.4. The new on-the-fly algorithm only stores canonical states in the hash table, and the search on a state $q$ is cut off whenever $\zeta(q)$ is found in the hash table, meaning that an equivalent but not necessarily identical state has been previously encountered in the search.

Because a deadlock state in the reduced state graph does not imply a deadlock state in the original state graph, a deadlock state is determined by checking whether $q' \neq q$ for every transition rule, but not whether $\zeta(q') \neq q$. This test exploits Lemma 3.3: we choose $q$ as a convenient representative of its equivalence class, and apply each transition rule to see if $q$ is a true deadlock state in the original state graph.

3.6 Efficient Heuristics

The complexity of the symmetry-reduction algorithm is directly related to the complexity of the canonizer $\zeta$. Unfortunately, computing a canonical representative is at least as hard as testing for graph isomorphism [CFJ93]:

Definition 3.6 (graph isomorphism) Graph $G(V, E)$ is a graph with vertex set $V$ and edge set $E$. Two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are isomorphic if there exists a one-to-one correspondence $\pi$ between their vertices and edges such that the incidence relationship is preserved, that is, $(x, y)$ belongs to $E_1$ if and only if $(\pi(x), \pi(y))$ belongs to $E_2$.

Theorem 3.4 Finding the canonical state is at least as hard as testing for graph isomorphism.
Symmetry Algorithm()
begin
  Reached = Unexpanded = \{ \zeta(q) | q \in \text{StartState} \};
  while Unexpanded \neq \emptyset do
    Remove a state q from Unexpanded;
    Deadlock = true;
    for each transition rule \( t \in T \) do
      if an error statement is executed in \( t \) on q then report error; endif;
      let \( q' = t(q) \) in
      if \( q' \) does not satisfy one of the invariants then report error; endif;
      if \( q' \neq q \) then Deadlock = false; endif;
      if \( \zeta(q') \) is not in Reached then put \( \zeta(q') \) in Reached and Unexpanded; endif
    endlet;
  endfor;
  if Deadlock then report deadlock; endif;
endwhile;
end

Figure 3.4: An on-the-fly symmetry reduction algorithm: The part highlighted by an underline represents the main difference of this algorithm from the basic algorithm shown in Figure 2.2.
Proof. Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two arbitrary finite directed graphs of $n$ nodes. We can represent each graph by an incidence matrix in a variable $v$ of the type

$$\text{array } [\alpha] \text{ of array } [\alpha] \text{ of } 0..1,$$

where $\alpha$ is a scalar set of size $n$, and $v[x][y] = 1$ if and only if $(x, y)$ belongs to the edge set $E_1$ and $E_2$ respectively.

Therefore, the graphs are isomorphic if and only if $\zeta(G_1) = \zeta(G_2)$, where $\zeta$ is a canonizer for states stored in the variable $v$. \qed

No polynomial-time algorithm is known for the testing of graph isomorphism. A straightforward implementation, which generates all equivalent states and chooses a lexicographical minimum state as the canonical state, is inefficient. When compared to a simple explicit state enumeration algorithm, this naive implementation often results in significantly increased time, even when there are vast reductions in the number of states explored, as shown in the results presented in [ID93a, ID93b], and in Table 3.1.

Two heuristics are described in the remainder of this section. The first one has been adapted from graph isomorphism heuristics. It is exact and guaranteed to generate a unique representative for each equivalence class. The second one uses an approximation algorithm, which may generate more than one representative for each equivalence class, resulting in a potentially larger reduced state graph.

3.6.1 Graph Isomorphism Heuristics

Many heuristics for testing graph isomorphism [Gou88, CK80, CG70, Ebe88, Mit88] can be used to speed up the construction of a unique representative for each equivalence class. This section describes a simple heuristic based on vertex invariants, and its adaptation to the symmetry reduction algorithm.

Vertex Invariants

Many graph isomorphism heuristics use vertex invariants, which are the properties of vertices that must be preserved under automorphism. Formally, a vertex invariant
inv is a function which labels the vertices of an arbitrary graph with integers so that similar vertices are assigned the same label. Examples of simple vertex invariants include the in-degree and the out-degree for the specified vertex.

Vertex invariants can be used to speed up the procedure to check whether or not two graphs are isomorphic. Given a graph \( G_1(V_1, E_1) \), a vertex invariant \( inv \) can be used to partition \( V_1 \) into \( k \) classes \( V_1^1, V_1^2, ..., V_1^k \), such that vertices in the same class are assigned the same label by \( inv \). Furthermore, we can arrange the classes so that the integer label for the vertices in \( V_1^i \) is smaller than the label for the vertices in \( V_1^j \), whenever \( i < j \). If another graph \( G_2(V_2, E_2) \) is isomorphic to \( G_1 \), the same vertex invariant \( inv \) should also partition \( V_2 \) into \( k \) classes \( V_2^1, V_2^2, ..., V_2^k \), such that \( V_2^i \) contains the same number of vertices as \( V_1^i \), and vertices in \( V_2^i \) have the same label as the vertices in \( V_1^i \). Furthermore, the one-to-one correspondence \( \pi \) that converts \( G_1 \) to \( G_2 \) must map each vertex in \( V_1^i \) to a vertex in \( V_2^j \). Therefore, if the \( i \)th class has \( p_i \) vertices, instead of \( n! \) mappings, where \( n \) is the number of vertices, only \( p_1! \times p_2! \times ... \times p_k! \) mappings need to be checked.

This algorithm for checking whether or not two graphs are isomorphic can be converted to an algorithm to generate a unique canonical graph \( \zeta(G) \) in the set of graphs isomorphic to a graph \( G \). Instead of \( n! \) isomorphic graphs, only \( p_1! \times p_2! \times ... \times p_k! \) isomorphic graphs are generated, and the lexicographically smallest one is chosen as the canonical graph.

Adaptation to Symmetry Reduction

To use heuristics with vertex invariants in a symmetry-reduction algorithm, we regard each value in each scalarset as a vertex in a graph. A vertex invariant is defined below to obtain an appropriate partition on the scalarset values, from which a canonical state can be generated in a shorter time.

A trivial partition \( \{V^1\} \) is used as the initial partition of the values in the scalarsets. This partition can be refined into better partitions according to the procedures shown below. The global variables used to store the states are divided into five categories, and each category provides a different way to refine a partition. In
these procedures, a symmetry array refers to an array with a scalarset indextype, and a scalarset variable refers to a variable with a scalarset type.

1. **A scalarset variable that is not an element of a symmetric array:**

   The value $i$ of this variable corresponds to a special vertex. If $i$ belongs to the class $V^k$, we can separate $V^k$ into two classes $\{i\}$ and $V^k \setminus \{i\}$ (with the label of $i$ smaller than the label of the other vertices).

2. **A symmetric array in which the elements do not involve any scalarset:**

   If the $i$th and $j$th elements have different values, $i$ and $j$ are labeled differently (ordered by the values in the array elements), and the partition is refined so that they belong to different classes.

3. **A symmetric array in which the elements are variables of a scalarset type other than the index type of the array:**

   This corresponds to a bipartite graph. First of all, if $i$ and $j$ have different in-degree or out-degree, the partition is refined so that $i$ and $j$ belong to different classes (ordered by the in-degree or the out-degree).

   Furthermore, if the $i$th and $j$th elements have scalarset values from two different classes, the partition is refined so that $i$ and $j$ belong to different classes (ordered by the labels of the array elements). This process is repeated until the partition remains the same.

4. **A symmetric array in which the elements are variables of the same scalarset type as the index type of the array:**

   This corresponds to a graph in general. The refinement is performed in the same way as the previous category, except for an extra case: If the $i$th element has value $i$, and the $j$th element has value other than $j$, the partition is refined so that $i$ and $j$ belong to different classes (with the label of $i$ smaller than the label of $j$).
Table 3.1: Performance of the graph isomorphism heuristics: While a naive algorithm requires a longer time to generate the canonical state graph, adaptation of graph isomorphism heuristics can reduce the time to less than 5% of the time required to generate the original state graph.

<table>
<thead>
<tr>
<th>System</th>
<th>DASH-C</th>
<th>DASH-C</th>
<th>ICCP</th>
<th>ICCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>symmetry</td>
<td>2 clusters &amp; 2 data values</td>
<td>3 clusters &amp; 3 data values w/o DMA</td>
<td>3 processors &amp; 2 data values</td>
<td>5 processors &amp; 1 data value with restricted messages</td>
</tr>
<tr>
<td>without reduction</td>
<td>191s</td>
<td>1,011s</td>
<td>96s</td>
<td>2,048s</td>
</tr>
<tr>
<td>naive canonicalization</td>
<td>480s</td>
<td>2,467s</td>
<td>49s</td>
<td>567s</td>
</tr>
<tr>
<td>heuristics canonicalization</td>
<td>154s</td>
<td>200s</td>
<td>24s</td>
<td>28s</td>
</tr>
</tbody>
</table>

5. A variable of complex type:

No refinement is performed, unless the variable can be broken down into smaller data structures that belong to the other categories. For example, an array of records can be broken into a record of arrays with simpler element types, and some of these arrays may fall into other categories. In such cases, refinement is performed for the parts that belong to other categories.

With the partition obtained by these refinements, a canonical state can be obtained by exhaustively checking each of the permutations that preserve the partitioning. Therefore, this heuristic drastically reduces the number of permutations that need to be tried, and results in a faster algorithm, as shown in Table 3.1.

### 3.6.2 Avoiding the Graph Isomorphism Problem

In some cases, the heuristics for graph isomorphism may not be fast enough. The following observation is used to make the algorithm even faster:

**Observation 3.1** Any function that maps each state to an equivalent state can be used in place of the canonizer while maintaining the soundness of the algorithm.

---

5In fact, if every global variable belongs to categories 1 or 2, only one permutation is performed, because every permutation that preserves the partitioning generates the same canonical state.
**Proof.** The verification algorithm will check each equivalence class at least once, which is sufficient to find all errors and deadlocks.

Therefore, instead of using a canonizer, a *normalizer* can be used to map the states in an equivalence class to a small set of representatives. The efficiency of a normalizer depends on how fast it can generate a representative from a state, and how often it generates a unique representative. Observation 3.1 gives a great deal of freedom in the choice of a normalizer, which can be fast and domain-specific.

The normalizer implemented in Mur$\phi$ also uses the graph isomorphism heuristic with vertex invariants to obtain a small set of permutations. However, instead of checking every one of these permutations to find the lexicographically smallest state as the unique canonical state, only a fixed number of permutations are performed. Therefore, it avoids the potential exponential complexity in the final phase, with the penalty that two equivalent states may map to two different representatives.

Fortunately, as shown in Table 3.2, the practical results show that about 10 explicit permutations are sufficient in many cases to map the states that are examined during the search to unique representatives in their corresponding equivalence class. The resulting reduced state graph has almost the same size as the canonicalized state graph, and it is generated in a much shorter time. The intuition is that if every processor has the same state, no matter how we permute a state, the same state will be generated. Therefore, instead of wasting time to perform $N!$ permutations, a fixed number of permutations (in fact, only 1 in this case) is sufficient to find the correct canonical state.

### 3.7 Implementation and Results

**Implementation**

Symmetry reduction can be easily implemented in a verification tool with an explicit state enumeration algorithm. Except for the symmetry-equivalent checking procedure $\zeta$, there is no change to the algorithm or the data structure. Except for the limited storage for the local variables used in $\zeta$, no extra storage is required. The
CHAPTER 3. VERIFYING SYMMETRIC SYSTEMS

<table>
<thead>
<tr>
<th>Symmetry on LIST1</th>
<th>3 remote cells and a network of 7 messages</th>
<th>4 remote cells and a network of 8 messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original size</td>
<td>8,893</td>
<td>560,185</td>
</tr>
<tr>
<td>Reduced size (Canonicalization)</td>
<td>1,069</td>
<td>13,044</td>
</tr>
<tr>
<td>Reduced size (Normalization)</td>
<td>1,077</td>
<td>13,497</td>
</tr>
<tr>
<td>Original time</td>
<td>5s</td>
<td>175s</td>
</tr>
<tr>
<td>New time (Canonicalization)</td>
<td>81s</td>
<td>4,056s</td>
</tr>
<tr>
<td>New time (Normalization)</td>
<td>11s</td>
<td>214s</td>
</tr>
</tbody>
</table>

Table 3.2: Performance of the normalization algorithm: A normalizer which checks 10 permutations requires a much shorter time to generate a reduced state graph that is almost the same size as the canonical state graph generated by a canonizer.

error trace on the original state graph can be reconstructed easily from the trace on the reduced state graph.

Verification Results

The reduction in the size of the state graph is closely related to the sizes of the equivalence classes. The larger the average size of the equivalence classes, the larger the reduction. With the permutations for a scalarset of size \(N\), an equivalence class has a maximum of \(N!\) states. Therefore, the state space is at most reduced by a factor of \(N!\). Using multiple symmetries, we increase the size of the equivalence classes. For \(n\) scalarsets with sizes \(N_1, \ldots, N_n\), the state space is at most reduced by a factor of \(N_1! \times N_2! \times \ldots \times N_n!\).

Savings close to maximum reduction are often realized for systems consisting of \(N\) replicated components, such as processors, as shown in Table 3.3. The rationale behind this observation is as follows: if the components are involved in reasonably complex operations, there are many situations where none of the components have exactly the same state. Hence, every non-trivial permutation of the state will result in a distinct state in the original state space, so the sizes of the equivalence classes are close to \(N!\).

However, there are some cases where \(N!\) reductions are not realized. For example, consider a network channel modeled by an array of size \(N\), where only the first \(n\)
### 3.7. IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th>System</th>
<th>PETERSON</th>
<th>MCS1</th>
<th>MCS2</th>
<th>DASH-L</th>
<th>LIST1</th>
<th>LIST2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetry</td>
<td>5 procs</td>
<td>4 procs</td>
<td>3 procs</td>
<td>3 clusters</td>
<td>4 nodes</td>
<td>4 nodes</td>
</tr>
<tr>
<td>Original size</td>
<td>628,868</td>
<td>554,221</td>
<td>3,240,032</td>
<td>55,366</td>
<td>560,185</td>
<td>1,382,001</td>
</tr>
<tr>
<td>Reduced size</td>
<td>6,770</td>
<td>23,636</td>
<td>540,219</td>
<td>9,313</td>
<td>23,410</td>
<td>57,616</td>
</tr>
<tr>
<td>Reduction</td>
<td>98.9%</td>
<td>95.7%</td>
<td>83.3%</td>
<td>83.2%</td>
<td>95.8%</td>
<td>95.8%</td>
</tr>
<tr>
<td>Original time</td>
<td>1.032s</td>
<td>194s</td>
<td>6,454s</td>
<td>188s</td>
<td>175s</td>
<td>1,177s</td>
</tr>
<tr>
<td>New time</td>
<td>12s</td>
<td>21s</td>
<td>384s</td>
<td>96s</td>
<td>15s</td>
<td>33s</td>
</tr>
<tr>
<td>Speedup</td>
<td>98.8%</td>
<td>96.5%</td>
<td>94%</td>
<td>49%</td>
<td>91.4%</td>
<td>95.5%</td>
</tr>
<tr>
<td>Theoretical maximum reduction</td>
<td>99.2%</td>
<td>95.8%</td>
<td>83.3%</td>
<td>83.3%</td>
<td>95.8%</td>
<td>95.8%</td>
</tr>
</tbody>
</table>

Table 3.3: Improvement in performance with scalarset: Close to $N_1! \times \ldots \times N_k!$ reductions are obtained, where $N_i$ is the size of the $i$th scalarset.
### Table 3.4: Inefficiency in handling symmetry in a message channel

<table>
<thead>
<tr>
<th>Symmetry on LIST1</th>
<th>3 remote cells and a network of 7 messages</th>
<th>4 remote cells and a network of 8 messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original size</td>
<td>8,893</td>
<td>560,185</td>
</tr>
<tr>
<td>Reduced size (Canonicalization)</td>
<td>1,069</td>
<td>13,044</td>
</tr>
<tr>
<td>Reduction (Canonicalization)</td>
<td>88%</td>
<td>97.7%</td>
</tr>
<tr>
<td>Theoretical Maximum Reduction</td>
<td>99.9%</td>
<td>99.9%</td>
</tr>
<tr>
<td>Original time</td>
<td>5s</td>
<td>175s</td>
</tr>
<tr>
<td>New time ( Canonicalization )</td>
<td>81s</td>
<td>4,056s</td>
</tr>
</tbody>
</table>

Elements store the actual messages in the channel, and the remaining elements contain empty messages. If the network does not preserve message orders, the ordering of the elements does not matter, and a scalarset can be used to index the array. But the reduction won’t be close to \( N! \) because permuting the empty messages always generates the same state. A naive algorithm would spend a large amount of time permuting the empty messages.

Such behaviors are demonstrated by the verification of ICCP, LIST1, and LIST2, all of which have an unordered network. The results for LIST1 are shown in Table 3.4, which show that a canonizer is very slow in this case, and the reductions are not close to the theoretical maximum.

Because of this inefficiency in modeling unordered channels using scalarsets, a new datatype, called *Multiset*, has been implemented to specify this situation. A multiset is also referred as a bag. It is essentially an array indexed by an anonymous scalarset type. A `choose` construct is used to nondeterministically select an element from a multiset and bind a parameter to a reference to the selected element. A special `add` construct and a special `remove` construct are used to put in and take out elements from the multiset. This new datatype allows the canonizer to ignore the permutation of the empty elements in the array, speeding up the verification by more than 90%, as shown in Table 3.5.
3.8. **RESTRICTED SYMMETRIES**

<table>
<thead>
<tr>
<th>Symmetry on LIST1</th>
<th>3 remote cells and a network of 7 messages</th>
<th>4 remote cells and a network of 8 messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original time</td>
<td>5s</td>
<td>175s</td>
</tr>
<tr>
<td>New time (Canonicalization on Scalarset)</td>
<td>81s</td>
<td>4,056s</td>
</tr>
<tr>
<td>New time (Normalization on Scalarset)</td>
<td>11s</td>
<td>214s</td>
</tr>
<tr>
<td>New time (Canonicalization on Scalarset &amp; Multiset)</td>
<td>3s</td>
<td>15s</td>
</tr>
</tbody>
</table>

Table 3.5: Improvement in performance with multiset: The multiset allows optimization in the canonizer that speeds up the verification by more than 90%.

### 3.8 Restricted Symmetries

Previous discussion in this chapter has concentrated on full symmetry represented by a scalarset, where every permutation on the scalarset value corresponds to an automorphism on the state graph. However, there are other classes of symmetries as well, such as reflexive ring symmetry, non-reflexive ring symmetry, and stabilization.

This section describes these symmetries and explains how they can be detected in a way similar to full symmetry. The proofs are omitted, but they follow the same sequence of analysis as the proof for scalarsets.

**Non-Reflexive Ring Symmetry**

A typical example for non-reflexive ring symmetry is a unidirectional local area ring network, in which each node has a predecessor and a successor.

To model a non-reflexive ring symmetry, a set of non-reflexive ring-indices $\alpha$ of size $N$ can be defined as a scalarset with two extra operations: given a ring-index $i$, $\text{pred}(i)$ gives the value $i - 1 \mod N$ and $\text{succ}(i)$ gives the value $i + 1 \mod N$. The corresponding set of permutations from which the automorphisms can be constructed is the automorphism group $G(H)$, where $H$ is the permutation mapping $i$ to $i + 1 \mod N$. The maximum reduction of a set of non-reflexive ring-indices is $N$. 

CHAPTER 3. VERIFYING SYMMETRIC SYSTEMS

Reflexive Ring Symmetry

A typical example for reflexive ring symmetry is the classical dining philosopher problem, in which each philosopher communicates directly to two neighbors only.

To model a reflexive ring symmetry, a reflexive ring-indices $\alpha$ of size $N$ can be defined as a scalar set with an associated set called `neighbor`. The elements of the set `neighbor` are the functions `pred` and `succ`, similar to the ones for non-reflexive ring symmetry. However, they cannot be accessed directly: any operation involving `pred` should have a corresponding operation for `succ`, as shown in the following example:

```plaintext
Type philosopher : ReflexiveRing ( N );
Var Phil : Array [ philosopher ] Of Record
    gotfork : Array [ neighbour(philosopher) ] Of Boolean;
End;
Ruleset i: philosopher;
    side1: neighbour(philosopher);
    side2: neighbour(philosopher) Do
    Rule `"get fork"'
        side1(i) ! = side2(i)
        & Phil [ side1(i) ] . gotfork [ side2(i) ] = false
        ==> Phil [ i ] . gotfork [ side1(i) ] = true;
    End;
End;

Ruleset i: philosopher
    side: neighbour(philosopher) Do
    Rule `"release fork"
    Phil [ i ] . gotfork [ side(i) ] = true
    ==> Phil [ i ] . gotfork [ side(i) ] = true;
    End;
End;
```

The corresponding set of permutations from which the automorphisms can be constructed is the automorphism group $G(H)$, where $H$ consists of two permutations, one of them mapping $i$ to $i - 1 \mod N$ and the other mapping $i$ to $i + 1 \mod N$. The maximum reduction of a non-reflexive ring-indices is $2N$. 
Stabilization

Sometimes, one or more of the components in a system may be special, and a scalarset cannot be used. Emerson and Sistla [ES93] called this stabilization. For example, proving a property about only processor 1 means that processor 1 is special, and any permutation mapping 1 to other values will not induce an automorphism.

The stabilization concept has been implemented in Murϕ using a *union* operator. A union concatenates a few enumerations and scalarsets together to form a new data type. For example, in a cache coherence protocol, if some of the components contain the physical memory, and some of them don’t, it may be modeled as:

```plaintext
TYPE
    Remote : Scalarset ( ProcCount - HomeCount );
    Home   : Scalarset ( HomeCount );
    Proc    : Union ( Home, Remote );
```

Therefore, both the remote nodes and the home nodes may be reordered among themselves, but a remote node cannot be mapped into a home node.

## 3.9 Comparison with Other Work

### Previous Work on Symmetry

The basic idea of exploiting symmetries to reduce a state space in automatic verification is not new. Lubachevsky used symmetry for automatic verification of a particular class of concurrent programs in 1984 [Lub84]. Aggarwal, Kurshan and Sabnani have applied symmetries for the verification of an alternating bit protocol [AKS83]. It was also described by Huber et al. [HJJ84] for high-level Petri nets, and Starke [Sta91] for deadlock and liveness checking in P/T nets.

However, the problems of detecting symmetry and extracting symmetry-equivalent relations automatically were not addressed, which makes application of symmetry-reduction difficult in practice.

### Minimal State Partitioning

Instead of using structural symmetry in a system, the basic minimal state partitioning methods [BFH90, BFH92, Fer93, LY92, ACH92] take advantage of bisimulation
relations on a state graph. Although symmetry and bisimulation relations are similar concepts, the resulting algorithms are different. Instead of generating the reduced state graph incrementally and checking the properties on-the-fly, minimal state partitioning methods recursively partition the set of states being examined, according to the transition relations of the system.

**Partial Order Methods**

Although both symmetry and partial order take advantage of equivalent structures in a state graph, there is no direct relationship between these two methods. Symmetry considers equivalent relations among different states, whereas partial order considers equivalent relations among different paths. There are cases in which partial order generates no reduction, but symmetry generates a large reduction, and vice versa.

**Other Applications of Symmetry**

Although the symmetry-reduction algorithm in Murφ only verifies simple safety properties and deadlock, symmetry has been applied to other types of verification as well [Eme96]. Huber et al. [HJJ84], Starke [Sta91], and Kensen [Jen96] have applied symmetry in the reachability analysis of Petri Nets. Clarke, Finkin, and Jha [CFJ93, CEFJ96] applied the idea to BDD-based symbolic model checking [BCM+90, CBM89, TSL+90]. Emerson and Sistla [ES93, ES96] applied the idea to CTL* model checking [CES86]. Jackson et al. [JJD96] used a similar idea of an unordered set for software simulation. Indeed, the idea of symmetry reduction is widely applicable to different kinds of systems, models of concurrency, and verification techniques.
Chapter 4

Verifying Data-Independent Systems

Chapter Overview

Data-independent systems can be verified using the same technique described in the previous chapter. Furthermore, this chapter describes how symmetry-reduction leads to a fully automatic verification of data-independent systems with arbitrary, unbounded, or infinite data domains.

4.1 Data-Independence

The cache coherence protocols described in Section 2.2 can be regarded as data-independent systems. The control flow of the protocols does not depend on the actual value of the data in a cache line. No computation is performed on the data value, and the correctness of the protocol depends only on whether two pieces of data are consistent or not.

To better illustrate data-independent systems and the effect of symmetry-reduction on these systems, let’s consider the (somewhat contrived, but simple) producer/consumer example in Figure 4.1. In this example, a state consists of the values in the producer, the channel, and the consumer. Initially, everyone has the same

\footnote{This chapter is based on materials published in [ID93a, ID93b, ID96a].}
data. When the producer has the same value as the consumer, it may produce a new value, as specified by the first rule. The value is then passed to the channel by the second rule, and finally passed to the receiver by the third rule. For this protocol, the data in the channel should be equal to either the one in the producer or the one in the consumer, as specified by the invariant.

Because the operations on $Data$ belong to the scalarset operations described in Section 3.3, it is modeled as a scalarset, and the symmetry reduction can be used to obtain a reduced state graph for verification.

As shown in Figure 4.1, the reduced state graph for $n$ data values, where $n \geq 2$, is isomorphic to the reduced state graph for 2 data values. This phenomenon is called data saturation, and the corresponding state graphs are called saturated models.

### 4.2 Detecting and Exploiting Data-Independence

Data-independent systems can often be described by a scalarset, and the data saturation phenomenon is guaranteed to happen if this scalarset is a data scalarset:

**Definition 4.1 (data scalarset)** A scalarset $\alpha$ is a data scalarset in a source program $P$ if $\alpha$ is not used as array indices or for statement indices.

Because of the strong relationship between the usual notion of data-independence and the symmetry in a data scalarset, we define data-independence as:

**Definition 4.2 (data-independence)** A protocol is data-independent with respect to a particular data domain, if the data domain can be declared as a data scalarset.

The data saturation phenomenon in a system described by a data scalarset can be formalized as follows:

**Theorem 4.1 (data saturation)** If $P$ is a Mur$\phi$ description, $\alpha$ is the name of a data scalarset in $P$, and $P_1$ and $P_2$ are programs identical to $P$ except that $\alpha$ is declared to be of size $N_1$ in $P_1$ and $N_2$ in $P_2$, then there exists a positive integer $N_\alpha$ such that the symmetry-reduced state graphs of $P_1$ and $P_2$ are isomorphic whenever $N_1 \geq N_\alpha$ and $N_2 \geq N_\alpha$. 
4.2. DETECTING AND EXPLOITING DATA-INDEPENDENCE

```plaintext
type Data : scalarset(2);

var Producer : Data; Channel : Data; Consumer : Data;

ruleset v : Data do
    startstate "Initial state with consistent data"
        Producer := v; Channel := v; Consumer := v end

    rule "Producer generates new data"
        Producer = Consumer \Rightarrow Producer := v end

    rule "Send the new data when the producer has a new and different value."
        Producer \neq Channel \Rightarrow Channel := Producer end

    rule "Receive the new data when the channel has a new and different value."
        Channel \neq Consumer \Rightarrow Consumer := Channel end

invariant "Consistent Data"
    (Producer = Channel \mid Channel = Consumer)

Figure 4.1: An example of data independence and data saturation: Because the operations for Data are symmetric, Data is modeled as a scalarset. Furthermore, the symmetry-reduced state graph for a scalarset of size \(n > 2\) is isomorphic to the one for a scalarset of size 2.
```
CHAPTER 4. VERIFYING DATA-INDEPENDENT SYSTEMS

The isomorphic state graphs obtained by symmetry-reduction are called *saturated models*. A complete proof of this theorem is given in Appendix A.3. The essence of the proof can be described as follows:

Because the data scalar set is not used in array indices, regardless of the exact size of the data scalar set, the global variables have a fixed number of variables with the data scalar set type. Let this number be $N$. For example, the system described in Figure 4.1 has three variables with data scalar set types.

Therefore, regardless of the exact size of the scalar set, a state can have at most $N$ different values. We can always find a permutation $\pi$ such that these $N$ values are mapped to the values from 1 to $N$. This permutation generates an equivalent state with values from 1 to $N$ only. For the system described in Figure 4.1, we can use the equivalent state with values 1 and 2 only as the canonical state for a symmetry-equivalence class.

On the other hand, during the execution of a transition rule, a Boolean expression like

$$\text{exists } i_1, i_2, \ldots, i_k : \text{data } \text{do } i_1 \neq i_2 \neq \ldots \neq i_k \text{ end}$$

can be used to distinguish whether or not the size of the scalar set is no less than $k$. Taking the $N$ global variables of the data scalar set type into account, a Boolean expression can be used to distinguish whether or not the data scalar set has $N + k$ elements or more. However, it cannot distinguish whether or not the data scalar set has $N + k$ elements or $N + k + 1$ elements, and similarly for data scalar set of larger sizes.

Although the bound given in the intuition is based on the number of data scalar set locations in global variables and the bounded variables, data saturation typically happens for a much smaller data scalar set. A simple run-time check can be used to determine when data saturation has occurred. Consider a Mur$\varphi$ description with a maximum of $k$ bounded variables of a data scalar set type. If the data scalar set has
size \( z \), and a state has \( n \) distinct data values, the verifier can compare \( z \) and \( n + k \) to check for saturation. If for all reachable states in the reduced state graph, \( z \) is smaller than \( n + k \), the reduced state graph is not a saturated model. Otherwise, the verification result is valid for all systems with data scalarset size larger than \( z \).

Therefore, it is possible to add a declaration for a scalarset of unknown cardinality to the description language, and to verify any system described by such scalarsets using symmetry reduction:

**Theorem 4.2** A system with a data scalarset of unspecified size can be verified by the verification of \( n \) systems, having data scalarsets of size 1 to \( n \), where \( n \) is size of the scalarset when saturation occurs.

### 4.3 Implementation and Results

#### Implementation

The only change to the symmetry reduction algorithm is the addition of the run-time check for saturation.

#### Verification Results

As shown in Table 4.1, the verification results confirmed the data saturation phenomenon. Both DASH-C and ICCP become saturated as the size of the data domain increases, and the reduced state graphs for systems with 4 or more data values are isomorphic.

### 4.4 Comparison with Other Work

The approach presented in this chapter is analogous to the approach suggested by Aggarwal et al. [AKS83] and Wolper [Wol86], which are summarized in Section 1.2.3. But their work requires the user to recognize that a protocol is data-independent and to transform the description manually in order to exploit data independence. In contrast to their methods, the use of data scalarsets requires no extra work for the user. The finite state space construction can be done automatically by the verifier, and the soundness of the algorithm is guaranteed.
Table 4.1: Data saturation: As the size of the data scalarset increases beyond 4, the size of the reduced state graph stops increasing. The reduced state graphs for data scalarsets of sizes beyond 4 are actually isomorphic.

Recently, Hojati and Brayton [HB95] exploited a similar idea to data path abstraction. Their definition of data-independent circuits can be considered as a particular instance of our definition, in which equality testing of two pieces of data is only allowed in the properties to be verified, but not in the transitions of the system. On the other hand, they have extended the idea of data-independence to semi-data-independence, in which other comparisons between two pieces of data are allowed. In both cases, they have proven theorems that are similar to the data saturation theorem: Their theorems state that systems with sufficiently large data domains have the same error behavior.
Chapter 5

Verifying Systems with Reversible Rules

Chapter Overview

This chapter defines reversible rules, and describes how they can be exploited in automatic formal verification. The key property of a reversible rule is that the original state before executing the rule can be re-constructed from the successor after executing the rule. Because of this property, the successor does not need to be stored in the memory; the original state can be re-constructed to represent the successor.

Two improvements to this basic scheme are also presented to remove false error reports and to reduce verification time.

5.1 Reversible Rules

In this section, we illustrate the key property of reversible rules by an example in a cache coherence protocol. As shown in Figure 5.1, suppose a processor has an invalid cache entry for a memory location and there is no outstanding request. The processor has two choices of actions: it can send a request for a shared copy, or send a request for an exclusive copy of the memory location. After either of these two transitions, the original state can be reconstructed by removing the message from the network and

---

\(^1\)This chapter is based on materials published in [ID96b].
changing the internal state of the processor back to invalid. Therefore, the transition rules corresponding to these transitions are called reversible rules.

Reversible rules often contribute to the state explosion in the state graph, because they are often local to the processors, that is, the executions of these rules are not affected by the environment of the processor. For example, the two reversible rules shown in Figure 5.1 generate $3^n$ states from a state of $n$ processors with invalid cache lines\(^2\). As shown in Figure 5.2, many similar subgraphs are generated by these reversible rules, and each subgraph is originated from a unique state.

The unique state from which each subgraph is generated is called the *progenitor* of the subgraph. The other states in the subgraph are called *transient states*. In order to reduce the memory usage in verification, transient states are not stored in the memory. When a transient state is generated during the construction of the state graph, we can reverse-execute the reversible rules a few times to reconstruct the

---

\(^2\)If the processors are symmetric, $(n+1)(n+2)/2$ equivalence classes are generated.
5.2. EXPLOITING REVERSIBLE RULES

In order to use the idea of reversible rules in automatic formal verification, we need to define precisely the required property of reversible rules. Based on this definition, we can design an algorithm to generate a reduced state graph in such a way that an error state is reachable in the original state graph only if a corresponding error state is reachable in the reduced state graph.
First of all, a reversible rule allows us to re-construct the original state before executing the rule:

**Definition 5.1 (reversible rule)** If a rule set \( T \) generates a state graph \( \langle Q, Q_0, \Delta, \text{error} \rangle \), a rule \( r \in T \) is a reversible rule if and only if

- for all \( q \in Q \), if \( q \neq \text{error} \), then \( r(q) \neq \text{error} \).
- there exists a function \( r^* \) such that for all \( q \in Q \), if there exists a unique \( q' \in Q \) such that \( q' \neq q \land r(q') = q \), then \( r^*(q) = q' \). Otherwise, \( r^*(q) = q \).
- there exists an integer \( n \) such that for all \( q \in Q \), we have \( (r^*)^{n+1}(q) = (r^*)^n(q) \).

As mentioned in the last section, the executions of these rules often depend only on the local state of a processor. Furthermore, the execution of a reversible rule usually won’t enable another reversible rule; in fact, it often disables another reversible rule. This property is captured as follows:

**Definition 5.2 (commutative reversible rule set)** If a rule set \( T \) generates a state graph \( \langle Q, Q_0, \Delta, \text{error} \rangle \), the subset \( U \subseteq T \) is a commutative reversible rule set if and only if

- every rule in \( U \) is a reversible rule; and
- for all \( q \in Q \) and \( r_1, r_2 \in U \), we have \( r_1^* (r_2^*(q)) = r_2^* (r_1^*(q)) \).

Although this property often results in large subgraphs in the original state graph, it also allows us to find a unique representative for each subgraph, called the progenitor, which eliminates the need to store the whole subgraph in memory:

**Definition 5.3 (progenitors)** If a rule set \( T \) generates a state graph \( \langle Q, Q_0, \Delta, \text{error} \rangle \), and \( U \subseteq T \) is a commutative reversible rule set, a state \( q \in Q \) is a progenitor if and only if for every \( r \in U \), \( r^*(q) = q \). A progenitor \( q \) is a progenitor of a state \( q' \) if and only if there exist \( r_1, ..., r_n \in U \), such that \( q \stackrel{r_1,...,r_n}{\rightarrow} q' \).

For example, if every reversible rule generates a new request message, a state with no request in the network is a progenitor.

The properties of a commutative reversible rule set guarantee that the progenitor is unique, and we denote the corresponding progenitor for a state \( q \) as \( \theta(q) \). The states that are not progenitors are called transient states.
5.2. EXPLOITING REVERSIBLE RULES

Theorem 5.1 (uniqueness of progenitor) If a rule set $T$ generates a state graph $(Q, Q_0, \Delta, error)$, and $U = \{r_1, \ldots, r_m\} \subseteq T$ is a commutative reversible rule set, there exists integer $n$ such that for all $q \in Q$, the unique progenitor $\theta(q)$ for $q$ is $(r_m^*)^n \ldots ((r_1^*)^n(q)) \ldots$.

Proof. First of all, we choose $n$ to be the integer such that for all $r_i \in U$ and $q \in Q$, $(r_i^*)^{n+1}(q) = (r_i^*)^n(q)$, and define $\theta(q)$ to be $(r_m^*)^n \ldots ((r_1^*)^n(q)) \ldots$. We can show that $\theta(q)$ is a progenitor of $q$ and that it is unique.

Progenitor: Because $U$ is a commutative reversible rule set, we can rearrange the order of application of $r_i^*$ such that for all $1 \leq i \leq m$,

$$\theta(q) = (r_i^*)^n \ldots ((r_{i+1}^*)^n \ldots ((r_{i-1}^*)^n \ldots ((r_1^*)^n(q)) \ldots)$$

Because for all $q' \in Q$, $(r_i^*)^{n+1}(q') = (r_i^*)^n(q')$, we have $r_i^*(\theta(q)) = \theta(q)$. Therefore $\theta(q)$ is in fact a progenitor.

Uniqueness: Given any progenitor $q'$ of $q$, there exist $r_1', \ldots, r_k' \in U$ such that $q' = r_k'(\ldots(r_1'(q))\ldots)$. Since we can rearrange the order of application of $r_i^*$, we can group the identical rules together and get $q' = (r_m^*)^{k_m} \ldots ((r_1^*)^{k_1}(q)) \ldots$ for some $k_i \geq 0$. There are two cases to consider:

- $k_m < n$:
  
  Since $q'$ is a progenitor, we have $q' = (r_m^*)^{n-k_m}(q')$ Therefore, $q' = (r_m^*)^{n-k_m} \ldots ((r_1^*)^{k_1}(q)) \ldots = (r_m^*)^n \ldots ((r_1^*)^{k_1}(q)) \ldots$ Repeat the argument for every $r_i \in U$, we have $q' = (r_m^*)^n \ldots ((r_1^*)^n(q)) \ldots = \theta(q)$.

- $k_m > n$:
  
  Since, for all $p \in Q$ and $1 \leq i \leq k$, $r_i^{n+1}(p) = r_i^n(p)$, we have $q' = (r_m^*)^{n+(k_m-n)} \ldots ((r_1^*)^{k_1}(q)) \ldots = (r_m^*)^n \ldots ((r_1^*)^{k_1}(q)) \ldots$. Repeat the argument for every $r_i \in U$, we have $q' = (r_m^*)^n \ldots ((r_1^*)^n(q)) \ldots = \theta(q)$.

$\square$
The progenitor $\theta(q)$ is used to represent the set of states reachable from $\theta(q)$ via the reversible rules. The resulting state graph is defined as follows:

**Definition 5.4 (reduced state graph by progenitors)** If a rule set $T$ generates a state graph $A = \langle Q, Q_0, \Delta, \text{error} \rangle$, and $U \subseteq T$ is a commutative reversible rule set, the reduced state graph is $A_U = \langle \theta(Q), \theta(Q_0), \theta(\Delta), \text{error} \rangle$, such that:

- $\theta(Q) = \{ \theta(q) \mid q \in Q \}$;
- $\theta(Q_0) = \{ \theta(q) \mid q \in Q_0 \}$; and
- $(q_1, q_2) \in \theta(\Delta)$ if and only if there exist $q \in Q$ and $t \in T \setminus U$ such that $\theta(q) = q_1$ and $\theta(t(q)) = q_2$.

The correctness of a verification performed on this reduced state graph is guaranteed, because every path in the original state graph has a corresponding path in the reduced state graph, as shown below as Lemma 5.1 and Theorem 5.2.

In order to simplify the subsequent lemmas and theorems in this chapter, we adapt the following conventions: we use (subscripted) $q$ to represent a state in $Q$, (subscripted) $r$ to represent a rule in $U$, (subscripted) $t$ to represent a rule in $T \setminus U$, and (subscripted) $k$ to represent a non-negative integer. Furthermore, we denote a transition in the reduced state graph as $q_1 \xrightarrow{t} q_2$ if there exists $q \in Q$ such that $\theta(q) = q_1$ and $\theta(t(q)) = q_2$.

**Lemma 5.1** If a rule set $T$ generates a state graph $\langle Q, Q_0, \Delta, \text{error} \rangle$, and $U \subseteq T$ is a commutative reversible rule set, whenever $q_0 \xrightarrow{r_1 \ldots r_k t} q$ is in $A$, we have $\theta(q_0) \xrightarrow{t} \theta(q)$ in $A_U$.

**Proof.** This follows directly from the definition of $\theta(\Delta)$.  

**Theorem 5.2 (soundness)** If a rule set $T$ generates a state graph $A = \langle Q, Q_0, \Delta, \text{error} \rangle$, and $U \subseteq T$ is a commutative reversible rule set, whenever $q$ is reachable from the initial states in $A$, $\theta(q)$ is also reachable from the initial states in $A_U$. 

Proof. Consider the following path from the initial state $q_0$ in the original state graph:

$$
q_0 \xrightarrow{r_1 \ldots r_{k_1}, t_1} q_1 \xrightarrow{r_{k_1+1} \ldots r_{k_2}, t_2} \ldots \xrightarrow{r_{m-1} \ldots r_m, t_m} q_m \xrightarrow{r_{m+1} \ldots r_{m+1}} q
$$

for some integers $k_i \geq 0$. By Lemma 5.1, $\theta(q_0) \xrightarrow{t_1} \theta(q_1) \xrightarrow{t_2} \ldots \xrightarrow{t_m} \theta(q_m)$ is a path from the initial state $\theta(q_0)$ in $A_U$ that reaches $\theta(q)$. 

\[\blacksquare\]

Corollary 5.1 If error is reachable from the initial states in $A$, error is also reachable from the initial states in $A_U$.

The on-the-fly reduction algorithm for error and invariant checking is shown in Figure 5.3. Every transient state represented by a progenitor is generated using a local search in the procedure \textit{Local\_Search()}. The successors of the progenitor and the transient states are generated using the rules in $T \setminus U$. The progenitors of these successors are then compared to the progenitors in the hash table to check whether or not they have been examined before.

This reduced state graph is a conservative approximation of the original state graph. If the reversible rules preserve all atomic prepositions in a stuttering-invariant LTL and $\forall$CTL* formula $f$, the original state graph satisfies $f$ if the reduced state graph satisfies $f$. For the notion of stuttering-invariant, see [Lam83]. The proof for stuttering-invariant LTL model checking follows directly from Lemma 5.1, and the proof for stuttering-invariant $\forall$CTL* model checking is similar to the one presented in [BBG+93].

For deadlock detection, every state in a subgraph generated from a progenitor can be checked explicitly during the local search to see if it is a deadlock state in the original state graph.

5.3 Improvements

Although the algorithm described in the previous section significantly reduces memory usage, it still has two problems. First of all, a reachable error state in the reduced state graph may not correspond to a reachable error state in the original state graph, resulting in a false error report. As shown in Figure 5.4, although a transient state is
Algorithm 1

begin
   Reached = Unexpanded = \{ \theta(q) \mid q \in Q_0 \};
   while Unexpanded \neq \emptyset do
      Remove a state s from Unexpanded;
      Local\_Search(s);
   endwhile;
end

Local\_Search(state s)

begin
   Local\_Reached = Local\_Unexpanded = \{s\};
   while Local\_Unexpanded \neq \emptyset do
      Remove a state s from Local\_Unexpanded;
      Generate\_Original\_Next\_States(s);
      for each transition rule r \in U do
         let s' = r(s) in
         if s' is not in Local\_Reached then
            Put s' in Local\_Reached and Local\_Unexpanded;
            endif;
      endlet;
   endwhile;
end

Generate\_Original\_Next\_States(state s)

begin
   for each transition rule t \in T \setminus U do
      let s' = t(s) in
      if s' = error then stop and report error; endif;
      if \theta(s') is not in Reached then
         Put \theta(s') in Reached and Unexpanded;
      endif;
   endlet;
endfor;
end

Figure 5.3: An on-the-fly reduction algorithm for reversible rules: The part highlighted by an underline or a left vertical line represents the main difference of this algorithm from the basic algorithm shown in Figure 2.2.
5.3. **IMPROVEMENTS**

![Diagram](image)

Figure 5.4: Potential false error report from reduction using reversible rules: Although a transient state is reachable from an initial state, its progenitor may not be reachable, and it may lead to a false error report.

reachable from an initial state, its progenitor may not be reachable, and it may lead to a false error report. Secondly, the time required to construct the reduced state graph is longer than the time required to generate the original state graph, since every reachable state in the original state graph is generated and examined explicitly during the local search.

This section describes two improvements to solve these problems.

### 5.3.1 Removing False Error Reports

Under an extra condition, called the *essential property*, the reduction algorithm is guaranteed not to report false errors. The essential property enforces that for every state pair \( q, q' \) and every essential rule \( r \) such that \( r(q) = q' \), if \( q' \) is reachable from the initial states, \( q \) is also reachable from the initial states:

**Definition 5.5 (essential rule)** If a rule set \( T \) generates a state graph \( \langle Q, Q_0, \Delta, \text{error} \rangle \), a commutative reversible rule set \( U \subseteq T \) is an essential rule set if and only if

- for all \( q \in Q_0 \), we have \( \theta(q) \in Q_0 \).

- for all \( r \in U, t \in T \setminus U \), if there exist distinct \( q_0, q_1, q'_1 \neq \text{error} \) such that \( q_0 \xrightarrow{t} q_1 \) and \( q'_1 \xrightarrow{r} q_1 \), then there exists \( q'_0 \in Q \) such that \( q'_0 \xrightarrow{r} q_0 \) and \( q'_0 \xrightarrow{t} q'_1 \).
Pictorially, the essential property can be summarized as:

\[
q_0 \xrightarrow{t} q_1 \quad \Rightarrow \quad q_0 \xrightarrow{t} q_1
\]

Lemma 5.2 If a rule set \( T \) generates a state graph \( \langle Q, Q_0, \Delta, \text{error} \rangle \), and \( U \subseteq T \) is a commutative reversible and essential rule set, whenever \( q_0 \xrightarrow{t} q_1 \) is in \( A_U \), there exist \( r_1, \ldots, r_k \in U \) such that \( q_0 \xrightarrow{r_1,\ldots,r_k} q_1 \) is in \( A \).

Proof. Since \( q_0 \xrightarrow{t} q_1 \) is in the reduced graph, there exist states \( q'_0, q'_1 \) such that \( \theta(q'_0) = q_0, \theta(q'_1) = q_1 \), and \( q'_0 \xrightarrow{t} q'_1 \).

If \( q_1 \xrightarrow{r'_1,\ldots,r'_z} q'_1 \), by recursively applying the essential property on \( r'_1, \ldots, r'_z \), there exists a state \( q''_0 \) such that \( q''_0 \xrightarrow{r'_1,\ldots,r'_z} q'_0 \) and \( q'_0 \xrightarrow{t} q_1 \). Consider the progenitor \( \theta(q''_0) \) of \( q''_0 \), there exists \( r_1, \ldots, r_k \) such that \( \theta(q''_0) \xrightarrow{r_1,\ldots,r_k} \theta(q'_0) \). Therefore, \( \theta(q''_0) \xrightarrow{r_1,\ldots,r_k} q'_0 \) and \( \theta(q''_0) = \theta(q'_0) = q_0 \). Hence, \( q_0 \xrightarrow{r_1,\ldots,r_k,t} q_1 \).

Pictorially, it can be illustrated as:

\[
\begin{array}{c}
q'_0 \xrightarrow{t} q'_1 \\
q_1 \xrightarrow{r'_1,\ldots,r'_z} q''_0 \xrightarrow{t} q_1 \\
q_0 \xrightarrow{r_1,\ldots,r_k} q_1
\end{array}
\]

Therefore, every state in the reduced state graph represents a reachable subgraph in the original state graph:

Theorem 5.3 (completeness) If a rule set \( T \) generates a state graph \( A = \langle Q, Q_0, \Delta, \text{error} \rangle \), and \( U \subseteq T \) is a commutative reversible and essential rule set, whenever \( q \) is reachable from the initial states in \( A_U \), \( q \) is also reachable from the initial states in \( A \).
5.3. IMPROVEMENTS

Proof. If $q_0 \xrightarrow{t_1} q_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} q$ is a path from the initial state $q_0$ in the reduced state graph, then by Lemma 5.2, we can find rules in $U$ such that $q_0 \xrightarrow{r_{k_1}, t_1} q_1 \xrightarrow{r_{k_1} + 1, \ldots, r_{k_2}, t_2} \ldots \xrightarrow{r_{k_{m-1}} + 1, \ldots, r_{k_m}, t_m} q$ is a path from the initial state $q_0$ in the original state graph.

\[ \square \]

Corollary 5.2 If $\text{error}$ is reachable from the initial states in $A_U$, $\text{error}$ is also reachable from the initial states in $A$.

Therefore, the same reduced state graph and the same algorithm shown in Figure 5.3 can be used for verification using a commutative, reversible, and essential rule set, with no false error reports. The correctness of the algorithm is given below:

Theorem 5.4 (soundness and completeness) If a rule set $T$ generates a state graph $A = \langle Q, Q_0, \Delta, \text{error} \rangle$, and $U \subseteq T$ is a commutative reversible and essential rule set, the reduced state graph $A_U$ is sound and complete for error and invariant checking.

Because of Lemma 5.1 and Lemma 5.2, the reduced state graph is sound and complete for stuttering-invariant LTL model checking. However, because of the lost of branching information within the subgraphs, the reduced state graph is still limited to $\forall\text{CTL}^*$ model checking. Fortunately, because the reduced state graph contains no unreachable state, it is a good approximation of the original state graph. Furthermore, a deadlock state is not reachable in the original state graph if and only if none of the states examined in the local search phase is a deadlock state.

5.3.2 Speeding Up the Reduction Algorithm

In the algorithm shown in Figure 5.3, every reachable transient state in the original state graph is generated during the local search phase of the algorithm, so the time required to generate the reduced state graph is longer than the time required to generate the original state graph. In this section, we present a condition, called singular property, which allows most of the transient states in the subgraph to be completely ignored during the construction of the reduced state graph. This reduces the execution time of the algorithm by more than 70% in some cases.
Figure 5.5: Singular property in a cache coherence protocol: The second transition on rule $t$ is redundant in the reduced state graph.

To illustrate the approach, consider once more the state graph of a cache coherence protocol in Figure 5.5. There are redundant transitions between subgraph $A$ and subgraph $B$: the transition $t$ can be taken both from $(w, I)$ and from $(w, w)$. In this case, it is not necessary to consider $(w, w)$ to generate the reduced state graph: every transition from $(w, w)$ is redundant.

The intuition is that the result of a transition often depends only on the immediate execution of at most one reversible rule. For example, consider a message-passing protocol in which each transition checks and removes at most one message from a network, and every reversible rule generates at least one message. If a transition $t$ depends only on the message generated by a reversible rule $r_1$, another reversible rule $r_2$ can be executed or reverse-executed without affecting the execution of $t$. Therefore, we only need to consider the transient states that are generated by exactly one reversible rule, and we can ignore the transient states that are generated by applying more than one reversible rule in sequence. The precise definition of this property is as follows:

**Definition 5.6 (singular property)** If a rule set $T$ generates a state graph $A = \langle Q, Q_0, \Delta, \text{error} \rangle$, a commutative reversible rule set $U \subseteq T$ is singular w.r.t. $T$ if and only if whenever $q_1 \xrightarrow{r_1,r_2} q_2$, we have either $q_1 \xrightarrow{r_1,t,r_2} q_2$ or $q_1 \xrightarrow{r_2,t,r_1} q_2$. 

\[ q_1 \xrightarrow{r_1,r_2} q_2 \]
Therefore, if there exists a transition between two subgraphs represented by two progenitors, there also exists a transition from the first progenitor or its immediate successor to the second subgraph:

**Lemma 5.3** If a rule set $T$ generates a state graph $A = (Q, Q_0, \Delta, \text{error})$, and a commutative reversible rule set $U \subseteq T$ is singular w.r.t. $T$, whenever $q \xrightarrow{r_1, \ldots, r_k, t} q'$, there exists $q''$ and $r_j$ such that $q \xrightarrow{r_j, t} q''$ and $\theta(q'') = \theta(q')$.

**Proof.** We prove the lemma by induction on $k$. In the base case, we have $q \xrightarrow{r_1, t} q'$ implies $q \xrightarrow{r_1, t} q'$, which is trivially true.

Assume the lemma is true for $k$, i.e., $q \xrightarrow{r_1, \ldots, r_k, t} q'$ implies $q \xrightarrow{r_j, t} q''$ for some integer $j$ such that $1 \leq j \leq k$ and $\theta(q'') = \theta(q')$.

Consider $q \xrightarrow{r_1, \ldots, r_k, t} q'$. There exists $p_1$ such that $q \xrightarrow{r_1} p_1 \xrightarrow{r_2, \ldots, r_k, t} q'$. By the induction hypothesis, there exists $p_2$ and some integer $j$ such that $2 \leq j \leq k + 1$, and $q \xrightarrow{r_1} p_1 \xrightarrow{r_j, t} p_2$, and $\theta(p_2) = \theta(q')$.

Because of the singular property, there exists $p_3$ such that $\theta(p_3) = \theta(p_2) = \theta(q')$, and either $q \xrightarrow{r_1, t} p_3$, or $q \xrightarrow{r_j, t} p_3$.

Lemma 5.3 implies that it is sufficient to apply the transition rules $T \setminus U$ only to the progenitors and their immediate successors:

**Theorem 5.5 (fast reduced state graph generation)** If a rule set $T$ generates a state graph $A = (Q, Q_0, \Delta, \text{error})$, and a commutative reversible rule set $U \subseteq T$ is singular w.r.t. $T$, $(q_1, q_2) \in \theta(\Delta)$ if and only if there exists $t \in T \setminus U$ such that $\theta(t(q_1)) = q_2$ or there exists $r \in U$ such that $\theta(t(r(q_1))) = q_2$.

Hence, the algorithm can be sped up to the one shown in Figure 5.6. As shown in Table 5.1, the practical results for ICCP confirmed that fewer states were examined and the verification finished in a much shorter time.

While this reduced state graph can be used for stuttering-invariant LTL model checking, a slight modification is needed for deadlock detection. In order to detect deadlock, all transient states must be generated and checked if it is a deadlock state in the original state graph. However, once it is determined that a transient state has a
Algorithm 2()
begin
    Reached = Unexpanded = \{ \theta(q) \mid q \in Q_0 \};
    while Unexpanded \neq \emptyset do
        Remove a state s from Unexpanded;
        Local Search(s);
    endwhile;
end

Local Search(state s)
begin
    Local Reached = \{s\}
    for each transition rule r \in U do
        let s' = r(s) in
        if s' is not in Local Reached then
            Put s' in Local Reached
            Generate Original Next States(s)
        endif
    endlet;
endfor;
end

Generate Original Next States(state s)
begin
    for each transition rule t \in T \setminus U do
        let s' = t(s) in
        if s' = error then stop and report error; endif;
        if \theta(s') is not in Reached then
            Put \theta(s') in Reached and Unexpanded;
        endif
    endlet;
endfor;
end

Figure 5.6: An on-the-fly reduction algorithm for singular reversible rules: The part highlighted by a left vertical line represents the main difference of this algorithm from the algorithm in Figure 5.3.
successor other than itself, the algorithm does not need to execute the non-reversible rules (and apply \( \theta \)) to find their next state.

## 5.4 Combining Reversible Rules and Symmetry

Reductions using reversible rules and symmetry can be combined to obtain even greater reduction. In order to combine the two techniques, a symmetric reversible rule set must be used:

**Definition 5.7 (symmetric rule set)** If a rule set \( T \) generates a state graph \( A = (Q, Q_0, \Delta, \text{error}) \), and \( H \) is a set of automorphisms on \( A \), a rule set \( U \subseteq T \) is symmetric if and only if for all \( q_1, q_2 \in Q, r \in U \) and \( h \in H \), whenever \( q_1 \xrightarrow{r} q_2 \), there exists \( r' \in U \) such that \( h(q_1) \xrightarrow{r'} h(q_2) \).

**Theorem 5.6** A set of automorphisms \( H \) on the state graph \( A \) is also a set of automorphisms on \( A_U \) if \( U \) is a commutative, reversible, and symmetric rule set.

**Proof.** If \( q \) is a progenitor, for all \( r \in U \), \( r_*(q) = q \). Because \( U \) is a symmetric rule set, for all \( r \in U \), \( r_*(h(q)) = q \) and \( h(q) \) is in fact a progenitor.

On the other hand, for all \( h \in H \) and \( t \in T \setminus U \), if \( q_0 \xrightarrow{t} q_1 \) is in the reduced state graph, there exists \( r_1, \ldots, r_k \in U \) such that \( q_0 \xrightarrow{r_1,\ldots,r_k,t} q_1 \) is in the original state graph. By the definition of automorphism, there exists \( r'_1, \ldots, r'_k \in U \) and \( t' \in T \setminus U \) such that

<table>
<thead>
<tr>
<th>ICCP: 4 processors</th>
<th>states stored</th>
<th>states examined</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (unordered network)</td>
<td>247,565</td>
<td>247,565</td>
<td>205s</td>
</tr>
<tr>
<td>Alg. 1</td>
<td>34,005</td>
<td>247,565</td>
<td>338s</td>
</tr>
<tr>
<td>Alg. 2</td>
<td>34,005</td>
<td>123,197</td>
<td>128s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ICCP: 5 processors</th>
<th>states stored</th>
<th>states examined</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (unordered network)</td>
<td>&gt; 6,500,000 states</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alg. 1</td>
<td>492,075</td>
<td>6,568,279</td>
<td>4 hours</td>
</tr>
<tr>
<td>Alg. 2</td>
<td>492,075</td>
<td>2,206,135</td>
<td>66 mins</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of the two reduction algorithms using reversible rules: Although the same reduced state graphs are obtained by both algorithms, the second algorithm examined fewer states and finished in less than 30% of the time required by the first algorithm.
h(q_0) \overset{r_1 \cdots r_k}{\Rightarrow} h(q_1) \text{ is in the original state graph. Therefore, } h(q_0) \overset{r}{\Rightarrow} h(q_1) \text{ is in the reduced state graph.}

Therefore, we can apply symmetry reduction on the reduced state graph obtained by reversible rules:

**Definition 5.8 (combined reduced state graph)** If a rule set $T$ generates a state graph $A = (Q, Q_0, \Delta, \text{error})$, $H$ is a set of automorphisms on $A$, and $U \subseteq T$ is a commutative, reversible, and symmetric rule set, the combined reduced state graph is $[A_U]_H = (\theta'(Q), \theta'(Q_0), \theta'(\Delta), \text{error}')$, defined as:

1. $\theta'(Q) = \{[\theta(q)] \mid q \in Q\}$

2. $\theta'(Q_0)' = \{[\theta(q)] \mid q \in Q_0\}$

3. $(q_1, q_2) \in \theta'(\Delta)$ if and only if there exist $q \in Q$ and $t \in T \setminus U$ such that $[\theta(q)] = q_1$ and $[\theta(t(q))] = q_2$.

### 5.5 Implementation and Results

**Implementation**

In the current implementation, the reversible rules are identified manually, and the user provides special reversed rules to re-construct the original state before executing the rule. The properties of these reversible rules are also checked manually. Section 5.6 describes how these properties could be determined by the verifier.

In the new algorithm, there is no change to the global hash table structure, which is only used for storing the progenitors. The extra memory used in the first algorithm consists of the temporary variables during execution of $\theta$, and a temporary hash table during the local search, which can be as small as required to store the largest subgraph in the original state graph. The extra memory for the second algorithm is even smaller; only up to $n + 1$ states are stored temporarily during the local search, where $n$ is the size of the reversible rule set. Because of the local search phase and the execution of $\theta$ to find the progenitor, the time required per state generated in the reduction algorithm is longer than that in the original algorithm.
### 5.6 Checking the Properties of Reversible Rules

In the current implementation, the reversible rules are identified and checked manually, and the reversed rules are provided by the user. This section discusses how this process could be performed automatically by a verifier. Because of the absent of channel primitives and implicit program locations (discussed below) in Mur$\varphi$, and the many possibilities of detecting reversible rules, this process is not automated in Mur$\varphi$ yet.

#### Verification Results

The reduction obtained by reversible rules depends on the number of reversible rules in the system, and how many of them can be enabled at the same time. For ICCP, a processor with an invalid cache line has two reversible choices to request for a shared or exclusive copy; a processor with a shared cache line has two reversible choices to request for promoting the shared copy to a exclusive copy or for removing the shared copy; a processor with an exclusive copy has one reversible choice to write back the data. Because of these five reversible rules, a large reduction in size and time is obtained, as shown in Table 5.2 and Table 5.3.

For LIST1 and LIST2, there are two situations, each with one reversible choice; the space reductions obtained are still quite large. DASH-C has only one situation with two reversible choices and DASH-L has only one situation with one reversible choice; therefore, the reductions are not as large as the other applications.

#### Table 5.2: Improvement on performance with singular reversible rules

<table>
<thead>
<tr>
<th></th>
<th>ICCP (p4) (unordered network)</th>
<th>LIST1 (P4)</th>
<th>LIST2 (p4) w/o DMA (p3)</th>
<th>DASH-C (P3)</th>
<th>DASH-L (p3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original size</td>
<td>247,365</td>
<td>301,029</td>
<td>329,601</td>
<td>26,925</td>
<td>55,366</td>
</tr>
<tr>
<td>Reduced size</td>
<td>34,005</td>
<td>112,784</td>
<td>162,736</td>
<td>15,751</td>
<td>36,728</td>
</tr>
<tr>
<td>Original time</td>
<td>205s</td>
<td>87s</td>
<td>250s</td>
<td>114s</td>
<td>188s</td>
</tr>
<tr>
<td>Reduced time</td>
<td>128s</td>
<td>72s</td>
<td>239s</td>
<td>85s</td>
<td>213s</td>
</tr>
</tbody>
</table>

$p\langle n \rangle$ : n-processor system
CHAPTER 5. VERIFYING SYSTEMS WITH REVERSIBLE RULES

<table>
<thead>
<tr>
<th></th>
<th>ICCP (p4)</th>
<th>LIST1 (p4)</th>
<th>LIST2 (p4)</th>
<th>DASH-C w/o DMA (p3)</th>
<th>DASH-L (p3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduced size (s)</td>
<td>11,814</td>
<td>13,044</td>
<td>13,959</td>
<td>4,575</td>
<td>9,313</td>
</tr>
<tr>
<td>Reduced size (s &amp; r)</td>
<td>1,760</td>
<td>4,926</td>
<td>6,894</td>
<td>2,672</td>
<td>6,170</td>
</tr>
<tr>
<td>Reduced time (s)</td>
<td>28s</td>
<td>13s</td>
<td>24s</td>
<td>63s</td>
<td>96s</td>
</tr>
<tr>
<td>Reduced time (s &amp; r)</td>
<td>13s</td>
<td>11s</td>
<td>22s</td>
<td>50s</td>
<td>96s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>p4</th>
<th>p5</th>
<th>p6</th>
<th>p7</th>
<th>p8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original size</td>
<td>247,565</td>
<td>&gt; 6,500,000 states</td>
<td>&gt; 1,500,000 states</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduced size (s)</td>
<td>11,814</td>
<td>68,879</td>
<td>358,078</td>
<td>49,045</td>
<td>121,302</td>
</tr>
<tr>
<td>Reduced size (s &amp; r)</td>
<td>1,760</td>
<td>6,021</td>
<td>18,118</td>
<td>49,045</td>
<td>121,302</td>
</tr>
<tr>
<td>Original time</td>
<td>205s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Reduced time (s)</td>
<td>28s</td>
<td>349s</td>
<td>3,762s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Reduced time (s &amp; r)</td>
<td>13s</td>
<td>98s</td>
<td>61.5s</td>
<td>3,283s</td>
<td>12,801s</td>
</tr>
</tbody>
</table>

s : Symmetry Reduction  
r : Reversible Rules Reduction  
p(n) : n-processor system

Table 5.3: Improvement on performance with both symmetry and reversible rules: Even greater reductions are obtained.

First of all, assume that the reversed rules are provided by the user, and that the reversed rules do indeed construct the original state before executing the reversible rules, we still need to check whether or not the set of reversible rules are commutative, essential and singular.

The determination of the commutative and singular properties can be facilitated by the notion of channels. A channel serves as an internal buffer where values can be stored. The two operations allowed are send and receive. Although channels are not implemented in Murφ, this concept can be implemented easily with an array.

Similar to the way that channels are important in the detection of the independence property for partial order reduction in SPIN [HGP92], they are also important in reduction using reversible rules. In partial order reduction, a send and a receive on the same channel are regarded as independent. In reduction using reversible rules, channels are used as serialization points, that is, a receive depends only on the previous execution of exactly one send. This is captured as the singular property. The commutative property makes use of symmetry in a unordered channel, which could probably be used in partial order reduction as well.
A sufficient but not necessary condition for each of the commutative, essential, and singular properties is summarized as follows:

**commutative:** Two reversible rules $r_1$ and $r_2$ are commutative if one of the following is true:

- the variables accessed by $r_1$ and $r_2$ are disjoint;
- the only common variables accessed by both $r_1$ and $r_2$ are channels in which the messages can be received out of order, and each rule only sends messages to these channels; or
- the execution of each rule disables the other rule.

**essential:** A rule $r$ is essential if both of the following are true:

- only $r$ assigns a certain value to a particular variable; and
- all other rules that check the variables written by $r$ also change the value of at least one of these variables.

**singular:** A reversible rule set $U$ is singular if for all rules $r_1, r_2 \in U$, and for all rules $t \in T \setminus U$, one of the following is true:

- the variables accessed by $t$ and one of the rules in $\{r_1, r_2\}$ are disjoint; or
- the only common variables accessed by $r_1, r_2$, and $t$ are channels, $t$ only receives from the channels once, and both $r_1$ and $r_2$ only send to the channels.

In the first case, because the rule $t$ only receives from the channels once, the channels serialize the messages from the two reversible rules, and only one message is visible to $t$.

On the other hand, reversible rules and the corresponding reversed rules can be detected automatically in many situations. The main difficulty in reconstructing the original state is that the original value of a variable before an assignment is lost. In some situations, the original value is implicitly specified in the program:
• In the case of concurrent program verification, each program has a program counter. In many cases, there is only one way to reach a certain location in a program. For example, consider an if statement "l₁ : if (x = 2) then l₂ : x := 3; end l₃ ;" where l₁, l₂, l₃ are labels for the corresponding locations. The only way to get to a state with l₂ active is to go through l₁. Therefore, given a state with x = 2 and l₂ active, we can replace l₂ with l₁ to reconstruct the state before executing the guard of the program.

• Consider the guarded command "rule x = 2 ⇒ x := 3; end", the original value of x before the execution of this rule must be 2. Therefore, given a state with x = 3, we can reconstruct the state before executing the rule by changing the value of x back to 2. Other similar cases include adding a new message into a network array, replacing an array element with the special undefined value ⊥.

• Consider the assignment "x := x + 1;", the original value of x can be obtained by subtracting 1 from the final value of x.

5.7 Comparison with Other Work

Partial Order Techniques

The commutative property and the singular property may look similar to the independent properties for partial order reduction methods [Val90, Val91, Val93, God90, GW93, GW94, God95, HP94, Pel94, Pel96]. However, the reduction proposed in this chapter depends only on the finite behavior of the reversible rule set and the finite behavior of the other rules w.r.t. to the reversible rule set. Therefore, the detection of the commutative property and singular property is much simpler than the calculation of a persistent or stubborn set.

In order to understand the differences better, two contrived examples are presented in this section. The first example is a system of n fully-independent processes with a single reversible transition:

\[ P_i : p_i \xrightarrow{a_i} q_i \]
5.7. COMPARISON WITH OTHER WORK

<table>
<thead>
<tr>
<th></th>
<th>states examined</th>
<th>states stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>original state graph</td>
<td>$2^n$</td>
<td>$2^n$</td>
</tr>
<tr>
<td>partial order reduction</td>
<td>$n+1$</td>
<td>$n+1$</td>
</tr>
<tr>
<td>reversible rule reduction</td>
<td>$2^n$</td>
<td>1</td>
</tr>
<tr>
<td>without singularity property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reversible rule reduction</td>
<td>$n+1$</td>
<td>1</td>
</tr>
<tr>
<td>with singularity property</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                |                  |                |
| $\langle p_1, p_2, \ldots, p_n \rangle$ |                  |                |
| $\downarrow a_1$ |  \langle q_1, p_2, \ldots, p_n \rangle | $a_1$           | $\langle q_1, p_2, \ldots, p_n \rangle$ |
| $\langle q_1, p_2, \ldots, p_n \rangle$ |                  |                |
| $\downarrow a_2$ |  \ldots | $a_2$           | $\langle p_1, q_2, \ldots, p_n \rangle$ |
| \ldots |                  |                |
| $\langle q_1, q_2, \ldots, q_n \rangle$ |                  |                |
| $\downarrow a_n$ |      |                |
| $\langle p_1, p_2, \ldots, q_n \rangle$ |                  |                |

Partial Order Reduction  Reversible Rules Reduction

Table 5.4: Comparison of partial order and reversible rules, I: The partial order method collapses redundant paths into a single path, whereas the singularity property shortens every path to length of one execution only.

The performance on this system is summarized in Table 5.4. While only 1 state is stored in the algorithm using reversible rules, the actual number of states examined is the same as the partial order algorithm. The actual transitions that are executed illustrate the main difference between the two approaches: The partial order method collapses redundant paths into a single path, whereas the singularity property shortens every path to a length of one execution only.

The second example has $n$ fully-independent processes with two reversible transitions:

$P_i: \quad q_i \xleftarrow{a_i} p_i \xrightarrow{b_i} r_i$

The performance on this system is summarized in Table 5.5. In this case, the algorithm using singular reversible rules performs much better than the partial order
Table 5.5: Comparison of partial order and reversible rules, II: The number of states examined by the partial order algorithm is exponential in the number of processes, whereas the number of states examined by the algorithm using singular reversible rules is linear in the number of processes.

However, if the transitions are not reversible, no reduction can be obtained from our algorithm. Therefore, methods using partial order and methods using reversible rules have their own domains of applications where one can perform better than the other. An interesting question for future investigation is how to combine these two methods.

Abstraction

The properties of the reversible rules make sure that every transient state in a subgraph can be generated from the progenitor via the reversible rules, and each state can be mapped back to the progenitor by reverse-execution of the reversible rules.
5.7. COMPARISON WITH OTHER WORK

Compared to conventional abstraction [GL93b], the method presented here does not require the user to provide a suitable abstract domain, and does not produce false negative results as one often finds with badly chosen abstract domains. It can be used for the verification of deadlock-freedom, error and invariant checking, and stuttering-invariant LTL model checking.

State space caching

The basic idea of discarding the transient states is similar to a state space caching algorithm [JJ91]: In the basic algorithm presented in this chapter, all reachable states are examined in the original state graph, but only a small portion of them are stored. However, we discard the states only when it can be determined from the existing states in the hash table that those discarded states have already been examined. Therefore, our method never examines a state more than once, whereas state space caching may examine the same state many times.

Invisible transitions

Independent to this work, Hillel Miller and Shmuel Katz [MK96] have investigated how to generate a reduced state graph by exploiting “invisible transitions”. Although the motivation is different, the transitions in the abstract state graph are similar: an abstract transition $t \in \mathcal{T}$ can be regarded as a series of original transitions $r_1 \ldots r_n \in \mathcal{R}$. However, their method takes at least as much time to generate the original state graph, with a potential of very large time overhead due to the re-visitation problem: a state that is removed from the state graph may be generated and examined many times. They rely on a complicated heuristic to solve this re-visitation problem. Instead of relying on heuristics, the method described in this chapter takes advantage of the reversible properties, which guaranteed only a predictable overhead (the cost of the function $reduce()$) to check whether or not a discarded state has been examined previously.
CHAPTER 5. VERIFYING SYSTEMS WITH REVERSIBLE RULES
Chapter 6

Verifying Scalable Systems

Chapter Overview

The techniques described until now have focused on verification of systems with a finite and known number of components. In contrast, this chapter describes an abstraction for verifying scalable systems with an arbitrary number of replicated components. The abstraction presented uses a set of repetition constructors to simplify the state space by ignoring the exact number of components in a system.

Although similar abstractions have previously been proposed to solve this problem (see Section 1.2.3 on previous work), they usually require a lot of effort and expertise from the user. The techniques presented in this chapter overcome these problems: through a static analysis of a description language, a verification tool can detect when to use such an abstraction, and a fully automatic algorithm is provided to construct an abstract state graph from a conventional description efficiently.

6.1 Abstraction Using Repetition Constructors

A lot of scalable systems can be verified using a set of repetition constructors to simplify the state space. This section illustrates what a repetition constructor is, and how the abstraction can be done.

\footnote{This chapter is based on materials published in [ID96c].}
A scalable system typically consists of a collection of components, including components that may be replicated from 1 to $n$ times. At any time, each component has a state (we say “the component is in the state”); the global state of the system at any time is the product of its component states.

In the state graph of a scalable system, we can often find many similar states that differ only in the number of components in certain states. For example, the states A and B shown in Figure 6.1 are two similar states: state A has two processors with an invalid cache line and one processor with a shared copy of a memory location, whereas state B has only one processor with an invalid cache line and two processors with a shared copy of a memory location. Otherwise the two states are exactly the same. As the number of components in the system increases, the number of such similar states increases exponentially: for a system with $n$ components and each component with $k$ possible component states, there can be up to $k^n$ similar states in the state graph. Because of this explosion, a verification algorithm which relies on an explicit search of the state graph can only verify systems with a small number of replicated components.

Fortunately, in many scalable systems, two similar states have similar successors as well. This property is defined later as the repetitive property. The intuition behind this property can be illustrated by the two transitions shown in Figure 6.1. In the first transition where the memory receives the read-request (depicted as RR) from processor 1, the successors generated from A and B are similar, differing only in the number of processors with an invalid cache line and in the number of processors with a shared copy of the memory location. In the second transition where the memory receives the write-request (depicted as WR) from processor 2, the successors are also similar: the successor of A has one processor with a shared copy of the memory location and a pending invalidation, whereas the successor of B has two processors with a shared copy of the memory location and a pending invalidation for each of these two processors.

One consequence of the repetitive property is that these systems can often be proved correct without modeling the precise number of replicated components. For example, suppose a multiprocessor has identical caches numbered 1 to 8, and that
6.1. **ABSTRACTION USING REPETITION CONSTRUCTORS**

---

**Figure 6.1:** Repetitive property in a cache coherence protocol: Consider states A and B (in a cache coherence protocol), which differ only in the number of components with state I and state S. The transitions show that their successors also differ only in the number of components with state I, state S, or state S with a pending invalidation.
a particular memory value is *invalid* in caches 1, 2, 3, 5, 6, 7 and *writable* in cache 4. An abstract state may be used to record that more than zero caches are *invalid*, and exactly one is *writable,* "forgetting" not only the number of processors in each state but also the ordering of the processors. Formally, the abstract state described in this chapter includes a mapping from component states to repetition constructors 0, 1, and +:

- **Null (0):** indicates zero instance.
- **Singleton (1):** indicates one and only one instance.
- **Plus (+):** indicates one or multiple instances.

This abstraction may convert an infinite state graph of a system with an arbitrary number of components into a finite abstract state graph. This abstract state graph is an approximation of the original state graph. This approximation is conservative for verification of invariants and ∀CTL* model checking: it never fails to report an error, but may report an error when none exists.

Similar abstractions have been proposed previously [Lub84, Dij85, CG87, PD95b, PNAD95, Pon95]. However, these methods require the user to write an executable description of the abstract behavior: For example, as summarized in Figure 6.2, in the symbolic state model proposed by Pong and Dubois, the system to be verified is *manually* modelled in a special format of the form \((s, [r_1, r_2, \ldots, r_n])\), where \(s\) contains all information that does not belong to any component, and each \(r_i\) contains all relevant information about the component \(i\). The next step is to rearrange the components so that components with the same state are clustered together: \((s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])\), where \(a_i\) denotes the number of \(q_k\)'s. This format allow the exact number of components \(a_i\) to be replaced with an abstracted number, such as +. Following this abstraction the transitions in the system are then rewritten *manually* in term of the abstract states, so that an abstract state graph can be constructed.

This executable description is different from the concrete description used for specification or synthesis, so their methods require more work, and raises the question of whether the concrete and abstract descriptions are consistent.
6.2. DETECTING REPETITIVE PROPERTY

Instead of requiring an executable abstract description from the user, we provide a high-level programming language in which a user can easily describe a protocol in its concrete form. The language extends the Murφ description language with a new datatype, called repetitiveID, to represent the indices of the replicated components. The Murφ compiler can then detect whether this datatype is used in a way that admits verification in an abstract state space.

To describe a scalable system in its concrete form, the replicated components are usually modeled in Murφ by defining a constant for the number of components (say CompCount), and defining a subrange CompID: 1 .. CompCount for the indices of the components. The local states of the components are stored in an array indexed by CompID. The rules describing the components are enclosed in a ruleset with a CompID.
parameter which represents the component to which the rule belongs. One of the
components “takes a step” when its rule is chosen and executed. Using this conven-
tion, a Murφ description becomes scalable, meaning that the number of replicated
components can be changed simply by modifying CompCount.

A new datatype, called repetitiveID, is used to determine whether a system can
be verified with the abstraction using repetition constructors, and to provide enough
information for Murφ to construct an abstract state graph. RepetitiveID is a sub-
type of conventional integer subrange (in fact, it is a subtype of scalarset), which
should be used for the indices of replicated identical components, such as processesors
in a multiprocessor. For example, we can change the subrange 1..numProcessor
to RepetitiveID(numProcessor), so that the verifier will verify the system in the
smaller abstract state space. Murφ automatically checks that certain restrictions are
satisfied so that the verification is sound. Since a member i of the repetitiveID type
is used as the name of a component in the description, it is natural to identify
i and the component, and refer to “component i” below.

A value of repetitiveID type can be assigned to variables, tested for equality with
other values, used as an array index, or bound in a RuleSet or a for loop, etc.
As presented below, there are six restrictions on the use of RepetitiveID. In spite of
these restrictions, the repetitiveID type can be used to model many systems, such
as bus-based multiprocessor cache coherence protocols [PD95b] and network-based
cache coherence protocols with a central or distributed directory [PNAD95, DDHY92,
LLG+90].

6.2.1 Restrictions for Abstract State Generation

Intuitively, the first goal of the restrictions on repetitiveID is to make sure that Murφ
can construct the abstract states by isolating the parts of the state corresponding to
the replicated components into a single array indexed by the repetitiveID type. The
first two restrictions make it possible to do this automatically:

1. The Murφ program has at most one RepetitiveID type. This restriction simplifies
   the analysis; otherwise, two repetitiveID types may interact with each other, and
in such cases, we may not be able to isolate two sets of replicated components into two arrays indexed by the two repetitiveID types.

2. *The elements of a symmetric array cannot contain another array indexed by the RepetitiveID.* A symmetric array is a multiset, or an array indexed by a scalar set or a repetitiveID. If this restriction is not satisfied, the data structure may be difficult to break down into a single linear array indexed by the repetitiveID.

The next two definitions describe when a component can be abstracted by the repetition constructors. A central-directory-based cache coherence protocol, similar to the one in Figure 6.1, is used to illustrate these definitions: consider a cache coherence protocol whose state includes an array of local processor control states, a multiset of messages representing a communication network, and a memory where each memory line has an owner field pointing to a processor that has a writable copy of the line, along with the data in the memory line. The messages in the network have to and from fields, which can be processor indices or a value representing the memory itself.

**Definition 6.1** *The component state of the component i includes the following state variables:*

- for every array A indexed by the components, the element A[i]. In the example, the local control state of processor i becomes part of the component state for i.

- for every multiset M that is not assigned to a component state by the previous case, the elements of M containing i and no other components. In the example, the messages between the memory and processor i become part of the component state for i.

In our example, the memory value (which does not contain a component index), the owner field (which is not in a multiset), and messages from one processor to another (which contain two component indices) are not included in any component states.

**Definition 6.2** *A component i is abstractable if and only if its component state contains all instances of i occurring in the global state, and contains no other component indices.*
In our example, a processor $i$ would not be abstractable if the *owner* field had the value $i$, or if there were a message between $i$ and another processor $j$.

The component states for two abstractable components $i$ and $j$ are considered to be the same if and only if the only difference between corresponding variables is that variables in component $i$ have the value $i$ and variables in component $j$ have the value $j$.

This notion of abstractable components enables Mur$\varphi$ to construct an abstract state automatically. As shown in Figure 6.3, without actually rearranging the state to the special form $(s, [q_1, \ldots, q_k])$ described in the previous section, the component states of two abstractable components can be compared, and, if the component states are the same, combined using the constructor $+$. Components that do not have the same state with other processors, and those that are not abstractable, can be described by the repetition constructor 1.

In the rest of this chapter, we regard an original state conceptually as being a pair $(s, [q_1, \ldots, q_k])$, where $[q_1, \ldots, q_k]$ is an array of component states indexed by the indices of the abstractable components, and $s$ is an assignment to the rest of the state variables.

### 6.2.2 Restrictions for Abstract Successor Generation

The remaining restrictions on the repetitiveID type ensure that the abstract successors (and therefore, the abstract state graph) can be constructed automatically. Intuitively, there are two goals to the remaining restrictions. The first one is to make sure that the components are symmetric, so that they can be reordered arbitrarily without changing the behavior of the system. The second goal is that, if components $c_1, \ldots, c_n$ have an identical component state $r$, these components also have an identical component state $r'$ in an immediate successor, and the value of $r'$ does not depend on the exact value of $n$. There is one exception: there can be a special *active* component associated with each transition rule, corresponding to the active thread of control in a interleaving model. The component state of this active component is treated differently from other component states, even if they are otherwise identical. An example of where this is useful is mutual exclusion: many components may be in
Figure 6.3: Automatic generation of abstract states: Without rearranging the state to the form $(s, [q_1, \ldots, q_k])$, the component states of two abstractable components can be compared. Since processors 1 and 2 have the same component state, one of them is removed from the state (by using the constructor 0, and assigning the special undefined value to the variables in the component states), and the repetition constructor of other one is changed to $+$. Processor 3 does not have the same state with any other processor, and processor 4 is in some state that is not abstractable (represented by '?' ); therefore, both of them are described by the repetition constructor 1.
identical states, waiting for a resource, but only the one that becomes active first will obtain it.

There are four restrictions on the use of RepetitiveID to ensure that these properties are true.

3. No “symmetry-breaking” operations (see Chapter 3). There are no literal constants in the repetitiveID type; arithmetic operations are not allowed; comparisons such as < are not allowed. This restriction allows us to rearrange and cluster components with the same component states, without affecting the behaviors of the state.

4. Bindings of the RepetitiveID type in RuleSets may not be nested. Because the component index of the active component is bound to the parameter in a rule set, this restriction associates at most one active component to each transition rule.

5. Bindings of the RepetitiveID type in for statements, exists expressions, and forall expressions may not be nested. The variables written by each iteration of a for statement on the RepetitiveID must be disjoint. This restriction forbids the counting of components, and also forbids abstractable components with the same component state to behave differently.

6. If a variable in the state has the RepetitiveID type, and its value corresponds to some abstractable components, the variable may not be used to index an array with RepetitiveID index type. This restriction forbids abstractable components with the same component state to behave differently.

The properties enforced by these restrictions can be summarized by two properties: the symmetry and the repetitive property.

Restriction 3 implies that repetitiveID is a subtype of scalarset, and every permutation of the indices generates an equivalent state, as described in Chapter 3. The equivalent class [q] of any state q can be denoted compactly in an exponent representation:
Definition 6.3 (exponent representation) An exponent representation of an equivalence class is of the form: \((s, \{q_1^{a_1}, \ldots, q_k^{a_k}\})\), where integers \(a_1, \ldots, a_k \geq 1\), and \(q_1, \ldots, q_k\) are distinct. It represents the set of states that are equivalent to \((s, [q_1, \ldots, q_1, \overbrace{q_k, \ldots, q_k}^{a_k}])\) where \(a_i\) denotes the number of \(q_i\)'s.

Restrictions 4, 5, and 6 enforce the repetitive property on the symmetry equivalence class, defined later in this section. However, before we define the repetitive property, we need to define the active component associated with a transition, and the representation that isolates the active component.

Definition 6.4 (active component) Given a Murφ rule generated by a ruleset over a repetitiveID type, the active component is the component referred by the value bound to the ruleset. If a Murφ rule is not generated by a ruleset over a repetitiveID type, there is no active component associated with this rule.

If the active component is abstractable, we can isolate the active component and the environment, using the following representation:

Definition 6.5 (active representation) Given an equivalence class in the exponent representation \(p = (s, \{q_1^{a_1}, \ldots, q_k^{a_k}\})\), and a transition rule \(t\) such that the active component in \(p\) has an abstractable component state \(q_i\), the active representation of \(p\), w.r.t. the rule \(t\), is defined as:

\[
\begin{align*}
(s, q_i, \{q_1^{a_1}, \ldots, q_i^{a_i-1}, \ldots, q_k^{a_k} \}) & \quad \text{if } a_i > 1; \text{ and} \\
(s, q_i, \{q_1^{a_1}, \ldots, q_i^{a_i-1}, q_i^{a_i+1}, \ldots, q_k^{a_k} \}) & \quad \text{if } a_i = 1,
\end{align*}
\]

representing the same set of states as \(p\).

Given an active representation \((s, q_i, \{q_1^{a_1}, \ldots, q_k^{a_k}\})\) for a transition rule \(t\), \(\{q_1^{a_1}, \ldots, q_k^{a_k}\}\) is called the abstracted environment of \(q_i\), and \(s\) is the unabstracted environment of \(q_i\).

If there is no active component for \(t\) (or if the active component is not abstractable), the active representation is of the form \((s, \{q_1^{a_1}, \ldots, q_k^{a_k}\})\), where \(\{q_1^{a_1}, \ldots, q_k^{a_k}\}\) is the abstracted environment, and \(s\) represents the unabstracted environment (and the active component).
The repetitive property is defined using the active representation:

**Definition 6.6 (repetitive property)** A system is repetitive if and only if, given an active representation \((s, q, \{q_{a_1}^1, \ldots, q_{a_k}^k\})\) for a transition rule \(t\), we have:

1. **Similarity in error behaviors**
   
   \(t\) executes an error statement on \((s, q, \{q_{a_1}^1, \ldots, q_{a_k}^k\})\) if and only if it executes an error statement on \((s, q, \{q_1, \ldots, q_k\})\).

2. **Similarity in successors**
   
   \(t\) transforms \((s, q, \{q_{a_1}^1, \ldots, q_{a_k}^k\})\) to \((s', r, \{r_{a_1}^1, \ldots, r_{a_k}^k\})\) if and only if \(t\) transforms \((s, q, \{q_1, \ldots, q_k\})\) to \((s', r, \{r_1, \ldots, r_k\})\), where \(r, r_1, \ldots, r_k\) may not be distinct or abstractable.

   And similarly for any equivalence class \(p' = (s, \{q_{a_1}^1, \ldots, q_{a_k}^k\})\) and a transition rule \(t\) without an abstractable active component.

Pictorially, the repetitive property may be summarized as:

\[
\begin{array}{ccc}
(s, q, \{q_1, \ldots, q_k\}) & \xrightarrow{t} & (s, q, \{q_{a_1}^1, \ldots, q_{a_k}^k\}) \\
\Downarrow & \Leftrightarrow & \Downarrow \\
(s', r, \{r_1, \ldots, r_k\}) & \xrightarrow{t} & (s', r, \{r_{a_1}^1, \ldots, r_{a_k}^k\})
\end{array}
\]

The repetitive property makes sure that the result of executing \(t\) on the states represented by the active representation \((s, q, \{q_1^1, \ldots, q_k^k\})\) does not depend on the exact values in \(a_1, \ldots, a_k\).

Although \(r, r_1, \ldots, r_k\) in the state generated by \(t\) may not be distinct or abstractable, the correct equivalence class represented by \((s', r, \{r_{a_1}^1, \ldots, r_{a_k}^k\})\) can be obtained by repartitioning the components into abstractable components and non-abstractable components, rearranging the abstractable components, and combining abstractable components with the same component state. In the rest of the paper, this process is implicitly assumed whenever a rule is executed.

The following theorem, proved in Appendix A.4, is used in the next section to construct the abstract state graph.

**Theorem 6.1** The system described by a Murϕ program with a repetitiveID is symmetric and repetitive w.r.t. the components named by the repetitiveID.
6.3 Automatic Abstraction

Once a verifier knows how to isolate the abstractable components in a state, and knows that the description is symmetric and repetitive, it can generate the abstract state graph using a fully automatic algorithm.

6.3.1 Abstract States

Conceptually, in order to generate an abstract state from an original state \( p \), the equivalence class \([p] = (s, \{q_1^{a_1}, \ldots, q_k^{a_k}\})\) is constructed first, and then an abstract state is constructed by replacing every \( a_i > 1 \) with the constructor \(+\). In the rest of the chapter, an abstract state is written in the form \((s, \{q_1^{e_1}, \ldots, q_k^{e_k}\})\), where each \( e_i \) is 1 or \(+\) (when the constructor is 0, the component state is omitted).

An original state \( a = (s, [r_1, \ldots, r_n]) \) is represented by an abstract state \( A = (s, \{q_1^{e_1}, \ldots, q_k^{e_k}\}) \) if the following conditions are satisfied:

- \( e_i = + \) if \( q_i \) occurs in \([r_1, \ldots, r_z]\) two or more times;
- \( e_i = 1 \) or \( e_i = + \) if \( q_i \) occurs in \([r_1, \ldots, r_z]\) exactly once;
- a component state does not appear in \( \{q_i, \ldots, q_k\} \) if it does not appear in \([r_1, \ldots, r_z]\).

The abstract states are partially ordered: \((s, \{q_1^{e_1}, \ldots, q_k^{e_k}\}) \leq (s, \{q_1^{e'_1}, \ldots, q_k^{e'_k}\})\) if and only if \( e_i = + \) implies \( e'_i = + \) for all \( 1 \leq i \leq k \). In this case, \((s, \{q_1^{e_1}, \ldots, q_k^{e_k}\})\) is said to cover \((s, \{q_1^{e_1}, \ldots, q_k^{e_k}\})\) (and the two states are comparable). An example is shown below:

\[
\begin{array}{c}
(s, \{q_1^+, q_2^+\}) \\
\downarrow \\
(s, \{q_1^+, q_2^-\}) \quad (s, \{q_1^+, q_2^+\}) \\
\downarrow \\
(s, \{q_1^-, q_2^+\}) \quad (s, \{q_1^+, q_2^+\}) \\
\end{array}
\]

The notation \( a \in A \) is used to indicate that abstract state \( A \) represents original state \( a \). The set of abstract states representing a particular concrete state has a
unique minimum element in this order; the abstraction function \texttt{abs} maps a concrete state to its minimum abstract representative.

In the actual implementation, Murφ does not explicitly rearrange the global variable to obtain \((s, [r_1, \ldots, r_n])\). Therefore, to obtain an abstract state, a slightly more complicated procedure is needed. First of all, an array of repetition constructors is declared internally and attached to every state. Initially, the constructor 1 is assigned to every element of this array. Because of symmetry (described in Chapter 3), many states are equivalent to the current state. Therefore, the unique representative \(\zeta(q)\) is then constructed to represent this set of equivalent states (still with the constructor 1 for every component). Finally, an abstract state is constructed by comparing every pair of the component states. If two component states are both abstractable and have the same component state, the constructor of one of them is converted to +. The constructor of the other is converted to 0, and the variables in the corresponding component state are assigned the special undefined value \(\perp\).

### 6.3.2 Abstract Transitions

In order to perform verification on the abstract state space, it is necessary to be able to generate the successors of an abstract state. How to do this is the fundamental problem that must be solved in any verification scheme using abstraction. There are several possible solutions:

- Write the abstract transition rules manually [PD93a].
- Write an abstract interpretation for the operations in the transition rules [CC92].
- Translate the transition rules into Boolean formulae and convert them to Boolean formulae for the abstract transition rules through a finite existential quantifier [GL93b].
- Generate the successors from every concrete state represented by an abstract state, and convert them back to a set of abstract successors.
However, the first two options are labor-intensive, and it is difficult to guarantee that the abstract behavior is consistent with the concrete behavior. The last two options require that the set of original states represented by an abstract state is finite, and it is impossible to apply them in our case where an infinite number of states is represented by an abstract state.

The key to our method is a variant of the fourth alternative, modified so that, instead of generating the successors from every state represented by an abstract state $A$, up to two representatives are chosen (see below). The successors of these representatives are generated by executing the original transition rules, and they are converted to abstract successors of $A$. The repetitive property, described in the previous section, ensures that these abstract successors represent the successors of all original states represented by $A$.

**Automatic construction of the abstract successors**

Given an abstract state $A$, the automatic construction algorithm chooses up to two representatives from the set of states represented by $A$. The choice of these representatives depends on the abstract state and on the active component in a transition rule. Given an abstract state $p = (s, \{q_1^{e_1}, ..., q_k^{e_k}\})$, and a transition $t$, there are three possible situations:

1. There is no active component for the transition $t$, or the active component is not abstractable in $p$, that is, it belongs to $s$.

2. The component $i$ in $p$ is active, with repetition constructor 1 and component state $q_i$.

3. The component $i$ in $p$ is active, with repetition constructor + and component state $q_i$.

In all three cases, the construction of the abstract successors consists of four steps, as depicted in Figure 6.4. In the first case, the verifier uses the symmetry-equivalence class $[p] = (s, \{q_1, ..., q_k\})$ as the representative for the abstract state. After executing $t$ on a canonical state in $[p]$ to obtain an original state $q'$, we can
restore the original array of repetition constructors on the equivalence class $[q'] = (s', \{r_1, \ldots, r_k\})$ to obtain $(s', \{r_1^e, \ldots, r_k^e\})$. Finally, the abstract successor can be obtained after repartitioning the components into abstractable components and non-abstractable components, rearranging the abstractable components, and combining abstractable components with the same component state.

The second case is similar to the first case, except that the active representation $(s, q_i, \{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_k\})$ is used instead of the equivalence class $[p] = (s, \{q_1, \ldots, q_k\})$. After executing $t$ on the active representation, the abstract successor can be recovered by restoring the original array of constructors.

The last case is the most difficult case. The repetitiveID restrictions allow a transition to have different effects on the active component and the other components with the same component state. Therefore, the states represented by the abstract state are partitioned into two sets of equivalence classes (where $a_i \in e_i$ means that $a_i = 1$ if $e_i = 1$; and $a_i > 1$ if $e_i = +$):

$$\{(s, \{q_1^{a_1}, \ldots, q_i^{a_i}, \ldots, q_k^{a_k}\}) \mid a_i \in e_i\}, \text{ and}$$

$$\{(s, \{q_1^{a_1}, \ldots, q_i^z, \ldots, q_k^{a_k}\}) \mid a_i \in e_i \text{ and } z \geq 2\}$$

Two representatives $(s, q_i, \{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_k^{a_k}\})$ and $(s, q_i, \{q_1, \ldots, q_i, \ldots, q_k\})$ are chosen from these two partitions. The equivalence classes of the successors of these two representatives can be generated by executing $t$ on the corresponding canonical states. The abstract successors can be recovered by restoring the original array of constructors, repartitioning the components into abstractable components and non-abstractable components, rearranging the abstractable components, and combining abstractable components with the same component state.

During the construction of the successors, if an error statement is executed, or if the invariant is false in the successors before restoring the constructors, the abstract successor is \textbf{error}. 


CASE 1
\[(s, \{q_1^e, \ldots, q_n^e\}) \]
\[\downarrow\]
\[(s, q, \{q_1, \ldots, q_n\}) \]
\[\rightarrow\]
\[(s', r, \{r_1, \ldots, r_n\}) \]
\[\rightarrow\]
\[(s', \{r_1^e, \ldots, r_k^e\}) \]
\[\rightarrow\]
abstract successor \(A\)

CASE 2
\[(s, \{q_1^e, \ldots, q_i^1, \ldots, q_n^e\}) \]
\[\downarrow\]
\[(s, q, \{q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n\}) \]
\[\rightarrow\]
\[(s', r, \{r_1, \ldots, r_{i-1}, r_i, r_{i+1}, \ldots, r_n\}) \]
\[\rightarrow\]
\[(s', \{r_1^e, \ldots, r_{i-1}^e, r_i^1, r_{i+1}^e, \ldots, r_k^e\}) \]
\[\rightarrow\]
abstract successor \(A\)

CASE 3
\[(s, \{q_1^e, \ldots, q_i^+ \ldots, q_n^e\}) \]
\[\rightarrow\]
\[(s, q, \{q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n\}) \]
\[\rightarrow\]
\[(s', r, \{r_1, \ldots, r_{i-1}, r_i, r_{i+1}, \ldots, r_n\}) \]
\[\rightarrow\]
\[(s', \{r_1^e, \ldots, r_{i-1}^e, r_i^1, r_{i+1}^e, \ldots, r_k^e\}) \]
\[\rightarrow\]
abstract successor \(A\)

abstract successor \(A\)

Figure 6.4: Automatic generation of abstract successors: The abstract successors of an abstract state can be generated by 1) choosing the right representative(s), 2) executing the original transition rules, 3) restoring the repetition constructors from the abstract state, 4) repartitioning the components into abstractable components and non-abstractable components, rearranging the abstractable components, and combining abstractable components with the same component state.
Because of the restrictions of the RepetitiveID, the abstract state graph generated this way has the following property:

**Property 6.1** Given two concrete states $a$ and $b$ that are reachable in the original state graph, such that $(a, b)$ is a transition in the original state graph, there exists abstract states $A$ and $B$ such that $a \in A$, $b \in B$, and $(A, B)$ is a transition in the abstract state graph.

Because of Property 6.1 and because the abstract start states represent all concrete start states, it can easily be proven by induction that for every state $q$ reachable from an initial state in the original state graph, at least one abstract state representing $q$ is reachable from an abstract initial state. It follows from this result that if the abstract state graph satisfies a $\forall$CTL* formula $f$, the original state graph also satisfies $f$ (c.f. [BBG93, DGG94]). Because the abstract state graph is an approximation, it may result in reports of non-existent errors, and it cannot be used for deadlock detection or $\exists$CTL* model checking.

### 6.3.3 On-the-Fly Abstraction Algorithms

The automatic construction discussed so far can be used to generate an abstract state graph for verifying systems of arbitrary sizes. However, the abstract state graph for the family of systems of different sizes may be infinite and the first algorithm may not terminate. Therefore, the algorithm used in Mur$\varphi$ uses a slightly different construction to generate an abstract state graph for verifying a system of a fixed size. Both algorithms are summarized in this section.

Although the algorithm used in Mur$\varphi$ verifies a system of a fixed size, the results may be generalized to systems of arbitrary sizes, using saturated state graphs. A saturated state graph obtained from a system of size $n$ also represents the behavior of a family of systems with sizes larger than $n$.

**Algorithm for verifying systems of arbitrary sizes**

An algorithm for verifying a system of an arbitrary size is shown in Figure 6.5. Start states in a scalable system typically consists of components in the same initial


6.3. AUTOMATIC ABSTRACTION

Simple Abstraction Algorithm()

begin

Reached = Unexpanded = {abs(q) | q ∈ StartState};

while Unexpanded ≠ ∅ do

Remove a state s from Unexpanded;

for each transition rule t ∈ T do

Generate the set of representatives, R from s;

for each representative r ∈ R do

if an error statement is executed in t on r then report error; endif

let r′ = t(r) in

if r′ does not satisfy one of the invariants then report error; endif

Recover the abstract successor s′ from r′, using the constructors in s;

if s′ is not in Reached then put s′ in Reached and Unexpanded; endif;

endif
endfor;
endfor;
endwhile;
end

Figure 6.5: A simple on-the-fly abstraction algorithm: The part highlighted by an underline represents the main difference of this algorithm from the basic algorithm shown in Figure 2.2.

component state. Therefore, the abstract start state of the form (s, {q0+}) is obtained from a start state in a finite system, by the abstraction function abs. This abstract start state represents the start states of a system of an arbitrary size. Using the construction described in the previous section, an abstract state graph is generated to verify the whole family of systems.

Algorithm for systems of fixed sizes

Although it is desirable to use a single abstract state graph to verify the whole family of systems with arbitrary numbers of replicated components, in many states, some of the replicated components may not have an abstractable component state. If there are infinitely many non-abstractable components, the abstract state graph is infinite, and the algorithm presented in Figure 6.5 will not terminate. Therefore, a restricted abstract state graph is used in Murϕ. A restricted abstract state graph represents
a system of a fixed size only, but the result of the verification may be automatically extended to a family of systems.

In a restricted abstract state graph, an abstract state maintains the total number of replicated components provided by the size of the repetitiveID type, while forgetting exactly how many components are in each component state.

**Definition 6.7 (Restricted Abstract State)** A restricted abstract state is an abstract state paired with a number representing the total number of replicated components.

We write \((s, \{q_1^c, \ldots, q_k^c\})_n\) to represent a restricted abstract state with \(n\) components.

A restricted abstract state graph for a system of size \(n\) can be generated using the previous algorithm, with three special cases added:

1. *If a restricted abstract state represents only states with fewer than \(n\) components, it is discarded.* An example is \((s, \{q_1^1, q_2^1\})_3\).

2. *If a restricted abstract state represents only states with more than \(n\) components, it is discarded.* An example is \((s, \{q_1^+, q_2^+\})_1\).

3. *If a restricted abstract state represents only one state with \(n\) components, it is converted to a restricted abstract state with only the constructor 1.* For example, the restricted abstract states \((s, \{q_1^1, q_2^1\})_2\) and \((s, \{q_1^+, q_2^+\})_2\) is the same as \((s, \{q_1^1, q_2^1\})_2\). Mur\(\varphi\) automatically converts both of them to \((s, \{q_1^1, q_2^1\})_2\).

These modifications guarantee that the restricted state graph is finite, since there are only a finite number of states in the original state graph for a system of a fixed size.

**Saturated state graphs**

Although the algorithm in Mur\(\varphi\) uses a restricted state graph, we can still use Mur\(\varphi\) to verify a family of systems of different sizes. Saturated models are used for this purpose, in the same framework as the one presented in Chapter 4.
During the verification process, if special cases 2 and 3 are not encountered, the abstract state graph obtained is the same as that for systems of larger sizes. We call this state graph a saturated state graph. A saturated state graph represents the behavior of all systems with sizes \( n \) or larger. The verification performed on this saturated state graph is therefore valid for all systems of size \( n \) or larger.

With a restricted abstract state graph, the verifier won't attempt to solve the problem for arbitrary sizes if the abstract state graph for arbitrary system sizes is infinite or too large to verify. If a saturated state graph is found, we can verify the system for arbitrary sizes by verifying systems from size 1 up to the size when the state graph becomes saturated.

A saturated state graph is guaranteed to exist if the global state does not store any component index explicitly. Many cache coherence protocols fall into this category [PD93b, PD93a, PD95b].

However, there are many situations in which a saturated state graph may not exist. A notable example is when there is a linked list structure that scales with the number of replicated components [PNAD95]. Pong et al. refer to these states as hybrid symbolic states. The following state is an example,

\[
(s \rightarrow q_1 \rightarrow \cdots \rightarrow q_k, \{q_{k+1}^e, \ldots, q_n^e\})
\]

The arrow from \( s \) means that \( s \) contains the index of a processor in component state \( q_1 \), and similarly for \( q_1 \) to \( q_{k-1} \). This part of the state corresponds to a linked list, which is not abstractable using the repetition constructors discussed in this chapter. Since the linked list may be arbitrarily long, the abstract state graph for arbitrary sizes is infinite, and there is no saturated restricted state graph.

## 6.4 Heuristics

The partial ordering of the abstract states introduces an efficiency issue that does not occur in conventional verification by state enumeration: it is wasteful to have two comparable states in the state graph — only the greater of the two states needs to be retained (in this case, we say the greater state covers the lessor). For example,
Efficient_Abstraction_Algorithm()
begin
  Reached = Unexpanded = {abs(q) | q ∈ StartState};
while Unexpanded ≠ ∅ do
  Remove a state s from Unexpanded;
  for each transition rule t ∈ T do
    Generate the set of representatives, R from s;
    for each representative r ∈ R do
      if an error statement is executed in t on r then report error; endif;
      let r' = t(r) in
      if r' does not satisfy one of the invariants then report error; endif;
      Recover the abstract next state s' from r', using the constructors in s;
      if s is not in Reached and s is not covered by a state in Reached then
        Remove the states in Reached and Unexpanded that are covered by s;
        Put s in Reached and Unexpanded;
      endif
    endfor;
  endfor;
endwhile;
End

Figure 6.6: An efficient on-the-fly abstraction algorithm: The part highlighted by an underline removes redundant abstract states in the abstract state graph.

(s, {q₁⁺, q₂⁺}) is redundant in the set of previously examined states when (s, {q₁⁺, q₂⁺}) is also in the set.

Given a set of abstract states, an abstract state in this set is called a maximal state if and only if none of the other states in the set covers it. Otherwise, it is a non-maximal state. A state should be inserted into the hash table only if it is maximal; when a state is added, some previously examined states may become non-maximal, and they should be removed. The final abstract state graph should contain only the states that are maximal in the set of reachable abstract states.

Therefore, the algorithm in Murϕ was modified to a more efficient algorithm, as shown in Figure 6.6. Two heuristics are presented in this section to reduce the time for checking whether a state is maximal in the set of previously generated states, and to reduce the number of non-maximal states generated.
Checking Maximal States

Checking whether an abstract state is maximal in a set of abstract states is important in the generation of the abstract state graph. A naive (and expensive) algorithm may exhaustively compare the state to every state in the set. This section describes a special hash table structure in which the checking can be done in practically constant time. The design of the special hash table is based on the following observation:

**Observation 6.1** Two abstract states (with constructors 1 and +) are comparable if and only if, except for the array of repetition constructors, the variables in the states have the same values.

For example, \((s, \{q_1^+, q_2^+, q_3^+\})\) covers \((s, \{q_1^1, q_2^1, q_3^1\})\); and, if we ignore the constructors \((1, +, +)\) and \((1, 1, +)\), they have the same values as the original state \((s, [q_1, q_2, q_3])\), which is called the signature of these abstract states.

Because of this observation, the hash table shown in Figure 6.7 is used to store the set of previously examined states. The abstract states with the same signature are hashed into the same entry in the hash table. Instead of the redundantly storing a list of abstract states with the same signature, we store the signature in a slot in the hash table, and maintain a linked list of constructor arrays. With this architecture, if a state \(p_1\) is comparable to a previously examined state \(p_2\), it will be hashed to the same location for \(p_2\). The constructor arrays in the linked list are then compared one-by-one to the constructor array in \(p_1\). Eventually, the constructor array of \(p_1\) is compared with the one for \(p_2\), and the non-maximal state is discarded.

The lists of constructor arrays are always much shorter than the list of previously examined states. In practice, the lists are very short. In ICCP, the maximum length is always less than 20. When an abstract state is hashed into a location with a list of constructor arrays, the average number of comparisons is smaller than 2, as shown in Table 6.1.

Reducing the Number of Non-Maximal States

In practice, although an abstract state graph has very few states, a lot of states that are non-maximal in the final state graph are temporarily stored and expanded. The
state to be checked 
\( (s, \{q_1^+, q_2^+\}) \) 

hashing w.r.t. the signature state 

hash table:

\[
\begin{array}{c}
(s, [q_1], q_2]) \\
\vdots \\
(s, [q_1, q_3]) \\
\end{array}
\]

linked list of constructor arrays

\[
\begin{array}{c}
1, +] \\
\vdots \\
+, 1] \\
\end{array}
\]

Figure 6.7: Hashing structure for efficient comparison of abstract states: Comparable states are hashed into the same location in the hash table.

<table>
<thead>
<tr>
<th>ICCP system of 9 processors</th>
<th>BFS</th>
<th>DFS</th>
<th>Priority BFS</th>
<th>Priority DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max length in lists</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Average number of comparisons when hashed to an occupied location</td>
<td>1.28</td>
<td>1.06</td>
<td>1.05</td>
<td>1.03</td>
</tr>
</tbody>
</table>

Table 6.1: Performance of the special hashing scheme: The maximum length of the lists of constructor arrays is small, and the average number of comparison performed is also close to one. BFS and DFS are conventional breadth-first-search and depth-first-search. Priority BFS and DFS are priority searches described in the next section.
efficiency of the abstraction algorithm depends on how fast the final set of maximal states are generated. As shown in Table 6.2, during the verification of ICCP using simple depth first search (DFS) and breadth first search (BFS), more than 75% of the time is spent in searching non-maximal states. The sizes of the intermediate models are also much larger than the final model.

The intuitive reason behind this is as follows: Many of the states generated by a transition are non-maximal in the final state graph, and yet the maximal states that cover them are not generated soon enough for the verifier to find out that they are non-maximal. Therefore, a lot of non-maximal states (and their successors) are generated, wasting both time and memory.

A best-first search strategy is used in Murϕ to improve the performance of the abstraction algorithm. Every abstract state is assigned a priority according to how many other abstract states it may cover. Such priorities can be assigned easily by counting the number of + constructors in the state. For example, the state $(s, \{q_1^+, q_2^+, q_3^+\})$ has a higher priority than $(s', \{r_1^+, r_2^+, r_3^+\})$, because three abstract states are covered by the first state and only one abstract state is covered by the second state.

The following observation leads to the design of a best-first search algorithm:

**Observation 6.2** The larger number of + constructors in an abstract state, the more likely that its successors have a large number of + constructors, and the more likely that they are maximal states in the final state graph.

In a best-first search, the successors of an abstract state with the largest number of + constructors are generated first, thereby decreasing the likelihood of generating a temporarily maximal state that will be removed later. As shown in Table 6.2, the priority searches perform much better than the simple searches. For ICCP, the number of non-maximal states examined is reduced from 106,528 down to 3,527 in a 9-processor system. The sizes of the intermediate models are always smaller than the final abstract model of 35,515 states, and the final model is constructed in a much shorter time than a simple BFS/DFS search strategy.
TABLE 6.2: Performance of the priority search strategies: The priority search strategies outperform the conventional search strategies by more than 65% in speed and 30% in memory.
6.5 Combining the Three Reduction Techniques

The abstraction using the repetition constructors can be combined with the reduction methods using symmetry and reversible rules.

Consider a Murφ description with a repetitiveID, scalarsets and a symmetric reversible rule set. The theorems in this section show that the reductions using symmetry and reversible rules can be performed on the abstract state graph obtained by using the repetition constructors. In the following definitions and theorems, we denote the original state graph as $A$, and the abstract state graph obtained from $A$ as $\text{abs}(A)$.

First of all, we define how a function on the original states can be converted into a function in the abstract state using the repetition constructors.

**Definition 6.8 (signature)** Given an abstract state $p = (s, \{q_1^{e_1}, \ldots, q_k^{e_k}\})$, the signature $\text{sig}(p)$ is the state $(s, [q_1, \ldots, q_k])$ in the original state graph.

**Definition 6.9** Given a function $f$ on the states in a state graph $A$, the corresponding abstracted function $\text{abs}(f)$ on the abstract states in $\text{abs}(A)$ is defined so that for all abstract states $p_1$ and $p_2$, $\text{abs}(f)(p_1) = p_2$ if and only if $f(\text{sig}(p_1)) = \text{sig}(p_2)$ and $p_1$ has the same list of repetition constructors as $p_2$.

Similarly, the automorphisms induced by the scalarsets on the original state graph can be converted to automorphisms in the abstract state graph:

**Theorem 6.2** Given a Murφ description with a repetitiveID and scalarsets, and an automorphism $h$ induced by the scalarsets on the state graph $A$, $\text{abs}(h)$ is an automorphism on the abstract state graph $\text{abs}(A)$.

**Proof.** Given a state $(s, [q_1, \ldots, q_k])$ in the original state graph, and a permutation $\pi$ on the scalarsets, the permuted state is of the form $(\pi(s), [\pi(q_1), \ldots, \pi(q_k)])$. This is because, according to the definition of a permutation on a state, a permutation on an array not indexed by a scalarset only permutes the values in the elements of the array, but it does not rearrange the position of elements.
Therefore, given an abstract state \( p = (s, \{ q_1^{e_1}, \ldots, q_k^{e_k} \}) \), the corresponding permutation \( \text{abs}(\pi) \) on \( p \) generates \( (\pi(s), \{ \pi(q_1)^{e_1}, \ldots, \pi(q_k)^{e_k} \}) \), which can be illustrated as follows:

\[
(p, [q_1, \ldots, q_k]) \quad \overset{\pi}{\Rightarrow} \quad (p, \{q_1^{e_1}, \ldots, q_k^{e_k}\})
\]

Consider the case for a transition rule \( t \) with an active component in the state \( q_i \) and \( e_i = + \). Two representatives \( p_1 \) and \( p_2 \) are used to generate the abstract next state for \( p \): \( (s, q_i, \{q_1, \ldots, q_k\}) \) and \( (s, q_i, \{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_k\}) \).

Similarly, for \( \text{abs}(\pi)(p) \), the same transition rule \( t \) has an active component in the component state \( \pi(q_i) \). The two corresponding representatives \( p_1' \) and \( p_2' \) are

\[
(\pi(s), \pi(q_i), \{\pi(q_1), \ldots, \pi(q_k)\}), \quad \text{and} \quad (\pi(s), \pi(q_i), \{\pi(q_1), \ldots, \pi(q_{i-1}), \pi(q_{i+1}), \ldots, \pi(q_k)\}).
\]

These two representatives for \( \text{abs}(p) \) are equivalent to the representatives for \( p \), and therefore, the abstract successors for \( \text{abs}(\pi)(p) \) are equivalent to the abstract successors for \( p \).

The remaining two cases, where a transition rule does not have an active component or a transition rule has an active component with state \( q_i \) and \( e_i = 1 \) can be proved in a similar way.

Additionally, to perform reduction using reversible rules on the abstract state graph, the set of reversible rules must be symmetric w.r.t. the replicated components, and the effect of the reversible rules must be localized to the active component:

**Definition 6.10** A rule \( r \) is localized if it accesses only the variables that belong to the component state of the active component of \( r \).

Therefore, given a state \( (s, [q_1, \ldots, q_n]) \) in the original state graph, the execution of a localized rule with an active component \( i \) generates a successor \( (s, [q_1, \ldots, q_{i-1}, r_i, q_{i+1}, \ldots, q_n]) \) for some component state \( r_i \).
To apply reduction using reversible rules in the abstract state graph, the minimum requirement is to be able to find an appropriate function to generate a progenitor abstract state from every abstract state \(p\). This progenitor abstract state should abstract the progenitors of all original states that can be abstracted into \(p\). Furthermore, to use the fast algorithm presented in Chapter 5, the abstracted version of the reversible rules should be singular.

Fortunately, the following theorem shows that, when the reversible rule set is localized, it is straightforward to adapt \(\theta\) (the function that maps an original state to its progenitor) for reduction using reversible rules on the abstract state graph:

**Theorem 6.3** If a rule set \(T\) generates a state graph \(A\), and \(U \subseteq T\) is a localized rule set, we have the following:

- Consider the case when \(U\) is a commutative reversible rule set for \(A\), and \(\theta\) maps a state in the original state graph to a unique progenitor. For every abstract state \(p\), if an original state \(c\) can be abstracted into \(p\), \(\theta(c)\) can be abstracted into \(\text{abs}(\theta)(p)\).

- If \(U\) is a singular rule set for \(A\), \(\text{abs}(U)\) is also a singular rule set for \(\text{abs}(A)\).

Since the abstraction using repetition constructors is an approximation, the essential property of the reversible rules is irrelevant.

**Proof.**

**Unique Progenitors:** Given an abstract state \(p = (s, \{q_1^{e_1}, \ldots, q_k^{e_k}\})\), and a reversible rule \(r \in U\) such that the active component has the component state \(q_i\) in \(p\). The successor of \(\text{sig}(p)\), after executing \(r\) is of the form \((s, [q_1, \ldots, q_{i-1}, r_i, q_{i+1}, \ldots, q_n])\), for some component state \(r_i\), and this successor can be abstracted into \((s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, r_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_n^{e_n}\})\).

Pictorially, this can be summarized as:

\[
\begin{array}{c}
(s, [q_1, \ldots, q_k]) \\
\downarrow r \\
(s, [q_1, \ldots, q_{i-1}, r_i, q_{i+1}, \ldots, q_n]) \\
\downarrow \text{abstraction} \\
(s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, r_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_n^{e_n}\})
\end{array}
\]
Other states in the form of \((s, [q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_k])\) can also be abstracted into \(p\). Because the reversible rule set is symmetric w.r.t. the replicated components, for each component \(j\) in the component state \(q_k\), there exists a corresponding reversible rule \(r_j\) such that \(r_j\) is the same as \(r\), except that the active component is the component \(j\). Because the reversible rules are localized, executing this set of \(a_i\) reversible rules generates the state

\[
(s, [q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_k, \ldots, q_k])
\]

which can be abstracted into \((s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, q_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_k^{e_k}\})\).

Pictorially, it can be summarized as:

\[
\begin{array}{c}
(s, [q_1, \ldots, q_k]) \\
\downarrow \quad r_{a_1}^{(a_1+\ldots+a_{i-1}+1)} \\
\vdots \\
\downarrow \quad r_{a_k}^{(a_1+\ldots+a_{i-1}+a_i)} \\
(s, [q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_k]) \\
\downarrow \text{abstraction} \\
(s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, q_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_k^{e_k}\})
\end{array}
\]

Similar diagrams are generated when the states before executing the reversible rules are generated by reverse-executing the reversible rules. That is, for every state that can be abstracted into \((s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, q_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_k^{e_k}\})\), the states before executing these reversible rules can be abstracted into \((s, \{q_1^{e_1}, \ldots, q_i^{e_i}, \ldots, q_k^{e_k}\})\).

Because the progenitors are generated by reverse-executing all reversible rules, the intermediate steps in the diagrams are irrelevant for the generation of the progenitors. Therefore, if an original state \(c\) can be abstracted into \(p\), \(\theta(c)\) can be abstracted into \(\text{abs}(\theta)(p)\).
Singularity: Because $U$ is a singular rule set, for all rules $r_1, r_2 \in U$ and $t \in T \setminus U$, whenever $q_1 \xrightarrow{r_1, r_2, t} q_2$, we have either $q_1 \xrightarrow{r_1, t} q_2$ or $q_1 \xrightarrow{r_2, t} q_2$.

Because of Property 6.1, the abstraction using the repetition constructors preserves every path, and therefore, whenever $\text{abs}(q_1) \xrightarrow{r_1, r_2, t} \text{abs}(q_2)$, we also have either $\text{abs}(q_1) \xrightarrow{r_1, t} \text{abs}(q_2)$ or $\text{abs}(q_1) \xrightarrow{r_2, t} \text{abs}(q_2)$.

Because of these two theorems, and because of Theorem 5.6 presented in Section 5.4, we can apply symmetry and reversible rule reductions on the abstract state graph obtained from the repetition constructors.

Furthermore, the reduction using reversible rules actually improves the abstraction using the repetition constructors. It removes most of the transient component states, so that a saturated abstract state graph is obtained at a smaller system size.

### 6.6 Implementation and Results

#### Implementation

To use the abstraction with repetition constructors, an array of repetition constructors is declared implicitly and attached to every state. Apart from this, the algorithm uses no extra memory, except for the temporary variables used in symmetry reduction and in storing up to two representatives for generating the successors. The special hash structure further reduces the memory usage by compressing a list of comparable abstract states. Instead of storing every comparable abstract state explicitly, the part common to every comparable abstract state, called the signature, is stored only once. The overhead in time is mostly spent on comparing two component states, and some time is wasted on examining intermediate non-maximal states.

#### Verification results

The verification results on the industrial cache coherence protocol (ICCP) indicate that the repetition constructors are good for verification of medium to infinite system sizes. As shown in Table 6.3, the reductions from this abstraction are small for a
system of fewer than 5 processors. However, the reduction increases significantly once the system has more than 5 processors.

In this protocol, because a processor with an exclusive copy of the cache line may directly forward the data to a processor requesting the copy, some processors may not be abstractable. Fortunately, the extent of forwarding is limited, and a saturated model is obtained with 14 processors.

We can estimate how close an abstract state graph is to saturating, by checking the number of restricted abstract states that represent only 1 original state with $n$ components. We call such states the full states of the restricted abstract state graph. As shown in the Table 6.4, the percentage of the full states decreases to 0 as the state graph approaches saturation.
6.6. IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th># of processors [ICCP]</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>size [unordered network]</td>
<td>10,077</td>
<td>247,565</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>size [sym. only]</td>
<td>1,781</td>
<td>11,814</td>
<td>68,879</td>
<td>358,078</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>size [rep. only]</td>
<td>1,770</td>
<td>11,206</td>
<td>57,790</td>
<td>257,692</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>size [sym./rev.]</td>
<td>431</td>
<td>1,760</td>
<td>6,021</td>
<td>18,118</td>
<td>49,045</td>
<td>121,302</td>
<td>-</td>
</tr>
<tr>
<td>time [unordered network]</td>
<td>5.1s</td>
<td>200s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>time [sym. only]</td>
<td>2.4s</td>
<td>28s</td>
<td>349s</td>
<td>3,762s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>time [rep. only]</td>
<td>4.4s</td>
<td>49s</td>
<td>497s</td>
<td>4,555s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>time [sym./rev.]</td>
<td>2.1s</td>
<td>13s</td>
<td>98s</td>
<td>615s</td>
<td>3,283s</td>
<td>12,801s</td>
<td>-</td>
</tr>
<tr>
<td>time [rev./rep.]</td>
<td>3.3s</td>
<td>27s</td>
<td>167s</td>
<td>811s</td>
<td>3,265s</td>
<td>7,593s</td>
<td>17,477s</td>
</tr>
<tr>
<td># of processors [ICCP]</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14 and up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size [rep./rev.]</td>
<td>38,146</td>
<td>38,485</td>
<td>38,329</td>
<td>38,269</td>
<td>38,269</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time [rep./rev.]</td>
<td>29,877s</td>
<td>37,983s</td>
<td>43,352s</td>
<td>48,410s</td>
<td>49,082s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sym. : Symmetry Reduction
rev. : Reversible Rules Reduction
rep. : Repetition Constructor Reduction

Table 6.3: Improvement in performance with repetitiveID: Through the combination of three reduction algorithms, the family of protocols with different numbers of processors was verified using less memory that had been required for a 4-processor system.

<table>
<thead>
<tr>
<th># of processors (ICCP)</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>427</td>
<td>1,590</td>
<td>4,542</td>
<td>10,587</td>
<td>19,485</td>
<td>28,927</td>
<td>35,515</td>
</tr>
<tr>
<td># of full states</td>
<td>359</td>
<td>1,135</td>
<td>2,756</td>
<td>5,298</td>
<td>7,830</td>
<td>8,589</td>
<td>6,800</td>
</tr>
<tr>
<td># of processors (ICCP)</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14 and up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>38,146</td>
<td>38,485</td>
<td>38,329</td>
<td>38,269</td>
<td>38,269</td>
<td></td>
<td></td>
</tr>
<tr>
<td># of full states</td>
<td>3,765</td>
<td>1,383</td>
<td>303</td>
<td>30</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Monitoring the progress towards a saturated model: As the system sizes increase, the percentage of full states decreases until it becomes 0 in a saturated model.
Chapter 7
Contributions and Future Work

7.1 Contributions

The aim of an effective debugging aid is to obtain an economic advantage by catching bugs reliably and early in the design process. Although automatic formal verification is reliable in catching bugs, the state explosion problem limits its use. In order to increase the effectiveness of automatic formal verification, it is necessary to develop state space reduction methods that are both easy to use and safe to apply.

This thesis provides three reduction algorithms that are easy to use and safe to apply. The contributions are summarized as follows:

- Fully automatic reduction algorithms have been developed to verify symmetric systems, data-independent systems, and scalable systems. New datatypes (scalarset, multiset, data scalarsets, and repetitiveID) have been designed to detect when such reduction algorithms can be used. In contrast to the existing abstractions using similar ideas, the translation of a conventional description to an abstract state graph is fully automatic; the user is not required to provide the abstraction mapping, or manually translate the description to an abstract model. Hence, the verification performed on the reduced state graph is guaranteed to be sound.

The incorporation of graph isomorphism heuristics, special hashing data structures, and priority search strategies have further reduced the resources required for verification using these abstractions.
Through the use of saturated models, a verification tool using these reduction algorithms can automatically extend the results of a finite system to systems with arbitrary sizes.

- A new automatic reduction method using reversible rules has been developed. The basic method has significantly reduced memory usage, and two improvements have been developed to take advantage of other properties in a system, so that no false error is reported, and reductions in time requirements are obtained.

- The reduction methods and the language extensions have been implemented in the Mur$\varphi$ verification system, through which the reductions are accessible to non-experts of formal verification. The three reduction methods are compatible and they can be used simultaneously.

The extended Mur$\varphi$ verification system has been applied to a range of practical applications, such as cache coherence protocols and distributed algorithms. Reductions of more than three orders of magnitude, in both time and memory requirements, have been obtained.

Two frameworks have been used consistently in this thesis. Firstly, the abstractions are performed in an explicit state enumeration setting. Instead of relying on user-provided abstract transition rules, or existentially quantified Boolean transition relations, the framework relies on a small set of concrete representatives from each abstract state. The abstract next states are generated by executing the concrete transition rules on these representatives. This enables us to translate a conventional description for an infinite system to a finite abstract model automatically.

Secondly, because verification of unbounded systems is in general undecidable, the algorithms have been designed to verify the system for a fixed size at the first attempt, rather than to verify an unbounded system. Through simple run-time checks for saturated models, a verifier may extend the results for a finite system to similar systems of larger sizes.
7.2 Future Work

The thesis presents three compatible reduction algorithms for explicit state enumeration. Future research is needed to investigate how well these techniques can be combined with other reduction methods, such as partial order reduction and symbolic methods.

Furthermore, all three reduction algorithms have great potential to be improved:

**Reduction using symmetry:** Emerson and Sistla have recently presented an improvement to symmetry reduction algorithm to efficiently handle fairness assumptions [ES95]. While they have designed an annotated quotient graph for CTL* model checking, they rely on the user to provide an appropriate automorphism group to represent symmetry.

The approach in this thesis provides an effective mean to discover symmetry and the appropriate automorphism group, and an efficient procedure for constructing representatives from equivalence classes. Therefore, the approach in this thesis complements their new algorithm nicely.

**Reduction using reversible rules:** The properties of reversible rules can probably be relaxed. At the moment, the set of reversible rules must be commutative, which limits the number of usable reversible rules. A more sophisticated method to allow more interaction among the reversible rules would further reduce the state space of a system.

**Reduction using repetition constructors:** It may be possible to relax the restrictions for the abstract state generation to handle multiple set of replicated components. Extensions to have more than one active component should be straightforward, with a similar but more complicated procedure to construct the abstract successors.

Extensions to use new repetition constructors would further reduce the state space of a system, and widen the range of practical systems in which similar abstractions are applicable. For example, Pong proposed the * repetition constructor [PD95b, Pon95] to represent zero or more components, and Dijkstra
used regular expression operators to abstract a ring or a list structure [Dij85].

Further research is needed to investigate how a verifier can automatically translate a conventional description to an abstract model in these cases.
Appendix A

Semantic Analysis and Proofs

A.1 Semantics of the Core Murφ Language

A simplified version of the Murφ description language is used to facilitate the semantic analysis of Murφ. This core language has the same expressive power as the actual Murφ language (except for the absent of while loop), but most of the syntactic sugar is removed.

The core language is shown in Figure A.1, using the Backus-Naur Form (BNF). In the subsequent discussion, ⊥ represents an undefined value, N represents the natural numbers, and N⊥ represents N ∪ {⊥}. [L → N⊥] represents the set of all functions with domain L and codomain N⊥.

A description consists of a set of type declarations, a single global state variable (without loss of generality, since multiple variables can be embedded in a record), an initialization rule and a transition rule.

The initialization rule and the transition rule consists of a statement in a simple sequential language, or a nondeterministic choice among simpler transition rules. The ruleset in the actual Murφ language can be translated into a nondeterministic choice over a parameter. The Boolean guard of a rule in the actual Murφ language can be translated into a if statement.
```latex
\begin{align*}
\langle \text{program} \rangle & \;::=\; \text{type} \langle \text{decls} \rangle \\
& \quad \text{var} \langle \text{id} \rangle \;::=\; \langle \text{typeExpr} \rangle \\
& \quad \text{init} \langle \text{rule} \rangle \\
& \quad \text{begin} \langle \text{rule} \rangle \;\text{end} \\
\langle \text{decls} \rangle & \;::=\; \langle \text{id} \rangle \;::=\; \langle \text{typeExpr} \rangle \{ \text{'} ; \langle \text{id} \rangle \;::=\; \langle \text{typeExpr} \rangle \} \\
\langle \text{typeExpr} \rangle & \;::=\; \langle \text{id} \rangle \\
& \quad | \langle \text{num} \rangle \;::=\; \langle \text{num} \rangle \\
& \quad | \text{record} \langle \text{decls} \rangle \;\text{end} \\
& \quad | \text{array} \{ \langle \langle \text{typeExpr} \rangle \rangle \} \;\text{of} \langle \text{typeExpr} \rangle \\
\langle \text{rule} \rangle & \;::=\; \langle \text{stmt} \rangle \\
& \quad | \text{if} \langle \text{boolExpr} \rangle \;\text{then} \langle \text{stmt} \rangle \;\text{end} \\
& \quad | \text{for} \langle \langle \langle \text{typeExpr} \rangle \rangle \rangle \;\text{do} \langle \text{stmt} \rangle \;\text{end} \\
\langle \text{stmt} \rangle & \;::=\; \text{error} \\
& \quad | \langle \text{var} \rangle \;::=\; \langle \text{term} \rangle \\
& \quad | \langle \langle \text{stmt} \rangle \rangle \;\text{if} \langle \text{boolExpr} \rangle \;\text{then} \langle \text{stmt} \rangle \;\text{end} \\
\langle \text{boolExpr} \rangle & \;::=\; \langle \text{term} \rangle \;::=\; \langle \text{term} \rangle \\
& \quad | \langle \langle \text{term} \rangle \rangle \;>\; \langle \text{term} \rangle \\
& \quad | \langle \langle \text{term} \rangle \rangle \;\langle \text{term} \rangle \\
& \quad | \langle \text{term} \rangle \;\langle \text{boolExpr} \rangle \;\text{\langle \text{term} \rangle} \\
& \quad | \langle \text{term} \rangle \;\langle \text{term} \rangle \;\text{\langle \text{term} \rangle} \\
\langle \text{term} \rangle & \;::=\; \langle \text{var} \rangle \;| \langle \text{num} \rangle \;| \perp \;| \langle \text{term} \rangle \;\langle \langle \text{term} \rangle \rangle \;| \langle \text{term} \rangle \;\langle \langle \text{term} \rangle \rangle \;| \langle \text{term} \rangle \;\langle \langle \text{term} \rangle \rangle \\
\langle \text{var} \rangle & \;::=\; \langle \text{id} \rangle \;| \langle \langle \text{var} \rangle \rangle \;| \langle \langle \text{var} \rangle \rangle \\
\langle \text{id} \rangle & \;::=\; \text{string} \\
\langle \text{num} \rangle & \;::=\; \text{number} \\
\end{align*}
```

Figure A.1: Syntax for the Simple Description Language
A.1. SEMANTICS OF THE CORE MUR$\varphi$ LANGUAGE

A.1.1 Abstract Transition Programs

A Mur$\varphi$ program specifies an abstract transition program: An abstract transition program is a quadruple $\langle L, T, I, R \rangle$. $L$ is a set of locations and $T : L \rightarrow \mathbb{N} \times \mathbb{N}$ maps each location $l$ to the lower bound and the upper bound of the values that can be assigned to that location. Each location can be assigned an undefined value or be assigned an integer value within the range specified by $T$. A state in the corresponding state graph is either an error state or an assignment of legal values to every location in $L$. $I$ is a set of constant initialization functions defining the set of initial states. $R$ is a set of transition functions from states to states.

An abstract transition program provides a compact representation of a state graph.

Definition A.1 The state graph generated by an abstract transition program $\langle L, T, I, R \rangle$ is $\langle Q, Q_0, \Delta, \text{error} \rangle$ where

- $Q \subseteq [L \rightarrow \mathbb{N}_1] \cup \{\text{error}\}$ and if $q \in Q$ and $q \neq \text{error}$, then for all $l \in L$ and $T(l) = (lb, ub)$, either $q(l) = \bot$ or $lb \leq q(l) \leq ub$;
- $Q_0 = \{q \in Q \mid \exists f \in I : f() = q\}$; and
- $\Delta = \{(p, q) \in Q \times Q \mid \exists f \in R : f(p) = q\}$.

A.1.2 Formal Semantics

The detailed semantics of the Mur$\varphi$ core language is presented in this section. We define here precisely the translation from a source program to an abstract transition program. Let the source program be

$$
type ts
var gv : \alpha
init ir
begin tr end.$$

The semantics of this program are defined by translating it to an abstract transition program $\langle L, T, I, R \rangle$, which, in turn, defines a state graph.

For simplicity in the definition of the semantics and subsequent proofs, we will not formalize type-checking and other semantic issues that are well-understood. We
assume that programs are well-formed syntactically and semantically. A complete semantics would define all of the static semantic errors, which could easily be checked in a compiler, using well-known algorithms.

In reality, there would also be run-time semantic errors, such as bounds errors when assigning a value to a subrange variable. We omit these checks from the semantics, because including them would complicate the definitions, without adding any particular interest. Moreover, checking for bounds errors could be made explicit using if, Boolean expressions, and the error statement. Hence, the only run-time error included in the semantics is execution of the error statement.

States and Environments

Each subrange, array, and record type has a unique type name. An id which is bound in a type declaration becomes synonymous with the name of the type expression appearing in the declaration. A literal (e.g. 2) always has a subrange type; given our type rules for subranges (which are similar to Pascal's), it doesn't particularly matter what the exact subrange type for the literal is, as long as it contains the integer value of the literal. α is used below to represent a type and also the set of its members in N.

Variables are represented abstractly using locations. Each location corresponds to a subrange variable. T(l) of a location l is a pair, consisting of a lower and an upper bound on the subrange type of the variable; and the location can be assigned an undefined value or be assigned an integer value within the range specified by T(l).

The program has two sets of locations: state locations, L, which are the components of the global variable, and index locations, I, which are parameter variables used to keep track of index values in the indexed □ for loop, and ∃ constructs. The set of all locations V is the union of L and I.

An environment is a function from V to N⊥, such that, the value in every location is consistent with type of the location. The environment that assigns ⊥ to every element of V is written as {}. The environment that assigns v to l and ⊥ to all other elements of V is written as (l → v). If s and s' are both environments, s • s' is the environment which assigns s'(l) to each location l unless s'(l) = ⊥, in which case it
assigns \( s(l) \) to the location. There is an additional \texttt{error} environment that results when an \texttt{error} statement is executed; for all environments \( s \), \( \texttt{error} \cdot s = \texttt{error} \). The set of all possible environments is denoted by \texttt{env}. The set of states is a subset of \texttt{env}, such that the value in every location in \( I \) is \( \bot \). \texttt{tostate} converts a non-error environment to a state by assigning \( \bot \) to every location in \( I \), and also maps \texttt{error} to itself.

**Translating the Variables into Locations**

It is convenient to represent locations as strings. Every variable has an associated string, but only the strings corresponding to subrange variables are locations. When a variable is neither a record field nor an array element, the string is simply the name of the variable. If the string associated with a record variable is \( l \), the string associated with the field \( f \) is \( l.f \). If the string associated with an array variable is \( l \), the string associated with the \( k \)th element is \( l[k] \). For example, with the declaration:

\begin{verbatim}
    type Pid : 1..2
    var P : array [ Pid ] of record CacheState : 0..3; CacheValue : 0..1 end,
\end{verbatim}

the set of locations \( L \) in the corresponding abstract transition program is:

\[ \{ 'P[0].CacheState', 'P[0].CacheValue', 'P[1].CacheState', 'P[1].CacheValue' \}. \]

More precisely, if \texttt{concat} is a function that concatenates a list of strings and numbers into a single string, a translation function \( T v \) can be defined such that, given a variable with name \( x \) and its type \( \alpha \), it produces a set of locations.

\[
T v('x', \alpha) = \begin{cases}
    \bigcup_{i \leq n} T v(\texttt{concat}( 'x', 'i', f_i), \alpha_i) & \text{if } \alpha = \texttt{record} f_1 : \alpha_1; \ldots; f_n : \alpha_n \text{ end} \\
    \bigcup_{i \in \beta} T v(\texttt{concat}( 'x', '[i', '[], \gamma), \gamma) & \text{if } \alpha = \texttt{array} [\beta] \text{ of } \gamma \\
    \{ 'x' \} & \text{if } \alpha \text{ is a subrange.}
\end{cases}
\]

With a global variable \( gv \) of type \( \alpha \), the set of state locations \( L \) in the corresponding abstract transition program is \( T v(gv, \alpha) \). The set of state locations is finite because all array index types are finite.
Translating the Variable References

For a variable reference $d$, the meaning $\llbracket d \rrbracket : env \to string$ is defined as:

$$\llbracket d \rrbracket (s) = \begin{cases} 
\text{concat}(\llbracket d' \rrbracket (s), \text{'f'}) & \text{if } d = d'.f \\
\text{concat}(\llbracket d' \rrbracket (s), \text{'[', } \llbracket t \rrbracket (s), \text{']'}) & \text{if } d = d'[t] \\
\text{'d'} & \text{otherwise,}
\end{cases}$$

where $s$ is an environment and $\llbracket t \rrbracket (s)$ is the integer value of the term $t$, defined later in this section.

It is an error if the type of the expression indexing an array is not a subrange. When the types of both the expression and the array index are subranges, it is acceptable for the two types to be different, so long as the index value falls within the bounds of the array index type. Out-of-range index values result in a run-time error. A program with a possible bounds error can always be rewritten to test for the bounds error explicitly and execute an error statement if a bounds error occurs. In order to simplify the formal semantics and proofs, this is assumed, so that bounds errors cannot occur in a well-formed program. In particular, if there are no bounds errors, the string obtained by $\llbracket d \rrbracket$ always represents a location in $V$, when $d$ is a variable of a subrange type.

Translating the Terms and Boolean Expressions

For a term $t$, the meaning $\llbracket t \rrbracket : env \to \mathbb{N}_+$ is defined as:

$$\llbracket t \rrbracket (s) = \begin{cases} 
n & \text{if } t = n \text{ for some integer constant } n \\
\text{s}(\llbracket d \rrbracket (s)) & \text{if } t \text{ is a subrange variable reference } d \\
\llbracket t_1 \rrbracket (s) + \llbracket t_2 \rrbracket (s) & \text{if } t = t_1 + t_2
\end{cases}$$

Similarly, the semantics can be extended to other operators. A term is illegal if its type is not a subrange type. (records and arrays are not terms).
Boolean expressions are handled analogously to terms. One of the cases is universal quantification, which uses an index variable:

$$[\forall (x : \alpha) e](s) = \bigwedge_{i \in \alpha} e(s \bullet (x \mapsto i)).$$

In this case, $\alpha$ must be a subrange.

Translating the Statements

For a statement $sm$, the meaning $[sm] : env \rightarrow env$ is defined as follows:

Given an environment $s$ and an assignment $d := t$, the meaning $[d := t](s)$ is $s \bullet (\lfloor d \rfloor(s) \mapsto \lfloor t \rfloor(s))$. It is an error to assign an out-of-bounds value to a location of subrange type. As with array indices, it is assumed that programs explicitly tests bounds, so that a bounds error never occurs during an assignment.

Given an environment $s$ and an if statement, the meaning is defined as:

$$[\text{if } b \text{ then } sm \text{ end}](s) = \begin{cases} [sm](s) & \text{if } [b](s) \text{ is true} \\ s & \text{otherwise} \end{cases}$$

The meaning of an compound statement is $[sm_1; sm_2](s) = [sm_2]( [sm_1](s) )$, and execution of an error statement results in an error environment: $[\text{error}](s) = \text{error}$.

Given an environment $s$ and a for statement, the meaning is defined as:

$$[\text{for } (x : \alpha) \text{ do } sm \text{ end}](s) = [sm_x(ub)](\ldots [sm_x(ub - 1)](\ldots [sm_x(lb)](s)\ldots))$$

where $lb$ and $ub$ are the lower bound and upper bound of the type $\alpha$, and we use $\lfloor sm_x(k) \rfloor(s)$ to denote $[sm](s \bullet (x \mapsto k))$.

Translating the Rules

The translation function for rules, $Tr$, maps a transition rule $tr$ (or initialization rule $ir$), and an environment $s$, to a set of transition functions. Each transition function is a function from states to states (environments that map every index location to $\perp$). In the following definition, $s$ is an environment which captures only the values of the $\square$ index location and $s_0$ represents a state to which an individual transition function
APPENDIX A. SEMANTIC ANALYSIS AND PROOFS

can be applied; the transition function extends $s_0$ by $s$ to bind the $\square$ index locations.

$$Tr(t, s) = \begin{cases} 
Tr(t_1, s) \cup Tr(t_2, s) & \text{if } t = t_1 \square t_2 \\
\cup_{i \in \alpha} Tr(t', s \bullet (x' \mapsto i)) & \text{if } \square (x : \alpha)t' \\
\{\lambda s_0. \text{tostate}([sm](s_0 \bullet s))\} & \text{if } t \text{ is a statement } sm.
\end{cases}$$

Therefore, for a source program with initialization rule $ir$ and transition rule $tr$, the set of transition functions in the abstract transition program, $\mathcal{R}$, is $Tr(tr, \langle \rangle)$, and the set of constant initialization functions, $\mathcal{I}$, is $\{f' \mid \exists f \in Tr(ir, \langle \rangle) : f'(\langle \rangle) = f(\langle \rangle)\}$.

The following lemma explains why it is possible to verify programs using explicit state enumeration:

**Lemma A.1** The abstract transition program derived from a source program has a finite number of transition functions in $\mathcal{R}$. The state graph has a finite number of states in $Q$.

**Proof.** All subranges are bounded, so there are only a finite number of locations. Every location has a finite range of values corresponding to its type, and so the state graph has a finite number of states. Since subranges have finite sizes, the $\square$ construct generates a finite number of transition functions. \qed

### A.2 Automorphism Induced by Scalarsets

Given a program with a scalarset $\alpha$, a permutation $\pi_\alpha$ on an environment maps the value $v$ of each location of type $\alpha$ to a new value, $\pi_\alpha(v)$. It also rearranges the locations when arrays are indexed by $\alpha$. The next three definitions define this concept precisely.

**Definition A.2 (permutation on values)** If $\alpha$ is a scalarset type, a permutation $\pi_\alpha : (\alpha \cup \{\bot\}) \to (\alpha \cup \{\bot\})$ is a bijection such that $\pi_\alpha(\bot) = \bot$.

A permutation can be applied to a location, with the effect of modifying array indices of type $\alpha$ and nothing else.
Definition A.3 (permutation on strings/locations) A permutation \( \pi_\alpha \) on \( \alpha \) can be extended to a permutation \( \pi_\alpha : \text{string} \to \text{string} \), defined as:

\[
\pi_\alpha (l) = \begin{cases} 
  l & \text{if } l \text{ is not of the form } \text{concat}(l', \cdot, f') \\
  \text{concat}(\pi_\alpha (l'), \cdot, f') & \text{if } l = \text{concat}(l', \cdot, f') \\
  \text{concat}(\pi_\alpha (l'), [\cdot, \pi_\alpha (k), \cdot]) & \text{if } l = \text{concat}(l', [\cdot, k, \cdot]) \text{ and the array} \\
  \text{concat}(\pi_\alpha (l'), [\cdot, k, \cdot]) & \text{corresponding to } l' \text{ has index type } \alpha \\
  \text{concat}(\pi_\alpha (l'), [\cdot, k, \cdot]) & \text{corresponding to } l' \text{ has index type other than } \alpha
\end{cases}
\]

For example, with an array of processors:

\[
\text{type } Pid : \text{scalarset} \ (2) \\
\text{var } P : \text{array} [ Pid ] \text{ of record CacheState : } 0..3; \text{ CacheValue : } 0..1 \text{ end}
\]

a permutation \( \pi_{\text{Pid}} \) mapping 0 to 1 will map location ‘P[0], CacheState’ to location ‘P[1], CacheState’.

A permutation on an environment or a state permutes its domain, permutes its codomain, and maps the \textbf{error} environment to itself.

Definition A.4 (permutation on environments/state) A permutation \( \pi_\alpha \) on \( \alpha \) can be extended to a permutation \( \pi_\alpha : \text{env} \to \text{env} \), defined as:

\[
\pi_\alpha (s) = \begin{cases} 
  \langle \rangle & \text{if } s = \langle \rangle \\
  (\pi_\alpha (l) \mapsto \pi_\alpha (v)) & \text{if } s = (l \mapsto v) \text{ and the type of location } l \text{ is } \alpha \\
  (\pi_\alpha (l) \mapsto v) & \text{if } s = (l \mapsto v) \text{ and the type of location } l \text{ is not } \alpha \\
  (\pi_\alpha (s_1)) \cdot (\pi_\alpha (s_2)) & \text{if } s = s_1 \cdot s_2 \\
  \text{error} & \text{if } s = \text{error}
\end{cases}
\]

The reader should note that by this definition, \( \pi_\alpha (s)(\pi_\alpha (l)) = \pi_\alpha (s(l)) \) if the type of \( l \) is \( \alpha \) and \( \pi_\alpha (s)(\pi_\alpha (l)) = s(l) \) otherwise.

The main result of this section is the following theorem.

Theorem A.1 (soundness theorem) Given a source program containing a scalarset \( \alpha \), every permutation \( \pi_\alpha \) on the states of the state graph \( A \) derived from the program is an automorphism on \( A \).

The proof of this theorem requires five lemmas. The soundness theorem and the lemmas are about the properties of a state graph derived from a source program. The
translation defined above is in two steps: the language semantics define an abstract transition program, and the abstract transition program defines a state graph. In the lemmas and the proof of the theorem, Let \( \langle L, T, I, R \rangle \) be the abstract transition program. All environments bind state and index locations from the source program, and all program constructs are assumed to appear in the original source program.

**Lemma A.2 (basic lemma)** For a scalar set \( \alpha \) and an environment \( s \), if \( d \) is a variable reference, then \( [d](\pi_\alpha(s)) = \pi_\alpha([d](s)) \). If \( t \) is a term of type \( \alpha \), then \( [t](\pi_\alpha(s)) = \pi_\alpha([t](s)) \); otherwise, \( [t](\pi_\alpha(s)) = [t](s) \).

For example, in a directory-based cache coherence protocol with the state in Figure 3.3, the processor state \( P[Dir] \) corresponding to the processor \( id \) in directory \( Dir \) should always point to the same processor, no matter how we permute the state:

\[
\begin{align*}
[P[Dir]](s') &= s('P[2]') = 1 \\
\pi_\alpha([P[Dir]](s)) &= \pi_\alpha(s('P[1]')) = \pi_\alpha(2) = 1
\end{align*}
\]

**Proof.** We prove both parts of the lemma in a single induction. For all variable references \( d \), if \( d \) does not have any record field reference or array indexing,

\[
[d](\pi_\alpha(s)) = 'd' \quad \text{definition of } [d] \\
= [d](s) \quad \text{definition of } [d]
\]

If \( d = d'.f \),

\[
[d](\pi_\alpha(s)) = \text{concat}([d'](\pi_\alpha(s)), 'f') \quad \text{definition of } [d'.f] \\
= \text{concat}(\pi_\alpha([d'](s)), 'f') \quad \text{induction hypothesis} \\
= \pi_\alpha(\text{concat}([d'](s), 'f')) \quad \text{permutation on a string/location} \\
= \pi_\alpha([d'.f](s)) \quad \text{definition of } [d'.f].
\]

If \( d = d'[t] \),

\[
\begin{align*}
[d](\pi_\alpha(s)) &= \text{concat}([d'](\pi_\alpha(s)), 't', [t](\pi_\alpha(s)), 't') \\
&\quad \text{definition of } [d'[t]] \\
&\quad \text{concat}(\pi_\alpha([d'](s)), 't', [t](s), 't') \\
\quad \text{if the type of } t \text{ is } \alpha \\
&\quad \text{concat}(\pi_\alpha([d'](s)), 't', [t](s), 't') \\
\quad \text{if the type of } t \text{ is not } \alpha \\
= \pi_\alpha(\text{concat}([d'](s), 't', [t](s), 't')) \quad \text{induction hypothesis} \\
= \pi_\alpha([d'[t]](s)) \quad \text{permutation on a string/location} \\
&\quad \text{definition of } [d'[t]].
\end{align*}
\]
For all terms \( t \), if \( t \) is an integer constant, the meaning is independent of \( s \), and the lemma holds trivially. If \( t \) is a variable reference \( d \) of type \( \alpha \),

\[
\llbracket t \rrbracket (\pi_\alpha(s)) = \pi_\alpha(s)(\llbracket d \rrbracket (\pi_\alpha(s))) \quad \text{definition of} \ \llbracket t \rrbracket \\
= \pi_\alpha(s)(\pi_\alpha(\llbracket d \rrbracket (s))) \quad \text{induction hypothesis} \\
= \pi_\alpha(s)[d](s)) \quad \text{permutation on an environment} \\
= \pi_\alpha(\llbracket t \rrbracket (s)) \quad \text{definition of} \ \llbracket t \rrbracket.
\]

If \( t \) is a variable reference \( d \) of type other than \( \alpha \),

\[
\llbracket t \rrbracket (\pi_\alpha(s)) = \pi_\alpha(s)(\llbracket d \rrbracket (\pi_\alpha(s))) \quad \text{definition of} \ \llbracket t \rrbracket \\
= \pi_\alpha(s)(\pi_\alpha(\llbracket d \rrbracket (s))) \quad \text{induction hypothesis} \\
= s(\llbracket t \rrbracket (s)) \quad \text{permutation on an environment} \\
= \llbracket t \rrbracket (s) \quad \text{definition of} \ \llbracket t \rrbracket.
\]

Finally, if \( t = t_1 + t_2 \), \( t_1 \) and \( t_2 \) cannot be of type \( \alpha \). By induction \( \llbracket t_1 \rrbracket (\pi_\alpha(s)) = \llbracket t_1 \rrbracket (s) \) and \( \llbracket t_2 \rrbracket (\pi_\alpha(s)) = \llbracket t_2 \rrbracket (s) \). So \( \llbracket t_1 + t_2 \rrbracket (\pi_\alpha(s)) = \llbracket t_1 \rrbracket (\pi_\alpha(s)) + \llbracket t_2 \rrbracket (\pi_\alpha(s)) = \llbracket t_1 \rrbracket (s) + \llbracket t_2 \rrbracket (s) = \llbracket t_1 + t_2 \rrbracket (s) \).

The proof would be similar for other operations on terms. \hfill \Box

**Lemma A.3 (Boolean lemma)** If \( \alpha \) is a scalarset, for every environment \( s \) and every Boolean expression \( e \), \( \llbracket e \rrbracket (\pi_\alpha(s)) = \llbracket e \rrbracket (s) \).

The intuition behind this lemma is that since the system is symmetrical and the two states are equivalent, they should satisfy the same set of predicates.

**Proof.** We prove the lemma by induction on the structure of the expression.

When \( e \) is of the form \( t_1 > t_2 \), the types of both \( t_1 \) and \( t_2 \) cannot be \( \alpha \). Therefore, \( \llbracket t_1 \rrbracket (\pi_\alpha(s)) = \llbracket t_1 \rrbracket (s) \) and \( \llbracket t_2 \rrbracket (\pi_\alpha(s)) = \llbracket t_2 \rrbracket (s) \), which implies that \( \llbracket t_1 > t_2 \rrbracket (\pi_\alpha(s)) = \llbracket t_1 > t_2 \rrbracket (s) \).

When \( e \) is of the form \( t_1 = t_2 \), if the type of one of the terms is \( \alpha \), the type of both \( t_1 \) and \( t_2 \) are \( \alpha \), therefore, \( \llbracket t_1 \rrbracket (\pi_\alpha(s)) = \pi_\alpha(\llbracket t_1 \rrbracket (s)) \) and \( \llbracket t_2 \rrbracket (\pi_\alpha(s)) = \pi_\alpha(\llbracket t_2 \rrbracket (s)) \). Since \( \pi_\alpha \) is a bijection, the equivalence or inequivalence of \( t_1 \) and \( t_2 \) is preserved by \( \pi_\alpha \).
On the other hand, if neither of the terms is of type $\alpha$, $\sem[t_1](\pi_\alpha(s)) = \sem[t_1](s)$ and $\sem[t_2](\pi_\alpha(s)) = \sem[t_2](s)$, which implies that $\sem[t_1 = t_2](\pi_\alpha(s)) = \sem[t_1 = t_2](s)$.

For the induction part, the cases for $\neg$ and $\land$ are straightforward. The case of $\forall$ when the quantification is over some type other than $\alpha$ follows immediately by the induction hypothesis. The case where the quantification is over $\alpha$ is also quite simple:

\[
\sem[\forall(x : \alpha) e](\pi_\alpha(s)) = \sem[e](\pi_\alpha(s) \cdot ('x' \mapsto i))
= \sem[e](\pi_\alpha(s) \cdot ('x' \mapsto \pi_\alpha(i)))
\]
definition of $\forall(x : \alpha)$

\[
\varepsilon \pi_\alpha(s) \cdot (\pi_\alpha('x') \mapsto \pi_\alpha(i))
\]
commutativity and associativity of $\land$

\[
\varepsilon \pi_\alpha(s) \cdot (\pi_\alpha('x') \mapsto i)
\]
permutation on an environment

\[
\sem[\forall(x : \alpha) e](s)
= \sem[\forall(x : \alpha) e](s)
\]
definition of $\forall(x : \alpha)$.

\[\square\]

**Lemma A.4 (statement lemma)** If $\alpha$ is a scalar set, for every environment $s$ and every statement $sm$, $\sem[sm](\pi_\alpha(s)) = \pi_\alpha(\sem[sm](s))$.

**Proof.** The proof is by induction on the structure of statements. A statement can be an **error**, assignment, **if**, **for** or a compound statement.

An **error** statement results in an **error** environment regardless of its environment, so the theorem is trivial in this case.

For a statement $sm$ of the form $d := t$,

\[
\sem[d := t](\pi_\alpha(s)) = \pi_\alpha(s) \cdot (\sem[d](\pi_\alpha(s)) \mapsto \sem[t](\pi_\alpha(s)))
\]
definition of $\sem[d := t]$

\[
\begin{cases}
\pi_\alpha(s) \cdot (\pi_\alpha(\sem[d](s)) \mapsto \sem[t](s)) \\
\pi_\alpha(s) \cdot (\pi_\alpha(\sem[d](s)) \mapsto \pi_\alpha(\sem[t](s)))
\end{cases}
\]
if the type of $t$ is $\alpha$

\[
\pi_\alpha(s) \cdot (\pi_\alpha(\sem[d](s)) \mapsto \pi_\alpha(\sem[t](s)))
\]
if the type of $t$ is not $\alpha$

\[
\pi_\alpha(s \cdot (\sem[d](s) \mapsto \sem[t](s)))
\]
permutation on an environment

\[
\pi_\alpha(\sem[d := t](s))
\]
definition of $\sem[d := t]$. 

\[\square\]
A.2. AUTOMORPHISM INDUCED BY SCALARSETS

For a statement \( sm \) of the form if \( b \) then \( sm' \) end, if \( \llbracket b \rrbracket(s) \) is true, then \( \llbracket b \rrbracket(\pi_\alpha(s)) \) is also true, by the Boolean lemma, so \( \llbracket sm \rrbracket(\pi_\alpha(s)) = \llbracket sm' \rrbracket(\pi_\alpha(s)) \). By the induction hypothesis, \( \llbracket sm' \rrbracket(\pi_\alpha(s)) = \pi_\alpha(\llbracket sm' \rrbracket(s)) = \pi_\alpha(\llbracket sm \rrbracket(s)) \). If \( \llbracket b \rrbracket(s) \) is false then \( \llbracket b \rrbracket(\pi_\alpha(s)) \) is also false, so \( \llbracket sm \rrbracket(\pi_\alpha(s)) = \pi_\alpha(s) = \pi_\alpha(\llbracket sm \rrbracket(s)) \).

For a statement \( sm \) of the form \( sm_1; sm_2 \), by the induction hypothesis,

\[
\begin{align*}
\llbracket sm \rrbracket(\pi_\alpha(s)) &= \llbracket sm_2 \rrbracket(\llbracket sm_1 \rrbracket(\pi_\alpha(s))) \\
&= \llbracket sm_2 \rrbracket(\pi_\alpha(\llbracket sm_1 \rrbracket(s))) \\
&= \pi_\alpha(\llbracket sm_2 \rrbracket(\llbracket sm_1 \rrbracket(s))) \\
&= \pi_\alpha(\llbracket sm \rrbracket(s)).
\end{align*}
\]

For a statement \( sm \) of the form for \( (v : \alpha) \) do \( sm' \) end,

\[
\begin{align*}
\llbracket sm \rrbracket(\pi_\alpha(s)) &= \llbracket sm\_v(ub) \rrbracket(\llbracket sm\_v(ub - 1) \rrbracket(\ldots \llbracket sm\_v(lb) \rrbracket(\pi_\alpha(s)) \ldots)) \\
&= \llbracket sm\_v(\pi_\alpha(ub)) \rrbracket(\llbracket sm\_v(\pi_\alpha(ub - 1)) \rrbracket(\ldots \llbracket sm\_v(\pi_\alpha(lb)) \rrbracket(\pi_\alpha(s)) \ldots)) \\
&= \pi_\alpha(\llbracket sm\_v(ub) \rrbracket(\llbracket sm\_v(ub - 1) \rrbracket(\ldots \llbracket sm\_v(lb) \rrbracket(s) \ldots))) \\
&= \pi_\alpha(\llbracket sm \rrbracket(s)) \\
&= \pi_\alpha(\llbracket sm \rrbracket(s)).
\end{align*}
\]

When \( f = \lambda s_0.\text{tostate}(\llbracket sm \rrbracket(s_0 \bullet s)) \), we define \( \pi_\alpha(f) \) to be \( \lambda s_0.\text{tostate}(\llbracket sm \rrbracket(s_0 \bullet \pi_\alpha(s))) \). In other words, \( \pi_\alpha \) is applied to the \( \square \) index locations that are bound in \( f \).

From the definition of \( \text{Tr} \), \( f \) is a member of \( \mathcal{R} \) exactly when \( \pi_\alpha(f) \) is a member of \( \mathcal{R} \).

**Lemma A.5 (transition lemma)** If \( \alpha \) is a scalarset type, \( s \) is a state, and \( f \) is a transition function in \( \mathcal{R} \), then \( \pi_\alpha(f)(\pi_\alpha(s)) = \pi_\alpha(f)(s) \).

**Proof.** Let \( f = \lambda s_0.\text{tostate}(\llbracket sm \rrbracket(s_0 \bullet s)) \) be a transition function in \( \mathcal{R} \) and \( s_0 \) be a state.

\[
\begin{align*}
\pi_\alpha(f)(\pi_\alpha(s_0)) &= \text{tostate}(\llbracket sm \rrbracket(\pi_\alpha(s_0) \bullet \pi_\alpha(s))) \\
&= \text{tostate}(\llbracket sm \rrbracket(\pi_\alpha(s_0) \bullet s)) \\
&= \text{tostate}(\pi_\alpha(\llbracket sm \rrbracket(s_0 \bullet s))) \\
&= \pi_\alpha(f(s_0))
\end{align*}
\]

\( \text{tostate} \) and \( \pi_\alpha \) commute. \( \square \)
Lemma A.6 (initialization lemma) \( f \in \mathcal{I} \) always implies \( \pi_\alpha(f) \in \mathcal{I} \).

**Proof.** It follows from the definition of \( \mathcal{T} \). \(\square\)

Now we can complete the proof of the soundness theorem:

**Proof.** (of the soundness theorem)

It is now quite simple to show that \( \pi_\alpha \) meets the conditions for an automorphism. First, \( \pi_\alpha \) on environments is a bijection.

The definition of an automorphism has three additional “if and only if” conditions. We prove implications, but since \( \pi_\alpha^{-1} \) is also a permutation, the converse of each implication holds, as well.

The translation of a well-formed source program to an abstract transition program guarantees that, for every \( (s, s') \in \Delta \), there is a transition function \( f \) such that \( s' = f(s) \). Because \( \pi_\alpha(f) \) is a transition function whenever \( f \) is, and by the transition lemma, \( \pi_\alpha(f)(\pi_\alpha(s)) = \pi_\alpha(f(s)) = \pi_\alpha(s') \), therefore, \( (\pi_\alpha(s), \pi_\alpha(s')) \) is in \( \Delta \).

Similarly, for every \( q \in Q_0 \), there exists \( f \in \mathcal{I} \) such that \( f() = q \). By the initialization lemma, there exists \( \pi_\alpha(f) \in \mathcal{I} \), and \( \pi_\alpha(q) = \pi_\alpha(f()) \in Q_0 \).

Finally, \( \pi_\alpha(\text{error}) = \text{error} \), by definition. \(\square\)

Every permutation of every scalarset is an automorphism on the state graph. If a source program has more than one scalarset, the union of all the permutations represents a larger set of automorphisms, enabling a larger reduction of the state graph. The verification using multiple symmetries at the same time is also sound, as the consequence of the theorems in Section 3.2:

**Corollary A.1** If a program \( P \) has scalarsets \( \alpha_1, \ldots, \alpha_n \), there are symmetries in the state graph \( A \) and we can use the symmetry-reduced state graph \( A/\approx_H \) to verify the properties, where \( H \) is the set of all permutations on the states w.r.t. \( \alpha_1, \ldots, \alpha_n \).
A.3 Data Saturation Induced by Data Scalarsets

In this section, we prove that the *data saturation* phenomenon occurs with data scalarset:

**Theorem A.2 (data saturation)** If $P$ is a Murϕ description, $α$ is the name of a data scalarset in $P$, $α$ is declared to be of size $N_1$ in $P_1$ and $N_2$ in $P_2$, then there exists a size $N_α$ such that the symmetry-reduced state graphs of $P_1$ and $P_2$ are isomorphic whenever $N_1 ≥ N_α$ and $N_2 ≥ N_α$.

To prove the data saturation theorem, we need to prove several lemmas. Let us first establish some assumptions and notation to be used throughout the rest of the section.

**The Source Programs**

The proof involves two source programs $P_1$ and $P_2$ containing a declaration of a data scalarset type named $α$. In general, symbols with subscript 1 pertain to $P_1$ and symbols with subscript 2 pertain to $P_2$. $P_1$ and $P_2$ are identical in every way except for the declaration of $α$. In $P_1$, $α$ is declared to be of size $N_1$, and the set of its elements is referred as $α_1$. Similarly, in $P_2$, $α$ is declared to be of size $N_2$, and the set of its elements is referred as $α_2$.

Since the data scalarset $α$ cannot be used as an array index, the set of locations in $P_1$ is exactly the same as the set of locations of $P_2$ (the only difference is that the locations with type $α$ have different possible values in the environments in $P_1$ and $P_2$). We call this set of locations $V$. Let $N_α$ be the number of locations of type $α$. We assume that $N_1 ≥ N_α$ and $N_2 ≥ N_α$.

When we refer to an environment below, it means a member of $[V → N_1]$, unless otherwise specified. For $j = 1, 2$, $s_j$ refers to an environment of $P_j$, and $β_j ⊆ α_j$ is the set of values actually appearing in the environment, that is,

$$\{s(l) \mid l \text{ is a location of type } α \text{ and } s(l) ≠ ⊥\}.$$

The meaning of the program text may be different in $P_1$ and $P_2$ because of the difference in the size of $α$. For a program construct $c$, let $[c]_1$ and $[c]_2$ denote the
meaning of $c$ in $P_1$ and $P_2$. For a rule $t$, let $Tr_1$ and $Tr_2$ denote the translation function for $t$ in $P_1$ and $P_2$. The difference in meaning only occurs when $\alpha$ appears literally in the text, that is, in $\square$ constructs of the form $\square(x: \alpha) t$ and in quantified Boolean expressions. Specifically, for $j = 1, 2$,

$$Tr_j(\square(x: \alpha) t, s) = \bigcup_{i \in \alpha_j} Tr_j(t, s \bullet (x' \mapsto i))$$

$$\forall(x: \alpha) b]_j(s) = \bigwedge_{i \in \alpha_j} [b]_j(s \bullet (x' \mapsto i)).$$

A statement may contain a Boolean expression, so $[sm]_1$ may differ from $[sm]_2$, also. These functions can be defined recursively by adding the appropriate subscript to $[\ ]$ uniformly in the original definitions.

When $\alpha$ does not appear in a program construct $c$, $[c]_1 = [c]_2$. In particular, $[d]_1 = [d]_2$ for all variables and $[t]_1 = [t]_2$ for all terms, since variables and terms never contain a literal $\alpha$.

In this section, the same permutation must be applied to several different types, so a permutation is redefined to be a bijection on $N_{\bot}$ which maps $\bot$ to itself. Throughout the section, only locations of type $\alpha$ are permuted.

**The General Canonicalizing Permutation**

Given an environment $s$, we construct a canonicalizing permutation $\pi_s$ on $N_{\bot}$. First of all, we fix a total order on the locations of type $\alpha$ in $V$: $l_1, \ldots, l_{N_{\alpha}}$ (any total order will suffice). Therefore, the environments are lexicographically ordered by the values in the locations of type $\alpha$. We define $\pi_s$ to map each environment $s$ to the lexicographically minimum equivalent environment $\pi_s(s)$:

- $\pi_s(\bot) = \bot$,

- for $i > 0$, if $s(l_i) \neq \bot$ and $s(l_i) \neq s(l_k)$ for every $k < i$, $\pi_s(s(l_i))$ is assigned the least nonnegative integer that has not been assigned to some $\pi_s(s(l_k))$, where $k < i$. 

• finally, for other value \( x \) such that \( x \neq s(l) \), for all locations \( l \) of type \( \alpha \), \( \pi_s(x) \) can be assigned in any way that makes \( \pi_s \) a permutation.

A canonicalization function \( \zeta \) can be defined such that \( \zeta(s) = \pi_s(s) \). The symbol \( \zeta \) will refer to this function for the remainder of the section.

Lemma A.7 \( \zeta \) is a canonicalization function.

Proof. We must demonstrate, for every equivalence class \([s]\), \( \zeta(s) \in [s] \) and that \( s' \in [s] \) implies \( \zeta(s') = \zeta(s) \). The first part is obvious, since by definition \( \zeta(s) = \pi_s(s) \).

Now, suppose \( \zeta(s') \neq \zeta(s) \). If there is a location of type other than \( \alpha \) such that \( \zeta(s')(l) \neq \zeta(s)(l) \), then \( s' \) cannot be equivalent to \( s \) because a permutation on \( \alpha \) cannot modify \( s(l) \) or \( s'(l) \). If \( \zeta(s') \) and \( \zeta(s) \) agree on all non-\( \alpha \) locations, let \( l \) be the least \( \alpha \) location at which they disagree, according to the total order on \( \alpha \)-locations used in the definition of \( \zeta \). Without loss of generality, suppose that \( \zeta(s)(l) < \zeta(s')(l) \). Then, by definition, there exists some \( l' < l \) such that \( \zeta(s)(l) = \zeta(s)(l') \) and \( \zeta(s')(l) \neq \zeta(s')(l') \). But then every permutation \( \pi \) will have \( \pi(s)(l) \neq \pi(s')(l) \) or \( \pi(s)(l') \neq \pi(s')(l') \), so \( s \) and \( s' \) cannot be equivalent.

□

Definition A.5 (\( \beta_1-\beta_2 \)-permutation) For \( \beta_1, \beta_2 \subseteq \mathbb{N} \), a \( \beta_1-\beta_2 \)-permutation \( \pi \) on \( \mathbb{N}_\bot \) is a permutation such that \( \pi(\bot) = \bot \) and for all \( v \in \beta_1 \), \( \pi(v) \in \beta_2 \), and vice versa.

The reader should note that a \( \beta_1-\beta_2 \)-permutation exists only if \( |\beta_1| = |\beta_2| \). The following lemma clarifies the relation between two states (which may be from different state graphs) that map to the same state under \( \zeta \).

Lemma A.8 (zeta equivalence lemma) Suppose that \( s_1 \) and \( s_2 \) are two environments. Then, \( \zeta(s_1) = \zeta(s_2) \) if and only if there exists a \( \beta_1-\beta_2 \)-permutation \( \pi \) such that \( \pi(s_1) = s_2 \).

Proof.

\( \Leftarrow \) Suppose there exists a \( \beta_1-\beta_2 \)-permutation \( \pi \) such that \( \pi(s_1) = s_2 \). Let \( \pi_{s_1} \) be the canonicalizing permutation for \( s_1 \), as defined above. Since \( \zeta(s_1) = \pi_{s_1}(s_1) \)
\[ \pi_s(\pi^{-1}(s_2)), \zeta(s_1) \] is equivalent to \( s_2 \). Because \( \zeta(s_1) \) is the lexicographical minimum, which is unique, the canonicalizing permutation \( \pi_{s_2} \) for \( s_2 \) is \( \pi_s \circ \pi^{-1} \).

Therefore,
\[
\zeta(s_2) = \pi_{s_2}(s_2) \\
= \pi_s(\pi^{-1}(s_2)) \\
= \pi_s(\pi^{-1}(\pi(s_1))) \\
= \pi_s(s_1) = \zeta(s_1).
\]

\( \Rightarrow \) By definition, there exist permutations \( \pi_{s_1} \) and \( \pi_{s_2} \) such that \( \zeta(s_1) = \pi_{s_1}(s_1) \) and \( \zeta(s_2) = \pi_{s_2}(s_2) \). \( \pi_{s_1}(s_1) = \pi_{s_2}(s_2) \), so for all \( v_1 \in \beta_1 \), there exists \( v_2 \in \beta_2 \) such that \( \pi_{s_1}(v_1) = \pi_{s_2}(v_2) \), and vice versa. Hence \( \pi = \pi_{s_2} \circ \pi_{s_1} \) is a \( \beta_1 \)-\( \beta_2 \)-permutation.

The next lemma asserts that for sufficiently large \( \alpha_1 \) and \( \alpha_2 \), if we have equivalent environments \( s_1 \) and \( s_2 \), and \( s_1 \) is extended to \( s'_1 \) by assigning a value to a previously undefined \( \alpha \)-location, we can extend \( s_2 \) to an environment \( s'_2 \) which is equivalent to \( s'_1 \).

**Lemma A.9 (permutation extension lemma)** Suppose \( s_1 \) and \( s_2 \) are two environments, \( \pi \) is a \( \beta_1 \)-\( \beta_2 \)-permutation such that \( s_2 = \pi(s_1) \), and \( s_1(l) = s_2(l) = \perp \) for some location \( l \) of type \( \alpha \).

For every \( i_1 \) in \( \alpha_1 \), there exists an \( i_2 \in \alpha_2 \) and a \((\beta_1 \cup \{i_1\})-(\beta_2 \cup \{i_2\})\)-permutation \( \pi' \) such that \( s_2 \bullet (l \mapsto i_2) = \pi'(s_1 \bullet (l \mapsto i_1)) \).

**Proof.** If \( i_1 \in \beta_1 \), set \( i_2 = \pi(i_1) \), and \( \pi' = \pi \). Otherwise, since \( s_2(l) = \perp \), \( |eta_2| < N_\alpha \), and since \( |\alpha_2| \geq N_\alpha \), there exists at least one value \( i_2 \in \alpha_2 - \beta_2 \). Therefore, we can choose an arbitrary value of \( i_2 \) from \( \alpha_2 - \beta_2 \), and define a \((\beta_1 \cup \{i_1\})-(\beta_2 \cup \{i_2\})\)-permutation \( \pi' \) such that \( \pi'(i_1) = i_2 \), and for \( i \in \beta_1 \), \( \pi'(i) = \pi(i) \) (for \( i \notin \beta_1 \cup \{i_1\} \), there are no constraints on \( \pi'(i) \) so long as it remains a permutation).  

**The Saturation Lemmas**

After the definition of the canonicalization function, we can proceed to prove the lemmas to support the data saturation theorem. The following asserts that a Boolean expression cannot distinguish between two scalar set values that could be assigned to a location, if those values do not appear in some other location in the same environment.
Lemma A.10 (interchangeability lemma) Suppose \(s_1\) and \(s_2\) are environments, \(b\) is a Boolean expression, \(l\) is a location of type \(\alpha\), and \(i\) and \(i'\) are values of type \(\alpha\).

If \(i \neq s_j(l)\) and \(i' \neq s_j(l)\) for every location \(l\) of type \(\alpha\), then \([b]_j(s_j \cdot ('x' \mapsto i)) = [b]_j(s_j \cdot ('x' \mapsto i'))\).

**Proof.** Let \(\pi\) be the permutation that swaps \(i\) and \(i'\) and leaves all other values fixed. Clearly \(\pi_j(s_j) = s_j\), since the range of \(s_j\) does not contain \(i\) or \(i'\). By the Boolean lemma, \([b]_j(s_j \cdot ('x' \mapsto i)) = [b]_j(s_j \cdot ('x' \mapsto i'))\). □

Lemma A.11 (Boolean saturation lemma) Suppose \(s_1\) and \(s_2\) are environments, and \(\pi\) is a \(\beta_1\)-\(\beta_2\)-permutation such that \(s_2 = \pi(s_1)\). Then, for every Boolean expression \(b\), \([b]_1(s_1) = [b]_2(s_2)\).

**Proof.** The proof is by induction on the structure of Boolean expressions.

For Boolean expressions of the form \(t_1 = t_2\) and \(t_1 \triangleright t_2\), \([b]_1 = [b]_2\), because \(\alpha\) does not appear literally in the expressions. So the lemma follows from the Boolean lemma, where \(\pi\) is the permutation. For Boolean expressions of the form \(\neg b'\) or \(b_1 \land b_2\), the lemma is immediate by the induction hypothesis. The case where \(b\) is \(\forall(x : \alpha')b'\), with \(\alpha' \neq \alpha\) is also straightforward.

Finally, suppose \(b\) is of the form \(\forall(x : \alpha)b'\). By definition, for \(j = 1, 2\), \([b]_j(s_j) = \bigwedge_{i \in \alpha_j} [\mathcal{V}]_j(s_j \cdot ('x' \mapsto i))\). This can be rewritten as \(\bigwedge_{i \in \beta_j} [\mathcal{V}]_j(s_j \cdot ('x' \mapsto i)) \land \bigwedge_{i \in \gamma_j} [\mathcal{V}]_j(s_j \cdot ('x' \mapsto i))\), where \(\gamma_j = \alpha_j - \beta_j\), the members of \(\alpha_j\) not in \(s_j\). We consider the cases where \(i \in \beta_j\) and \(i \in \gamma_j\) separately.

**Case 1:** \(i \in \beta_j\)

We show that \(\bigwedge_{i \in \beta_j} [\mathcal{V}]_1(s_1 \cdot ('x' \mapsto i)) = \bigwedge_{i \in \beta_j} [\mathcal{V}]_2(s_2 \cdot ('x' \mapsto i))\).

If \(i \in \beta_1\), the set of non-\(\perp\) values assigned to locations of type \(\alpha\) in \(s_1 \cdot ('x' \mapsto i)\) remains \(\beta_1\). Since \(\pi\) is a \(\beta_1\)-\(\beta_2\)-permutation, the set non-\(\perp\) values assigned to locations of type \(\alpha\) in \(s_2 \cdot ('x' \mapsto \pi(i))\) remains \(\beta_2\). By the induction hypothesis, for all \(i_1 \in \beta_1\), \([\mathcal{V}]_1(s_1 \cdot ('x' \mapsto i_1)) = [\mathcal{V}]_2(s_2 \cdot ('x' \mapsto \pi(i_1))).\)
Hence, we have
\[
\bigwedge_{i \in \beta_1} [\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto i)) = \bigwedge_{i \in \beta_1} [\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto \pi(i)))
\]
Since the image of \( \beta_1 \) under \( \pi \) is \( \beta_2 \), this is equal to \( \bigwedge_{i \in \beta_2} [\mathcal{B}]_2 (s_2 \cdot ('x' \mapsto i)) \).

**Case 2: \( i \in \gamma_j \)**

We show that \( \bigwedge_{i \in \gamma_1} [\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto i)) = \bigwedge_{i \in \gamma_2} [\mathcal{B}]_2 (s_2 \cdot ('x' \mapsto i)) \).

Because the value of location 'x' in \( s_1 \) and \( s_2 \) is \( \perp \), \( \beta_1 \) and \( \beta_2 \) are smaller than \( N_\alpha \). Since \( \alpha_1 \geq N_\alpha \) and \( \alpha_2 \geq N_\alpha \), neither \( \gamma_1 \) nor \( \gamma_2 \) is empty.

By the interchangeability lemma, \([\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto i)) \) yields the same value for every \( i \in \gamma_1 \), so if \( \gamma_1 \) is nonempty, we may choose a particular value \( i_1 \in \gamma_1 \), knowing that
\[
\bigwedge_{i \in \gamma_1} [\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto i)) = [\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto i_1)).
\]

By the permutation extension lemma, there exists \( i_2 \in \alpha_2 - \beta_2 \) and a \( (\beta_1 \cup \{i_1\}) \)-permutable \( \pi' \) such that
\[
s_2 \cdot ('x' \mapsto i_2) = \pi'(s_1 \cdot ('x' \mapsto i_1)).
\]

Now, by the induction hypothesis,
\[
[\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto i_1)) = [\mathcal{B}]_2 (\pi'(s_1 \cdot ('x' \mapsto i_1))) = [\mathcal{B}]_2 (s_2 \cdot ('x' \mapsto i_2)).
\]

By the interchangeability lemma, again,
\[
[\mathcal{B}]_2 (s_2 \cdot ('x' \mapsto i_2)) = \bigwedge_{i \in \gamma_2} [\mathcal{B}]_2 (s_2 \cdot ('x' \mapsto i)).
\]

Therefore,
\[
\bigwedge_{i \in \gamma_1} [\mathcal{B}]_1 (s_1 \cdot ('x' \mapsto i)) = \bigwedge_{i \in \gamma_2} [\mathcal{B}]_2 (s_2 \cdot ('x' \mapsto i)).
\]

Combining the \( \beta \) and \( \gamma \) cases gives \([\forall (x : \alpha) B]_1 (s_1) = [\forall (x : \alpha) B]_2 (s_2)\). \(\square\)
Lemma A.12 (statement saturation lemma) Suppose $s_1$ and $s_2$ are environments and $\pi$ is a $\beta_1$-$\beta_2$-permutation such that $s_2 = \pi(s_1)$. Then, for every statement $sm$, $\pi([sm]_1(s_1)) = [sm]_2(s_2)$.

Proof. The proof is by induction on the structure of statements. $sm$ can be of several forms:

If $sm$ is error, the result is obvious (independent of $\alpha_i$).

Suppose $sm$ is an assignment, $d := t$. $[\ ]_1 = [\ ]_2$ for both definitions and terms, so in this case, by the statement lemma, $\pi([d := t]_1(s_1)) = [d := t]_2(\pi(s_1)) = [d := t]_2(s_2)$.

If $sm$ is a conditional if $b$ then $sm'$ end, the Boolean saturation lemma asserts that $[b]_1(s_1) = [b]_2(s_2)$. If $[b]_1(s_1)$ is false, $s_1$ and $s_2$ are not changed by $sm$, so the lemma holds. If $[b]_1(s_1)$ is true, then for $j = 1, 2$, $[sm]_j(s_j) = [sm']_j(s_j)$, and by the induction hypothesis, the lemma holds.

The lemma holds when $sm$ is $sm_1; sm_2$ or a for loop, by repeated application of the induction hypothesis (recall that for statement may not be indexed by a data scalarset such as $\alpha$).

The following lemma asserts that for every transition and initialization rule in $P_1$, there is a corresponding rule in $P_2$.

Lemma A.13 (rule saturation lemma) Suppose $s_1$ and $s_2$ are environments, and let $\langle L, \mathcal{T}_j, \mathcal{I}_j, \mathcal{R}_j \rangle$ be the abstract transition program for $P_j$.

For every transition rule $f_1 \in \mathcal{R}_1$, if $\zeta(s_1) = \zeta(s_2)$, there exists $f_2 \in \mathcal{R}_2$ such that $\zeta(f_1(s_1)) = \zeta(f_2(s_2))$. For every initialization rule $f_1 \in \mathcal{I}_1$, there exists $f_2 \in \mathcal{I}_2$ such that $\zeta(f_1()) = \zeta(f_2())$.

Proof.

Let $f_1$ be any rule in $\mathcal{R}_1$. The main issue in the proof is to show that $\square$ index locations bound in $f_1$ can be permuted to give a transition rule $f_2$ in $\mathcal{R}_2$.

By definition, $f_1(s_1) = \text{tostate}([sm]_1(s_1 \cdot s))$, for some statement $sm$. The locations in $s_1$ and $s$ with defined values are disjoint, since $s$ is an environment that assigns values only to $\square$ index locations, and $s_1$ is a state (an environment that assigns $\perp$ to $\square$ index locations),
By the zeta equivalence lemma, there exists a $\beta_1$-$\beta_2$-permutation $\pi$ such that $s_2 = \pi(s_1)$, since $\zeta(s_1) = \zeta(s_2)$. Let $\beta'_1$ be the set of non-$\bot$ $\alpha_2$ values assigned to $s$. By repeated application of the permutation extension lemma, there exists a set $\beta'_2 \subseteq \alpha_2$ and a $(\beta_1 \cup \beta'_1)$-$\beta_2$-$\beta'_2$-permutation $\pi'$ such that $s_2 \cdot \pi'(s) = \pi'(s_1 \cdot s)$.

Since $\pi'(s)$ maps the index locations to the values in $\alpha_2$, by the definition of $Tr_2$, there exists $f_2 = \lambda s_0.\text{tostate}(\llbracket sm \rrbracket_2(s_0 \cdot \pi'(s)))$ in $R_2$. By the statement saturation lemma,

$$\llbracket sm \rrbracket_2(s_2 \cdot \pi'(s)) = \llbracket sm \rrbracket_2(\pi'(s_1 \cdot s)) = \pi'(\llbracket sm \rrbracket_1(s_1 \cdot s)).$$

Because $\pi = \text{tostate} \circ \pi'$, we have $f_2(s_2) = \pi(f_1(s_1))$.

Similarly, if $f_1 \in I_1$, there exists $f_2 \in I_2$ such that $\zeta(f_1) = \zeta(f_2)$. \hfill $\square$

Now we can complete the proof for the data saturation theorem.

**Proof.** (of the data saturation theorem)

For $i = 1, 2$, suppose $\langle L, T_i, I_i, R_i \rangle$ and $A_i = \langle Q_i, Q_{0i}, \Delta_i, \text{error}_i \rangle$ are the abstract transition program and the state graph for $P_i$.

**Step 1:** First we show that $\zeta(Q_1) = \zeta(Q_2)$.

We show that, for every $s_1 \in Q_1$, there exists an $s_2 \in Q_2$ such that $\zeta(s_1) = \zeta(s_2)$. The converse follows by symmetry. For every $s_1 \in Q_1$ and $s_1 \neq \text{error}$, let $\beta_1 \subseteq \alpha_1$ be the set of non-$\bot$ values assigned to locations of type $\alpha$ in $s_1$. Since the number of locations of type $\alpha$ is $N_\alpha$, $|\beta_1| \leq N_\alpha$. Since $|\alpha_2| \geq N_\alpha$, there exists $\beta_2 \subseteq \alpha_2$ of the same size as $\beta_1$ and a $\beta_1$-$\beta_2$-permutation $\pi$. The set of non-$\bot$ values assigned to locations of type $\alpha$ in $s_2 = \pi(s_1)$ is $\beta_2$, therefore, $s_2$ is a member of $Q_2$, and $\zeta(s_1) = \zeta(s_2)$ by the zeta equivalence lemma.

**Step 2:** We show in this step that every transition $(q_1, q'_1)$ in $\zeta(A_1)$, is also transition $(q_1, q'_2)$ in $\zeta(A_2)$. The converse follows by symmetry.

For every transition $(\zeta(q_1), \zeta(q'_1))$ in $\zeta(A_1)$, there exists $s_1, s'_1 \in Q_1$ such that $(s_1, s'_1) \in \Delta_1$. Therefore there exists $f_1 \in R_1$ such that $f_1(s_1) = s'_1$.

From the proof in step 1, we know that there exists a state $s_2 \in Q_2$ such that $\zeta(s_1) = \zeta(s_2)$. By the rule saturation lemma, there exists $f_2 \in R_2$ such that
\[ \zeta(f_1(s_1)) = \zeta(f_2(s_2)). \] Since \((s_2, f_2(s_2)) \in \Delta_2, (\zeta(s_2), \zeta(f_2(s_2)) = (\zeta(s_1), \zeta(s'_1))\) is a transition in \(\zeta(A_2)\).

The proof that the initial states are the same in \(\zeta(A_1)\) and \(\zeta(A_2)\) is similar.

Finally, the error states \(\zeta(A_1)\) and \(\zeta(A_2)\) are isomorphic, since they are connected by the isomorphic transitions.

Since the two canonicalized graphs are isomorphic, the symmetry-reduced state graphs of \(P_1\) and \(P_2\) are isomorphic.

This proof applies equally well when the size of \(\alpha\) is infinite. A system with a scalar set of infinite size cannot be verified using conventional search algorithms, because there can be an infinite number of reachable states. Even worse, use of an infinite scalar set as an index in the \(\square\) construct yields an infinite set of transition functions. However, since the symmetry-reduced state graph is finite in all respects, verification using symmetry reduction is straightforward.

### A.4 Repetitive Property Enforced by RepetitiveIDs

This section presents the proof of the following theorem:

**Theorem A.3** The system described by a Murϕ program with a repetitiveID is symmetric and repetitive w.r.t. the components named by the repetitiveID.

Since repetitiveID is a subtype of scalar set, Theorem A.1 has already shown that a system with a repetitiveID is symmetric w.r.t. the components named by the repetitiveID. The remainder of this section concentrates on showing that the system is repetitive, which is stated here again for convenience:

**Definition A.6 (repetitive property)** A system is repetitive if and only if, given an active representation \((s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\})\) for a transition rule \(t\), we have:

**Similarity in error behaviors:**

\(t\) executes an error statement on \((s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\})\) if and only if it executes an error statement on \((s, q, \{q_1, \ldots, q_k\})\).
Similarity in successors:

t transforms \( (s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\}) \) to \( (s', r, \{r_1^{a_1}, \ldots, r_k^{a_k}\}) \) if and only if t transforms \( (s, q, \{q_1, \ldots, q_k\}) \) to \( (s', r, \{r_1, \ldots, r_k\}) \), where \( r, r_1, \ldots, r_k \) may not be distinct or abstractable.

And similarly for any equivalence class \( p' = (s, \{q_1^{a_1}, \ldots, q_k^{a_k}\}) \) and a transition rule \( t \) without an abstractable active component.

First of all, the restriction 4 in page 96 on repetitiveIDs ensures that there is at most one active component for each transition rule, and this active component can be abstractable or non-abstractable. Therefore, given an equivalence class \( (s, \{q_1^{a_1}, \ldots, q_k^{a_k}\}) \) and a transition rule \( t \), there are three cases to consider.

In the first two cases, if there is no active component associated with \( t \), or if the active component is not abstractable, the state \( (s, q, q_1, q_1, \ldots, q_1, \ldots, q_k) \) can be used as the canonical state to generate the successor after executing \( t \). In the third case, if the active component is abstractable, the active representation \( (s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\}) \) is generated, and the state \( (s, q, q_1, \ldots, q_1, \ldots, q_k, \ldots, q_k) \) can be used as the canonical state to generate the successor after executing \( t \).

To simplify the discussion and proof, the remainder of this section only consider the third case, in which the transition rule \( t \) has an abstractable active component.

We also denote the active representation \( (s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\}) \) as \( p' \), \( (s, q, \{q_1, \ldots, q_k\}) \) as \( p \), the state \( (s, q, q_1, \ldots, q_1, \ldots, q_k, \ldots, q_k) \) as \( c' \), and \( (s, q, q_1, \ldots, q_k) \) as \( c \).

To carry out the proof of Theorem 6.1, the relationship between the abstractable component indices in the states \( c' \) and \( c \) is defined in terms of a pair of mappings:

**Definition A.7** The mapping \( \text{exp} \) maps the abstractable active component in the state \( c = (s, [q,q_1,\ldots,q_k]) \) to the active component in the state \( c' = (s, [q_1,\ldots,q_1,\ldots,q_k,\ldots,q_k]) \). It maps the abstractable component index \( j+1 \) in the state \( c \) to the abstractable component index \( 1 + a_1 + a_2 + \ldots + a_{j-1} + 1 \) in the state \( c' \).

The mapping \( \text{exp}^{-1} \) maps the abstractable active component in the state \( c' \) to the active component in the state \( c \). It maps the abstractable component index \( 1 + a_1 + a_2 + \ldots + a_{j-1} + b \) in the state \( c' \), where \( 1 \leq b \leq a_j \), to the abstractable component index \( j+1 \) in the state \( c \).
Pictorially, the mapping $\exp$ and $\exp^{-1}$ can be summarized as:

\[
\begin{align*}
\exp &: (s, \overline{q, q_1, \ldots, q_1, q_2, \ldots, q_2, \ldots, q_k}) \\
&\quad \rightarrow (s, \overline{q_1, q, \ldots, q, q_2, \ldots, q_2, \ldots, q_k}) \\
\exp^{-1} &: (s, \overline{q, q_1, q_2, \ldots, q_2, \ldots, q_k}) \\
&\quad \rightarrow (s, \overline{q, q_1, q_2, \ldots, q_2, \ldots, q_k})
\end{align*}
\]

For example, given states $c'_1 = (s, \overline{q, q_1, q_1, q_2, q_2})$ and $c_1 = (s, \overline{q, q_1})$, $\exp^{-1}$ maps the 5th and 6th abstractable components in $c'_1$ to the 3rd abstractable component in $c_1$, and $\exp$ maps the 3rd abstractable component in $c_1$ to the 5th abstractable component in $c'_1$.

The restriction 4 and 5 in page 96 on repetitiveIDs guarantees that at most one parameter variable of the repetitiveID type is declared in a ruleset and at most one parameter variable of the repetitiveID type is declared in a for loop or a universal quantifier. To simplify the discussion, we consider only the cases in which there is exactly one ruleset parameter with the repetitiveID type, and the value of this parameter refers to an abstractable active component. In these cases, an active representation $p' = (s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\})$ is used, and the state $c' = (s, \overline{q_1, \ldots, q_1, \ldots, q_k, \ldots, q_k})$ is used as the canonical state to generate the successor after executing $t$. An environment may extend the state $c'$ in two different ways:

1. $(c' \bullet (\langle x' \mapsto 1 \rangle) \cdot c_0)$, and
2. $(c' \bullet (\langle x' \mapsto 1 \rangle) \cdot (\langle y' \mapsto j \rangle) \cdot c_0)$,

where $c_0$ corresponds to the mappings of other parameter variables not of the repetitiveID type, $x$ is the name of the repetitiveID variable declared in a ruleset, $y$ is the name of the variable declared in a for loop or a universal quantifier, and $j$ is a repetitiveID index. Note that the canonical state $c'$ is chosen from the equivalence
class $p'$ so that the active abstractable component is the first abstractable component in $c'$. To simplify the notation, the part $\langle x' \mapsto 1 \rangle$ is combined into $c_0$; and the two environments are denoted as $(c' \bullet c_0)$ and $(c' \bullet \langle y' \mapsto j \rangle \bullet c_0)$.

The proof of the Theorem 6.1 is organized in a similar list of lemmas as the corresponding proof for scalarset, with the permutation $\pi_a$ replaced by $\exp$ and $\exp^{-1}$. Similar to the cases with permutations, $\exp$ and $\exp^{-1}$ on the abstractable component indices are extended to those on string/locations, and environments/state:

**Definition A.8** $\exp$ (and similarly for $\exp^{-1}$) on the abstractable component indices can be extended to $\exp$ on the string/locations, defined as:

$$
\exp(l) = \begin{cases} 
  l & \text{if } l \text{ is not of the form } \text{concat}(l', \langle j', f' \rangle) \\
  \text{concat}(\exp(l'), \langle j', f' \rangle) & \text{if } l = \text{concat}(l', \langle j', f' \rangle) \\
  \text{concat}(\exp(l'), \langle [j', \exp(k), i'] \rangle) & \text{if } l = \text{concat}(l', \langle [j', k, i'] \rangle) \text{ and the array corresponding to } l' \text{ is indexed by the repetitiveID} \\
  \text{concat}(\exp(l'), \langle [j', k, i'] \rangle) & \text{if } l = \text{concat}(l', \langle [j', k, i'] \rangle) \text{ and the array corresponding to } l' \text{ is not indexed by the repetitiveID}
\end{cases}
$$

**Definition A.9** $\exp$ and $\exp^{-1}$ on the abstractable component indices can be extended to $\exp$ and $\exp^{-1}$ on the environments, defined as:

- $\exp(c \bullet c_0) = (c' \bullet c_0)$, and $\exp(c \bullet \langle y' \mapsto j \rangle \bullet c_0) = (c' \bullet \langle y' \mapsto \exp(j) \rangle \bullet c_0)$,
- $\exp^{-1}(c' \bullet c_0) = (c \bullet c_0)$, and $\exp^{-1}(c' \bullet \langle y' \mapsto j \rangle \bullet c_0) = (c \bullet \langle y' \mapsto \exp^{-1}(j) \rangle \bullet c_0)$.

**Lemma A.14** (basic lemma) Given a program with a repetitiveID, a variable reference $d$, a term $t$ in a program, and two environments $e_2$ and $e_1$ obtained by extending states $c'$ and $c$, we have

$$
[d](\exp(e_1)) = \exp([d](e_1)), \text{ and } \exp^{-1}([d](e_2)) = [d](\exp^{-1}(e_2)).
$$

If the type of $t$ is the repetitiveID,

$$
[t](\exp(e_1)) = \exp([t](e_1)), \text{ and } \exp^{-1}([t](e_2)) = [t](\exp^{-1}(e_2)).
$$

Otherwise

$$
[t](\exp(e_1)) = [t](e_1), \text{ and } [t](e_2) = [t](\exp^{-1}(e_2)).
$$
Proof. We prove the first part of the lemma for both variable references and terms in a single induction. The proof for the second part with $\exp^{-1}$ is similar to the first part with $\exp$.

For all variable references $d$, if $d$ does not have any record field reference or array indexing,

$$[d](\exp(e_1)) = 'd' \quad \text{definition of } [d]$$
$$= \exp('d') \quad \text{definition of } \exp$$
$$= \exp([d](e_1)) \quad \text{definition of } [d]$$

If $d = d'.f$,

$$[d](\exp(e_1)) = \concat([d'](\exp(e_1)), 'f') \quad \text{definition of } [d'.f]$$
$$= \concat(\exp([d'](e_1)), 'f') \quad \text{induction hypothesis}$$
$$= \exp(\concat([d'](e_1), 'f')) \quad \text{definition of } \exp$$
$$= \exp([d](e_1)) \quad \text{definition of } [d'.f]$$

If $d = d'[t]$,

$$[d](\exp(e_1)) = \begin{cases} 
\concat([d'](\exp(e_1)), [t], [d](\exp(e_1)), 't') & \text{definition of } [d'[t]] \\
\concat(\exp([d'](e_1)), [t], \exp([d](e_1)), 't') & \text{induction hypothesis} \\
\begin{cases} 
\concat(\exp([d'](e_1)), [t], [d](e_1), 't') & \text{if the type of } t \text{ is the repetitiveID} \\
\exp(\concat([d'](e_1), [t], [d](e_1), 't')) & \text{if the type of } t \text{ is not the repetitiveID} \\
\end{cases} & \text{definition of } \exp \\
\exp([d](e_1)) & \text{definition of } [d'[t]] 
\end{cases}$$

For all terms $t$, if $t$ is an integer constant, the meaning is independent of $s$, and the lemma holds trivially. If $t$ is a variable reference $d$ of the repetitiveID type,

$$[t](\exp(e_1)) = \exp(e_1)([d](\exp(e_1))) \quad \text{definition of } [t]$$
$$= \exp(e_1)(\exp([d](e_1))) \quad \text{induction hypothesis}$$
$$= \exp(e_1)([d](e_1)) \quad \text{definition of } \exp$$
$$= \exp([t](e_1)) \quad \text{definition of } [t].$$
If \( t \) is a variable reference \( d \) of type other than \( \alpha \),

\[
[t](\exp(e_1)) = \exp(e_1)([d](\exp(e_1))) \quad \text{definition of } [t] \\
= \exp(e_1)(\exp([d](e_1))) \quad \text{induction hypothesis} \\
= (e_1)([d](e_1)) \quad \text{definition of } \exp \\
= [t](e_1) \quad \text{definition of } [t].
\]

Finally, if \( t = t_1 + t_2 \), the type of \( t_1 \) and \( t_2 \) cannot be the repetitiveID. By induction \( \| t_1 \| (\exp(e_1)) = \| t_1 \| (e_1) \) and \( \| t_2 \| (\exp(e_1)) = \| t_2 \| (e_1) \). So \( \| t_1 + t_2 \| (\exp(e_1)) = \| t_1 \| (\exp(e_1)) + \| t_2 \| (\exp(e_1)) = \| t_1 \| (e_1) + \| t_2 \| (e_1) = \| t_1 + t_2 \| (e_1) \).

The proof would be similar for other operations on terms. \( \square \)

**Lemma A.15 (Boolean lemma)** Given a program with a repetitiveID, a Boolean expression \( e \), and two environments \( c_2 \) and \( e_1 \) obtained by extending states \( c' \) and \( c \), we have \( \llbracket e \rrbracket (\exp(e_1)) = \llbracket e \rrbracket (e_1) \), and \( \llbracket e \rrbracket (e_2) = \llbracket e \rrbracket (\exp^{-1}(e_2)) \).

**Proof.** We prove the lemma by induction on the structure of the expression.

When \( e \) is over the form \( t_1 > t_2 \), the types of both \( t_1 \) and \( t_2 \) cannot be the repetitiveID. Therefore, \( \llbracket t_1 \rrbracket (\exp(e_1)) = \llbracket t_1 \rrbracket (e_1) \) and \( \llbracket t_2 \rrbracket (\exp(e_1)) = \llbracket t_2 \rrbracket (e_1) \), which implies that \( \llbracket t_1 > t_2 \rrbracket (\exp(e_1)) = \llbracket t_1 > t_2 \rrbracket (e_1) \).

When \( e \) is of the form \( t_1 = t_2 \), and neither of the terms is of the repetitiveID type, we have \( \llbracket t_1 \rrbracket (\exp(e_1)) = \llbracket t_1 \rrbracket (e_1) \) and \( \llbracket t_2 \rrbracket (\exp(e_1)) = \llbracket t_2 \rrbracket (e_1) \), which implies that \( \llbracket t_1 = t_2 \rrbracket (\exp(e_1)) = \llbracket t_1 = t_2 \rrbracket (e_1) \).

The situation becomes more complicated if the type of one of the terms is the repetitiveID. In this case, both \( t_1 \) and \( t_2 \) must be variables of the repetitiveID type. If \( e_1 = (c \bullet \alpha_0) \), the only possible variables are global variables in a state. Because of the restriction 6 on the repetitiveID, the non-active abstractable component states cannot be accessed. Therefore, \( t_1 \) and \( t_2 \) can only be variables in the non-abstractable part (and the active component state) in \( c \), which have the same values in \( c' \). Since \( \exp(e_1) = (c' \bullet \alpha_0) \), we have \( \llbracket t_1 = t_2 \rrbracket (\exp(e_1)) = \llbracket t_1 = t_2 \rrbracket (e_1) \). On the other hand, if \( e_1 = (c \bullet (y \mapsto j) \bullet \alpha_0) \), \( t_1 \) and \( t_2 \) may be variables in the abstractable component \( j \) in \( c \). However the variables in the abstractable component \( j \) in \( c \) have the same values as the variables in the abstractable component \( \exp(j) \) in \( c' \). Since \( \exp(e_1) = (c' \bullet (y \mapsto \exp(j)) \bullet \alpha_0) \), we have \( \llbracket t_1 = t_2 \rrbracket (\exp(e_1)) = \llbracket t_1 = t_2 \rrbracket (e_1) \).
A.4. REPETITIVE PROPERTY ENFORCED BY REPETITIVE IDs

Similarly, the base cases for \(\exp^{-1}\) are also true.

For the induction part, the cases for \(\neg\) and \(\land\) are straightforward. For the universal quantifier, the case when the quantification is over some type other than the repetitiveID follows immediately by the induction hypothesis.

For the case where the quantification is over the repetitiveID, because of the restriction 5 on the repetitiveID, \(e_1\) must be of the form \((c \cdot c_0)\). Therefore, if the repetitiveID type is \(\alpha\), we have

\[
[\forall(y : \alpha)e](\exp (e_1)) = \land_{j \in \alpha} [e] (c' \cdot (y' \rightarrow j) \cdot c_0)
\]

\[
= [e] (c' \cdot (y' \rightarrow 1) \cdot c_0)
\]

\[
\land_{2 \leq j \leq 1+a_1} [e] (c' \cdot (y' \rightarrow j) \cdot c_0)
\]

\[
\land \ldots \land_{2+a_1 + \ldots + a_{k-1} \leq 1+a_1 + \ldots + a_k} [e] (c' \cdot (y' \rightarrow j) \cdot c_0)
\]

\[
\text{expand}
\]

\[
= [e] (c \cdot (y' \rightarrow 1) \cdot c_0)
\]

\[
\land_{2 \leq j \leq 1+a_1} [e] (c \cdot (y' \rightarrow 2) \cdot c_0)
\]

\[
\land \ldots \land_{2+a_1 + \ldots + a_{k-1} \leq 1+a_1 + \ldots + a_k} [e] (c \cdot (y' \rightarrow k+1) \cdot c_0)
\]

\[
\text{induction hypothesis: } [e](e_2) = [e](\exp^{-1}(e_2))
\]

\[
= \land_{1 \leq j \leq k+1} [e] (c \cdot (y' \rightarrow j) \cdot c_0)
\]

\[
\text{simplify}
\]

\[
= [\forall(y : \alpha)e](e_1)
\]

\[
\text{definition of } \forall(y : \alpha).
\]

And similarly for \( [\forall(y : \alpha)e](e_2) = [\forall(y : \alpha)e](\exp^{-1}(e_2)) \).

\[\square\]

Lemma A.16 (statement lemma) Given a program with a repetitiveID, a statement \(sm\) and two environments \(e_2\) and \(e_1\) obtained by extending states \(c'\) and \(c\), if \(e_2 = (c' \cdot c_0)\) and \(e_1 = (c \cdot c_0)\), we have \([sm](\exp(e_1)) = \exp([sm](e_1))\), and \(\exp^{-1}([sm](e_2)) = [sm](\exp^{-1}(e_2))\).

Proof. The proof is by induction on the structure of statements, with the help of the following auxiliary property for a statement enclosed inside a for loop:

if \(e_2 = (c' \cdot (y' \rightarrow j) \cdot c_0)\), and \(e_1 = (c \cdot (y' \rightarrow \exp^{-1}(j)) \cdot c_0)\), the statement \(sm\) changes only the component state of the component \(k\) in \(e_2\) and the component state of the component \(\exp^{-1}(j)\) in \(e_1\). Both components are changed to the same final component states.
This property is enforced by the restriction 5 on the repetitive ID. When environments $e_2 = (c' \bullet (y' \mapsto j) \bullet e_0)$, and $e_1 = (c \bullet (y' \mapsto \text{exp}^{-1}(j)) \bullet e_0)$ are encountered before the execution of a statement, the statement must be within a for loop. Since the variables written by each iteration are disjoint, the only possible changes made by the statement is the component state of the component referred by the variable $y$. Furthermore, because of the variables being disjoint, the control flow within the statement depends only on the component state, which is the same in $e_1$ and $e_2$. Therefore, the final component states are the same after executing the statement in $e_1$ and $e_2$.

Base on this property, we can prove the lemma by induction on the structure of the statements. An error statement results in an error environment regardless of its environment, so the theorem is trivial in this case.

For a statement $sm$ of the form $d := t$, if $sm$ is executed on $e_2 = (c' \bullet e_0)$ or $e_1 = (c \bullet e_0)$, $sm$ must not be enclosed within a for loop. Therefore, $d$ and $t$ can only reference the non-abstractable part (or the component state of the active component) of $c'$ in $e_2$, which is the same as the corresponding part in $c$ in $e_1$. This implies that $[sm](\text{exp}(e_1)) = \text{exp}([sm](e_1))$, and $\text{exp}^{-1}([sm](e_2)) = [sm](\text{exp}^{-1}(e_2))$.

For a statement $sm$ of the form if $b$ then $sm'$ end, if $[b](e_1)$ is true, then $[b](\text{exp}(e_1))$ is also true, by the Boolean lemma, so $[sm](\text{exp}(e_1)) = [sm'](\text{exp}(e_1))$. By the induction hypothesis, $[sm'](\text{exp}(e_1)) = \text{exp}([sm'](e_1)) = \text{exp}([sm](e_1))$. If $[b](e_1)$ is false then $[b](\text{exp}(e_1))$ is also false, so $[sm](\text{exp}(e_1)) = \text{exp}(e_1) = \text{exp}([sm](e_1))$.

For a statement $sm$ of the form $sm_1; sm_2$, by the induction hypothesis,

$$
[sm](\text{exp}(e_1)) = [sm_2][[sm_1](\text{exp}(e_1))]
= [sm_2](\text{exp}([sm_1](e_1)))
= \text{exp}([sm_2]([sm_1](e_1)))
= \text{exp}([sm](e_1)).
$$

For a statement $sm$ of the form for $(y : \alpha)$ do $sm'$ end, where $\alpha$ is the repetitive ID type, we have

$$
[sm](\text{exp}(e_1)) = [sm'_y(1 + a_1 + \ldots + a_k)][[[sm'_y(1 + a_1 + \ldots + (a_k - 1))](\ldots
\ldots [sm'_y(1)][(\text{exp}(e_1))\ldots])])
\text{definition of }[sm]
$$
On the other hand, because of the auxiliary property, we also have:

\[
\begin{align*}
\llbracket sm_y'(1) \rrbracket (\exp(e_1)) &= \exp(\llbracket sm_y'(1) \rrbracket)(e_1) \\
\llbracket sm_y'(1 + a_1 + \ldots + a_j) \rrbracket (\ldots \llbracket sm_y'(1 + a_1 + \ldots + a_j - 1) \rrbracket)(\exp(e_1)) &= \exp(\llbracket sm_y'(1 + j) \rrbracket)(e_1)
\end{align*}
\]  

(A.1)

Therefore,

\[
\begin{align*}
\llbracket sm \rrbracket (\exp(e_1)) &= [sm_y'(1 + a_1 + \ldots + a_k)](\ldots [sm_y'(1)](\exp(e_1)) \ldots) \\
&= \exp(\llbracket sm_y'(k + 1) \rrbracket(\ldots [sm_y'(1)](e_1) \ldots)) \\
&= \exp(\llbracket sm \rrbracket(e_1)) \\
&= \exp(\llbracket sm \rrbracket(e_2)) \\
&= \exp^{-1}(\llbracket sm \rrbracket(e_2))
\end{align*}
\]  

(A.2)

The same analysis can be repeated for \(\exp^{-1}(\llbracket sm \rrbracket(e_2)) = [sm](\exp^{-1}(e_2))\).

\[\square\]

**Lemma A.17 (transition lemma)** Given a program with a repetitiveID, a transition rule \(t\), and two environments \(e_2\) and \(e_1\) obtained by extending states \(e'\) and \(e\), we have \(\llbracket t \rrbracket(\exp(e_1)) = \exp(\llbracket t \rrbracket(e_1))\), and \(\exp^{-1}(\llbracket t \rrbracket(e_2)) = \llbracket t \rrbracket(\exp^{-1}(e_2))\).

**Proof.** It follows trivially from the statement lemma. \[\square\]

Theorem 6.1 can be obtained by combining the Boolean lemma and the transition lemma.
Appendix B

Murφ Descriptions

This appendix contains the Murφ description of the DASH cache coherence protocol.
APPENDIX B. MURφ DESCRIPTIONS

/* Declarations */

The number of clusters is defined by ProcCount. To simplify the description, only one of the clusters have memory (HomeCount). The number of clusters without memory is given by RemoteCount.

The directory used by the protocol is a full mapped directory. Instead of using bit map as in the real implementation, array of cluster ID is used to simply the rules used to manipulate the directory.

Array is used to model every individual cluster-to-cluster network channel. The size of the array is estimated to be larger than the maximum possible number of messages in the network.

All the addresses are in the cache-able address space.

*/

-- Constant Declarations

parseInt */

-- File: adash/tc.p

-- Content: abstract Dash protocol with elementary operations and extended DMA operations

-- Specification decision:

1. Each cluster is modeled by a cache and a RAC to store outstanding operations.
2. The cluster acts as a single processor/global shared cache.
3. Separate network channels are used.
4. Aliases are used extensively.

-- Summary of results:

1. A 'non-obvious' bug is rediscovered.
2. No bug is discovered for the final version of the protocol.

-- Options:

- The code for the bug is retained in the description.
- To see the result of using a erroneous protocol, use option flag 'bug'.
- An option flag 'nohome' is used to switch on/off the local memory action. This enables us to simplify the protocol to examine the behavior when the number of processors increases.

-- Reference:

   The Stanford DASH Multiprocessor.
2. Daniel Lenoski.
   DASH Prototype System.
   PhD thesis, Stanford University,
   (available online)
   (available online)

-- Last Modified: November 96

---File: adash/tc.p

-- Description:

The number of clusters is defined by ProcCount. To simplify the description, only one of the clusters have memory (HomeCount). The number of clusters without memory is given by RemoteCount.

The directory used by the protocol is a full mapped directory. Instead of using bit map as in the real implementation, array of cluster ID is used to simply the rules used to manipulate the directory.

Array is used to model every individual cluster-to-cluster network channel. The size of the array is estimated to be larger than the maximum possible number of messages in the network.

All the addresses are in the cache-able address space.

*/

-- Constant Declarations

parseInt */

-- File: adash/tc.p

-- Content: abstract Dash protocol with elementary operations and extended DMA operations

-- Specification decision:

1. Each cluster is modeled by a cache and a RAC to store outstanding operations.
2. The cluster acts as a single processor/global shared cache.
3. Separate network channels are used.
4. Aliases are used extensively.

-- Summary of results:

1. A 'non-obvious' bug is rediscovered.
2. No bug is discovered for the final version of the protocol.

-- Options:

- The code for the bug is retained in the description.
- To see the result of using a erroneous protocol, use option flag 'bug'.
- An option flag 'nohome' is used to switch on/off the local memory action. This enables us to simplify the protocol to examine the behavior when the number of processors increases.

-- Reference:

   The Stanford DASH Multiprocessor.
2. Daniel Lenoski.
   DASH Prototype System.
   PhD thesis, Stanford University,
   (available online)
   (available online)

-- Last Modified: November 96

---File: adash/tc.p

-- Description:

The number of clusters is defined by ProcCount. To simplify the description, only one of the clusters have memory (HomeCount). The number of clusters without memory is given by RemoteCount.

The directory used by the protocol is a full mapped directory. Instead of using bit map as in the real implementation, array of cluster ID is used to simply the rules used to manipulate the directory.

Array is used to model every individual cluster-to-cluster network channel. The size of the array is estimated to be larger than the maximum possible number of messages in the network.

All the addresses are in the cache-able address space.

*/

-- Constant Declarations

parseInt */

-- File: adash/tc.p

-- Content: abstract Dash protocol with elementary operations and extended DMA operations

-- Specification decision:

1. Each cluster is modeled by a cache and a RAC to store outstanding operations.
2. The cluster acts as a single processor/global shared cache.
3. Separate network channels are used.
4. Aliases are used extensively.

-- Summary of results:

1. A 'non-obvious' bug is rediscovered.
2. No bug is discovered for the final version of the protocol.

-- Options:

- The code for the bug is retained in the description.
- To see the result of using a erroneous protocol, use option flag 'bug'.
- An option flag 'nohome' is used to switch on/off the local memory action. This enables us to simplify the protocol to examine the behavior when the number of processors increases.

-- Reference:

   The Stanford DASH Multiprocessor.
2. Daniel Lenoski.
   DASH Prototype System.
   PhD thesis, Stanford University,
   (available online)
   (available online)

-- Last Modified: November 96

---File: adash/tc.p

-- Description:

The number of clusters is defined by ProcCount. To simplify the description, only one of the clusters have memory (HomeCount). The number of clusters without memory is given by RemoteCount.

The directory used by the protocol is a full mapped directory. Instead of using bit map as in the real implementation, array of cluster ID is used to simply the rules used to manipulate the directory.

Array is used to model every individual cluster-to-cluster network channel. The size of the array is estimated to be larger than the maximum possible number of messages in the network.

All the addresses are in the cache-able address space.
-- the type of reply that go into the reply network
-- Cacheable and DMA operations:

ReplyType:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>-- basic op. -- read reply</td>
</tr>
<tr>
<td>IVAR</td>
<td>-- basic op. -- waiting for a read reply</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- waiting for a read reply</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- waiting for a read reply</td>
</tr>
<tr>
<td>UNR</td>
<td>-- with ownership transfer</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- waiting for a read excl. reply</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- waiting for invalid ack</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- validating read/read</td>
</tr>
<tr>
<td>UNR</td>
<td>-- with ownership request</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- waiting for a DMA read reply</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- waiting for update acknowledge</td>
</tr>
<tr>
<td>UNR</td>
<td>-- basic op. -- waiting for a DMA write ack.</td>
</tr>
</tbody>
</table>

-- record for the requests in the network

Request:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
</tbody>
</table>

-- record for the reply in the network

Reply:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
</tbody>
</table>

-- States in the Remote Access Cache (RAC):
-- a) maintaining the states of currently outstanding requests,
-- b) buffering replies from network
-- c) supplementing the functionality of caches

RACState:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
</tbody>
</table>

-- States of data in the cache

CacheState:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
<tr>
<td>UNR</td>
<td>-- invalid</td>
</tr>
</tbody>
</table>

-- Directory Controller and the Memory
-- a) Contains directory DRAM
-- b) Forwards local requests to remotes
-- c) Responds to MPBIDE with directory information
-- d) Stores locks and lock queues

procedure Set.Dir.1st_Entry: h: Home; n: Proc;

begin
    undefine Home[h].Dir[n].Entries;
    Home[h].Dir[n].Entries[0] := n;
end;
-- Sending Reply
-- ---------------------------

-- send read reply
--- Aux = home cluster
Procedure Send_R_Reply(Dst, Src, Home : Proc;
Addr : Address; Val : Value);
Begin
Send_Reply(ACK, Dst, Src, Home, Addr, Val, 0);
End;

-- send negative ack to requesting cluster
Procedure Send_NA_B(Proc);
Addr : Address;
Begin
Send_Reply(NAK, Dst, Src, Aux, Addr, Undefined, Val);
End;

-- send invalidate acknowledge from shared remote clusters
--- Aux = where the request originally is from
Procedure Send_Inv_Ack(Dst, Src, Aux : Proc;
Addr : Address);
Begin
Send_Reply(SACK, Dst, Src, Aux, Addr, Undefined, 0);
End;

-- send exclusive remote reply to requesting cluster
--- Aux = where the request originally is from
Procedure Send_R_Ex_Reply(Dst, Src, Aux : Proc;
Addr : Address; Val : Value;
InvCount : NodeCount);
Begin
Alias Count : ReplyNet[Src][Dst].Count
Do
Assert(Count != ChanMax) "Reply Channel is full!"
If(InvCount > 0) Then
Send_Reply(ACK, Dst, Src, Aux, Addr, Val, 0);
Else
Send_Reply(NAK, Dst, Src, Aux, Addr, Undefined, 0);
End
End;

-- send dirty transfer ack to new master
Procedure Send_Dirty_Transfer_Ack(Dst, Src : Proc;
Addr : Address);
Begin
Send_Reply(SACK, Dst, Src, Undefined, Addr, Undefined, 0);
End;

-- Sending DMA Request
-- ---------------------------

-- DMA Send request to home
Procedure Send_DMA_R_Req(H : Proc;
Addr : Address);
Begin
Send_Req(DH, Dst, Src, Undefined, Addr, Undefined);
End;

-- DMA Send request to remote
Procedure Send_DMA_R_Req_RAC(Dst, Src : Proc;
Addr : Address);
Begin
Send_Req(DH, Dst, Src, Undefined, Addr, Undefined);
End;

-- DMA write request from Local
Procedure Send_DMA_W_Req_H(Dst, Src : Proc;
Addr : Address;
Val : Value);
Begin
Send_Req(DH, Dst, Src, Undefined, Addr, Val);
End;

-- send DMA update count to local
Procedure Send_DMA_Update_Count(Dst, Src : Proc;
Addr : Address;
SharedCount : NodeCount);
Begin
Send_Reply(ACK, Dst, Src, Addr, Undefined, SharedCount);
End;

-- send DMA update request from Local
Procedure Send_DMA_Update_Rq(Dst, Src, Aux : Proc;
Addr : Address;
Val : Value);
Begin
Send_Req(DH, Dst, Src, Aux, Addr, Val);
End;

-- send DMA update request from home
--- forward DMA update request from home
Procedure Send_DMA_W Req_RAC(Dst, Src, Aux : Proc;
Addr : Address;
Val : Value);
Begin
Send_Req(DH, Dst, Src, Aux, Addr, Val);
End;

-- Sending DMA Reply
-- ---------------------------

-- DMA read reply
Procedure Send_DMA_R_Reply(Dst, Src, Aux : Proc;
Addr : Address;
Val : Value);
Begin
Send_Reply(ACK, Dst, Src, Aux, Addr, Val);
End;

-- send DMA write Acknowledge
Procedure Send_DMA_W_Ack(Dst, Src, Aux : Proc;
Addr : Address);
Begin
Send_Reply(ACK, Dst, Src, Aux, Addr, Val, 0);
End;

-- send DMA update acknowledge to local
Procedure Send_DMA_Update_Ack(Dst, Src : Proc;
Addr : Address);
Begin
Send_Reply(SACK, Dst, Src, Address, Undefined, 0);
End;

/* Rule Sets for fundamental memory access and DMA access
1) CPU la : basic home memory requests from CPU
CPU lb : DMA home memory requests from CPU
2) CPU la : basic remote memory requests from CPU
CPU lb : DMA remote memory requests from CPU
3) PCPU la : handling basic requests to PCPU at memory cluster
PCPU lb : handling DMA requests to PCPU at memory cluster
4) PCPU la : handling basic requests to PCPU in remote cluster
PCPU lb : handling DMA requests to PCPU in remote cluster
5) MPUC lab : handling basic and DMA replies to MPUC in any cluster.*/
APPENDIX B. MURφ DESCRIPTIONS

<snip content from the page>,

APPENDIX B. MURφ DESCRIPTIONS

/*

CPU Ia

The rule sets non-deterministically issue requests for local cache-able memory. The requests include read, exclusive read.

Two sets of Rules:
Rule 'Local Memory Read Request'
Rule 'Local Memory Read Exclusive Request'

Issue messages:
  RD_RAC
  RDX_RAC
  INV

Ruleset n:
  Proc Do

Ruleset h:
  Home Do

Ruleset a:
  Address Do

Alias
  RAC: Proc[n].RAC[h][a];
  Cache: Proc[n].Cache[h][a];
  Mem: Home[h].Mem[a];

Do
  begin
    switch RAC.State
    case INVAL:
      /* no pending event */
      switch Dir.State
      case Dirty_Remote:
        /* send request to master cluster */
        RAC.State := WRDO;
        Send_R_Req_RAC(Dir.Entries[i][b][h][a]);
      else
        /* get from memory */
        switch Cache.State
        case Locally_Exmod:
          /* write back local memory through snoopy protocol */
          Cache.State := Locally_Shared;
          Mem := Cache.Value;
          Case Locally_Shared:
          /* other cache supply data */
          Case Not_Locally_Cached:
          /* update cache */
          Cache.State := Locally_Shared;
          Cache.Value := Mem;
          end;
        end;
      end;
      end;
    case WRDO:
      /* conflict */
      end;
    else
      /* WRD, RRD, WRX, WRRX, WRXN, WRRXN */
      Assert ( RAC.State != WRD & RAC.State != WRD )
      'Funny RAC state at home cluster';
      /* conflict */
    end;
    end;
end;

begin
  switch RAC.State
  case INVAL:
    /* no pending event */
    switch Dir.State
    case Unached:
      /* get from memory */
      invalidate local copy through snoopy protocol
      Cache.State := Locally_Exmod;
      Cache.Value := Mem;
      Case Shared_Remote:
      /* other cache supply data */
      Assert (Dir.State /= Uncached) /"Inconsistent Directory";
    case WRDO:
      /* conflict */
    else
      /* WRD, RRD, WRX, WRRX, WRXN, WRRXN */
      Assert ( RAC.State != WRD & RAC.State != WRD )
      'Funny RAC state at home cluster';
      /* conflict */
    end;
    end;
  case WRX:
    /* merge */
    end;
  else
    /* WRD, RRD, WRX, WRRX, WRXN, WRRXN */
    Assert ( RAC.State != WRD & RAC.State != WRD )
    'Funny RAC state at home cluster';
    /* conflict */
  end;
end;
end;

begin
  switch RAC.State
  case INVAL:
    /* no pending event */
    switch Dir.State
    case Unached:
      /* get from memory */
      invalidate all remote shared read copies
      RAC.State := WRDN;
      RAC.InvCount := Dir.SharedCount;
    for i : NodeCount Do
      if i < RAC.InvCount Then
        Send_Inv_Req(Dir.Entries[i][b][h][a]);
      end;
    end;
    Dir.State := Uncached;
    Dir.SharedCount := 0;
    undefine Dir.Entries;
    Case Dirty_Remote:
    /* send request to master cluster */
    (switch requesting processor to do other jobs)
    RAC.State := WRDX;
    Send_R_Ex_Req_RAC(Dir.Entries[i][b][h][a]);
    end;
    end;
  case WRX:
    /* merge */
    end;
  else
    /* WRD, RRD, WRX, WRRX, WRXN, WRRXN */
    Assert ( Dir.State = Unached ) "Inconsistent Directory";
    Case WRDO:
    /* conflict */
    else
      /* WRD, RRD, WRX, WRRX, WRXN, WRRXN */
      Assert ( RAC.State = WRD & RAC.State = WRD )
      'Funny RAC state at home cluster';
      /* conflict */
    end;
    end;
end;

end;
end;
*/

------------------------------
-- Home CPU issue read request
-- confirmed with tables: net.tbl, rc.tbl, rc_3.tbl, rc_up.tbl
------------------------------

------------------------------
-- Home CPU issue read exclusive request
-- confirmed with tables: net.tbl, rc.tbl, rc_3.tbl, rc_up.tbl
------------------------------

------------------------------
-- Rule 'Local Memory Read Request'
------------------------------

------------------------------
-- Rule 'Local Memory Read Exclusive Request'
------------------------------

------------------------------
-- Rule 'Local Memory Read Request'
------------------------------

------------------------------
-- Rule 'Local Memory Read Exclusive Request'
------------------------------

------------------------------
-- Rule 'Local Memory Read Request'
------------------------------
The rule sets non-deterministically issue requests for local cacheable memory. The requests include DMA read and DMA write.

Two sets of rules:
- Rule "DMA Local Memory Read Request"
- Rule "DMA Local Memory Write Request"

Issue messages:
- DRD_RAC
- DWR_RAC
- DUP

Ruleset n:

\[
\text{Begin}\]

Begin Switch Dir.State
Case Uncached:
- switch Dir.State
Case Shared_Remote:
- update remote master copy
  RAC.State := RAC.State := VRAND;
  RAC.Value := VRAND;
  Send_DMU_Update_RAC(Dir.Entries[i], n, n, v);
End; --for;
End; --switch;

Case Dirty_Remote:
- update remote master copy
  RAC.State := VRAND;
  RAC.Value := VRAND;
  Send_DMU_Update_RAC(Dir.Entries[i], n, n, v);
End; --switch;

Case WINV:
- local master copy
  write local cache
  If (Cache.State = Locally_Shared) Then
    Cache.Value := Cache.Value := v;
  Else
    If (Cache.State = Locally_Shared) Then
      Cache.Value := v;
      End; --if;
  End; --else;
Else
- VRAND, VRAND, VRAND, VRAND, VRAND, VRAND
  Assert (RAC.State = VRAND & RAC.State = VRAND)
  funny RAC state at home cluster!
  -- conflict
End; --else
End; --switch;

Case VWIN:
- local master copy
  write local cache
  If (Cache.State = Locally_Shared) Then
    Proc[i].Cache[i][a].Value := v;
  Else
    If (Cache.State = Locally_Shared) Then
      Cache.Value := v;
      End; --if;
  End; --else
Else
- VRAND, VRAND, VRAND, VRAND, VRAND, VRAND
  Assert (RAC.State = VRAND) "VRAND at home cluster";
End; --switch;

End; --rule -- DMA Local Memory Read Request

Ruleset v:

\[
\text{Value Do}\]

Rule "DMA Local Memory Write Request"
( h = n ) & home

Begin
Switch RAC.State
Case INVAL:
- no pending event
Switch Dir.State
Case Uncached:
- Switch Cache.State
  Case Locally_Cached:
    write to memory
    Mem := v;
  Case Locally_Shared:
    write to memory and cache
    Cache.Value := v;
    Mem := v;
  Case Locally_Exmod:
    write to cache
    Cache.Value := v;
End; --switch;

Case Shared_Remote:
- shared by remote cache
  a) update local cache
  b) update memory
  c) update remote cache
Switch Cache.State
Case Locally_Cached:
- write to memory and cache
  Cache.Value := v;
  Mem := v;
  Case Locally_Exmod:
    update local cache
    Cache.Value := v;
End; --switch;

Case Shared_Remote:
- update remote master copy
  RAC.State := VRAND;
  RAC.Value := VRAND;
  Send_DMU_Update_RAC(Dir.Entries[i], n, n, v);
End; --if;
End; --for;
End; --switch;

Case Dirty_Remote:
- update remote master copy
  RAC.State := VRAND;
  RAC.Value := VRAND;
  Send_DMU_Update_RAC(Dir.Entries[i], n, n, v);
End; --switch;

Case WINV:
- local master copy
  write local cache
  If (Cache.State = Locally_Shared) Then
    Proc[i].Cache[i][a].Value := v;
  Else
    If (Cache.State = Locally_Shared) Then
      Cache.Value := v;
      End; --if;
  End; --else
Else
- VRAND, VRAND, VRAND, VRAND, VRAND, VRAND
  Assert (RAC.State = VRAND) "VRAND at home cluster";
End; --switch;

End; --rule -- DMA write
End; --ruleset; -- v

End; --alias; -- RAC, Cache, Dir, Mem
End; --ruleset; -- n
End; --ruleset; -- b
End; --ruleset; -- n
The rule sets non-deterministically issue requests for remote cacheable memory. The requests include read, exclusive read, and explicit writeback.

Three sets of rules:
Rule "Remote Memory Read Request"
Rule "Remote Memory Read Exclusive Request"
Rule "Explicit Writeback request"

Issue messages:
RD
RDX
WB

Ruleset n:
Ruleset h:
Ruleset a:

Alias
RAC:
Proc[<n>], Proc[a];
Cache:
Proc[x], Cache[a]<x>;

--- Remote CPU issues read request --
--- confirmed with tables net.tbl, rc.tbl1, rc.tbl3, up.tbl
---

Rule "Remote Memory Read Request" (h = n)
---
Begin
Switch RAC.State
Case INVAL:
-- no pending event
Switch Cache.State
Case Non_Locally_Cached:
-- send request to home cluster
RAC.State := WRD;
Send_R_Req_H(h,n,a);
Else
-- other cache supply data using snoopy protocol
End;
Case WINV:
-- other cache supply data
Assert (Cache.State = Locally_Exmod )
"WINV with Exmod not asserted";
Case WRD:
-- conflict
Assert (Cache.State = Locally_Exmod )
"WRD with Exmod asserted";
Case RRD:
-- conflict
Assert (Cache.State = Non_Locally_Cached )
"RRD with funny cache state";
Case WRDO:
Error ( "remote cluster with WRDO" );
Case WDMAR:
-- conflict
Assert (Cache.State = Locally_Exmod )
"WDMAR with Locally_Exmod";
Case WUP:
-- conflict
Case WDMAW:
-- conflict
Assert (Cache.State = Locally_Exmod )
"WDMAR with Locally_Exmod";
End; -- switch;
End; -- rule -- Remote Memory Read Request

--- Remote CPU issues read exclusive request --
--- confirmed with tables net.tbl, rc.tbl1, rc.tbl3, up.tbl
---

Rule "Remote Memory Read Exclusive Request" (h = n)
---
Begin
Switch RAC.State
Case INVAL:
-- no pending event
Switch Cache.State
Case Locally_Exmod:
-- other cache supply data
Else
-- send request to home cluster
RAC.State := WRDX;
Send_R_Req_H(h,n,a);
End;
Case WUP:
-- conflict
Case WDMAW:
-- conflict
Assert (Cache.State = Locally_Exmod )
"WDMAR with Locally_Exmod";
End; -- switch;
End; -- rule -- Remote Memory Read Exclusive Request

--- Remote CPU issues explicit writeback request --
---

Rule "Explicit Writeback request" (h = n)
---
Begin
If ( RAC.State = WUP ) Then
-- retry later
Else
-- send request to home cluster
Assert ( RAC.State = INVAL )
| RAC.State := WRD;
Send_WB_Req(h,n,a,Cache.Value);
Cache.State := Non_Locally_Cached;
Undefine Cache.Value;
End;
End; -- switch;
End; -- rule -- Explicit Writeback request

--- Remote CPU issues read exclusive request --
--- confirmed with tables net.tbl, rc.tbl1, rc.tbl3, up.tbl
---

Rule "Remote Memory Read Exclusive Request" (h = n)
---
Begin
Switch RAC.State
Case INVAL:
-- no pending event
Switch Cache.State
Case Locally_Exmod:
-- other cache supply data
Else
-- send request to home cluster
RAC.State := WRD;
Send_R_Req_H(h,n,a);
End;
Case WINV:
-- other cache supply data
Assert (Cache.State = Locally_Exmod )
"WINV with Exmod not asserted";
Case WRD:
-- conflict
Assert (Cache.State = Locally_Exmod )
"WRD with Exmod asserted";
Case RRD:
-- conflict
Case WRDO:
Error ( "remote cluster with WRDO" );
Case WDMAR:
-- conflict
Assert (Cache.State = Locally_Exmod )
"WDMAR with Locally_Exmod";
Case WUP:
-- conflict
Case WDMAW:
-- conflict
Assert (Cache.State = Locally_Exmod )
"WDMAR with Locally_Exmod";
End; -- switch;
End; -- rule -- Remote Memory Read Exclusive Request
The rule sets non-deterministically issue requests for remote cache-able memory. The requests include DMA read, DMA write.

Two sets of rules:
Rule "DMA Remote Memory Read Request"
Rule "DMA Remote Memory Write Request"

Issue messages:
DRD_H
DWR_H

Ruleset n: Proc Do
Ruleset h: Home Do
Ruleset a: Address Do

Alias
RAC : Proc[n].RAC[x];
Cache : Proc[n].Cache[b][x];

Begin
Switch Cache.State
Case Non_Locally_Cached:
Switch RAC.State
Case INVAL:
RAC.State := WDMAR;
Send_DMA_R_Req_H (b,n,a);
Case WINV:
Error "Inconsistent RAC and Cache";
Else
WRD, WRDX, RRD, WDMAR, WUP, WDMAW
Assert ( RAC.State := WDMAR ) 'Inconsistent directory';

Case Locally_Shared:
Assert ( RAC.State := VINV )
& RAC.State := WROD & RAC.State := MRD & RAC.State := MRRO & RAC.State := VINV ) 'Inconsistent directory';

Case Locally_Exmod:
Assert ( RAC.State := WDMAR )
& RAC.State := WROD & RAC.State := MRD & RAC.State := MRRO & RAC.State := VINV ) 'Inconsistent directory';

End;

End; -- rule -- DMA Remote Memory Request

Ruleset v : Value Do

Rule "DMA Remote Memory Write Request" ( h != n )

Begin
Switch RAC.State
Case INVAL:
Switch Cache.State
Case Locally_Exmod:
-- update cache
Cache.Value := v;
Case Locally_Shared:
-- no update to local cache
-- update will be requested by home cluster later.
-- send update request
RAC.State := WDMAR;
RAC.Value := v;
Send_DMA_V_Req_R(b,n,a,v);
Case Non_Locally_Cached:
-- send update request
RAC.State := WDMAR;
RAC.Value := v;
Send_DMA_V_Req_R(b,n,a,v);
End; --switch;

Case VINV:
-- update local cache
Assert ( Cache.State = Locally_Exmod )
'VINV with Exmod not asserted';
Cache.Value := v;
Else
WRD, WROD, WRDX, RRD, WDMAR, WUP, WDMAW
Assert ( RAC.State := WROD ) 'WROD at remote cluster';
-- conflict or merge
End;

End; -- rule -- DMA Remote Memory Write Request

Ruleset a:

-- remote CPU issues DMA write request
-- confirmed with tables net.tbl, rc.tbl, rc.tbl, rc.tbl, rc.tbl
-- up.tbl

Ruleset n:

-- remote CPU issues DMA read request
-- confirmed with tables net.tbl, rc.tbl, rc.tbl, rc.tbl, rc.tbl
-- up.tbl

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:

Ruleset h:

Ruleset a:

Ruleset n:
APPENDIX B. MURΦ DESCRIPTIONS

-- PCPU handles read request to home cluster
-- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
-- up_up.tbl
--------------------
Rule "handle read request to home"
ReqChan : ReqNet/|Src|/|Dst|/; Message/|0|/; Message/|0|/
Begin
Alias
RAC : Procs[|Dst|].RAC[|Addr|]; Cache : Procs[|Src|].Cache[|Dst|][|Addr|];
Dir : Homes[|Dst|].Dir[|Addr|]; Men : Homes[|Dst|].Men[|Addr|];
Do
Switch RAC.State
Case WINV:
RAC : Procs[|Dst|].RAC[|Addr|];
Cache : Procs[|Src|].Cache[|Dst|][|Addr|];
-- some one has a shared copy.
-- send copy to remote cluster
Cache.State := Locally_Shared;
Men := Cache.Value;
End;
Dir.State := Shared_Remote;
Dir.SharedCount := 1;
Set.Dir.Ist_Entry(Dir, Addr, Src);
Send.R_Reply(Src, Dir, Addr, Men);
Consumes_Request(Src, Dir);
Else
WINV, WRD, WRDO, WRDX, RRD, WDMAR, WUP, WDMAW
-- invalidation every shared copy
if(/bug/) then
-- ####### bug!
-- if you want protocol with no bug,
-- use the following:
-- send invalidation anyway to
-- master-copy-requesting cluster, which has
-- already invalidated the cache
for i : NodeCount Do
Send.Inv.Req(Dir.Entries[i], Dir, Addr, Men, Dir.SharedCount);
ed
else
-- ####### bug!
-- if you want protocol with bug
-- rediscovered by Murphi which is a subtle bug
-- that the designer once overlooked
-- discovered it by substantial simulation, use
-- the following:
-- no invalidation to master-copy-requesting
-- cluster, which has already invalidated the cache
for i : DirIndex Do
if (i < Dir.SharedCount) then
Send.Inv.Req(Dir.Entries[i], Src, Dir, Addr, Men);
ed
else
Send.Inv.Req(Src, Dir, Addr, Men, Dir.SharedCount);
ed
end
-- some one has a shared copy.
-- send copy to remote cluster
Add.Dir.Entries(Dir, Addr, Src);
Send.R_Reply(Src, Dir, Addr, Men);
Consumes_Request(Src, Dir);
Case Dir.Shared_Remote:
-- some one has a master copy.
-- forward request to master cluster
Send.R_Req(RAC.Dir.Entries[0], Dir, Src, Addr);
Consumes_Request(Src, Dir);
ed;
-- switch;
ed;
alias := RAC, Cache, Dir, Men;
ed;
-- rule "handle read request to home"
--------------------

Then
Send_R_Ex_Reply(Src, Dst, Dst, Addr, Mem, Dir, SharedCount);  
Else
Send_R_Ex_Reply(Src, Dst, Dst, Addr, Mem, Dir, SharedCount);  
End;
End;  
End;  
-- bug or no bug

Dir(State) := Dirty_Remote;
Set_Dir_Entry(Dst, Addr, Src);
Consume_Request(Src, Dst);

Case Dirty_Remote:
-- some one has a master copy,
-- forward request to master cluster
Send_R_Ex_Req_RAC(Entries[0], Dst, Src, Addr);
Consume_Request(Src, Dst);
End;  
End;  
End;  
-- switch;

--- PCPU handles sharing writeback request to home cluster
Rule 'handleSharingWriteback request to home'
ReqChan.Count > 0
& Request = SHWB
---
Begin
Alias:
v : ReqNet(Src, Dst, Messages[0], Value);
Dir : Home(Dst).Dir[Addr];
Mem : Home(Dst).Mem[Addr]  
Do
Assert (Dir.State = Dirty_Remote)
'Writeback to non dirty remote memory';
Assert (Dir.Entries[0] = Src)
'Explicit writeback by non owner';
Mem := v;
Dir.State := Uncached;
Dir.SharedCount := 0;
Undefine Dir.Entries;
Consume_Request(Src, Dst);
End;  
-- alias : v, Dir, Mem
End;  
-- rule -- handle writeback

--- PCPU handles dirty transfer request to home cluster
Rule 'handleDirtyTransfer request to home'
ReqChan.Count > 0
& Request = DXFER
---
Begin
Alias:
Dir : Home(Dst).Dir[Addr]  
Do
Assert (Dir.State = Dirty_Remote)
'Dirty transfer for non dirty remote memory';
Assert (Dir.Entries[0] = Src)
'Dirty transfer by non owner';
Set_Dir_Entry(Dst, Addr, Aux);
Send_Dirty_Transfer_Ack(Aux, Dst, Addr);
Consume_Request(Src, Dst);
End;  
-- alias : Dir
End;  
-- rule -- handle dirty transfer to home
PCPU Ib

PCPU handles DMA requests to Home for cache-able memory.

Two sets of rules:
- Rule "handle DMA read request to home"
- Rule "handle DMA write request to home"

Handle message:
- DRD_H
- DWR_H

Ruleset Dst : Proc Do
Ruleset Src : Proc Do
Alias
ReqChan : ReqNet[Src|Dest]|Message[0].MsgType;
Addr : ReqNet[Src|Dest].Message[0].Addr;
Aux : ReqNet[Src|Dest].Message[0].Aux
Do

PCPU handles DMA read request to home

Begin

Alias
RAC : Proc[Dest].RAC[Dest].Addr;
Cache : Proc[Dest].Cache[Dest].Addr;
Dir : Home[Dest].Dir[Addr];
Mem : Home[Dest].Mem[Addr];

Do

Switch RAC.State
Case WINV:

- cannot release copy

  Assert (Cache.State /= Locally_Exmod);
  Send_DMA_R_Repl(Src, Dest, Addr, Cache.Value);

  Consume.Request(Src, Dest);
End;

Case INVAL, WRD, WRDO, WRDX, RRD, WDMAR, WUP, WDMAW:

  Switch Dir.State
  Case Uncached:

    If (Cache.State /= Locally_Exmod) Then
      Cache.Value /= v;
      Case Locally_Shared:
      Cache.Value /= v;
      Case Non_Locally_Cached:
      v := v;
      End;
      Send_DMA_W_Ack(Src, Dest, Addr);
      Consume.Req(Src, Dest);
    End;
    Send_DMA_Upd_Repl(Dir.Entries[0], Dest, Addr, v);
    Consume.Req(Src, Dest);
  End;

  Case Dirty_Remote:

    update remote master copy
    Send_DMA_Upd_Repl(Dir.Entries[0], Dest, Addr, v);
    Consume.Req(Src, Dest);

  End;

End;

End;

- alias : RAC, Cache, Dir, Mem
End -- rule -- handle DMA read request to home

PCPU handles DMA write request to Home

Begin

Alias
v : ReqNet[Src|Dest].Message[0].Value;
RAC : Proc[Dest].RAC[Dest][Addr];
Cache : Proc[Dest].Cache[Dest][Addr];
Dir : Home[Dest].Dir[Addr];
Mem : Home[Dest].Mem[Addr];

Do

Switch RAC.State
Case UNIV:

  cannot release copy

  Assert (Cache.State = Locally_Exmod);
  Send_DMA_W_Repl(Src, Dest, Addr, Cache.Value);

  Consume.Request(Src, Dest);
End;

Else -- UNIV, WRD, WRDO, WRDX, RUP, WDMAR:

  Switch Dir.State
  Case Uncached:

    update local data
    Switch Cache.State
    Case Locally_Exmod:
      Cache.Value /= v;
      Case Locally_Shared:
      Cache.Value /= v;
      End;
      Case Non_Locally_Cached:
      v := v;
      End;
      Send_DMA_W_Ack(Src, Dest, Addr);
      Consume.Request(Src, Dest);
    End;
    Send_DMA_Upd_Count(Src, Dest, Addr, Dir.SharedCount);
    Consume.Request(Src, Dest);
  End;

  Case Dirty_Remote:

    update remote master copy
    Send_DMA_Upd_Repl(Dir.Entries[0], Dest, Addr, v);
    Consume.Req(Src, Dest);

  End;

End;

End;

End -- rule -- handle DMA write request to home
PCPU: #a

PCPU handles basic requests to non-home for cacheable memory.

Three sets of rules:

- Rule "Handle read request to remote cluster" 
- Rule "Handle invalidate request to remote cluster" 
- Rule "Handle read exclusive request to remote cluster"

Handle Messages:

- RD_RAC
- INV
- RD_RAC

Rule "Handle read request to remote cluster"

Alias

/* */

Handle Messages:

PCPU: Proc

Begin

Rule Set Dst: Proc Do
Rule Set Src: Proc Do
Alias

ReqChan: ReqNet[Src][Dst];
Request: ReqNet[Src][Dst].Message[0].Mtype;
Addr: ReqNet[Src][Dst].Message[0].Addr;
Aux: ReqNet[Src][Dst].Message[0].Aux
Do

-- PCPU handles read request to remote cluster
-- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl, up_mp.tbl
-- Case DRDX: ambiguous in their table

Rule "Handle read request to remote cluster"

Begin

Case DRDX:

Assert ( Dst != Src ) "Invalidation to Local Memory";
If ( Dst == Aux )

End;

Case WINV:

Cache: Proc[Dst].Cache[Src][Addr];
RAC: Proc[Dst].RAC[Src][Addr];

Send[Inv_Ack](Aux, Dst, Src, Addr);

Else

End;

Case WRD:

End;

Case WRX:

End;

Case WINV:

End;

End; /* */

-- PCPU handles invalidate request

Rule "Handle invalidate request to remote cluster"

Begin

Alias

RAC: Proc[Dst].RAC[Src][Addr];
Cache: Proc[Dst].Cache[Src][Addr];

Send[NAK](Aux, Dst, Src, Addr);

End;

Else

Assert ( Dst != Aux ) "Invalidation to Local Memory";
If ( Dst == Aux )

End;

End; /* */

-- PCPU handles invalidate request to initiating cluster

Rule "Handle invalidate request to remote cluster"

Begin

Alias

RAC: Proc[Dst].RAC[Src][Addr];
Cache: Proc[Dst].Cache[Src][Addr];

Send[NAK](Aux, Dst, Src, Addr);

Else

End;
APPENDIX B. MURφ DESCRIPTIONS

---

PCPU handles Read exclusive request to remote cluster
- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl, up_mp.tbl

---

Rule 'Handle read exclusive request to remote cluster'
 ReqChan.Count > 0
 & Request = RDX_RAC
 =>

Begin
 Alias:
 RAC: Proc[Dst], RAC[Src][Addr];
 Cache: Proc[Dst].Cache[Src][Addr];
 Do
 Switch RAC.State
 Case WINV:
   -- cannot release copy
   Assert (Cache.State = Locally_Exmod )
   "WINV with Exmod not asserted";
   Send_RK_NA_K(Aux, Dst, Src, Addr);
   Consume_Request(Aux, Dst);
 Else
   -- inval, WRD, WRDO, WRD, WRD, WRD, WDMAR, WRD, WDMAR.
   Assert (RAC.State = WRD) "WRD in remote cluster";
   Switch Cache.State
   Case Locally_Exmod:
     -- has master copy; dirty transfer data
     If (Src = Aux) Then
       -- request from home cluster
       Send_RK_Reply(Aux, Dst, Src, Addr, Cache.Value, 0); 
     Else
       -- request from remote cluster
       Send_RK_Reply(Aux, Dst, Src, Addr, Cache.Value, 1);
       Send_Dirty_Transfer_Req(Src, Dst, Aux, Addr);
     End;
     Cache.State := Non_Locally_Cached;
     Consume_Request(Aux, Dst);
   Else
     -- cannot release
     -- possible situation is:
     -- WRD or still waiting for reply
     Send_RK_NA_K(Aux, Dst, Src, Addr);
     Consume_Request(Aux, Dst);
   End;
   => alias: RAC, Cache
 End; => ruleset; => Dst
---

PCPU handles DMA requests to non-home for cache-able memory.

Three sets of rules:
- Rule 'Handle DMA read request to remote cluster'
- Rule 'Handle DMA update request to remote cluster'
- Rule 'Handle DMA write request to remote cluster'

---

Rule 'Handle DMA read request to remote cluster'
 ReqChan.Count > 0
 & Request = DRD_RAC
 =>

Begin
 Alias:
 RAC: ReqNet[Src][Dst];
 Request: ReqNet[Src][Dst].Message[0].Msgtype;
 Addr: ReqNet[Src][Dst].Message[0].Addr;
 Aux: ReqNet[Src][Dst].Message[0].Aux;
 Do
 Switch RAC.State
 Case WINV:
   Send_RK_NA_K(Aux, Dst, Src, Addr);
   Consume_Request(Src, Dst);
 Else
   -- inval, WRD, WRDO, RRD, WRD, WRD, WDMAR, WRD, WDMAR.
   Assert (RAC.State = WRD) "WRD in remote cluster";
   Switch Cache.State
   Case Locally_Exmod:
     -- reply with new data
     Send_DMA_R_Reply(Aux, Dst, Src, Addr, Cache.Value);
     Consume_Request(Src, Dst);
   Else
     -- cannot release
     -- possible situation is:
     -- WRD or still waiting for reply
     -- i.e. request message received before reply
     Send_RK_NA_K(Aux, Dst, Src, Addr);
     Consume_Request(Src, Dst);
   End;
   => alias: RAC, Cache
 End; => rule; => handle DMA read request to remote cluster
---

PCPU handles DMA update request

---

Rule 'Handle DMA update request to remote cluster'
 ReqChan.Count > 0
 & Request = DUP
 =>

Begin
 Alias:
 v: ReqNet[Src][Dst].Value[0].Value;
 RAC: Proc[Dst], RAC[Src][Addr];
 Cache: Proc[Dst].Cache[Src][Addr];
 Do
 Switch RAC.State
 Case WINV:
   Assert (Cache.State = Locally_Exmod )
   "WINV with Exmod not asserted";
   Cache.Value := v;
 Case WRD:
   RAC.State := RRD;
---

PCPU handles DMA write request to remote cluster

---

Rule 'Handle DMA write request to remote cluster'
 ReqChan.Count > 0
 & Request = WRD_RAC
 =>

Begin
 Alias:
 RAC: Proc[Dst], RAC[Src][Addr];
 Cache: Proc[Dst].Cache[Src][Addr];
 Do
 Switch RAC.State
 Case WINV:
   Send_RK_NA_K(Aux, Dst, Src, Addr);
   Consume_Request(Aux, Dst);
 Case WRD:
   RAC.State := RRD;
---
Case RRD:
  no change
Else
  -- IVAL, WRDO, WRDX, WDMAR, WUP, WDMAW.
  Assert (RAC.State != WRDO) 'WRDO in remote cluster!';
  Switch Cache.State
  Case Locally_Shared:
    Cache.Value := v;
    Case_Hen_Locally_Cached:
      -- cache already given up data
      -- possible situation :
      -- DMA update causes an RRD,
      -- therefore not up-to-date copy in cache
      -- while home cluster still thinks there
      -- is a shared copy. If you want to
      -- introduce some errors, add next line
      -- Error 'checking if we model flush'!
    Case_Locally_Shared:
      -- possible situation is :
      -- arise if the bug I is used/.
      -- otherwise not.
      -- 1) cluster : get shared copy
      -- 2) home tries DMA update
      -- 3) cluster : has given up data and get a master copy
      -- Error 'add possible situation IV!'
    Cache.Value := v;
    -- requested updated Exclusive copy ?
  End;
End;
Send_DMA_Update_Ack(Aux, Dst, Src, Addr);
Consume_Request(Src, Dst);
End;
-- rule -- handle DMA update request to remote cluster

RCPU handles DMA write request to remote cluster
-- confirmed with table net.tbl, rc_1.tbl, rc_3.tbl,
-- except simplified in handling NAK on WDMAW
*/
RCPU Iab
RCPU handles cache-able and DMA acknowledgments and replies.
Four sets of rules:
  Rule 'handle Acknowledgment'
  Rule 'handle negative Acknowledgment'
  Rule 'handle Supplementary Acknowledgment'
Handle message:
  ACK
  SACK
  NAK

-- confirmed with table net_up.tbl
-- except simplified in handling NAK on WDMAW
*/
Ruleset Dst Do Proc Do
Ruleset Src Do Proc Do
Alias
ReplyChan : ReplyNet[Src, Dst, Addr];
Reply : ReplyNet[Src, Dst, Messages[0], Addr];
Aux : ReplyNet[Src, Dst, Messages[0], Aux];
v : ReplyNet[Src, Dst, Messages[0], Value];
ICount : ReplyNet[Src, Dst, Messages[0], invCount]
Do
Rule 'handle Acknowledgment'
ReplyChan.Count := 0 & Reply = ACK
  -- basic operation -- read reply
  -- basic operation -- read exclusive reply
  -- (inv count = 0)
  -- DMA operation -- read reply
  -- DMA operation -- write acknowledge
Rule 'handle negative Acknowledgment'
ReplyChan.Count := 0 & Reply = NAK
  -- pending read, i.e. read reply
  -- Error 'ACK in INVAL RAC state!'
Case VUNY:
  -- cannot update copy
  Assert (Cache.State = Locally_Refresh) 'VUNY with Refresh not asserted.';
  Send_DMA_Update_ACK(Aux, Dst, Src, Addr);
  Consume_Request(Src, Dst);
Else
  -- IVAL, WRD, WRDX, WDMAR, WUP, WDMAW.
  Switch Cache.State
  Case VUNY:
    -- cannot update copy
    Assert (Cache.State = Locally_Refresh) 'VUNY with Refresh not asserted.';
    Send_DMA_Update_ACK(Aux, Dst, Src, Addr);
    Consume_Request(Src, Dst);
Else
  -- cannot update copy
  -- possible situation is :
  -- DMA update also waiting for reply
  -- i.e. request message received before reply
  Send_DMA_ACK(Aux, Dst, Src, Addr);
  Consume_Request(Src, Dst);
End;
APPENDIX B. MURφ DESCRIPTIONS

-- pending exclusive read, i.e. exclusive read reply
-- no invalidation is required
Cache.State := Locally_Exmod;
Cache.Value := v;
if (Dest = Aux)
then
   alias
   Dir := Home[Dest].Dir[Addr];
do
   -- getting master copy back in home cluster
   -- no shared copy in the network
   Dir.State := Uncached;
   Dir.SharedCount := 0;
   undefine Dir.Entries;
end -- alias := Dir
end;
undefine RAC;
RAC.State := INVALID;
RAC.InvCount := 0;
consume_reply(src, dst);

-- invalidated pending event, ignore reply
undefine RAC;
RAC.State := INVALID;
RAC.InvCount := 0;
consume_reply(src, dst);

-- pending DMA read, i.e. read reply
-- return data to cpu
undefine RAC;
RAC.State := INVALID;
RAC.InvCount := 0;
consume_reply(src, dst);

-- pending DMA write, i.e. write acknowledge
undefined RAC;
RAC.State := INVALID;
RAC.InvCount := 0;
consume_reply(src, dst);
else
   -- VSYNC, VSP
   error "RAC in funny RAC state!"
end -- switch;
end; -- alias := RAC, Cache, Dir, Mem
end -- rule -- handle ACK

----------- handle negative Acknowledgment -----------
ReplyChan.Count := 0;
& Reply := RAC
-- ANY kind of operation -- negative acknowledge

begin
alias
RAC := Proc[Dest], RAC[Aux][Addr];
Cache := Proc[Dest], Cache[Aux][Addr];
do
switch RAC.State
case INVALID:
   error "NAK in INVALID RAC state!"
end case;
usr:
   undefine RAC;
   RAC.State := INVALID;
   RAC.InvCount := 0;
   consume_reply(src, dst);
end case;
usr:
   undefine RAC;
   RAC.State := INVALID;
   RAC.InvCount := 0;
   consume_reply(src, dst);
end case;
usr:
   undefine RAC;
   RAC.State := INVALID;
   RAC.InvCount := 0;
   consume_reply(src, dst);
end case;
usr:
   undefine RAC;
   RAC.State := INVALID;
   RAC.InvCount := 0;
   consume_reply(src, dst);
end case;
else
   -- Winv, WUP
   error "Cache is Exmod while WDMAW outstanding";
   Cache.Value := RAC.Value;
   undefine RAC;
   RAC.State := INVALID;
   RAC.InvCount := 0;
   case Dirty_Remote:
       undefine RAC;
       RAC.State := WDMAW;
       Send_DMA_W Req_RAC(Dir.Entries[Dest], Dest, Addr, RAC.Value);
   end case;
end else;
end switch;
end;
end;
end; -- alias := Dir, Mem
else
   -- if
   -- remote cluster issuing DMA write
   switch Cache.State
   case Locally_Exmod:
      error "Cache is Exmod while WDMAW outstanding";
      Cache.Value := RAC.Value;
      undefine RAC;
      RAC.State := INVALID;
      RAC.InvCount := 0;
   else
      undefine RAC;
      RAC.State := WDMAW;
      Send_DMA_W Req_B(Aux, Dest, Addr, RAC.Value);
   end case;
end switch;
else
   -- switch
   VSP, VUP
   error "NAK in funny RAC state!"
end switch;
end;
end; -- alias := RAC
end; -- rule -- handle NAK
Rule "Handle Indirect Acknowledgment"
ReplyChan.Count > 0
& Reply = IACK
   -- basic operation  -- read exclusive reply
   -- (inv count = 0)
   -- DMA operation  -- acknowledgment with update count
Begin:
Alias:
RAC : Proc[Dst].RAC[Aux][Addr];
Cache : Proc[Dst].Cache[Aux][Addr];
Do
Switch RAC.State
Case INVAL:
   -- no pending event
   Error "Read exclusive reply in INVAL RAC state";
Case WRDX:
   -- pending exclusive read, i.e. exclusive read reply
   -- require invalidation
   Cache.State := Locally_Exmod;
   Cache.Value := v;
   If ( Dst = Aux ) Then
      -- getting master copy back in home cluster
      Alias:
      Dir : Home[Dst].Dir[Addr];
      Do
      Error "Already sent invalidations to copy ??";
      Dir.State := Uncached;
      Dir.SharedCount := 0;
      Undefine Dir.Entries;
      End;
      End;
      -- alias := Dir;
      RAC.InvCount := ICount - RAC.InvCount;
      RAC.State := WRDX;
      If ( RAC.InvCount = 0 ) Then
         -- all invalidation acks received
         Undefine RAC;
         RAC.State := INV;
         RAC.InvCount := 0;
      End;
      Consume_Reply(Src, Dst);
      Case WDMAW:
      RAC.InvCount := ICount + RAC.InvCount;
      RAC.State := WUP;
      If ( RAC.InvCount = 0 ) Then
         -- all invalidation acks received
         Undefine RAC;
         RAC.State := INV;
         RAC.InvCount := 0;
      End;
      Consume_Reply(Src, Dst);
      Else
      WRD, WRDO, RRD, WDMAR, WUP,
      Error "Read exclusive reply in funny RAC state";
      End;
      End;
      -- alias := RAC, Cache, Dir
      End;
      -- rule -- IACK

Rule "Handle Supplementary Acknowledgment"
ReplyChan.Count > 0
& Reply = SACK
   -- basic operation  -- invalidate acknowledge
   -- basic operation  -- dirty transfer acknowledge
   -- DMA operation  -- update acknowledge
Begin:
Alias:
RAC : Proc[Dst].RAC[Aux][Addr];
Do
-- inv_Ack, Dirty_Transfer_Ack
Switch RAC.State
Case INVAL:
   -- no pending event
   Error "Invalidate acknowledge in INVAL RAC state";
Case WINV:
   -- get invalidation acknowledgment
   RAC.InvCount := RAC.InvCount + 1;
   If ( RAC.InvCount = 0 ) Then
      -- get all invalidation acks
      Undefine RAC;
      RAC.State := INV;
      RAC.InvCount := 0;
   End;
   Consume_Reply(Src, Dst);
Case WRDX:
   -- get invalidation acknowledgment before reply
   RAC.InvCount := RAC.InvCount + 1;
   Consume_Reply(Src, Dst);
Case WUP:
   RAC.InvCount := RAC.InvCount - 1;
   If ( RAC.InvCount = 0 ) Then
      Undefine RAC;
      RAC.State := INV;
      RAC.InvCount := 0;
   End;
   Consume_Reply(Src, Dst);
Case WDMAW:
   -- why is it a plus ??
   RAC.InvCount := RAC.InvCount + 1;
   Consume_Reply(Src, Dst);
Else
   WRD, WRDO, RRD, WDMAR,
   Error "Invalidate acknowledge in funny RAC state";
   End;
   -- alias := RAC
   End;
   -- rule -- SACK
   End;
   -- alias := ReplyChan, Reply, Addr, Aux, v, ICount
   End;
   --ruleset := -- Src
   End;
   --ruleset := -- Dst
APPENDIX B. MURφ DESCRIPTIONS

/*
 *    -- rule for non-deterministically change the master copy
 */
Ruleset v : Value Do
Ruleset h : Home Do
Ruleset n : Proc Do
Ruleset a : Address Do

Rule 'modifying value at cache' 
Procs[n].Cache[h][a].State = Locally_Exmod

Begin
Procs[n].Cache[h][a].Value := v;
End;
End; -- ruleset -- a
End; -- ruleset -- n
End; -- ruleset -- h
End; -- ruleset -- v

/*
 *    Start state
 */

the memory can have arbitrary value

Ruleset v : Value Do

Startstate
Begin
For h : Home Do
Home[h].Dir[a].State := Uncached;
Home[h].Dir[a].SharedCount := 0;
Home[h].Mem[a] := v;
Undefine Home[h].Dir[a].Entries;
End;
End;

For n : Proc Do
For h : Home do
For a : Address Do
Procs[n].Cache[h][a].State := Non_Locally_Cached;
Procs[n].RAC[h][a].State := INVALID;
Undefine Procs[n].Cache[h][a].Value;
Undefine Procs[n].RAC[h][a].Value;
Procs[n].RAC[h][a].InvCount := 0;
End;
End;
End;

For Src : Proc Do
For Dst : Proc Do
Redirect[Src][Dst][Dir].Count := 0;
Undefine Redirect[Src][Dst][Dir].Messages;
RedirectSrc[Src][Dir].Count := 0;
Undefine RedirectSrc[Src][Dir].Messages;
End;
End;
End; -- startstate

End; -- ruleset -- v

/*-------------------------------------------------------------*/

/*
 *    invariant 'Globally invalid RAC state at Home Cluster'
 *    invariant 'Globally invalid RAC state at Local Cluster'
 *    invariant 'Only a single master copy exist'
 *    invariant 'Irrelevant data is set to zero'
 *    invariant 'Consistency within Directory'
 *    invariant 'Consistency of data'
 *    invariant 'Adequate invalidations with Read Exclusive request'
 */

/*
 *    invariant 'Globally invalid RAC state at Home Cluster'
 */
Forall n : Proc Do
Forall h : Home Do
Forall a : Address Do
(h != n)
{ ( Procs[n].RAC[h][a].State := WRD 
 & Procs[n].RAC[h][a].State := RRD ) }
End
End
End; -- globally invalid RAC state at Home Cluster

/*
 *    invariant 'Globally invalid RAC state at Local Cluster'
 */
Forall n : Proc Do
Forall h : Home Do
Forall a : Address Do
(h = n)
{ ( Procs[n].RAC[h][a].State := WRD ) }
End
End
End; -- globally invalid RAC state at Local Cluster

/*
 *    invariant 'Only a single master copy exist'
 */
Forall n1 /: Proc Do
Forall n2 /: Proc Do
Forall h /: Home do
Forall a /: Address Do
! (#1 /= n2 
 & Procs[n1].Cache[h][a].State := Locally_Exmod 
 & Procs[n2].Cache[h][a].State := Locally_Exmod )
End
End
End
End; -- only a single master copy exist

/*
 *    invariant 'Irrelevant data is set to zero'
 */
Forall n : Proc Do
Forall h : Home Do
Forall a : Address Do
( Home[h].Dir[a].State := Uncached 
 & Home[h].Dir[a].SharedCount := 0 )
&
( Forall i : 0..DirMax-1 Do 
 i > Home[h].Dir[a].SharedCount 
 -> IsUndefined(Homes[h].Dir[a].Entries[i])
End )
&
( Procs[n].Cache[h][a].State := Non_Locally_Cached 
 -> IsUndefined(Procs[n].Cache[h][a].Value) )
&
( Procs[n].RAC[h][a].State := INVALID 
 -> ( IsUndefined(Procs[n].RAC[h][a].Value) 
 & Procs[n].RAC[h][a].InvCount := 0 ) )
End
End
End; -- irrelevant data is set to zero

End; -- ruleset -- v
Invariant "Consistency within Directory"

Forall h /: Home Do
  Forall a /: Address Do
    ( Homes[h].Dir[a].State = Uncached
    & Homes[h].Dir[a].SharedCount = 0 )
    | ( Homes[h].Dir[a].State = Dirty_Remote
    & Homes[h].Dir[a].SharedCount = 1 )
    | ( Homes[h].Dir[a].State = Shared_Remote
    & Homes[h].Dir[a].SharedCount = 0 ) & Homes[h].Dir[a].SharedCount /= 0
    &forall i /: DirIndex Do
      Forall j /: DirIndex Do
        ( i /= j
        & i < Homes[h].Dir[a].SharedCount
        & j < Homes[h].Dir[a].SharedCount )
      )
      | ( Homes[h].Dir[a].Entries[i] /= Homes[h].Dir[a].Entries[j] )
  End
End;
-- Consistency within Directory

Invariant "Condition for existence of master copy of data"

Forall n /: Proc Do
  Forall h /: Home Do
    Forall a /: Address Do
      ( Procs[n].Cache[h][a].State = Locally_Exmod
      & Procs[n].RAC[h][a].State = INV
      & Procs[n].RAC[h][a].State = WINV )
    End
End;
-- Condition for existence of master copy of data

Invariant "Consistency of data"

Forall n /: Proc Do
  Forall h /: Home Do
    Forall a /: Address Do
      ! ( Procs[n].Cache[h][a].State = Locally_Shared
      & Procs[n].Cache[h][a].Value = Homes[h].Mem[a]
      & Homes[h].Dir[a].State = Dirty_Remote
      & ! ( Exists i /: 0..ChanMax Do
      & ReqNet[h][n].Count
      & ReqNet[h][n].Messages[i].Type = INV )
    ) & ! ( Exists i /: 0..ChanMax Do
      & RespNet[h][n].Count
      & RespNet[h][n].Messages[i].Type = SHWB )
    ) & ! ( Exists i /: 0..ChanMax Do
      & ( i < ReqNet[h][n].Count
      & ReqNet[h][n].Messages[i].Aux = n )
    ) & ! ( Exists i /: 0..ChanMax Do
      & ( i < RespNet[h][n].Count
      & RespNet[h][n].Messages[i].Aux = ACK )
    ) & ! ( Exists i /: 0..ChanMax Do
      & ( i < RespNet[h][n].Count
      & RespNet[h][n].Messages[i].Aux = DCP )
    ) & Procs[n].RAC[h][a].State = WINAV
    & ! ( Exists i /: 0..ChanMax Do
      & ( i < ReqNet[h][n].Count
      & ReqNet[h][n].Messages[i].Type = WINAV )
    ) End
End
End;
-- Consistency of data
Bibliography


BIBLIOGRAPHY


[JJ91] C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-

[JJD96] Daniel Jackson, Somesh Jha, and Craig A. Damon. Faster checking of software
specifications by eliminating isomorphs. ACM Conference on Principles of
Programming Languages, January 1996.

[KMOS94] Robert P. Kurshan, Michael Merritt, Ariel Orda, and Sonia R. Sachs. A
structural linearization principle for processes. Formal Methods in System

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. IEEE Trans-

[Lam80] Leslie Lamport. ‘sometime’ is sometimes ‘not never’, on the temporal logic
of programs. 7th ACM Symposium on Principles of programming languages,
1980.

[Lam83] L. Lamport. What good is temporal logic. Information Processing 83, pages

[LG81] G. Levin and D. Gries. A proof technique for communicating sequential pro-

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and
John Hennessy. The directory-based cache coherence protocol for the dash
multiprocessor. 17th International Symposium on Computer Architecture,
1990.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich We-
ber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The


[LSU89] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. VHDL: Hardware Description

[Lub84] Boris D. Lubachevsky. An approach to automating the verification of compact

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. 24th

synchronization on shared-memory multiprocessors. ACM Transactions on


