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

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.

⟨ 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.

⟨ 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.

⟨ signed ⟩
John C. Mitchell

Approved for the University Committee on Graduate Studies:

⟨ 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 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.

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

List of Tables

List of Figures

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].

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, Dil96]. 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* [BBLS92, 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, Pel94, Pel96] 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 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 time, 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.

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.

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, HJJJ84, 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 $\forall$CTL* model checking). The reduction using the repetitive property is sound for $\forall$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 Murφ description language was designed to be the simplest high level language that supports nondeterministic, scalable descriptions. Murφ meets these particular goals (especially simplicity) better than many hardware and protocol description languages [Bri86, BD87, Ora88, Hol91b, LSU89, TM91]. The Murφ language is inspired by the Chandy and Misra's Unity language [CM88]. A Murφ description of a system is typically at a high level of abstraction. Most features in Murφ 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 Murφ 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 Murφ compiler generates a C++ program from a description in the Murφ 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, Murφ 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 Murφ description consists of constant and type declarations, variable declarations, invariants declarations, and rule declarations. As an example, the Murφ description of a simple mutual exclusion protocol is shown in Figure 2.1. From this description, Murφ 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 **startstate**s; and the states reachable from the initial states are generated by recursively apply the transition **rule**s to the initial states. The verifier checks that the **invariant**s are true in every reachable state.

```
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 ⇒ 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
```

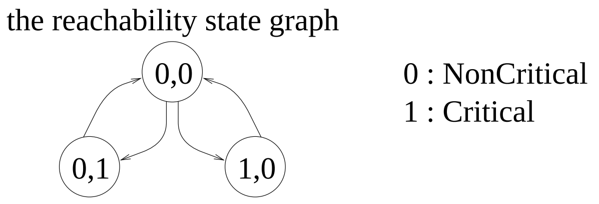


Figure 2.1: A Murφ description of a simple mutual exclusion protocol

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 **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:

rule "Leaving Critical Section"
    $P[i]$ = Critical $\Rightarrow$ P[i] := NonCritical; **end**

The condition in this rule is $P[i] =$ Critical, which makes sure that the process i is originally in the critical section. The action in this rule is $P[i] :=$ 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.

- Murφ 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.

> 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 $\perp$ 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* $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$*, where* Q *is a set of states,* $Q_0 \subseteq Q$ *is a set of initial states,* $\mathbf{error} \in Q$ *is a unique error state, and* $\Delta \subseteq Q \times Q$ *is a transition relation with the property that* $q = \mathbf{error}$ *whenever* $(\mathbf{error}, q) \in \Delta$*.*

The special error state, **error**, is used as the next state whenever an invariant is violated or an error statement is executed.

In Murφ, a state graph is defined implicitly by a set of *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 $q_2 = t(q_1)$.

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

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 $1 \leq i \leq n$.*

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

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 $q_1 \longrightarrow q_2$, denote $q_2 = t(q_1)$ as $q_1 \xrightarrow{t} q_2$, and denote $q_n = t_n(...t_1(q_0)...)$ as $q_0 \xrightarrow{t_1...t_n} q_n$.

The algorithm in Murφ, as shown in Figure 2.2, checks whether **error** or a deadlock state is reachable. It is an *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 *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.

```
**Hashtable** $Reached$ ;
**Queue** $Unexpanded$ ;
**Boolean** $Deadlock$ ;

Simple_Algorithm()
begin
    $Reached = Unexpanded = \{q \mid q \in StartState\}$;
    while $Unexpanded \neq \phi$ 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 $q'$ is not in $Reached$ then put $q'$ in $Reached$ and $Unexpanded$; endif;
            endlet;
        endfor;
        if $Deadlock$ then report deadlock; endif;
    endwhile;
end
```

Figure 2.2: A simple on-the-fly algorithm used in Murφ

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:

DASH-C	the directory-based cache coherence protocol for the DASH multiprocessor [LLG$^+$90].
ICCP	an industrial directory-based cache coherence protocol described in [DDHY92] (an early version of the cache coherence protocol in Sun S3.mp systems).

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.

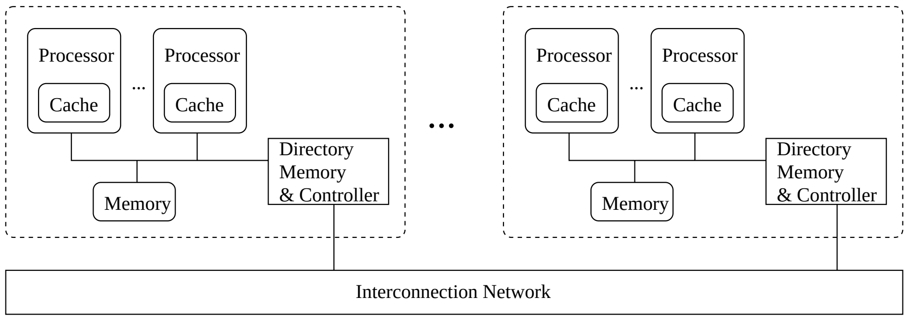


Figure 2.3: The DASH multiprocessor: The processors are arranged into clusters, which communicate with one another through an interconnection network.

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) :

```
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:

```
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

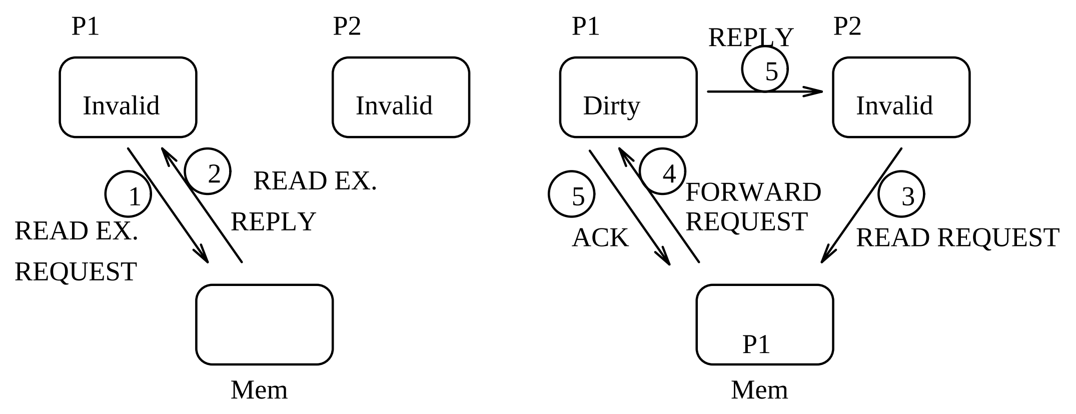


Figure 2.4: Examples of cache coherence transactions: In the first transaction, processor 1 requests for an exclusive copy of a memory location, and the memory replies with the appropriate data. In the second transaction, processor 2 requests for a shared copy of the same location, and the memory forwards 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 **ruleset**s, such as:

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

Ruleset Dst : 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:

```
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:

PETERSON	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).

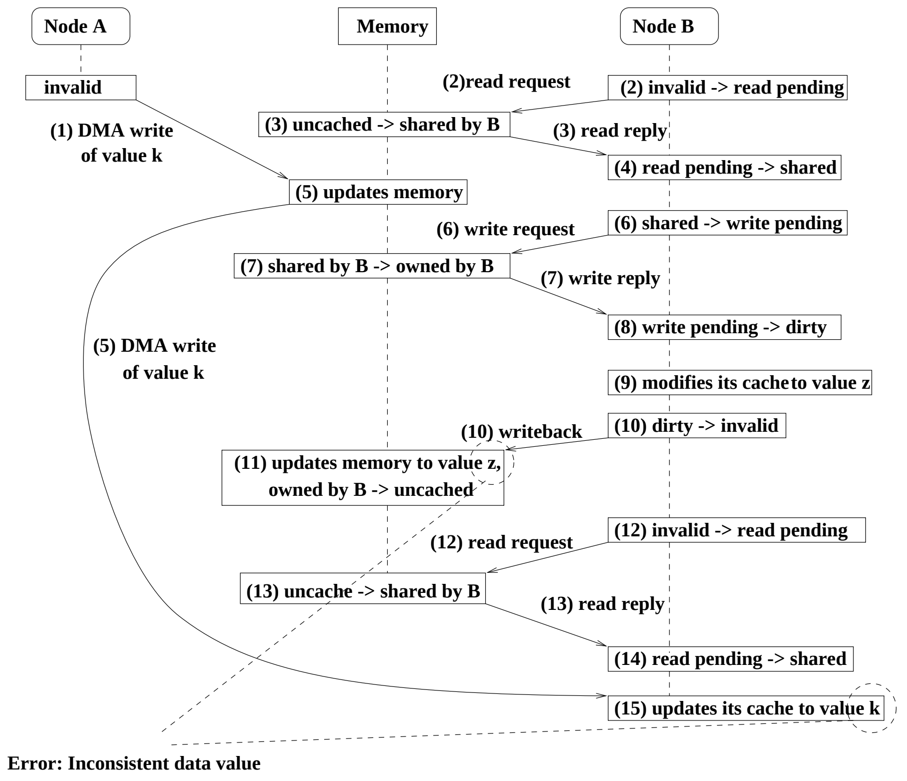
The Murφ verifier[1] was able to detect an error in the DASH protocol with two remote nodes, after exploring 14,825 states in 63 seconds. The counter-example constructed is shown in Figure 2.5.

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.


Node A
Memory
Node B
invalid
(2)read request
(2) invalid -> read pending
(3) uncached -> shared by B
(1) DMA write of value k
(3) read reply
(4) read pending -> shared
(5) updates memory
(6) write request
(6) shared -> write pending
(7) shared by B -> owned by B
(7) write reply
(8) write pending -> dirty
(5) DMA write of value k
(9) modifies its cacheto value z
(10) writeback
(10) dirty -> invalid
(11) updates memory to value z, owned by B -> uncached
(12) invalid -> read pending
(12) read request
(13) uncache -> shared by B
(13) read reply
(14) read pending -> shared
(15) updates its cache to value k
Error: Inconsistent data value

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
9. B modifies its cache
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 update 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.

system	DASH-C with 2 remote nodes and 2 data values	ICCP with 3 processors and 2 data values
size	41,848	150,794
time	191s	96s

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

Table 2.1: Verification results using the basic algorithm, I

system	DASH-C with 3 remote nodes and 3 data values without DMA operations	ICCP with 5 processors and 1 data values with at most 7 outstanding messages
size	210,663	2,093,231
time	1,011s	2,048s

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

[1]This chapter is based on materials published in [ID93a, ID93b, ID96a].

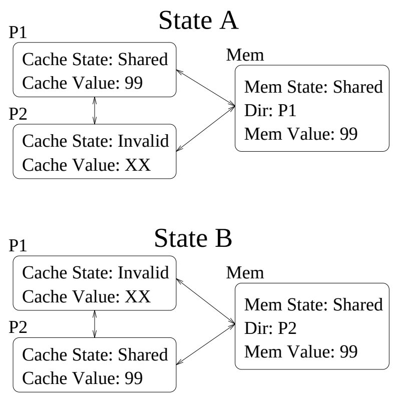


Figure 3.1: Symmetry in a cache coherence protocol: A cache coherence protocol considers all processors to be identical. While states A and B have different processors caching the data from the memory, each of them have exactly one processor caching the data, and exactly one processor having an invalid 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* $A = \langle Q, Q_0, \Delta, \mathbf{error}\rangle$ *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* $h(\mathbf{error}) = \mathbf{error}$.

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* $H \cup \{\mathbf{id}\}$ *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 **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 $A = \langle Q, Q_0, \Delta, \mathbf{error}\rangle$ w.r.t. a congruence $\approx$ is $A/\!\approx \; = \langle Q', Q_0', \Delta', \mathbf{error}'\rangle$, where $Q' = \{[q] : q \in Q\}$, $Q_0' = \{[q] : q \in Q_0\}$, $\mathbf{error}' = [\mathbf{error}]$, 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 $A = \langle Q, Q_0, \Delta, \mathbf{error}\rangle$ 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 Δ', 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, $h(p_0), \ldots, h(p_{i-1}), h(p_i)$ 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* [**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
```

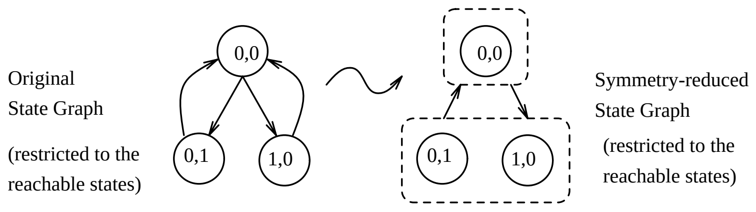


Figure 3.2: An example on the use of scalarset: Because the operations for *Pid* are symmetric, it can be converted from a subrange to a scalarset.

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 $=$. 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 α is a scalarset type, a permutation $\pi_\alpha : (\alpha \cup \{\perp\}) \rightarrow (\alpha \cup \{\perp\})$ is a bijection such that $\pi_\alpha(\perp) = \perp$.*

In this definition, we use the same symbol α to present the set of values represented by the scalarset α, excluding the special undefined value $\perp$.

Extending this notion of permutation, we can obtain symmetry-equivalent states by permuting the scalarset entries of a state[4]:

- 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, π_α permutes a scalarset α of size 4 so that 0 maps to 0, 1 maps to 2, 2 maps to 1, and 3 maps to 3. When π_α is used to permute a state, π_α 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 α, every permutation π_α on the states of the state graph A derived from the program is an automorphism on A.*

[4]When 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.

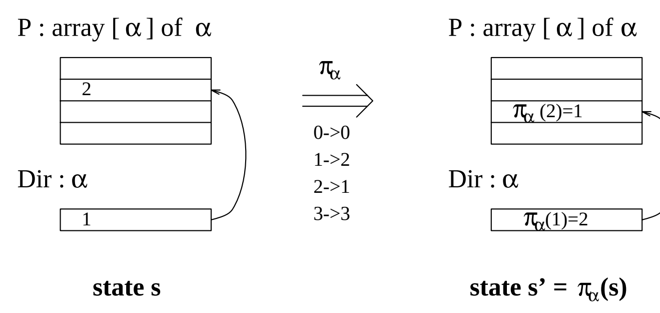


Figure 3.3: Permutation on state variables: Any permutation on the scalarset values induces a permutation on the state variables, which can be used to construct a symmetry-equivalent state.

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 $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$ and a set of automorphisms H on A, a* canonizer *ζ 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 mimimum 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 ζ. 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 π 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
    $Reached = Unexpanded = \{\underline{\zeta(q)} \mid q \in StartState\}$;
    while $Unexpanded \neq \phi$ 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 $\underline{\zeta(q')}$ is not in $Reached$ then put $\underline{\zeta(q')}$ 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

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

where α is a scalarset 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 ζ is a canonizer for states stored in the variable v. $\square$

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 π that converts G_1 to G_2 must map each vertex in V_1^i to a vertex in V_2^i. Therefore, if the ith 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 ith and jth 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 ith and jth 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 ith element has value i, and the jth 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).

system	DASH-C	DASH-C	ICCP	ICCP
symmetry	2 clusters & 2 data values	3 clusters & 3 data values w/o DMA	3 processors & 2 data values	5 processors & 1 data values with restricted messages
without reduction	191s	1,011s	96s	2,048s
naive canonicalization	480s	2,467s	49s	567s
heuristics canonicalization	154s	200s	24s	28s

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.

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[5]. 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 ζ while maintaining the soundness of the algorithm.*

[5]In 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 Murφ 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 ζ, there is no change to the algorithm or the data structure. Except for the limited storage for the local variables used in ζ, no extra storage is required. The

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

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

system	DASH-C	DASH-C	ICCP	ICCP
symmetry	2 clusters & 2 data values	3 clusters & 3 data values w/o DMA	3 processors & 2 data values	5 processors & 1 data values with restricted messages
Original size	41,848	210,663	150,794	2,093,231
Reduced size	10,466	8,251	12,577	18,962
Reduction	75%	96.1%	91.7%	99.1%
Original time	191s	1,011s	96s	2,048s
New time	154s	200s	24s	28s
Speedup	19%	80%	75%	98.7%
Theoretical maximum reduction	75%	97.2%	91.7%	99.2%

System	PETERSON	MCS1	MCS2	DASH-L	LIST1	LIST2
Symmetry	5 procs	4 procs	3 procs	3 clusters	4 nodes	4 nodes
Original size	628,868	554,221	3,240,032	55,366	560,185	1,382,001
Reduced size	6,770	23,636	540,219	9,313	23,410	57,616
Reduction	98.9%	95.7%	83.3%	83.2%	95.8%	95.8%
Original time	1,032s	194s	6,454s	188s	175s	1,177s
New time	12s	21s	384s	96s	15s	53s
Speedup	98.8%	96.5%	94%	49%	91.4%	95.5%
Theoretical maximum reduction	99.2%	95.8%	83.3%	83.3%	95.8%	95.8%

Table 3.3: Improvement in performance with scalarset: Close to $N_1! \times ... \times N_k!$ reductions are obtained, where N_i is the size of the ith scalarset.

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

Table 3.4: Inefficiency in handling symmetry in a message channel: The reduction obtained is less than $N!$, where N is the size of the scalarset representing the indices of the messages. A naive canonizer spends a large amount of time in permuting the empty messages; a normalizer was able to reduce the time to reasonable amount.

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.

Symmetry on LIST1	3 remote cells and a network of 7 messages	4 remote cells and a network of 8 messages
Original time	5s	175s
New time (Canonicalization on Scalarset)	81s	4,056s
New time (Normalization on Scalarset)	11s	214s
New time (Canonicalization on Scalarset & Multiset)	3s	15s

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 α of size N can be defined as a scalarset with two extra operations: given a ring-index i, **pred**(i) gives the value $i-1 \bmod \mathrm{N}$ and **succ**(i) gives the value $i+1 \bmod \mathrm{N}$. 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 $i+1 \bmod \mathrm{N}$. The maximum reduction of a set of non-reflexive ring-indices is N.

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 α of size N can be defined as a scalarset 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:

```
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 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:

```
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. [HJJJ84] 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, BFH$^+$92, Fer93, LY92, ACH$^+$92] 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 vise 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. [HJJJ84], Starke [Sta91], and Kensen [Jen96] have applied symmetry in the reachability analysis of Petri Nets. Clarke, Filkorn 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 [1]

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

[1]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 α is a* data scalarset *in a source program P if α is not used as array indices or* **for** *statement indices.*

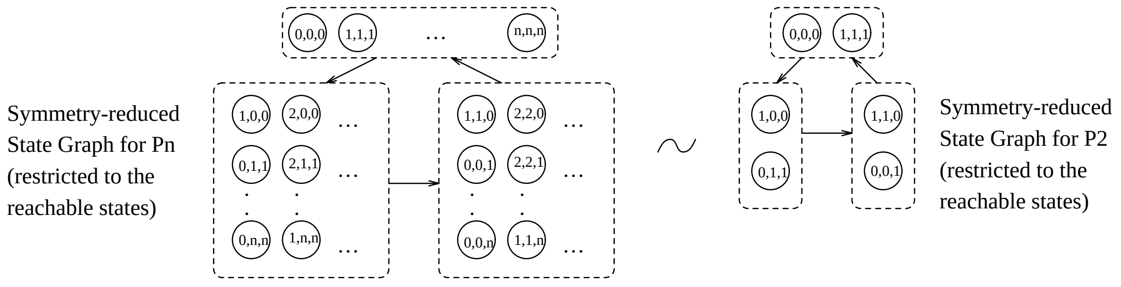
Because of the strong relationship between the usual notion of data-independence and the symmetry in a data scalarset, we define data-independence as:

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 descibed by a data scalarset can be formalized as follows:

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

```
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
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"
    $(Producer = Channel \mid Channel = Consumer)$
```



Symmetry-reduced State Graph for Pn (restricted to the reachable states)
0,0,0
1,1,1
...
n,n,n
1,0,0
2,0,0
0,1,1
2,1,1
0,n,n
1,n,n
1,1,0
2,2,0
0,0,1
2,2,1
0,0,n
1,1,n
~
0,0,0
1,1,1
1,0,0
0,1,1
1,1,0
0,0,1
Symmetry-reduced State Graph for P2 (restricted to the reachable states)

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.

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 scalarset is not used in array indices, regardness of the exact size of the data scalarset, the global variables have a fixed number of variables with the data scalarset type. Let this number be N. For example, the system described in Figure 4.1 has three variables with data scalarset types.
>
> Therefore, regardless of the exact size of the scalarset, a state can have at most N different values. We can always find a permutation π 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
>
> $$\textbf{exists } i_1, i_2, ..., i_k : \textit{data} \textbf{ do } i_1 \neq i_2 \neq ... \neq i_k \textbf{ end}$$
>
> can be used to distinguish whether or not the size of the scalarset is no less than k. Taking the N global variables of the data scalarset type into account, a Boolean expression can be used to distinguish whether or not the data scalarset has $N + k$ elements or more. However, it cannot distinguish whether or not the data scalarset has $N + k$ elements or $N + k + 1$ elements, and similarly for data scalarset of larger sizes.

Although the bound given in the intuition is based on the number of data scalarset locations in global variables and the bounded variables, data saturation typically happens for a much smaller data scalarset. A simple run-time check can be used to determine when data saturation has occurred. Consider a Murφ description with a maximum of k bounded variables of a data scalarset type. If the data scalarset 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.

# of possible data values	1	2	3	4	5	...	n
DASH-C (3 processors) without DMA operations							
Original size	26,925	91,254	210,663	exceeding 400,000 states			
Reduced size	4,575	7,741	8,251	8,276	8,276	...	8,276
ICCP (4 processors) with restriction in the number of messages							
Original size	200,913	762,114	2,034,099	exceeding 3,000,000 states			
Reduced size	8,534	16,169	18,619	18,851	18,851	...	18,851

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 [1]

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].

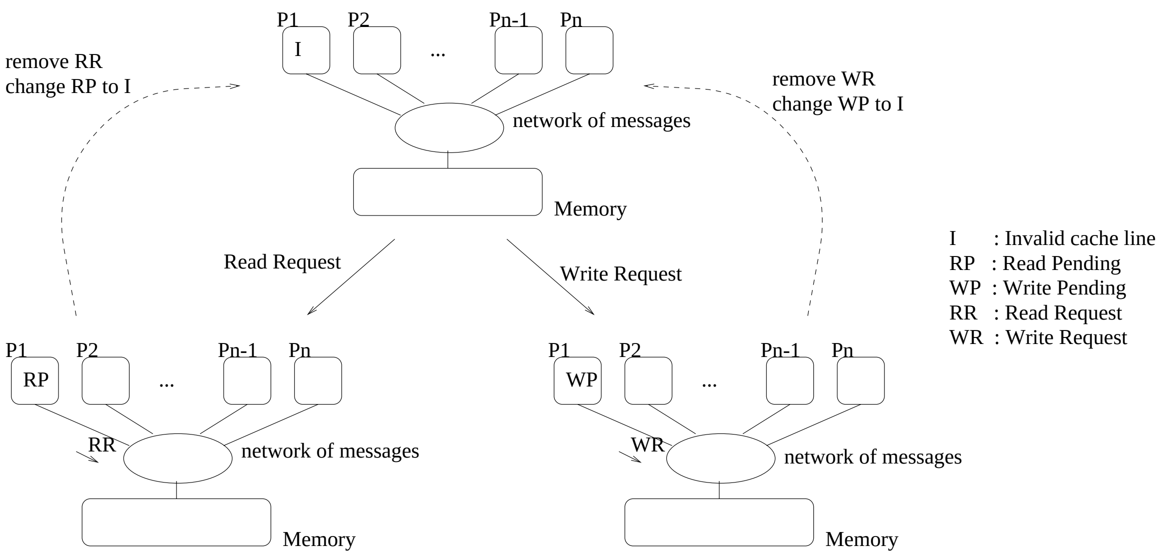


Figure 5.1: Reversible rules in a cache coherence protocol: The original state can be reconstructed by removing the message from the network and changing the internal state of the processor back to invalid.

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 time to re-construct the

[2]If the processors are symmetric, $(n + 1)(n + 2)/2$ equivalence classes are generated.

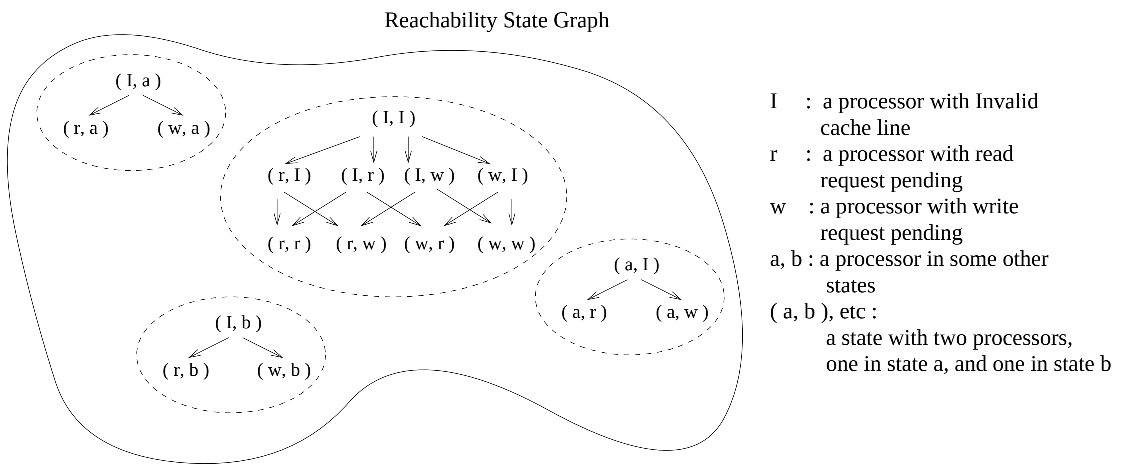

Reachability State Graph
(I, a)
(r, a)
(w, a)
(I, I)
(r, I)
(I, r)
(I, w)
(w, I)
(r, r)
(r, w)
(w, r)
(w, w)
(a, I)
(a, r)
(a, w)
(I, b)
(r, b)
(w, b)
I : a processor with Invalid cache line
r : a processor with read request pending
w : a processor with write request pending
a, b : a processor in some other states
(a, b), etc : a state with two processors, one in state a, and one in state b

Figure 5.2: State explosion due to reversible rules: A set of reversible rules generates a lot of similar subgraphs in the reachability state graph, causing the number of states to explode as the number of processors increases.

progenitor, and store only the progenitor in the memory. The precise definitions of reversible rules and the reduction algorithm are presented in the next section.

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 $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, a rule $r \in T$ is* a reversible rule *if and only if*

- *for all $q \in Q$, if $q \neq \mathbf{error}$, then $r(q) \neq \mathbf{error}$.*
- *there exists a function r^* such that for all $q \in Q$, if there exists a unique $q' \in Q$ such that $q' \neq q \wedge r(q') = q$, 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 $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, 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 $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, 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 $q \xrightarrow{r_1,...,r_n} q'$.*

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*.

Theorem 5.1 (uniqueness of progenitor) *If a rule set T generates a state graph $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, and $U = \{r_1, ..., r_m\} \subseteq T$ 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 $(r_m^*)^n(...((r_1^*)^n(q))...)$.*

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 $(r_m^*)^n(...((r_1^*)^n(q))...)$. 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 $1 \leq i \leq m$,

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

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 $r'_1, ..., r'_k \in U$ such that $q' = r_k'^*(...(r_1'^*(q))...)$. Since we can rearrange the order of application of $r_i'^*$, we can group the identical rules together and get $q' = (r_m^*)^{k_m}(...((r_1^*)^{k_1}(q))...)$ 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, $q' = (r_m^*)^{n-k_m}((r_m^*)^{k_m}(...((r_1^*)^{k_1}(q))...)) = (r_m^*)^n(...((r_1^*)^{k_1}(q))...)$ Repeat the argument for every $r_i \in U$, we have $q' = (r_m^*)^n(...((r_1^*)^n(q))...) = \theta(q)$.

$k_m > n$**:**

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

□

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

- *$\theta(Q) = \{\theta(q) | q \in Q\}$;*
- *$\theta(Q_0) = \{\theta(q) | q \in Q_0\}$; 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 $q_1 \overset{t}{\Longrightarrow} q_2$ 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 $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, 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 $\theta(q_0) \overset{t}{\Longrightarrow} \theta(q)$ 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 $A = \langle Q, Q_0, \Delta, \mathbf{error}\rangle$, 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:

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

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

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 *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 ∀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 ∀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
    $Reached = Unexpanded = \{\ \underline{\theta(q)}\ \mid q \in Q_0\}$;
    while $Unexpanded \neq \emptyset$ do
        Remove a state $s$ from $Unexpanded$;
        $\underline{\text{Local\_Search(s);}}$
    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;
|         endfor;
|     endwhile;
| end

Generate_Original_Next_States(state s)
begin
    for each transition rule $t \in \underline{\ T \setminus U\ }$ do
        let $s' = t(s)$ in
            if $s'$ = error then stop and report error; endif;
            if $\underline{\theta(s')}$ is not in $Reached$ then
                Put $\underline{\theta(s')}$ 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.

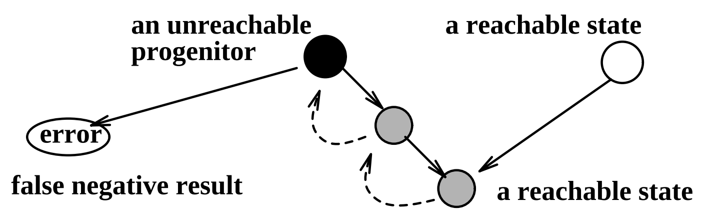


Figure 5.4: Potential false error report from reduction using reversible rules: Although a transient state is reachable from an initial state, its progenitor may not be reachable, and it may lead to a false error report.

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 $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, a communtative 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 $q_0, q_1, q_1' \neq$ **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 $q_0' \xrightarrow{t} q_1'$.*

Pictorally, the essential property can be summarized as:

$$\begin{tikzcd}
q_0 \arrow[rr, "t"] & & q_1 & & & q_0 \arrow[rr, "t"] & & q_1 \\
& q_1' \arrow[ur, "r"] & & \Rightarrow & q_0' \arrow[ur, "r"] \arrow[rr, "t"] & & q_1' \arrow[ur, "r"] &
\end{tikzcd}$$

Lemma 5.2 *If a rule set T generates a state graph $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, and $U \subseteq T$ is a commutative reversible and essential rule set, whenever $q_0 \overset{t}{\Longrightarrow} q_1$ is in A_U, there exist $r_1, ..., r_k \in U$ such that $q_0 \overset{r_1,...,r_k,t}{\longrightarrow} q_1$ is in A.*

Proof. Since $q_0 \overset{t}{\Longrightarrow} q_1$ 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 $q_0' \overset{t}{\longrightarrow} q_1'$.

If $q_1 \overset{r_1',...,r_z'}{\longrightarrow} q_1'$, by recursively applying the essential property on $r_1', ..., r_z'$, there exists a state q_0'' such that $q_0'' \overset{r_1',...,r_z'}{\longrightarrow} q_0'$ and $q_0'' \overset{t}{\longrightarrow} q_1$. Consider the progenitor $\theta(q_0'')$ of q_0'', there exists $r_1, ..., r_k$ such that $\theta(q_0'') \overset{r_1,...,r_k}{\longrightarrow} q_0''$. Therefore, $\theta(q_0'') \overset{r_1,...,r_k,r_1',...,r_z'}{\longrightarrow} q_0'$ and $\theta(q_0'') = \theta(q_0') = q_0$. Hence, $q_0 \overset{r_r,...,r_k,t}{\longrightarrow} q_1$.

Pictorially, it can be illustrated as:

$$\begin{tikzcd}
& q_0' \arrow[rr, "t"] & & q_1' & \Rightarrow & q_0' \arrow[rr, "t"] & & q_1' \\
& & q_1 \arrow[ur, "r_1', ..., r_z'"'] & & q_0'' \arrow[ur, "r_1', ..., r_z'"'] \arrow[rr, "t"] & & q_1 \arrow[ur, "r_1', ..., r_z'"'] & \\
q_0 \arrow[uur, "..."'] & & & q_0 \arrow[ur, "r_1, ..., r_k"'] & & & &
\end{tikzcd}$$

□

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 $A = \langle Q, Q_0, \Delta, \mathbf{error}\rangle$, 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.*

Proof. If $q_0 \xRightarrow{t_1} q_1 \xRightarrow{t_2} ... \xRightarrow{t_m} q$ 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 $q_0 \overset{r_1,...,r_{k_1},t_1}{\longrightarrow} q_1 \overset{r_{k_1+1},...,r_{k_2},t_2}{\longrightarrow} ... \overset{r_{k_{m-1}+1},...,r_{k_m},t_m}{\longrightarrow} q$ is a path from the initial state q_0 in the original state graph.

□

Corollary 5.2 *If* **error** *is reachable from the initial states in A_U,* **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 $A = \langle Q, Q_0, \Delta, \textbf{error} \rangle$, 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 ∀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.

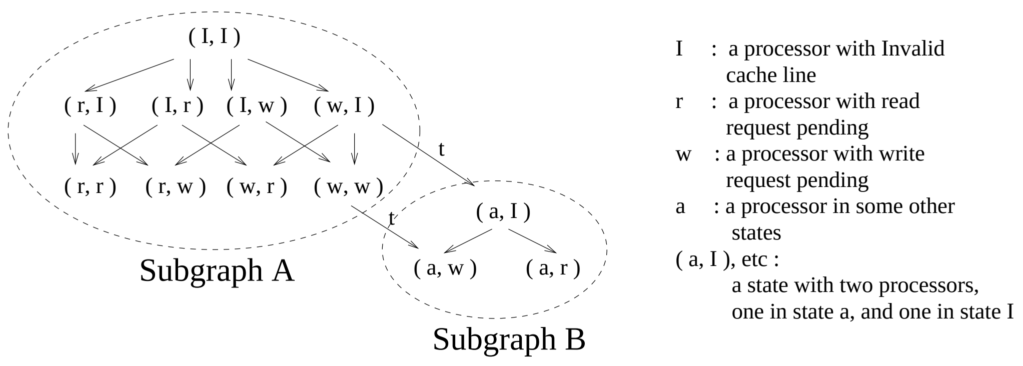

(I, I)
(r, I)
(I, r)
(I, w)
(w, I)
(r, r)
(r, w)
(w, r)
(w, w)
t
t
(a, I)
(a, w)
(a, r)
Subgraph A
Subgraph B
I : a processor with Invalid cache line
r : a processor with read request pending
w : a processor with write request pending
a : a processor in some other states
(a, I), etc : a state with two processors, one in state a, and one in state I

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 $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$, a commutative reversible rule set $U \subseteq T$ is singular w.r.t. T if and only if whenever $q_1 \overset{r_1, r_2, t}{\longrightarrow} q_2$, we have either $q_1 \overset{r_1, t, r_2}{\longrightarrow} q_2$ or $q_1 \overset{r_2, t, r_1}{\longrightarrow} q_2$.*

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 $A = \langle Q, Q_0, \Delta, \mathbf{error}\rangle$, 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 $q \xrightarrow{r_1,\ldots,r_{k+1},t} q'$. There exists p_1 such that $q \xrightarrow{r_1} p_1 \xrightarrow{r_2,\ldots,r_{k+1},t} q'$. 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 $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$, 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 speeded 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
    $Reached = Unexpanded = \{\ \theta(q) \quad | \ q \in Q_0\}$;
    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.

ICCP: 4 processors	states stored	states examined	time
Original (unordered network)	247,565	247,565	205s
Alg. 1	34,005	247,565	338s
Alg. 2	34,005	123,197	128s
ICCP: 5 processors	states stored	states examined	time
Original unordered network)	> 6,500,000 states		
Alg. 1	492,075	6,568,279	4 hours
Alg. 2	492,075	2,206,135	66 mins

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.

successor other than itself, the algorithm does not need to execute the non-reversible rules (and apply θ) 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 $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$, 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 $q_0 \overset{t}{\Longrightarrow} q_1$ is in the reduced state graph, there exists $r_1, ..., r_k \in U$ such that $q_0 \overset{r_1,...,r_k,t}{\Longrightarrow} q_1$ is in the original state graph. By the definition of automorphism, there exists $r'_1, ..., r'_k \in U$ and $t' \in T \setminus U$ such that

$h(q_0) \overset{r'_1,\ldots,r'_k,t'}{\Longrightarrow} h(q_1)$ is in the original state graph. Therefore, $h(q_0) \overset{t'}{\Longrightarrow} h(q_1)$ 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 $A = \langle Q, Q_0, \Delta, \textbf{error}\rangle$, 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 $[A_U]_H = \langle \theta'(Q), \theta'(Q_0), \theta'(\Delta), \textbf{error}'\rangle$, defined as:*

- $\theta'(Q) = \{[\theta(q)] | q \in Q\}$
- $\theta'(Q_0)' = \{[\theta(q)] | q \in Q_0\}$
- $(q_1, q_2) \in \theta'(\Delta)$ *if and only if there exist* $q \in Q$ *and* $t \in T \backslash U$ *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 θ, 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 θ to find the progenitor, the time required per state generated in the reduction algorithm is longer than that in the original algorithm.

	ICCP (p4)	LIST1 (P4)	LIST2 (p4)	DASH-C w/o DMA (p3)	DASH-L (p3)
	(unordered network)				
Original size	247,565	301,029	329,601	26,925	55,366
Reduced size	34,005	112,784	162,736	15,751	36,728
Original time	205s	87s	250s	114s	188s
Reduced time	128s	72s	239s	85s	213s

$p\langle n \rangle$: n-processor system

Table 5.2: Improvement on performance with singular reversible rules: Depending on the number of reversible rules, reductions in memory of up to 86% are obtained.

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.

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 Murφ, and the many possibilities of detecting reversible rules, this process is not automated in Murφ yet.

	ICCP (p4)	LIST1 (p4)	LIST2 (p4)	DASH-C	DASH-L (p3)
	(unordered network)			w/o DMA (p3)	
Reduced size (s)	11,814	13,044	13,959	4,575	9,313
Reduced size (s & r)	1,760	4,926	6,894	2,672	6,170
Reduced time (s)	28s	13s	24s	63s	96s
Reduced time (s & r)	13s	11s	22s	50s	96s

ICCP	p4	p5	p6	p7	p8
Original size (unordered network)	247,565	> 6,500,000 states			
Reduced size (s)	11,814	68,879	358,078	> 1,500,000 states	
Reduced size (s & r)	1,760	6,021	18,118	49,045	121,302
Original time	205s	–	–	–	–
Reduced time (s)	28s	349s	3,762s	–	–
Reduced time (s & r)	13s	98s	615s	3,283s	12,801s

s	: Symmetry Reduction
r	: Reversible Rules Reduction
p$\langle n \rangle$	: 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 send 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 "l_1 : **if** $(x = 2)$ **then** l_2 : $x := 3$; **end** l_3 :," where l_1, l_2, l_3 are labels for the corresponding locations. The only way to get to a state with l_2 active is to go through l_1. Therefore, given a state with $x = 2$ and l_2 active, we can replace l_2 with l_1 to reconstruct the state before executing the guard of the program.

- Consider the guarded command "**rule** $x = 2 \Rightarrow x := 3$; **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 $\perp$.

- 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:

$$P_i : \quad p_i \xrightarrow{a_i} q_i$$

	states examined	states stored
original state graph	2^n	2^n
partial order reduction	$n+1$	$n+1$
reversible rule reduction without singularity property	2^n	1
reversible rule reduction with singularity property	$n+1$	1

$$\begin{tikzcd}
\langle p_1, p_2, ..., p_n \rangle \arrow[d, "a_1"] \\
\langle q_1, p_2, ..., p_n \rangle \arrow[d, "a_2"] \\
\cdots \arrow[d, "a_n"] \\
\langle q_1, q_2, ..., q_n \rangle
\end{tikzcd}$$

$$\begin{tikzcd}
\langle p_1, p_2, ..., p_n \rangle \arrow[r, "a_1"] \arrow[dr, "a_2"] \arrow[ddr, phantom, "\cdots"] \arrow[dddr, "a_n"'] & \langle q_1, p_2, ..., p_n \rangle \\
& \langle p_1, q_2, ..., p_n \rangle \\
& \cdots \\
& \langle p_1, p_2, ..., q_n \rangle
\end{tikzcd}$$

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

	states examined	states stored
original state graph	3^n	3^n
partial order reduction	$2^{(n+1)} - 1$	$2^{(n+1)} - 1$
reversible rule reduction without singularity property	3^n	1
reversible rule reduction with singularity property	$2n + 1$	1

$$\begin{tikzcd}
 & & & \langle p_1, p_2, ..., p_n\rangle \arrow[dll, "a_1"'] \arrow[drr, "b_1"] & & & \\
 & \langle r_1, p_2, ..., p_n\rangle \arrow[dl, "a_2"'] \arrow[dr, "b_2"] & & & & \langle q_1, p_2, ..., p_n\rangle \arrow[dl, "a_2"'] \arrow[dr, "b_2"] & \\
 ... & & ... & & ... & & ...
\end{tikzcd}$$

$$\begin{tikzcd}
\langle p_1, p_2, ..., p_n\rangle \arrow[r, "a_1"] \arrow[dr, "b_1"] \arrow[ddr, phantom, "..."] \arrow[dddr, "b_n"] & \langle q_1, p_2, ..., p_n\rangle \\
 & \langle r_1, p_2, ..., p_n\rangle \\
 & ... \\
 & \langle p_0, p_1, ..., r_n\rangle
\end{tikzcd}$$

Partial Order Reduction

Reversible Rules Reduction

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.

method, reducing the number of states examined to 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.

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 $\overset{t}{\Longrightarrow}$ can be regarded as a series of original transitions $\xrightarrow{r_1,\ldots,r_n,t}$. 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 6

Verifying Scalable Systems

Chapter Overview [1]

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.

[1]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

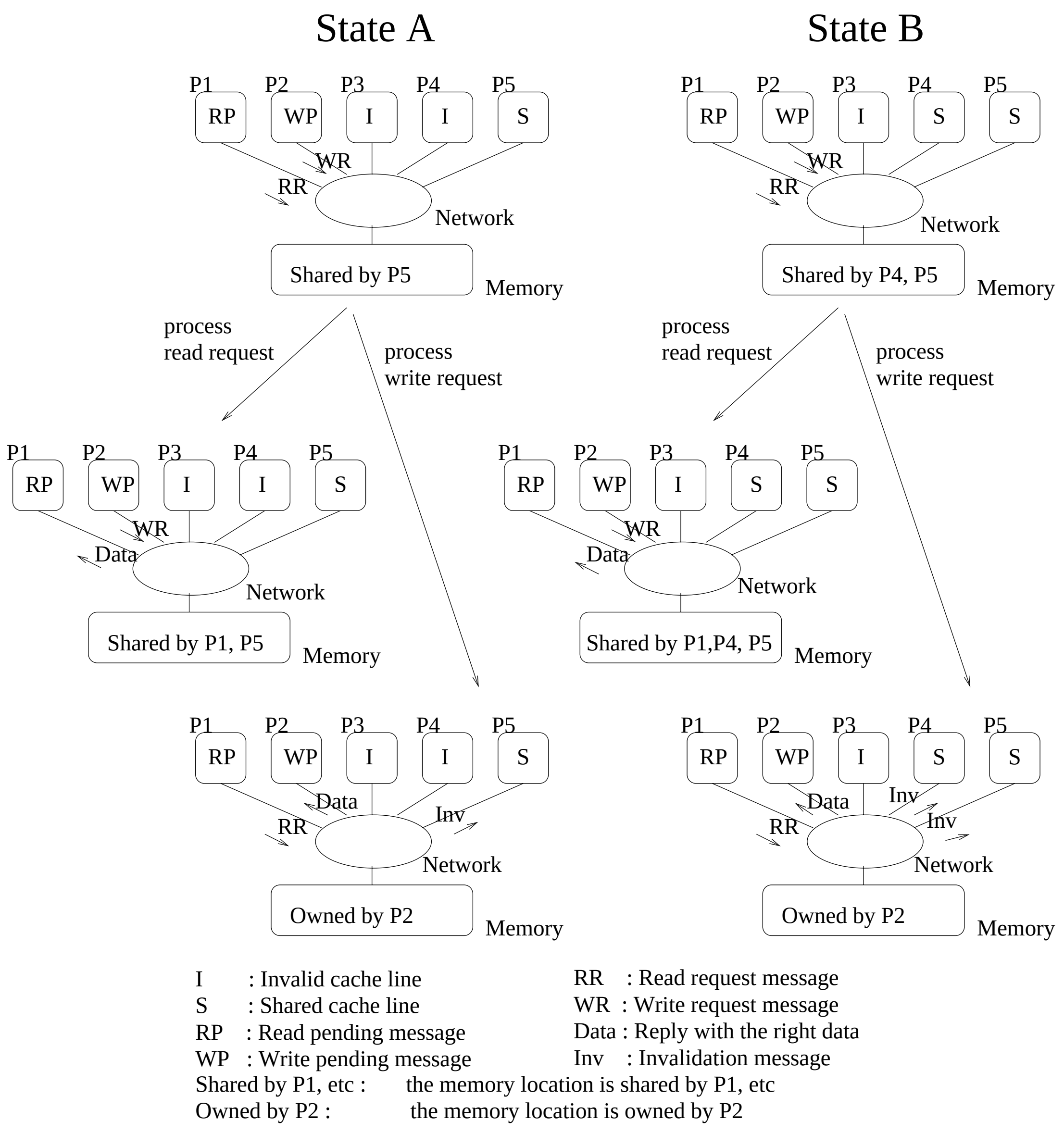

State A
State B
P1 P2 P3 P4 P5
RP WP I I S
RP WP I S S
WR
RR
Data
Inv
Network
Shared by P5
Shared by P4, P5
Shared by P1, P5
Shared by P1,P4, P5
Owned by P2
Memory
process read request
process write request
I : Invalid cache line
S : Shared cache line
RP : Read pending message
WP : Write pending message
Shared by P1, etc : the memory location is shared by P1, etc
Owned by P2 : the memory location is owned by P2
RR : Read request message
WR : Write request message
Data : Reply with the right data
Inv : Invalidation message

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.

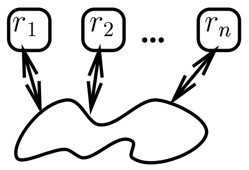
a particular memory value is *invalid* in caches 1,2,3,5,6,7 and *writable* in cache 4. An abstract state may be used to record that more than zero caches are *invalid*, and exactly one is *writable*, "forgetting" not only the number of processors in each state but also the ordering of the processors. 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 $\forall$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_i'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.



modelled as

$$\longrightarrow \quad (s, [r_1, r_2, \ldots, r_n])$$

rearranged to

$$\longrightarrow \quad (s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$$

abstracted into

$$\longrightarrow \quad (s, \{q_1^+, \ldots, q_k^+\})$$

automatic verification using
manually translated
abstract transitions

$$\longrightarrow \quad \text{an abstract state graph}$$

Figure 6.2: Manual translation to use repetition constructors: In previous work, in order to generate an abstract state graph, the user has to explicitly model the system in a special format, and to rewrite the transitions in term of the abstract states.

In the remainder of this chapter, a fully automatic abstraction algorithm is described, which does not require an abstract executable description.

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 convention, 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 subtype 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 processors 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 scalarset 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 Murφ 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 $c_1, ..., c_n$ 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

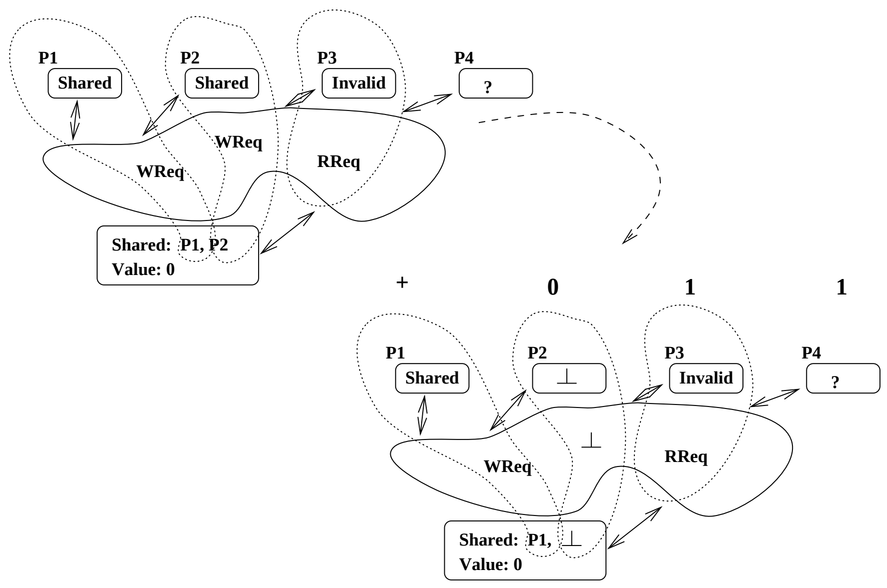


Figure 6.3: Automatic generation of abstract states: Without rearranging the state to the form $(s, [q_1, \ldots, q_k])$, the component states of two abstractable components can be compared. Since processors 1 and 2 have the same component state, one of them is removed from the state (by using the constructor 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* **RuleSet***s 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 $a_1, ..., a_k \geq 1$, and $q_1, ..., q_k$ are distinct. It represents the set of states that are equivalent to $(s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$ where a_i denotes the number of q_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:*

$$\begin{array}{ll} (s, q_i, \{q_1^{a_1}, \ldots, q_i^{(a_i)-1}, \ldots, q_k^{a_k}\}) & \textit{if } a_i > 1\textit{; and} \\ (s, q_i, \{q_1^{a_1}, \ldots, q_{i-1}^{a_{(i-1)}}, q_{i+1}^{a_{(i+1)}}, \ldots, q_k^{a_k}\}) & \textit{if } a_i = 1, \end{array}$$

representing the same set of states as p.

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

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 $(s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\})$ 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 $p' = (s, \{q_1^{a_1}, ..., q_k^{a_k}\})$ and a transition rule t without an abstractable active component.

Pictorially, the repetitive property may be summarized as:

$$\begin{tikzcd}
(s, q, \{q_1, \ldots, q_k\}) \arrow[d, "t"] & \Leftrightarrow & (s, q, \{q_1^{a_1}, \ldots, q_k^{a_k}\}) \arrow[d, "t"] \\
(s', r, \{r_1, \ldots, r_k\}) & & (s', r, \{r_1^{a_1}, \ldots, r_k^{a_k}\})
\end{tikzcd}$$

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

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 $(s', r, \{r_1^{a_1}, \ldots, r_k^{a_k}\})$ 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 $\{q_1, \ldots, q_k\}$ if it does not appear in $[r_1, \ldots, r_z]$.

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

$$\begin{tikzcd}
& (s, \{q_1^+, q_2^+\}) \arrow[dl, dash] \arrow[dr, dash] & \\
(s, \{q_1^1, q_2^+\}) \arrow[dr, dash] & & (s, \{q_1^+, q_2^1\}) \arrow[dl, dash] \\
& (s, \{q_1^1, q_2^1\}) &
\end{tikzcd}$$

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 **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 $[p] = (s, \{q_1, \ldots, q_k\})$ 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 $(s', \{r_1^{e_1}, \ldots, r_k^{e_k}\})$. 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 $e_i = +$) :

$$\left\{(s, \{q_1^{a_1}, \ldots, q_i^{1}, \ldots, q_k^{a_k}\}) \ \mid \ a_i \in e_i\right\}, \text{ and}$$

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

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 **error**.

CASE 1

$$\begin{tikzcd}
(s, \{q_1^{e_1}, ..., q_n^{e_n}\}) \arrow[d] \\
(s, q_i, \{q_1, ..., q_n\}) \arrow[d, "t"'] \\
(s', r, \{r_1, ..., r_n\}) \arrow[d, "\text{restoring constructors}"] \\
(s', \{r_1^{e_1}, ..., r_k^{e_k}\}) \arrow[d, "\text{re-partition and combining}"] \\
\text{abstract successor } A
\end{tikzcd}$$

CASE 2

$$\begin{tikzcd}
(s, \{q_1^{e_1}, ..., q_i^1, ..., q_n^{e_n}\}) \arrow[d] \\
(s, q_i, \{q_1, ..., q_{i-1}, q_{i+1} ..., q_n\}) \arrow[d, "t"'] \\
(s', r, \{r_1, ..., r_{i-1}, r_{i+1}, ..., r_n\}) \arrow[d, "\text{restoring constructors}"] \\
(s', \{r_1^{e_1}, ..., r_{i-1}^{e_{i-1}}, r^1, r_{i+1}^{e_{i+1}}, ..., r_k^{e_k}\}) \arrow[d, "\text{re-partition and combining}"] \\
\text{abstract successor } A
\end{tikzcd}$$

CASE 3

$$\begin{tikzcd}
& (s, \{q_1^{e_1}, ..., q_i^+, ..., q_n^{e_n}\}) \arrow[dl] \arrow[dr] & \\
(s, q_i, \{q_1, ..., q_{i-1}, q_{i+1}, ..., q_n\}) \arrow[d, "t"'] & & (s, q_i, \{q_1, ..., q_i, ..., q_n\}) \arrow[d, "t"'] \\
(s', r, \{r_1, ..., r_{i-1}, r_{i+1}, ..., r_n\}) \arrow[d, "\text{restoring constructors}"] & & (s', r, \{r_1, ..., r_i, ..., r_n\}) \arrow[d] \\
(s', \{r_1^{e_1}, ..., r_{i-1}^{e_{i-1}}, r^1, r_{i+1}^{e_{i+1}}, ..., r_k^{e_k}\}) \arrow[d, "\text{re-partition and combining}"] & & (s', \{r_1^{e_1}, ..., r^1, r_i^+, ..., r_k^{e_k}\}) \arrow[d] \\
\text{abstract successor } A & & \text{abstract successor } B
\end{tikzcd}$$

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. [BBG+93, 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 Murφ 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 Murφ 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

```
Simple_Abstraction_Algorithm()
begin
    $Reached = Unexpanded = \{\underline{\textbf{abs}(q)} \mid q \in StartState\}$;
    while $Unexpanded \neq \phi$ do
        Remove a state $s$ from $Unexpanded$;
        for each transition rule $t \in T$ do
            Generate the set of representatives, $R$ from $s$;
            for each representative $r \in R$ 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 $(s, \{q_0^+\})$ 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 $(s, \{q_1^{e_1}, \ldots, q_k^{e_k}\})|_n$ 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 $(s, \{q_1^1, q_2^+\})|_2$ and $(s, \{q_1^+, q_2^1\})|_2$ is the same as $(s, \{q_1^1, q_2^1\})|_2$. Murφ 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 Murφ uses a restricted state graph, we can still use Murφ 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,

$$(s \rightarrow q_1 \rightarrow ... \rightarrow q_k, \{q_{k+1}^{e_{k+1}}, ..., q_n^{e_n}\})$$

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
    $Reached = Unexpanded = \{\mathbf{abs}(q) \mid q \in StartState\}$;
    while $Unexpanded \neq \phi$ do
        Remove a state $s$ from $Unexpanded$;
        for each transition rule $t \in T$ do
            Generate the set of representatives, $R$ from $s$;
            for each representative $r \in R$ 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
                endlet;
            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.

$(s, \{q_1^1, q_2^+\})$ is redundant in the set of previously examined states when $(s, \{q_1^+, q_2^+\})$ 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.

6.4.1 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, $(s, \{q_1^1, q_2^+, q_3^+\})$ covers $(s, \{q_1^1, q_2^1, q_3^+\})$; 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.

6.4.2 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

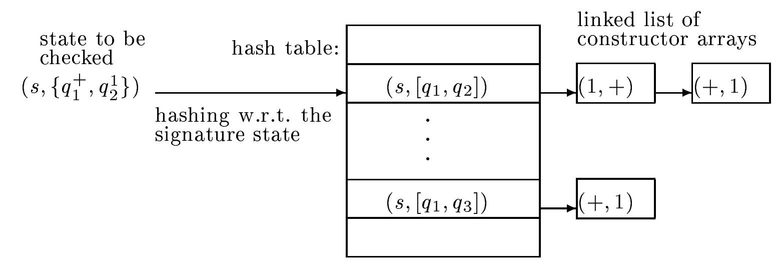


Figure 6.7: Hashing structure for efficient comparison of abstract states: Comparable states are hashed into the same location in the hash table.

ICCP system of 9 processors	BFS	DFS	Priority BFS	Priority DFS
Max length in lists	17	9	6	7
Average number of comparisons when hashed to an occupied location	1.28	1.06	1.05	1.03

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 $(s, \{q_1^1, q_2^+, q_3^+\})$ has a higher priority than $(s', \{r_1^1, r_2^1, r_3^+\})$, 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 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.

ICCP system of 9 processors	BFS	DFS	Priority BFS	Priority DFS
Final size	35,515	35,515	35,515	35,515
Maximum intermediate size	49,259	36,968	35,515	35,515
Number of Non-maximal States Examined	106,528	93,430	3,527	3,263
Time required	51,425s	49,060s	18,400s	17,477s

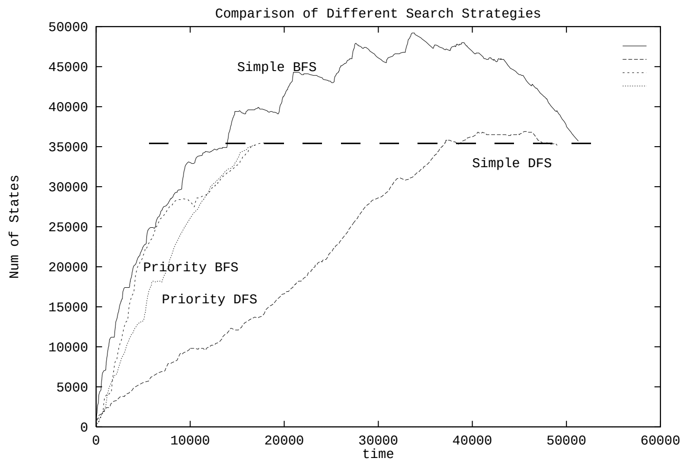


Table 6.2: Performance of the priority search strategies: The priority search strategies outperform the conventional search strategies by more than 65% in speed and 30% in memory.

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 $\mathbf{abs}(A)$.

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 $\mathbf{sig}(p)$ 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 $\mathbf{abs}(f)$ on the abstract states in $\mathbf{abs}(A)$ is defined so that for all abstract states p_1 and p_2, $\mathbf{abs}(f)(p_1) = p_2$ if and only if $f(\mathbf{sig}(p_1)) = \mathbf{sig}(p_2)$ 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, $\mathbf{abs}(h)$ is an automorphism on the abstract state graph $\mathbf{abs}(A)$.*

Proof. Given a state $(s, [q_1, \ldots, q_k])$ in the original state graph, and a permutation π 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 $p = (s, \{q_1^{e_1}, \dots, q_k^{e_k}\})$, the corresponding permutation $\mathbf{abs}(\pi)$ on p generates $(\pi(s), \{\pi(q_1)^{e_1}, \dots, \pi(q_k)^{e_k}\})$, which can be illustrated as follows:

$$\begin{tikzcd}
(s, [q_1, \dots, q_k]) \arrow[d, "\pi"] & & (s, \{q_1^{e_1}, \dots, q_k^{e_k}\}) \arrow[d, "\mathbf{abs}(\pi)"] \\
(\pi(s), [\pi(q_1), \dots, \pi(q_k)]) \arrow[rr, phantom, "\Leftrightarrow"] & & (\pi(s), \{\pi(q_1)^{e_1}, \dots, \pi(q_k)^{e_k}\})
\end{tikzcd}$$

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: $(s, q_i, \{q_1, \dots, q_k\})$ and $(s, q_i, \{q_1, \dots, q_{i-1}, q_{i+1}, \dots, q_k\})$.

Similarly, for $\mathbf{abs}(\pi)(p)$, 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

$$(\pi(s), \pi(q_i), \{\pi(q_1), \dots, \pi(q_k)\}), \text{ and}$$

$$(\pi(s), \pi(q_i), \{\pi(q_1), \dots, \pi(q_{i-1}), \pi(q_{i+1}), \dots, \pi(q_k)\}).$$

These two representatives for $\mathbf{abs}(p)$ are equivalent to the representatives for p, and therefore, the abstract successors for $\mathbf{abs}(\pi)(p)$ 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 $(s, [q_1, \dots, q_n])$ in the original state graph, the execution of a localized rule with an active component i generates a successor $(s, [q_1, \dots, q_{i-1}, r_i, q_{i+1}, \dots, q_n])$ 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 θ (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 θ 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 $\mathbf{abs}(\theta)(p)$.*
- *If U is a singular rule set for A, $\mathbf{abs}(U)$ is also a singular rule set for $\mathbf{abs}(A)$.*

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 $\mathbf{sig}(p)$, 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 $(s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, r_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_k^{e_k}\})$.

Pictorially, this can be summarized as:

$$\begin{tikzcd}
(s, [q_1, \ldots, q_k]) \arrow[d, "r"] \\
(s, [q_1, \ldots, q_{i-1}, r_i, q_{i+1}, \ldots, q_n]) \arrow[d, "\text{abstraction}"] \\
(s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, r_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_k^{e_k}\})
\end{tikzcd}$$

Other states in the form of $(s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$ 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_i, 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

$$(s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_{i-1}, \ldots, q_{i-1}}_{a_{i-1}}, \underbrace{r_i, \ldots, r_i}_{a_i}, \underbrace{q_{i+1}, \ldots, q_{i+1}}_{a_{i+1}}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}]),$$

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

Pictorially, it can be summarized as:

$$\begin{tikzcd}
(s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}]) \arrow[d, "r_{(a_1+\ldots+a_{i-1}+1)}"] \\
\ldots \arrow[d, "r_{(a_1+\ldots+a_{i-1}+a_i)}"] \\
(s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_{i-1}, \ldots, q_{i-1}}_{a_{i-1}}, \underbrace{r_i, \ldots, r_i}_{a_i}, \underbrace{q_{i+1}, \ldots, q_{i+1}}_{a_{i+1}}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}]) \arrow[d, "\text{abstraction}"] \\
(s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, r_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_k^{e_k}\})
\end{tikzcd}$$

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 $(s, \{q_1^{e_1}, \ldots, q_{i-1}^{e_{i-1}}, r_i^{e_i}, q_{i+1}^{e_{i+1}}, \ldots, q_k^{e_k}\})$, 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 $\mathbf{abs}(\theta)(p)$.

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 $q_1 \xrightarrow{r_1,t,r_2} q_2$ or $q_1 \xrightarrow{r_2,t,r_1} q_2$.

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

□

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.

# of processors (ICCP)	3	4	5	6	7	8	9
size (unordered network)	10,077	247,565	–	–	–	–	–
size (sym. only)	1,781	11,814	68,879	358,078	–	–	–
size (rep. only)	1,770	11,206	57,790	257,692	–	–	–
size (sym./rev.)	434	1,760	6,021	18,118	49,045	121,302	–
size (rev./rep.)	427	1,590	4,542	10,587	19,485	28,927	35,515
time (unordered network)	5.1s	205s	–	–	–	–	–
time (sym. only)	2.4s	28s	349s	3,762s	–	–	–
time (rep. only)	4.4s	49s	497s	4,555s	–	–	–
time (sym./rev.)	2.1s	13s	98s	615s	3,283s	12,801s	–
time (rep./rev.)	3.3s	27s	167s	811s	3,265s	7,593s	17,477s

# of processors (ICCP)	10	11	12	13	14 and up
size (rep./rev.)	38,146	38,485	38,329	38,269	38,269
time (rep./rev.)	29,871s	37,903s	43,352s	48,410s	49,932s

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.

# of processors (ICCP)	3	4	5	6	7	8	9
size	427	1,590	4,542	10,587	19,485	28,927	35,515
# of full states	359	1,135	2,756	5,298	7,830	8,589	6,800

# of processors (ICCP)	10	11	12	13	14 and up
size	38,146	38,485	38,329	38,269	38,269
# of full states	3,765	1,383	303	30	0

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 Murφ 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 Murφ 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, $\perp$ represents an undefined value, $\mathbf{N}$ represents the natural numbers, and $\mathbf{N}_\perp$ represents $\mathbf{N} \cup \{\perp\}$. $[L \to \mathbf{N}_\perp]$ represents the set of all functions with domain L and codomain $\mathbf{N}_\perp$.

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 $\Box$ among simpler transition rules. The ruleset in the actual Murφ language can be translated into a nondeterministic choice $\Box$ over a parameter. The Boolean guard of a rule in the actual Murφ language can be translated into a **if** statement.

```
⟨program⟩    ::=  type ⟨decls⟩
                  var ⟨id⟩ ‘:’ ⟨typeExpr⟩
                  init ⟨rule⟩
                  begin ⟨rule⟩ end

⟨decls⟩      ::=  ⟨id⟩ ‘:’ ⟨typeExpr⟩ { ‘;’ ⟨id⟩ ‘:’ ⟨typeExpr⟩ }

⟨typeExpr⟩   ::=  ⟨id⟩
                  | ⟨num⟩ ‘..’ ⟨num⟩
                  | record ⟨decls⟩ end
                  | array ‘[’ ⟨typeExpr⟩ ‘]’ of ⟨typeExpr⟩

⟨rule⟩       ::=  ⟨stmt⟩
                  | ‘□’ ‘(’ ⟨id⟩ ‘:’ ⟨typeExpr⟩ ‘)’ ⟨rule⟩
                  | ⟨rule⟩ ‘□’ ⟨rule⟩

⟨stmt⟩       ::=  error
                  | ⟨var⟩ ‘:=’ ⟨term⟩
                  | if ⟨boolExpr⟩ then ⟨stmt⟩ end
                  | for ‘(’ ⟨id⟩ ‘:’ ⟨typeExpr⟩ ‘)’ do ⟨stmt⟩ end
                  | ⟨stmt⟩ ‘;’ ⟨stmt⟩

⟨boolExpr⟩   ::=  ⟨term⟩ ‘=’ ⟨term⟩
                  | ⟨term⟩ ‘>’ ⟨term⟩
                  | ‘¬’ ⟨boolExpr⟩
                  | ⟨boolExpr⟩ ‘∧’ ⟨boolExpr⟩
                  | ‘∀’ ‘(’ ⟨id⟩ ‘:’ ⟨typeExpr⟩ ‘)’ ⟨boolExpr⟩
                  | ‘(’ ⟨boolExpr⟩ ‘)’

⟨term⟩       ::=  ⟨var⟩ | ⟨num⟩ | ⊥ | ⟨term⟩ ‘+’ ⟨term⟩
⟨var⟩        ::=  ⟨id⟩ | ⟨var⟩ ‘.’ ⟨id⟩ | ⟨var⟩ ‘[’ ⟨term⟩ ‘]’
⟨id⟩         ::=  string
⟨num⟩        ::=  number
```

⟨⟩	denotes nonterminals;
bold face or ‘’	denotes terminals;
{}	denotes repetition zero or more times;
a | b	denotes either a or b;

Figure A.1: Syntax for the Simple Description Language

A.1.1 Abstract Transition Programs

A Murφ program specifies an abstract transition program: An abstract transition program is a quadruple $\langle L, \mathcal{T}, \mathcal{I}, \mathcal{R}\rangle$. L is a set of *locations* and $\mathcal{T} : L \to \mathbf{N} \times \mathbf{N}$ 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 $\mathcal{T}$. A state in the corresponding state graph is either an **error** state or an assignment of legal values to every location in L. $\mathcal{I}$ is a set of constant *initialization functions* defining the set of initial states. $\mathcal{R}$ 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* $\langle L, \mathcal{T}, \mathcal{I}, \mathcal{R}\rangle$ *is* $\langle Q, Q_0, \Delta, \mathbf{error}\rangle$ *where*

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

A.1.2 Formal Semantics

The detailed semantics of the Murφ 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 $\langle L, \mathcal{T}, \mathcal{I}, \mathcal{R}\rangle$, 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 $\mathbf{N}$.

Variables are represented abstractly using locations. Each location corresponds to a subrange variable. $\mathcal{T}(l)$ 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 $\mathcal{T}(l)$.

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 $\Box$, **for** loop, and $\forall$ constructs. The set of all locations V is the union of L and I.

An *environment* is a function from V to $\mathbf{N}_\perp$, such that, the value in every location is consistent with type of the location. The environment that assigns $\perp$ to every element of V is written as $\langle\rangle$. The environment that assigns v to l and $\perp$ to all other elements of V is written as $(l \mapsto v)$. If s and s' are both environments, $s \bullet s'$ is the environment which assigns $s'(l)$ to each location l unless $s'(l) = \perp$, in which case it assigns $s(l)$ to the location. There is an additional **error** environment that results when an **error** statement is executed; for all environments s, $\textbf{error} \bullet s = \textbf{error}$. The set of all possible environments is denoted by *env*. The set of states is a subset of *env*, such that the value in every location in I is $\perp$. **tostate** converts a non-error environment to a state by assigning $\perp$ to every location in I, and also maps **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 kth element is '$l[k]$'. For example, with the declaration:

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

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

$$\{\text{`}P[0].\mathit{CacheState}\text{'}, \text{`}P[0].\mathit{CacheValue}\text{'}, \text{`}P[1].\mathit{CacheState}\text{'}, \text{`}P[1].\mathit{CacheValue}\text{'}\}.$$

More precisely, if **concat** is a function that concatenates a list of strings and numbers into a single string, a translation function *Tv* can be defined such that, given a variable with name x and its type α, it produces a set of locations.

$$\mathit{Tv}(\text{`}x\text{'}, \alpha) = \begin{cases} \bigcup_{1 \le i \le n} \mathit{Tv}(\textbf{concat}(\text{`}x\text{'}, \text{`}.f_i\text{'}), \alpha_i) & \text{if } \alpha = \textbf{record}\ f_1 : \alpha_1; \ldots; f_n : \alpha_n\ \textbf{end} \\ \bigcup_{i \in \beta} \mathit{Tv}(\textbf{concat}(\text{`}x\text{'}, \text{`[}\text{'}, i, \text{`]}\text{'}), \gamma) & \text{if } \alpha = \textbf{array}\ [\beta]\ \textbf{of}\ \gamma \\ \{\text{`}x\text{'}\} & \text{if } \alpha \text{ is a subrange.} \end{cases}$$

With a global variable gv of type α, the set of state locations L in the corresponding abstract transition program is $\mathit{Tv}(gv, \alpha)$. 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 $[\![d]\!] : env \rightarrow string$ is defined as:

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

where s is an environment and $[\![t]\!](s)$ 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 $[\![d]\!]$ 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 $[\![t]\!] : env \rightarrow \mathbf{N}_\perp$ is defined as:

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

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:

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

In this case, α must be a subrange.

Translating the Statements

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

Given an environment s and an assignment $d := t$, the meaning $[\![d := t]\!](s)$ is $s \bullet ([\![d]\!](s) \mapsto [\![t]\!](s))$. 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:

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

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: $[\![\textbf{error}]\!](s) = \textbf{error}$.

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

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

where lb and ub are the lower bound and upper bound of the type α, and we use $[\![sm_x(k)]\!](s)$ to denote $[\![sm]\!](s \bullet (\text{‘}x\text{’} \mapsto k))$.

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 $\Box$ index location and s_0 represents a state to which an individual transition function can be applied; the transition function extends s_0 by s to bind the $\Box$ index locations.

$$Tr(t,s) = \begin{cases} Tr(t_1,s) \cup Tr(t_2,s) & \text{if } t = t_1 \Box\, t_2 \\ \bigcup_{i\in\alpha} Tr(t', s \bullet (\text{`}x\text{'} \mapsto i)) & \text{if } t = \Box\ (x:\alpha)t' \\ \{\lambda s_0.\mathbf{tostate}(\llbracket sm \rrbracket(s_0 \bullet s))\} & \text{if } t \text{ is a statement } sm. \end{cases}$$

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 $\{f' \mid \exists f \in Tr(ir, \langle\rangle) : f'() = f(\langle\rangle)\}$.

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 $\Box$ construct generates a finite number of transition functions. $\Box$

A.2 Automorphism Induced by Scalarsets

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

Definition A.2 (permutation on values) *If α 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 α and nothing else.

Definition A.3 (permutation on strings/locations) *A permutation π_α on α can be extended to a permutation π_α : string $\rightarrow$ string, defined as:*

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

For example, with an array of processors:

type $\;\;$ *Pid* : **scalarset** (2)

var $\;\;$ *P* : **array** [*Pid*] **of record** *CacheState* : 0..3; *CacheValue* : 0..1 **end**

a permutation π_{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 **error** environment to itself.

Definition A.4 (permutation on environments/state) *A permutation π_α on α can be extended to a permutation π_α : env $\rightarrow$ env, defined as:*

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

The reader should note that by this definition, $\pi_\alpha(s)(\pi_\alpha(l)) = \pi_\alpha(s(l))$ if the type of l is α 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 α, every permutation π_α 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 $\langle L, \mathcal{T}, \mathcal{I}, \mathcal{R}\rangle$ 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 scalarset α 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 α, 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:

$$\begin{array}{rl} [\![P[Dir]]\!](s') & = s'(\text{`}P[2]\text{'}) = 1 \\ \pi_\alpha([\![P[Dir]]\!](s)) & = \pi_\alpha(s(\text{`}P[1]\text{'})) = \pi_\alpha(2) = 1 \end{array}$$

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,

$$\begin{array}{rcll} [\![d]\!](\pi_\alpha(s)) & = & \text{`}d\text{'} & \text{definition of } [\![d]\!] \\ & = & [\![d]\!](s) & \text{definition of } [\![d]\!] \end{array}$$

If $d = d'.f$,

$$\begin{array}{rcll} [\![d]\!](\pi_\alpha(s)) & = & \mathbf{concat}([\![d']\!](\pi_\alpha(s)), \text{`}.f\text{'}) & \text{definition of } [\![d'.f]\!] \\ & = & \mathbf{concat}(\pi_\alpha([\![d']\!](s)), \text{`}.f\text{'}) & \text{induction hypothesis} \\ & = & \pi_\alpha(\mathbf{concat}([\![d']\!](s), \text{`}.f\text{'})) & \text{permutation on a string/location} \\ & = & \pi_\alpha([\![d'.f]\!](s)) & \text{definition of } [\![d'.f]\!]. \end{array}$$

If $d = d'[t]$,

$$\begin{array}{rcll} [\![d]\!](\pi_\alpha(s)) & = & \mathbf{concat}([\![d']\!](\pi_\alpha(s)), \text{`['}, [\![t]\!](\pi_\alpha(s)), \text{`]'}) & \text{definition of } [\![d'[t]]\!] \\ & = & \left\{ \begin{array}{l} \mathbf{concat}(\pi_\alpha([\![d']\!](s)), \text{`['}, \pi_\alpha([\![t]\!](s)), \text{`]'}) \\ \quad \text{if the type of } t \text{ is } \alpha \\ \mathbf{concat}(\pi_\alpha([\![d']\!](s)), \text{`['}, [\![t]\!](s), \text{`]'}) \\ \quad \text{if the type of } t \text{ is not } \alpha \end{array} \right. & \text{induction hypothesis} \\ & = & \pi_\alpha(\mathbf{concat}([\![d']\!](s), \text{`['}, [\![t]\!](s), \text{`]'})) & \text{permutation on a string/location} \\ & = & \pi_\alpha([\![d'[t]]\!](s)) & \text{definition of } [\![d'[t]]\!]. \end{array}$$

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 α,

$$\begin{array}{rcll} [\![t]\!](\pi_\alpha(s)) & = & \pi_\alpha(s)([\![d]\!](\pi_\alpha(s))) & \text{definition of } [\![t]\!] \\ & = & \pi_\alpha(s)(\pi_\alpha([\![d]\!](s))) & \text{induction hypothesis} \\ & = & \pi_\alpha(s([\![d]\!](s))) & \text{permutation on an environment} \\ & = & \pi_\alpha([\![t]\!](s)) & \text{definition of } [\![t]\!]. \end{array}$$

If t is a variable reference d of type other than α,

$$\begin{array}{rcll} [\![t]\!](\pi_\alpha(s)) & = & \pi_\alpha(s)([\![d]\!](\pi_\alpha(s))) & \text{definition of } [\![t]\!] \\ & = & \pi_\alpha(s)(\pi_\alpha([\![d]\!](s))) & \text{induction hypothesis} \\ & = & s([\![d]\!](s)) & \text{permutation on an environment} \\ & = & [\![t]\!](s) & \text{definition of } [\![t]\!]. \end{array}$$

Finally, if $t = t_1 + t_2$, t_1 and t_2 cannot be of type α. By induction $[\![t_1]\!](\pi_\alpha(s)) = [\![t_1]\!](s)$ and $[\![t_2]\!](\pi_\alpha(s)) = [\![t_2]\!](s)$. So $[\![t_1 + t_2]\!](\pi_\alpha(s)) = [\![t_1]\!](\pi_\alpha(s)) + [\![t_2]\!](\pi_\alpha(s)) = [\![t_1]\!](s) + [\![t_2]\!](s) = [\![t_1 + t_2]\!](s)$.
The proof would be similar for other operations on terms. $\square$

Lemma A.3 (Boolean lemma) *If α is a scalarset, for every environment s and every Boolean expression e, $[\![e]\!](\pi_\alpha(s)) = [\![e]\!](s)$.*

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 α. Therefore, $[\![t_1]\!](\pi_\alpha(s)) = [\![t_1]\!](s)$ and $[\![t_2]\!](\pi_\alpha(s)) = [\![t_2]\!](s)$, which implies that $[\![t_1 > t_2]\!](\pi_\alpha(s)) = [\![t_1 > t_2]\!](s)$.

When e is of the form $t_1 = t_2$, if the type of one of the terms is α, the type of both t_1 and t_2 are α, therefore, $[\![t_1]\!](\pi_\alpha(s)) = \pi_\alpha([\![t_1]\!](s))$ and $[\![t_2]\!](\pi_\alpha(s)) = \pi_\alpha([\![t_2]\!](s))$. Since π_α is a bijection, the equivalence or inequivalence of t_1 and t_2 is preserved by π_α.

On the other hand, if neither of the terms is of type α, $[\![t_1]\!](\pi_\alpha(s)) = [\![t_1]\!](s)$ and $[\![t_2]\!](\pi_\alpha(s)) = [\![t_2]\!](s)$, which implies that $[\![t_1 = t_2]\!](\pi_\alpha(s)) = [\![t_1 = t_2]\!](s)$.

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

$$\begin{array}{lll}
[\![\forall(x:\alpha)e]\!](\pi_\alpha(s)) & = \bigwedge_{i\in\alpha}[\![e]\!](\pi_\alpha(s) \bullet (\text{`}x\text{'} \mapsto i)) & \text{definition of } \forall(x:\alpha) \\
& = \bigwedge_{i\in\alpha}[\![e]\!](\pi_\alpha(s) \bullet (\text{`}x\text{'} \mapsto \pi_\alpha(i))) & \text{commutativity and} \\
& & \quad\text{associativity of } \wedge \\
& = \bigwedge_{i\in\alpha}[\![e]\!](\pi_\alpha(s) \bullet (\pi_\alpha(\text{`}x\text{'}) \mapsto \pi_\alpha(i))) & \pi_\alpha(\text{`}x\text{'}) = \text{`}x\text{'} \\
& = \bigwedge_{i\in\alpha}[\![e]\!](\pi_\alpha(s \bullet (\text{`}x\text{'} \mapsto i))) & \text{permutation on} \\
& & \quad\text{an environment} \\
& = \bigwedge_{i\in\alpha}[\![e]\!](s \bullet (\text{`}x\text{'} \mapsto i)) & \text{induction hypothesis} \\
& = [\![\forall(x:\alpha)e]\!](s) & \text{definition of } \forall(x:\alpha).
\end{array}$$

□

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

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 $d := t$,

$$\begin{array}{lll}
[\![d := t]\!](\pi_\alpha(s)) & = \pi_\alpha(s) \bullet ([\![d]\!](\pi_\alpha(s)) \mapsto [\![t]\!](\pi_\alpha(s))) & \text{definition of } [\![d := t]\!] \\
& = \left\{\begin{array}{l} \pi_\alpha(s) \bullet (\pi_\alpha([\![d]\!](s)) \mapsto \pi_\alpha([\![t]\!](s))) \\ \quad\text{if the type of } t \text{ is } \alpha \\ \pi_\alpha(s) \bullet (\pi_\alpha([\![d]\!](s)) \mapsto [\![t]\!](s)) \\ \quad\text{if the type of } t \text{ is not } \alpha \end{array}\right. & \text{basic lemma} \\
& = \pi_\alpha(s \bullet ([\![d]\!](s) \mapsto [\![t]\!](s))) & \text{permutation on} \\
& & \quad\text{an environment} \\
& = \pi_\alpha([\![d := t]\!](s)) & \text{definition of } [\![d := t]\!].
\end{array}$$

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

For a statement sm of the form $sm_1; sm_2$, by the induction hypothesis,

$$\begin{aligned} [\![sm]\!](\pi_\alpha(s)) &= [\![sm_2]\!]([\![sm_1]\!](\pi_\alpha(s))) \\ &= [\![sm_2]\!](\pi_\alpha([\![sm_1]\!](s))) \\ &= \pi_\alpha([\![sm_2]\!]([\![sm_1]\!](s))) \\ &= \pi_\alpha([\![sm]\!](s)). \end{aligned}$$

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

$$\begin{aligned} [\![sm]\!](\pi_\alpha(s)) &= [\![sm'_v(ub)]\!]([\![sm'_v(ub-1)]\!](\ldots [\![sm'_v(lb)]\!](\pi_\alpha(s))\ldots)) \\ &\qquad \text{definition of } [\![sm]\!] \\ &= [\![sm'_v(\pi_\alpha(ub))]\!]([\![sm'_v(\pi_\alpha(ub-1))]\!](\ldots [\![sm'_v(\pi_\alpha(lb))]\!](\pi_\alpha(s))\ldots)) \\ &\qquad \text{property of } \textbf{for} \text{ statements indexed by a scalarset} \\ &= [\![sm'_v(\pi_\alpha(ub))]\!]([\![sm'_v(\pi_\alpha(ub-1))]\!](\ldots \pi_\alpha([\![sm'_v(lb)]\!](s)\ldots)) \\ &\qquad \text{induction hypothesis} \\ &= \ldots \\ &= \pi_\alpha([\![sm'_v(ub)]\!]([\![sm'_v(ub-1)]\!](\ldots [\![sm'_v(lb)]\!](s)\ldots))) \\ &\qquad \text{repeated application of the induction hypothesis} \\ &= \pi_\alpha([\![sm]\!](s)) \\ &\qquad \text{definition of } [\![sm]\!] \end{aligned}$$

□

When $f = \lambda s_0.\textbf{tostate}([\![sm]\!](s_0 \bullet s))$, we define $\pi_\alpha(f)$ to be $\lambda s_0.\textbf{tostate}([\![sm]\!](s_0 \bullet \pi_\alpha(s)))$. In other words, π_α is applied to the $\square$ index locations that are bound in f. From the definition of 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 α 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 $f = \lambda s_0.\textbf{tostate}([\![sm]\!](s_0 \bullet s))$ be a transition function in $\mathcal{R}$ and s_0 be a state.

$$\begin{aligned} \pi_\alpha(f)(\pi_\alpha(s_0)) &= \textbf{tostate}([\![sm]\!](\pi_\alpha(s_0) \bullet \pi_\alpha(s))) && \text{definition of } \pi_\alpha(f) \\ &= \textbf{tostate}([\![sm]\!](\pi_\alpha(s_0 \bullet s))) && \text{permutation on an environment} \\ &= \textbf{tostate}(\pi_\alpha([\![sm]\!](s_0 \bullet s))) && \text{statement lemma} \\ &= \pi_\alpha(f(s_0)) && \textbf{tostate} \text{ and } \pi_\alpha \text{ commute.} \end{aligned}$$

□

Lemma A.6 (initialization lemma) *$f \in \mathcal{I}$ always implies $\pi_\alpha(f) \in \mathcal{I}$.*

Proof. It follows from the definition of *Tr*. □

Now we can complete the proof of the soundness theorem:

Proof. (of the soundness theorem)

It is now quite simple to show that π_α meets the conditions for an automorphism. First, π_α on environments is a bijection.

The definition of an automorphism has three additional "if and only if" conditions. We prove implications, but since π_α^{-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 Δ.

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, $\pi_\alpha(\mathbf{error}) = \mathbf{error}$, by definition. □

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 $N_1 \geq N_\alpha$ and $N_2 \geq N_\alpha$.*

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 $N_1 \geq N_\alpha$ and $N_2 \geq N_\alpha$.

When we refer to an environment below, it means a member of $[V \rightarrow \mathbf{N}_\perp]$, unless otherwise specified. For $j = 1, 2$, s_j refers to an environment of P_j, and $\beta_j \subseteq \alpha_j$ is the set of values actually appearing in the environment, that is,

$$\{s(l) \mid l \text{ is a location of type } \alpha \text{ and } s(l) \neq \perp.\}.$$

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 α appears literally in the text, that is, in $\Box$ constructs of the form $\Box\ (x : \alpha)\ t$ and in quantified Boolean expressions. Specifically, for $j = 1, 2$,

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

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

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 α 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 α.

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

The General Canonicalizing Permutation

Given an environment s, we construct a canonicalizing permutation π_s on $\mathbf{N}_\perp$. First of all, we fix a total order on the locations of type α 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 α. We define π_s to map each environment s to the lexicographically minimum equivalent environment $\pi_s(s)$:

- $\pi_s(\perp) = \perp$,
- for $i > 0$, if $s(l_i) \neq \perp$ 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 α, $\pi_s(x)$ can be assigned in any way that makes π_s a permutation.

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

Lemma A.7 *ζ 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 α such that $\zeta(s')(l) \neq \zeta(s)(l)$, then s' cannot be equivalent to s because a permutation on α cannot modify $s(l)$ or $s'(l)$. If $\zeta(s')$ and $\zeta(s)$ agree on all non-α locations, let l be the least α location at which they disagree, according to the total order on α-locations used in the definition of ζ. 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 π will have $\pi(s)(l) \neq \pi(s')(l)$ or $\pi(s)(l') \neq \pi(s')(l')$, so s and s' cannot be equivalent. $\square$

Definition A.5 (β_1-β_2-permutation) *For $\beta_1, \beta_2 \subseteq \mathbf{N}$, a β_1-β_2-permutation π on $\mathbf{N}_\perp$ is a permutation such that $\pi(\perp) = \perp$ and for all $v \in \beta_1$, $\pi(v) \in \beta_2$, and* vice versa.

The reader should note that a β_1-β_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 ζ.

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 β_1-β_2-permutation π such that $\pi(s_1) = s_2$.*

Proof.

$\Leftarrow$ Suppose there exists a β_1-β_2-permutation π such that $\pi(s_1) = s_2$. Let π_{s_1} be the canonicalizing permutation for s_1, as defined above. Since $\zeta(s_1) = \pi_{s_1}(s_1) =$

$\pi_{s_1}(\pi^{-1}(s_2))$, $\zeta(s_1)$ is equivalent to s_2. Because $\zeta(s_1)$ is the lexicographical minimum, which is unique, the canonicalizing permutation π_{s_2} for s_2 is $\pi_{s_1} \circ \pi^{-1}$.

Therefore,

$$\begin{aligned}\zeta(s_2) &= \pi_{s_2}(s_2)\\ &= \pi_{s_1}(\pi^{-1}(s_2))\\ &= \pi_{s_1}(\pi^{-1}(\pi(s_1)))\\ &= \pi_{s_1}(s_1) = \zeta(s_1).\end{aligned}$$

$\Rightarrow$ By definition, there exist permutations π_{s_1} and π_{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 $\pi = \pi_{s_2}^{-1} \circ \pi_{s_1}$ is a β_1-β_2-permutation.

□

The next lemma asserts that for sufficiently large α_1 and α_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 α-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, π is a β_1-β_2-permutation such that $s_2 = \pi(s_1)$, and $s_1(l) = s_2(l) = \perp$ for some location l of type α.*

For every i_1 in α_1, there exists an $i_2 \in \alpha_2$ and a $(\beta_1 \cup \{i_1\})$-$(\beta_2 \cup \{i_2\})$-permutation π' 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 $s_2(l) = \perp$, $|\beta_2| < N_\alpha$, 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 π' 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 scalarset 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 α, and i and i' are values of type α.*

If $i \neq s_j(l)$ and $i' \neq s_j(l)$ for every location l of type α, then $[\![b]\!]_j(s_j \bullet (\text{‘}x\text{’} \mapsto i)) = [\![b]\!]_j(s_j \bullet (\text{‘}x\text{’} \mapsto i'))$.

Proof. Let π 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, $[\![b]\!]_j(s_j \bullet (\text{‘}x\text{’} \mapsto i)) = [\![b]\!]_j(\pi(s_j \bullet (\text{‘}x\text{’} \mapsto i))) = [\![b]\!]_j(s_j \bullet (\text{‘}x\text{’} \mapsto i')))$. $\square$

Lemma A.11 (Boolean saturation lemma) *Suppose s_1 and s_2 are environments, and π is a β_1-β_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 $t_1 > t_2$, $[\![b]\!]_1 = [\![b]\!]_2$, because α does not appear literally in the expressions. So the lemma follows from the Boolean lemma, where π is the permutation. For Boolean expressions of the form $\neg b'$ or $b_1 \wedge b_2$, 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 $j = 1, 2$, $[\![b]\!]_j(s_j) = \bigwedge_{i \in \alpha_j} [\![b']\!]_j(s_j \bullet (\text{‘}x\text{’} \mapsto i))$. This can be rewritten as $\bigwedge_{i \in \beta_j} [\![b']\!]_j(s_j \bullet (\text{‘}x\text{’} \mapsto i)) \wedge \bigwedge_{i \in \gamma_j} [\![b']\!]_j(s_j \bullet (\text{‘}x\text{’} \mapsto i))$, where $\gamma_j = \alpha_j - \beta_j$, the members of α_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 $\bigwedge_{i \in \beta_1} [\![b']\!]_1(s_1 \bullet (\text{‘}x\text{’} \mapsto i)) = \bigwedge_{i \in \beta_2} [\![b']\!]_2(s_2 \bullet (\text{‘}x\text{’} \mapsto i))$.

If $i \in \beta_1$, the set of non-$\perp$ values assigned to locations of type α in $s_1 \bullet (\text{‘}x\text{’} \mapsto i)$ remains β_1, Since π is a β_1-β_2-permutation, the set non-$\perp$ values assigned to locations of type α in $s_2 \bullet (\text{‘}x\text{’} \mapsto \pi(i))$ remains β_2. By the induction hypothesis, for all $i_1 \in \beta_1$, $[\![b']\!]_1(s_1 \bullet (\text{‘}x\text{’} \mapsto i_1)) = [\![b']\!]_2(s_2 \bullet (\text{‘}x\text{’} \mapsto \pi(i_1))$.

Hence, we have

$$\bigwedge_{i\in\beta_1} [\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto i)) = \bigwedge_{i\in\beta_1} [\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto \pi(i)))$$

Since the image of β_1 under π is β_2, this is equal to $\bigwedge_{i\in\beta_2}[\![b']\!]_2(s_2 \bullet (\text{`}x\text{'} \mapsto i))$.

Case 2: $i \in \gamma_j$

We show that $\bigwedge_{i\in\gamma_1}[\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto i)) = \bigwedge_{i\in\gamma_2}[\![b']\!]_2(s_2 \bullet (\text{`}x\text{'} \mapsto i))$.

Because the value of location 'x' in s_1 and s_2 is $\perp$, β_1 and β_2 are smaller than N_α. Since $\alpha_1 \geq N_\alpha$ and $\alpha_2 \geq N_\alpha$, neither γ_1 nor γ_2 is empty.

By the interchangeability lemma, $[\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto i))$ yields the same value for every $i \in \gamma_1$, so if γ_1 is nonempty, we may choose a particular value $i_1 \in \gamma_1$, knowing that

$$\bigwedge_{i\in\gamma_1} [\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto i)) = [\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto i_1)).$$

By the permutation extension lemma, there exists $i_2 \in \alpha_2 - \beta_2$ and a $(\beta_1 \cup \{i_1\})$-$(\beta_2 \cup \{i_2\})$-permutation π' such that

$$s_2 \bullet (\text{`}x\text{'} \mapsto i_2) = \pi'(s_1 \bullet (\text{`}x\text{'} \mapsto i_1)).$$

Now, by the induction hypothesis,

$$[\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto i_1)) = [\![b']\!]_2(\pi'(s_1 \bullet (\text{`}x\text{'} \mapsto i_1))) = [\![b']\!]_2(s_2 \bullet (\text{`}x\text{'} \mapsto i_2)).$$

By the interchangeability lemma, again,

$$[\![b']\!]_2(s_2 \bullet (\text{`}x\text{'} \mapsto i_2)) = \bigwedge_{i\in\gamma_2} [\![b']\!]_2(s_2 \bullet (\text{`}x\text{'} \mapsto i)).$$

Therefore,

$$\bigwedge_{i\in\gamma_1} [\![b']\!]_1(s_1 \bullet (\text{`}x\text{'} \mapsto i)) = \bigwedge_{i\in\gamma_2} [\![b']\!]_2(s_2 \bullet (\text{`}x\text{'} \mapsto i)).$$

Combining the β and γ 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 π is a β_1-β_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 α_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 α). $\square$

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, $f_1(s_1) = \textbf{tostate}([\![sm]\!]_1(s_1 \bullet s))$, 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 β_1-β_2-permutation π such that $s_2 = \pi(s_1)$, since $\zeta(s_1) = \zeta(s_2)$. Let β_1' be the set of non-$\perp$ α_2 values assigned to s. By repeated application of the permutation extension lemma, there exists a set $\beta_2' \subseteq \alpha_2$ and a $(\beta_1 \cup \beta_1')$-$(\beta_2 \cup \beta_2')$-permutation π' such that $s_2 \bullet \pi'(s) = \pi'(s_1 \bullet s)$.

Since $\pi'(s)$ maps the index locations to the values in α_2, by the definition of Tr_2, there exists $f_2 = \lambda s_0.\textbf{tostate}([\![sm]\!]_2(s_0 \bullet \pi'(s)))$ in $\mathcal{R}_2$. By the statement saturation lemma,

$$[\![sm]\!]_2(s_2 \bullet \pi'(s)) = [\![sm]\!]_2(\pi'(s_1 \bullet s)) = \pi'([\![sm]\!]_1(s_1 \bullet s)).$$

Because $\pi = \textbf{tostate} \circ \pi'$, we have $f_2(s_2) = \pi(f_1(s_1))$.

Similarly, if $f_1 \in \mathcal{I}_1$, there exists $f_2 \in \mathcal{I}_2$ such that $\zeta(f_1()) = \zeta(f_2())$. □

Now we can complete the proof for the data saturation theorem.

Proof. (of the data saturation theorem)

For $i = 1, 2$, suppose $\langle L, \mathcal{T}_i, \mathcal{I}_i, \mathcal{R}_i \rangle$ and $A_i = \langle Q_i, Q_{0i}, \Delta_i, \textbf{error}_i \rangle$ 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 $s_1 \neq \textbf{error}$, let $\beta_1 \subseteq \alpha_1$ be the set of non-$\perp$ values assigned to locations of type α in s_1. Since the number of locations of type α is N_α, $|\beta_1| \leq N_\alpha$. Since $|\alpha_2| \geq N_\alpha$, there exists $\beta_2 \subseteq \alpha_2$ of the same size as β_1 and a β_1-β_2-permutation π. The set of non-$\perp$ values assigned to locations of type α in $s_2 = \pi(s_1)$ is β_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 $f_1 \in \mathcal{R}_1$ 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 $f_2 \in \mathcal{R}_2$ 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 α is infinite. A system with a scalarset 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 scalarset as an index in the □ 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 scalarset, 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 $p' = (s, \{q_1^{a_1}, ..., q_k^{a_k}\})$ *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 $(s, \{q_1^{a_1}, ..., q_k^{a_k}\})$ 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 $(s, [\underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$ 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 $(s, q, \{q_1^{a_1}, ..., q_k^{a_k}\})$ is generated, and the state $(s, [q, \underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$ can be used as the canonical state to generate the successor after executing t.

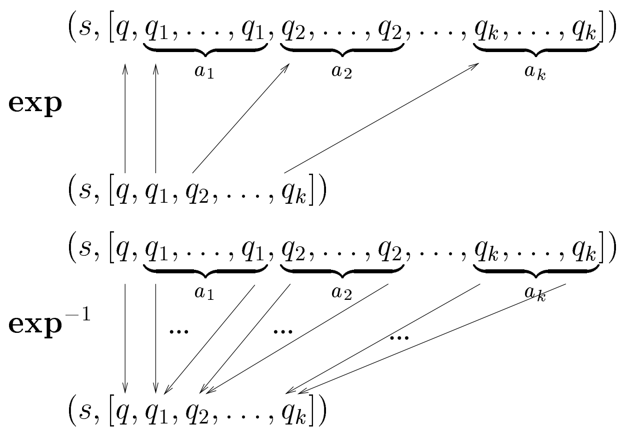
To simplify the discussion and proof, the remainder of this section only consider the third case, in which the transition rule t has an abstractable active component. We also denote the active representation $(s, q, \{q_1^{a_1}, ..., q_k^{a_k}\})$ as p', $(s, q, \{q_1, ..., q_k\})$ as p, the state $(s, [q, \underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$ 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* **exp** *maps the abstractable active component in the state* $c = (s, [q, q_1, \ldots, q_k])$ *to the active component in the state* $c' = (s, [q, \underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$*. It maps the abstractable component index* $j + 1$ *in the state c to the abstractable component index* $1 + a_1 + a_2 + ... + a_{j-1} + 1$ *in the state* c'*.*

The mapping $\mathbf{exp}^{-1}$ *maps the abstractable active component in the state* c' *to the active component in the state c. It maps the abstractable component index* $1 + a_1 + a_2 + ... + a_{j-1} + b$ *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:



$(s, [q, \underbrace{q_1, \ldots, q_1}_{a_1}, \underbrace{q_2, \ldots, q_2}_{a_2}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$
exp
$(s, [q, q_1, q_2, \ldots, q_k])$
$(s, [q, \underbrace{q_1, \ldots, q_1}_{a_1}, \underbrace{q_2, \ldots, q_2}_{a_2}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$
exp^{-1}
...
...
...
$(s, [q, q_1, q_2, \ldots, q_k])$

For example, given states $c_1' = (s, [q, q_1, q_1, q_1, q_2, q_2])$ and $c_1 = (s, [q, q_1, q_2])$, **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 $p' = (s, q, \{q_1^{a_1}, ..., q_k^{a_k}\})$ is used, and the state $c' = (s, [q, \underbrace{q_1, \ldots, q_1}_{a_1}, \ldots, \underbrace{q_k, \ldots, q_k}_{a_k}])$ 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. $(c' \bullet (\text{‘}x\text{’} \mapsto 1) \bullet c_0)$, and
2. $(c' \bullet (\text{‘}x\text{’} \mapsto 1) \bullet (\text{‘}y\text{’} \mapsto j) \bullet c_0)$,

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 ('x' $\mapsto 1$) is combined into c_0; and the two environments are denoted as $(c' \bullet c_0)$ and $(c' \bullet (\text{'}y\text{'} \mapsto j) \bullet c_0)$.

The proof of the Theorem 6.1 is organized in a similar list of lemmas as the corresponding proof for scalarset, with the permutation π_α replaced by **exp** and $\mathbf{exp}^{-1}$. Similar to the cases with permutations, **exp** and $\mathbf{exp}^{-1}$ on the abstractable component indices are extended to those on string/locations, and environments/state:

Definition A.8 **exp** *(and similarly for* $\mathbf{exp}^{-1}$*) on the abstractable component indices can be extended to* **exp** *on the string/locations, defined as:*

$$\mathbf{exp}(l) = \begin{cases} l & \text{if } l \text{ is not of the form } \mathbf{concat}(l', \text{'}.f\text{'}) \\ & \quad \text{or } \mathbf{concat}(l', \text{'['}, k, \text{']'}) \\ \mathbf{concat}(\mathbf{exp}(l'), \text{'}.f\text{'}) & \text{if } l = \mathbf{concat}(l', \text{'}.f\text{'}) \\ \mathbf{concat}(\mathbf{exp}(l'), \text{'['}, \mathbf{exp}(k), \text{']'}) & \text{if } l = \mathbf{concat}(l', \text{'['}, k, \text{']'}) \text{ and the array} \\ & \quad \text{corresponding to } l' \text{ is indexed} \\ & \quad \text{by the repetitiveID} \\ \mathbf{concat}(\mathbf{exp}(l'), \text{'['}, k, \text{']'}) & \text{if } l = \mathbf{concat}(l', \text{'['}, k, \text{']'}) \text{ and the array} \\ & \quad \text{corresponding to } l' \text{ is not indexed} \\ & \quad \text{by the repetitiveID} \end{cases}$$

Definition A.9 **exp** *and* $\mathbf{exp}^{-1}$ *on the abstractable component indices can be extended to* **exp** *and* $\mathbf{exp}^{-1}$ *on the environments, defined as:*

- $\mathbf{exp}(c \bullet c_0) = (c' \bullet c_0)$, and $\mathbf{exp}(c \bullet (\text{'}y\text{'} \mapsto j) \bullet c_0) = (c' \bullet (\text{'}y\text{'} \mapsto \mathbf{exp}(j)) \bullet c_0)$,
- $\mathbf{exp}^{-1}(c' \bullet c_0) = (c \bullet c_0)$, and $\mathbf{exp}^{-1}(c' \bullet (\text{'}y\text{'} \mapsto j) \bullet c_0) = (c \bullet (\text{'}y\text{'} \mapsto \mathbf{exp}^{-1}(j)) \bullet c_0)$.

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*

$$[\![d]\!](\mathbf{exp}(e_1)) = \mathbf{exp}([\![d]\!](e_1)), \text{ and } \mathbf{exp}^{-1}([\![d]\!](e_2)) = [\![d]\!](\mathbf{exp}^{-1}(e_2)).$$

If the type of t is the repetitiveID,

$$[\![t]\!](\mathbf{exp}(e_1)) = \mathbf{exp}([\![t]\!](e_1)), \text{ and } \mathbf{exp}^{-1}([\![t]\!](e_2)) = [\![t]\!](\mathbf{exp}^{-1}(e_2)).$$

Otherwise

$$[\![t]\!](\mathbf{exp}(e_1)) = [\![t]\!](e_1), \text{ and } [\![t]\!](e_2) = [\![t]\!](\mathbf{exp}^{-1}(e_2)).$$

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 $\mathbf{exp}^{-1}$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,

$$\begin{array}{rcll} [\![d]\!](\mathbf{exp}(e_1)) & = & \text{`}d\text{'} & \text{definition of } [\![d]\!] \\ & = & \mathbf{exp}(\text{`}d\text{'}) & \text{definition of } \mathbf{exp} \\ & = & \mathbf{exp}([\![d]\!](e_1)) & \text{definition of } [\![d]\!] \end{array}$$

If $d = d'.f$,

$$\begin{array}{rcll} [\![d]\!](\mathbf{exp}(e_1)) & = & \mathbf{concat}([\![d']\!](\mathbf{exp}(e_1)), \text{`}.f\text{'}) & \text{definition of } [\![d'.f]\!] \\ & = & \mathbf{concat}(\mathbf{exp}([\![d']\!](e_1)), \text{`}.f\text{'}) & \text{induction hypothesis} \\ & = & \mathbf{exp}(\mathbf{concat}([\![d']\!](e_1), \text{`}.f\text{'})) & \text{definition of } \mathbf{exp} \\ & = & \mathbf{exp}([\![d]\!](e_1)) & \text{definition of } [\![d'.f]\!]. \end{array}$$

If $d = d'[t]$,

$$\begin{array}{rcll} [\![d]\!](\mathbf{exp}(e_1)) & = & \mathbf{concat}([\![d']\!](\mathbf{exp}(e_1)), \text{`['}, [\![t]\!](\mathbf{exp}(e_1)), \text{`]'}) & \text{definition of } [\![d'[t]]\!] \\ & = & \left\{ \begin{array}{l} \mathbf{concat}(\mathbf{exp}([\![d']\!](e_1)), \text{`['}, \mathbf{exp}([\![t]\!](e_1)), \text{`]'})) \\ \quad \text{if the type of } t \text{ is the repetitiveID} \\ \mathbf{concat}(\mathbf{exp}([\![d']\!](e_1)), \text{`['}, [\![t]\!](e_1), \text{`]'})) \\ \quad \text{if the type of } t \text{ is not the repetitiveID} \end{array} \right. & \text{induction hypothesis} \\ & = & \mathbf{exp}(\mathbf{concat}([\![d']\!](e_1), \text{`['}, [\![t]\!](e_1), \text{`]'})) & \text{definition of } \mathbf{exp} \\ & = & \mathbf{exp}([\![d]\!](e_1)) & \text{definition of } [\![d'[t]]\!] \end{array}$$

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,

$$\begin{array}{rcll} [\![t]\!](\mathbf{exp}(e_1)) & = & \mathbf{exp}(e_1)([\![d]\!](\mathbf{exp}(e_1))) & \text{definition of } [\![t]\!] \\ & = & \mathbf{exp}(e_1)(\mathbf{exp}([\![d]\!](e_1))) & \text{induction hypothesis} \\ & = & \mathbf{exp}((e_1)([\![d]\!](e_1))) & \text{definition of } \mathbf{exp} \\ & = & \mathbf{exp}([\![t]\!](e_1)) & \text{definition of } [\![t]\!]. \end{array}$$

If t is a variable reference d of type other than α,

$$\begin{array}{rcll} [\![t]\!](\mathbf{exp}(e_1)) & = & \mathbf{exp}(e_1)([\![d]\!](\mathbf{exp}(e_1))) & \text{definition of } [\![t]\!] \\ & = & \mathbf{exp}(e_1)(\mathbf{exp}([\![d]\!](e_1))) & \text{induction hypothesis} \\ & = & (e_1)([\![d]\!](e_1)) & \text{definition of } \mathbf{exp} \\ & = & [\![t]\!](e_1) & \text{definition of } [\![t]\!]. \end{array}$$

Finally, if $t = t_1 + t_2$, the type of t_1 and t_2 cannot be the repetitiveID. By induction $[\![t_1]\!](\mathbf{exp}(e_1)) = [\![t_1]\!](e_1)$ and $[\![t_2]\!](\mathbf{exp}(e_1)) = [\![t_2]\!](e_1)$. So $[\![t_1 + t_2]\!](\mathbf{exp}(e_1)) = [\![t_1]\!](\mathbf{exp}(e_1)) + [\![t_2]\!](\mathbf{exp}(e_1)) = [\![t_1]\!](e_1) + [\![t_2]\!](e_1) = [\![t_1 + t_2]\!](e_1)$.

The proof would be similar for other operations on terms. □

Lemma A.15 (Boolean lemma) *Given a program with a repetitiveID, a Boolean expression e, and two environments e_2 and e_1 obtained by extending states c' and c, we have $[\![e]\!](\mathbf{exp}(e_1)) = [\![e]\!](e_1)$, and $[\![e]\!](e_2) = [\![e]\!](\mathbf{exp}^{-1}(e_2))$.*

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, $[\![t_1]\!](\mathbf{exp}(e_1)) = [\![t_1]\!](e_1)$ and $[\![t_2]\!](\mathbf{exp}(e_1)) = [\![t_2]\!](e_1)$, which implies that $[\![t_1 > t_2]\!](\mathbf{exp}(e_1)) = [\![t_1 > t_2]\!](e_1)$.

When e is of the form $t_1 = t_2$, and neither of the terms is of the repetitiveID type, we have $[\![t_1]\!](\mathbf{exp}(e_1)) = [\![t_1]\!](e_1)$ and $[\![t_2]\!](\mathbf{exp}(e_1)) = [\![t_2]\!](e_1)$, which implies that $[\![t_1 = t_2]\!](\mathbf{exp}(e_1)) = [\![t_1 = t_2]\!](e_1)$.

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 $e_1 = (c \bullet c_0)$, 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 $\mathbf{exp}(e_1) = (c' \bullet c_0)$, we have $[\![t_1 = t_2]\!](\mathbf{exp}(e_1)) = [\![t_1 = t_2]\!](e_1)$. On the other hand, if $e_1 = (c \bullet (\text{‘}y\text{’} \mapsto j) \bullet c_0)$, 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 $\mathbf{exp}(j)$ in c'. Since $\mathbf{exp}(e_1) = (c' \bullet (\text{‘}y\text{’} \mapsto \mathbf{exp}(j)) \bullet c_0)$, we have $[\![t_1 = t_2]\!](\mathbf{exp}(e_1)) = [\![t_1 = t_2]\!](e_1)$.

Similarly, the base cases for $\mathbf{exp}^{-1}$are also true.

For the induction part, the cases for $\neg$ and $\wedge$ 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 $(c \bullet c_0)$. Therefore, if the repetitiveID type is α, we have

$$\begin{array}{rcl}
[\![\forall(y:\alpha)e]\!](\mathbf{exp}(e_1)) & = & \bigwedge_{j\in\alpha}[\![e]\!](c' \bullet (\text{`}y\text{'} \mapsto j) \bullet c_0) \\
& & \quad \text{definition of } \forall(y:\alpha) \\
& = & [\![e]\!](c' \bullet (\text{`}y\text{'} \mapsto 1) \bullet c_0) \\
& & \wedge \bigwedge_{2\le j\le 1+a_1}[\![e]\!](c' \bullet (\text{`}y\text{'} \mapsto j) \bullet c_0) \\
& & \wedge \ldots \wedge \bigwedge_{2+a_1+\ldots a_{k-1}\le j\le 1+a_1+\ldots+a_k}[\![e]\!](c' \bullet (\text{`}y\text{'} \mapsto j) \bullet c_0) \\
& & \quad \text{expand} \\
& = & [\![e]\!](c \bullet (\text{`}y\text{'} \mapsto 1) \bullet c_0) \\
& & \wedge \bigwedge_{2\le j\le 1+a_1}[\![e]\!](c \bullet (\text{`}y\text{'} \mapsto 2) \bullet c_0) \\
& & \wedge \ldots \wedge \bigwedge_{2+a_1+\ldots a_{k-1}\le j\le 1+a_1+\ldots+a_k}[\![e]\!](c \bullet (\text{`}y\text{'} \mapsto k+1) \bullet c_0) \\
& & \quad \text{induction hypothesis: } [\![e]\!](e_2) = [\![e]\!](\mathbf{exp}^{-1}(e_2)) \\
& = & \bigwedge_{1\le j\le k+1}[\![e]\!](c \bullet (\text{`}y\text{'} \mapsto j) \bullet c_0) \\
& & \quad \text{simplify} \\
& = & [\![\forall(y:\alpha)e]\!](e_1) \\
& & \quad \text{definition of } \forall(y:\alpha).
\end{array}$$

And similarly for $[\![\forall(y:\alpha)e]\!](e_2)) = [\![\forall(y:\alpha)e]\!](\mathbf{exp}^{-1}(e_2))$. □

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 $e_2 = (c' \bullet c_0)$ and $e_1 = (c \bullet c_0)$, we have $[\![sm]\!](\mathbf{exp}(e_1)) = \mathbf{exp}([\![sm]\!](e_1))$, and $\mathbf{exp}^{-1}([\![sm]\!](e_2)) = [\![sm]\!](\mathbf{exp}^{-1}(e_2))$,*

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 $e_2 = (c' \bullet (\text{`}y\text{'} \mapsto j) \bullet c_0)$, and $e_1 = (c \bullet (\text{`}y\text{'} \mapsto \mathbf{exp}^{-1}(j)) \bullet c_0)$, the statement sm changes only the component state of the component k in e_2 and the component state of the component $\mathbf{exp}^{-1}(j)$ in e_1. Both components are changed to the same final component states.

This property is enforced by the restriction 5 on the repetitiveID. When environments $e_2 = (c' \bullet (\text{`}y\text{'} \mapsto j) \bullet c_0)$, and $e_1 = (c \bullet (\text{`}y\text{'} \mapsto \mathbf{exp}^{-1}(j)) \bullet c_0)$ 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 $e_2 = (c' \bullet c_0)$ or $e_1 = (c \bullet c_0)$, 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 $[\![sm]\!](\mathbf{exp}(e_1)) = \mathbf{exp}([\![sm]\!](e_1))$, and $\mathbf{exp}^{-1}([\![sm]\!](e_2)) = [\![sm]\!](\mathbf{exp}^{-1}(e_2))$,

For a statement sm of the form **if** b **then** sm' **end**, if $[\![b]\!](e_1)$ is true, then $[\![b]\!](\mathbf{exp}(e_1))$ is also true, by the Boolean lemma, so $[\![sm]\!](\mathbf{exp}(e_1)) = [\![sm']\!](\mathbf{exp}(e_1))$. By the induction hypothesis, $[\![sm']\!](\mathbf{exp}(e_1)) = \mathbf{exp}([\![sm']\!](e_1)) = \mathbf{exp}([\![sm]\!](e_1))$. If $[\![b]\!](e_1)$ is false then $[\![b]\!](\mathbf{exp}(e_1))$ is also false, so $[\![sm]\!](\mathbf{exp}(e_1)) = \mathbf{exp}(e_1) = \mathbf{exp}([\![sm]\!](e_1))$.

For a statement sm of the form $sm_1; sm_2$, by the induction hypothesis,

$$\begin{array}{rcl} [\![sm]\!](\mathbf{exp}(e_1)) & = & [\![sm_2]\!]([\![sm_1]\!](\mathbf{exp}(e_1))) \\ & = & [\![sm_2]\!](\mathbf{exp}([\![sm_1]\!](e_1))) \\ & = & \mathbf{exp}([\![sm_2]\!]([\![sm_1]\!](e_1))) \\ & = & \mathbf{exp}([\![sm]\!](e_1)). \end{array}$$

For a statement sm of the form **for** $(y : \alpha)$ **do** sm' **end**, where α is the repetitiveID type, we have

$$\begin{array}{rcl} [\![sm]\!](\mathbf{exp}(e_1)) & = & [\![sm'_y(1 + a_1 + ... + a_k)]\!]([\![sm'_y(1 + a_1 + ... + (a_k - 1))]\!](\ldots \\ & & \ldots [\![sm'_y(1)]\!](\mathbf{exp}(e_1)) \ldots)) \\ & & \text{definition of } [\![sm]\!] \end{array}$$

On the other hand, because of the auxiliary property, we also have:

$$\begin{aligned} [\![sm'_y(1)]\!](\mathbf{exp}(e_1)) &= \mathbf{exp}([\![sm'_y(1)]\!](e_1) \quad (A.1) \\ [\![sm'_y(1+a_1+...+a_j)]\!](\ldots[\![sm'_y(1+a_1+...+a_{j-1}+1)]\!](\mathbf{exp}(e_1)) & \\ &= \mathbf{exp}([\![sm'_j(1+j)]\!](e_1) \quad (A.2) \end{aligned}$$

Therefore,

$$\begin{aligned} [\![sm]\!](\mathbf{exp}(e_1)) &= [\![sm'_y(1+a_1+...+a_k)]\!]([\![sm'_y(1+a_1+...+(a_k-1))]\!](\ldots \\ &\qquad \ldots[\![sm'_y(1)]\!](\mathbf{exp}(e_1))\ldots)) \\ &\qquad \text{definition of } [\![sm]\!] \\ &= \mathbf{exp}([\![sm'_y(k+1)]\!](\ldots[\![sm'_y(1)]\!](e_1)\ldots))) \\ &\qquad \text{Eqn. (A.1) and (A.2)} \\ &= \mathbf{exp}([\![sm]\!](e_1)) \\ &\qquad \text{definition of } [\![sm]\!]. \end{aligned}$$

The same analysis can be repeated for $\mathbf{exp}^{-1}([\![sm]\!](e_2)) = [\![sm]\!](\mathbf{exp}^{-1}(e_2))$.

□

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 c' and c, we have $[\![t]\!](\mathbf{exp}(e_1)) = \mathbf{exp}([\![t]\!](e_1))$, and $\mathbf{exp}^{-1}([\![t]\!](e_2)) = [\![t]\!](\mathbf{exp}^{-1}(e_2))$.*

Proof. It follows trivially from the statement lemma. □

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.

```
----------------------------------------------------------------
-- Copyright (C) 1996 by the Board of Trustees of
-- Leland Stanford Junior University.
--
-- This description is provided to serve as an example of the
-- use of the Murphi description language and verifier, and as a
-- benchmark example for other verification efforts.
--
-- License to use, copy, modify, sell and/or distribute this
-- description and its documentation for any purpose is hereby
-- granted without royalty, subject to the following terms and
-- conditions:
--
-- 1.  The above copyright notice and this permission notice
--     must appear in all copies of this description.
--
-- 2.  The Murphi group at Stanford University must be
--     acknowledged in any publication describing work that
--     makes use of this example.
--
-- Nobody vouches for the accuracy or usefulness of this
-- description for any purpose.
----------------------------------------------------------------

----------------------------------------------------------------
-- Author:      C. Norris Ip
--
-- File:        adash.m
--
-- 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)  Simplification:
--              RAC is not used to store data; the cluster
--              acts as a single processor/global shared cache
--          3)  Separate network channels are used.
--              (request and reply)
--          4)  Aliases are used extensively.
--
-- Summary of result:
--          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 'bug1'.
--
--          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.
--
-- References:
--          1) Daniel Lenoski, James Laudon, Kourosh
--             Gharachorloo, Wlof-Dietrich Weber, Anoop Gupta,
--             John Hennessy, Mark Horowitz and Monica Lam.
--             The Stanford DASH Multiprocessor.
--             Computer, Vol 25 No 3, p.63-79, March 1992.
--             (available online)
--
--          2) Daniel Lenoski,
--             DASH Prototype System,
--             PhD thesis, Stanford University,
--             Chapter 3, 1992.
--             (available online)
--
-- Last Modified:        November 96
----------------------------------------------------------------

/*
Declarations

The number of clusters is defined by 'ProcCount'.  To simplify
the description, only some 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
--------------------------------------------------
Const
  HomeCount:    1;      -- number of homes.
  RemoteCount:  2;      -- number of remote nodes.
  ProcCount:    HomeCount+RemoteCount;
                        -- number of processors with cache.
  AddressCount: 1;      -- number of address at
                        -- each home cluster.
  ValueCount:   2;              -- number of data values.
  DirMax:       ProcCount-1;    -- number of directory entries
                                -- that can be kept (full map)
  ChanMax:      2*ProcCount*AddressCount*HomeCount;
                        -- buffer size in a single channel

  -- options
  bug1:         false;  -- options to introduce the
                        -- bug described or not
  nohome:       true;   -- options to switch off processors
                        -- at Home to simplify the protocol.

--------------------------------------------------
-- Type Declarations
--------------------------------------------------
Type
  Home:      1..HomeCount;
                -- can be changed to Scalarset(HomeCount);
  Remote:    (HomeCount+1)..(HomeCount+RemoteCount);
                -- can be changed to Scalarset(RemoteCount);
  Proc:      1..HomeCount+RemoteCount;
                -- can be changed to Union{Home, Remote};
  Address:   1..AddressCount;
                -- can be changed to Scalarset(AddressCount);
  Value:     1..ValueCount;
                -- can be changed to Scalarset(ValueCount);
  DirIndex:  0..DirMax-1;
  NodeCount: 0..ProcCount-1;

  -- the type of requests that go into the request network
  -- Cache-able and DMA operations
  RequestType:
    enum{
      RD_H,     -- basic op. -- read request to Home (RD)
      RD_RAC,   -- basic op. -- read request to Remote (RD)
      RDX_H,    -- basic op. -- read excl. req. to Home (RDX)
      RDX_RAC,  -- basic op. -- read excl. req. to Remote (RDX)
      INV,      -- basic op. -- invalidate request
      WB,       -- basic op. -- Explicit Write-back
      SHWB,     -- basic op. -- sharing writeback request
      DXFER,    -- basic op. -- dirty transfer request
      DRD_H,    -- DMA op.   -- read request to Home (DRD)
      DRD_RAC,  -- DMA op.   -- read request to Remote (DRD)
      DWR_H,    -- DMA op.   -- write request to Home (DWR)
      DWR_RAC,  -- DMA op.   -- write request to Remote (DWR)
      DUP       -- DMA op.   -- update request
      };
  -- the type of reply that go into the reply network
  -- Cache-able and DMA operations
  ReplyType:
    enum{
      ACK,   -- basic op.       -- read reply
             -- basic op.       -- read excl. reply
                                -- (inv count = 0)
             -- DMA op.         -- read reply
             -- DMA op.         -- write acknowledge
      NAK,   -- ANY kind of op. -- negative acknowledge
      IACK,  -- basic op.       -- read exclusive reply
                                -- (inv count !=0)
             -- DMA op.         -- acknowledge with update count
      SACK   -- basic op.       -- invalidate acknowledge
             -- basic op.       -- dirty transfer acknowledge
             -- DMA op.         -- update acknowledge
      };

  -- record for the requests in the network
  Request:
    Record
      Mtype: RequestType;
      Aux:   Proc;
      Addr:  Address;
      Value: Value;
    End;

  -- record for the reply in the network
  Reply:
    Record
      Mtype:    ReplyType;
      Aux:      Proc;
      Addr:     Address;
      InvCount: NodeCount;
      Value:    Value;
    End;

  -- States in the Remote Access Cache (RAC) :
  -- a) maintaining the state of currently outstanding requests,
  -- b) buffering replies from the network
  -- c) supplementing the functionality of caches
  RAC_State:
    enum{
      INVAL,   -- Invalid
      WRD,     -- basic op. -- waiting for a read reply
      WRDO,    -- basic op. -- waiting for a read reply
                            -- with ownership transfer
      WRDX,    -- basic op. -- waiting for a read excl. reply
      WINV,    -- basic op. -- waiting for invalidate ack.
      RRD,     -- basic op. -- invalidated read/read
                            -- with ownership request
      WDMAR,   -- DMA op.   -- waiting for a DMA read reply
      WUP,     -- DMA op.   -- waiting for update acknowledges
      WDMAW    -- DMA op.   -- waiting for a DMA write ack.
      };

  -- State of data in the cache
  CacheState:
    enum{
        Non_Locally_Cached,
        Locally_Shared,
        Locally_Exmod
        };

Type
  -- Directory Controller and the Memory
  -- a) Contains directory DRAM
  -- b) Forwards local requests to remotes,
  --    and replies to remote requests
  -- c) Responds to MPBUS with directory information
  -- d) Stores locks and lock queues
  HomeState:
    Record
      Mem: Array[Address] of Value;
      Dir: Array[Address] of
             Record
               State:  enum{ Uncached, Shared_Remote,
                             Dirty_Remote};
               SharedCount:  0..DirMax;
               Entries:      Array[DirIndex] of Proc;
             End;
    End;

  -- 1. Snoopy Caches
  -- 2. Pseudo-CPU (PCPU)
  --    a) Forwards remote CPU requests to local MPBUS
  --    b) Issues cache line invalidations and lock grants
  -- 3. Reply Controller (RC)
  --    a) Remote Access Cache (RAC) stores state of
  --       on-going memory requests and remote replies.
  --    b) Per processor invalidation counters (not implemented)
  --    c) RAC snoops on bus
  ProcState:
    Record
      Cache: Array[Home] of Array[Address] of
               Record
                 State: CacheState;
                 Value: Value;
               End;
      RAC:   Array[Home] of Array[Address] of
               Record
                 State:    RAC_State;
                 Value:    Value;
                 InvCount: NodeCount;
               End;
    End;

--------------------------------------------------
-- Variable Declarations
--
-- Clusters 0..HomeCount-1 : Clusters with distributed memory
-- Clusters HomeCount..ProcCount-1 :
--   Simplified Clusters without memory.
-- ReqNet : Virtual network with cluster-to-cluster channels
-- ReplyNet : Virtual network with cluster-to-cluster channels
--------------------------------------------------
Var
  ReqNet:   Array[Proc] of Array[Proc] of
              Record
                Count:    0..ChanMax;
                Messages: Array[0..ChanMax-1] of Request;
              End;
  ReplyNet: Array[Proc] of Array[Proc] of
              Record
                Count:    0..ChanMax;
                Messages: Array[0..ChanMax-1] of Reply;
              End;
  Procs:    Array[Proc] of ProcState;
  Homes:    Array[Home] of HomeState;

/*
Procedures
-- Directory handling functions
-- Request Network handling functions
-- Reply Network handling functions
-- Sending request
-- Sending Reply
-- Sending DMA Request
-- Sending DMA Reply
*/
--------------------------------------------------
-- Directory handling functions
-- a) set first entry in directory and clear other entries
-- b) add node to directory if it does not already exist
--------------------------------------------------
Procedure Set_Dir_1st_Entry( h : Home;
                             a : Address;
                             n : Proc);
Begin
  Undefine Homes[h].Dir[a].Entries;
  Homes[h].Dir[a].Entries[0] := n;
End;
```

```
Procedure Add_to_Dir_Entries( h : Home;
                              a : Address;
                              n : Proc);
Begin
  Alias SharedCount: Homes[h].Dir[a].SharedCount
  Do
    If ( Forall i:0..DirMax-1 Do
           ( i < SharedCount )
           -> ( Homes[h].Dir[a].Entries[i] != n )
         End )
    Then
      Homes[h].Dir[a].Entries[SharedCount] := n;
      SharedCount := SharedCount + 1;
    End;
  End;
End;

--------------------------------------------------
-- Request Network handling functions
-- a) A request is put into the end of a specific
--    channel connecting the source Src and the
--    destination Dst.
-- b) Request is only consumed at the head of the
--    queue, forming a FIFO ordered network channel.
--------------------------------------------------
Procedure Send_Req( t : RequestType;
                    Dst, Src, Aux : Proc;
                    Addr : Address;
                    Val : Value );
Begin
  Alias Count : ReqNet[Src][Dst].Count
  Do
    Assert ( Count != ChanMax ) "Request Channel is full";
    ReqNet[Src][Dst].Messages[Count].Mtype := t;
    ReqNet[Src][Dst].Messages[Count].Aux := Aux;
    ReqNet[Src][Dst].Messages[Count].Addr := Addr;
    ReqNet[Src][Dst].Messages[Count].Value := Val;
    Count := Count + 1;
  End;
End;

Procedure Consume_Request( Src, Dst: Proc);
Begin
  Alias Count : ReqNet[Src][Dst].Count
  Do
    For i: 0..ChanMax -2 Do
      ReqNet[Src][Dst].Messages[i]
      := ReqNet[Src][Dst].Messages[i+1];
    End;
    Undefine ReqNet[Src][Dst].Messages[Count-1];
    Count := Count - 1;
  End;
End;

--------------------------------------------------
-- Reply Network handling functions
-- a) A Reply is put into the end of a specific
--    channel connecting the source Src and the
--    destination Dst.
-- b) Reply is only consumed at the head of the queue,
--    forming a FIFO ordered network channel.
--------------------------------------------------
Procedure Send_Reply( t : ReplyType;
                      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";
    ReplyNet[Src][Dst].Messages[Count].Mtype := t;
    ReplyNet[Src][Dst].Messages[Count].Aux := Aux;
    ReplyNet[Src][Dst].Messages[Count].Addr := Addr;
    ReplyNet[Src][Dst].Messages[Count].Value := Val;
    ReplyNet[Src][Dst].Messages[Count].InvCount := InvCount;
    Count := Count + 1;
  End;
End;
```

```
Procedure Consume_Reply( Src, Dst : Proc);
Begin
  Alias Count : ReplyNet[Src][Dst].Count
  Do
    For i: 0..ChanMax -2 Do
      ReplyNet[Src][Dst].Messages[i]
       := ReplyNet[Src][Dst].Messages[i+1];
    End;
    Undefine ReplyNet[Src][Dst].Messages[Count-1];
    Count := Count - 1;
  End;
End;

--------------------------------------------------
-- Sending request
--------------------------------------------------

-- send read request to home cluster
Procedure Send_R_Req_H( Dst, Src : Proc;
                        Addr : Address);
Begin
  Send_Req(RD_H, Dst, Src, Undefined, Addr, Undefined);
End;

-- send read request to dirty remote block
--      Aux = where the request originally is from
Procedure Send_R_Req_RAC( Dst, Src, Aux : Proc;
                          Addr : Address);
Begin
  Send_Req(RD_RAC, Dst, Src, Aux, Addr, Undefined);
End;

-- send sharing writeback to home cluster
--      Aux = new sharing cluster
Procedure Send_SH_WB_Req( Dst, Src, Aux : Proc;
                          Addr : Address;
                          Val : Value);
Begin
  Send_Req(SHWB, Dst, Src, Aux, Addr, Val);
End;

-- send invalidate request to shared remote clusters
--      Aux = where the request originally is from
Procedure Send_Inv_Req( Dst, Src, Aux : Proc;
                        Addr : Address);
Begin
  Send_Req(INV, Dst, Src, Aux, Addr, Undefined);
End;

-- send read exclusive request
Procedure Send_R_Ex_Req_RAC( Dst, Src, Aux : Proc;
                             Addr : Address);
Begin
  Send_Req(RDX_RAC, Dst, Src, Aux, Addr, Undefined);
End;

-- send read exclusive local request
Procedure Send_R_Ex_Req_H( Dst, Src : Proc;
                           Addr : Address);
Begin
  Send_Req(RDX_H, Dst, Src, Undefined, Addr, Undefined);
End;

-- send dirty transfer request to home cluster
Procedure Send_Dirty_Transfer_Req( Dst, Src, Aux : Proc;
                                   Addr : Address);
Begin
  Send_Req(DXFER, Dst, Src, Aux, Addr, Undefined);
End;

-- Explicit writeback request
Procedure Send_WB_Req( Dst, Src : Proc;
                       Addr : Address;
                       Val : Value);
Begin
  Send_Req(WB, Dst, Src, Undefined, Addr, Val);
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_NAK( Dst, Src, Aux : Proc;
                    Addr : Address);
Begin
  Send_Reply(NAK, Dst, Src, Aux, Addr, Undefined, 0);
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 read 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(IACK, Dst, Src, Aux, Addr, Val, InvCount);
    End; --if;
  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, Src, Addr, Undefined, 0);
End;

--------------------------------------------------
-- Sending DMA Request
--------------------------------------------------
-- DMA Read request to home
Procedure Send_DMA_R_Req_H( Dst, Src : Proc;
                            Addr : Address);
Begin
  Send_Req(DRD_H, Dst, Src, Undefined, Addr, Undefined);
End;

-- DMA Read request to remote
Procedure Send_DMA_R_Req_RAC( Dst, Src, Aux : Proc;
                              Addr : Address);
Begin
  Send_Req(DRD_RAC, Dst, Src, Aux, Addr, Undefined);
End;

-- send DMA write request from Local
Procedure Send_DMA_W_Req_H( Dst, Src : Proc;
                            Addr : Address;
                            Val : Value);
Begin
  Send_Req(DWR_H, 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(IACK, Dst, Src, Src, Addr, Undefined, SharedCount);
End;

-- send DMA update request from Local
Procedure Send_DMA_Update_Req( Dst, Src, Aux : Proc;
                               Addr : Address;
                               Val : Value);
Begin
  Send_Req(DUP, 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(DWR_RAC, 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, 0);
End;

-- send DMA write Acknowledge
Procedure Send_DMA_W_Ack( Dst, Src, Aux : Proc;
                          Addr : Address);
Begin
  Send_Reply(ACK, Dst, Src, Aux, Addr, Undefined, 0);
End;

-- DMA update acknowledge to local
Procedure Send_DMA_Update_Ack( Dst, Src, Aux : Proc;
                               Addr : Address);
Begin
  Send_Reply(SACK, Dst, Src, Aux, Addr, Undefined, 0);
End;

/*
Rule Sets for fundamental memory access and DMA access
1) CPU Ia   : basic home memory requests from CPU
   CPU Ib   : DMA home memory requests from CPU
2) CPU IIa  : basic remote memory requests from CPU
   CPU IIb  : DMA remote memory requests from CPU
3) PCPU Ia  : handling basic requests to PCPU at memory cluster
   PCPU Ib  : handling DMA requests to PCPU at memory cluster
4) PCPU IIa : handling basic requests to PCPU in remote cluster
   PCPU IIb : handling DMA requests to PCPU in remote cluster
5) RCPU Iab : handling basic and DMA replies to RCPU in
              any cluster.
*/
```

```
/*
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 : Procs[n].RAC[h][a];
  Cache : Procs[n].Cache[h][a];
  Dir : Homes[h].Dir[a];
  Mem : Homes[h].Mem[a]
Do
  --------------------------------------------------
  -- Home CPU issue read request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "Local Memory Read Request"
    ( h = n ) & !nohome
  ==>
  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[0],h,h,a);
      Else
        -- get from memory
        Switch Cache.State
        Case Locally_Exmod:
          -- write back local master through snoopy protocol
          Cache.State := Locally_Shared;
          Mem := Cache.Value;
        Case Locally_Shared:
          -- other cache supply data
        Case Non_Locally_Cached:
          -- update cache
          Cache.State := Locally_Shared;
          Cache.Value := Mem;
        End; --switch;
      End; --switch;

    Case WRDO:
      -- merge
    Else
      -- WRD, RRD, WINV, WRDX, WDMAR, WUP, WDMAW.
      Assert ( RAC.State != WRD & RAC.State != RRD )
             "Funny RAC state at home cluster";
      -- conflict
    End;
  End; -- rule -- Local Memory Read Request

  --------------------------------------------------
  -- Home CPU issue read exclusive request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "Local Memory Read Exclusive Request"
    ( h = n ) & !nohome
  ==>
  Begin
    Switch RAC.State
    Case INVAL:
      -- no pending event
      Switch Dir.State
      Case Uncached:
        -- get from memory
        -- invalidate local copy through snoopy protocol
        Cache.State := Locally_Exmod;
        Cache.Value := Mem;
      Case Shared_Remote:
        -- get from memory
        Cache.State := Locally_Exmod;
        Cache.Value := Mem;
        -- invalidate all remote shared read copies
        RAC.State := WINV;
        RAC.InvCount := Dir.SharedCount;
        For i : NodeCount Do
          If ( i < RAC.InvCount ) Then
            Send_Inv_Req(Dir.Entries[i],h,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[0],h,h,a);
      End; --switch;

    Case WINV:
      -- other local processor already get the dirty copy
      -- other Cache supply data
      Assert ( Dir.State = Uncached ) "Inconsistent Directory";

    Case WRDX: -- merge
      Switch Dir.State
      Case Uncached:
        -- only arise in case of:
        -- remote cluster WB
      Case Shared_Remote:
        -- only arise in case of:
        -- remote cluster WB and RD
      Case Dirty_Remote:
        -- merge
      End; --switch;

    Case WRDO:
      -- conflict
    Else
      -- WRD, RRD, WDMAR, WUP, WDMAW.
      Assert ( RAC.State != WRD & RAC.State != RRD )
             "Funny RAC state at home cluster";
      -- conflict
    End; --switch;
  End; -- rule -- local memory read exclusive request

End; --alias; -- RAC, Cache, Dir, Mem
End; --ruleset; -- a
End; --ruleset; -- h
End; --ruleset; -- n
```

```
/*
CPU Ib

The rule sets non-deterministically issue requests for local
cache-able 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 : Proc Do
Ruleset h : Home Do
Ruleset a : Address Do
Alias
  RAC : Procs[n].RAC[h][a];
  Cache : Procs[n].Cache[h][a];
  Dir : Homes[h].Dir[a];
  Mem : Homes[h].Mem[a]
Do
  --------------------------------------------------
  -- Home CPU issue DMA read request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl up_mp.tbl
  --------------------------------------------------
  Rule "DMA Local Memory Read Request"
   ( h = n ) & !nohome
  ==>
  Begin
    Switch Dir.State
    Case Uncached:
      -- if RAC = INVAL => other cache supply data
      -- else           => conflict
    Case Shared_Remote:
      -- if RAC = INVAL => other cache supply data
      -- else           => conflict
    Case Dirty_Remote:
      Switch RAC.State
      Case INVAL:
        -- send DMA read request
        RAC.State := WDMAR;
        Send_DMA_R_Req_RAC(h,n,n,a);
      Case WINV:
        Error "inconsistent directory";
      Case WDMAR:
        -- merge
      Else
        -- WRD, WRDO, WRDX, RRD, WUP, WDMAW.
        Assert ( RAC.State != WRD & RAC.State != RRD )
               "Funny RAC state at home cluster";
        -- conflict
      End;
    End;
  End; -- rule -- DMA Local Memory Read Request

  --------------------------------------------------
  -- Home CPU issue DMA write request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Ruleset v:Value Do
  Rule "DMA Local Memory Write Request"
    ( h = n ) & !nohome
  ==>
  Begin
    Switch RAC.State
    Case INVAL:
      -- no pending event
      Switch Dir.State
      Case Uncached:
        Switch Cache.State
        Case Non_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 Non_Locally_Cached:
          -- no in local cache
        Case Locally_Shared:
          -- update local cache
          Cache.Value := v;
        Case Locally_Exmod:
          Error "Shared_remote with Exmod asserted";
        End; --switch;
        Mem := v;
        RAC.State := WUP;
        RAC.InvCount := Dir.SharedCount;
        For i:NodeCount Do
          If ( i < Dir.SharedCount ) Then
            Send_DMA_Update_Req(Dir.Entries[i], n,n,a,v);
          End; --if;
        End; --for;

      Case Dirty_Remote:
        -- update remote master copy
        RAC.State := WDMAW;
        RAC.Value := v;
        Send_DMA_W_Req_RAC(Dir.Entries[0], n,n,a,v);
      End; --switch;

    Case WINV:
      -- local master copy
      -- write local cache
      If ( Cache.State = Locally_Exmod ) Then
        Procs[n].Cache[h][a].Value := v;
      Elsif ( Cache.State = Locally_Shared ) Then
        Cache.Value := v;
        Mem := v;
      End; --if;

    Else
      -- WRD, WRDO, WRDX, RRD, WDMAR, WUP, WDMAW
      Assert ( RAC.State != WRD ) "WRD at home cluster";
    End; --switch;
  End; -- rule -- DMA write
  End; --ruleset; -- v

End; --alias; -- RAC, Cache, Dir, Mem
End; --ruleset; -- a
End; --ruleset; -- h
End; --ruleset; -- n
```

```
/*
CPU IIa

The rule sets non-deterministically issue requests for remote
cache-able memory.  The requests include read, exclusive read,
Explicit write back.

Three sets of rules:
  Rule "Remote Memory Read Request"
  Rule "Remote Memory Read Exclusive Request"
  Rule "Explicit Writeback request"

Issue messages:
         RD_H
         RDX_H
         WB
*/

Ruleset n : Proc Do
Ruleset h : Home Do
Ruleset a : Address Do
Alias
  RAC : Procs[n].RAC[h][a];
  Cache : Procs[n].Cache[h][a]
Do

  --------------------------------------------------
  -- remote CPU issues read request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.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:
      -- RAC take dirty ownership (simplified)
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted";
    Case WRD:
      -- merge
      Assert ( Cache.State = Non_Locally_Cached )
             "WRD with data Locally_Cached";
    Case WRDX:
      -- conflict
      Assert ( Cache.State != Locally_Exmod )
             "WRDX with data Locally_Exmod";
    Case RRD:
      -- conflict
      Assert ( Cache.State = Non_Locally_Cached )
             "WRDX with funny cache state";
    Case WRDO:
      Error "remote Cluster with WRDO";
    Case WDMAR:
      -- conflict
      Assert ( Cache.State = Non_Locally_Cached )
             "WRD with data Locally_Cached";
    Case WUP:
      -- conflict
    Case WDMAW:
      -- conflict
      Assert ( Cache.State != Locally_Exmod )
             "WRDX with data Locally_Exmod";
    End; --switch;
  End; -- rule -- Remote Memory Read Request

  --------------------------------------------------
  -- remote CPU issues read exclusive request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.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_Ex_Req_H(h,n,a);
      End;

    Case WINV:
      -- other cache supply data
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted";
    Case WRDX:
      -- merge
      Assert ( Cache.State != Locally_Exmod )
             "WRDX with Exmod asserted";
    Case WRD:
      -- conflict
      Assert ( Cache.State != Locally_Exmod )
             "WRD with Exmod asserted";
    Case RRD:
      -- conflict
      Assert ( Cache.State != Locally_Exmod )
             "RRD with Exmod asserted";
    Case WRDO:
      Error "remote cluster with WRDO";
    Case WDMAR:
      -- conflict
      Assert ( Cache.State != Locally_Exmod )
             "WRD with Exmod asserted";
    Case WUP:
      -- conflict
    Case WDMAW:
      -- conflict
    End; --switch;
  End; -- rule -- Remote Memory Read Exclusive Request

  --------------------------------------------------
  -- remote CPU issues explicit writeback request
  --------------------------------------------------
  Rule "Explicit Writeback request"
    ( h != n )
    & ( Cache.State = Locally_Exmod )
  ==>
  Begin
    If ( RAC.State = WINV ) Then
      -- retry later
    Else
      -- send request to home cluster
      Assert ( RAC.State = INVAL
             | RAC.State = WUP ) "Inconsistent Directory";
      Send_WB_Req(h,n,a,Cache.Value);
      Cache.State := Non_Locally_Cached;
      Undefine Cache.Value;
    End;
  End; -- rule -- Explicit Writeback request

End; --alias; -- RAC, Cache
End; --ruleset; -- a
End; --ruleset; -- h
End; --ruleset; -- n
```

```
/*
CPU IIb

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 : Procs[n].RAC[h][a];
  Cache : Procs[n].Cache[h][a]
Do

  --------------------------------------------------
  -- remote CPU issues DMA read request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "DMA Remote Memory Read Request"
    ( h != n )
  ==>
  Begin
    Switch Cache.State
    Case Non_Locally_Cached:
      Switch RAC.State
      Case INVAL:
        RAC.State := WDMAR;
        Send_DMA_R_Req_H(h,n,a);
      Case WINV:
        Error "Inconsistent RAC and Cache";
      Else -- WRD, WRDX, RRD, WRDO, WDMAR, WUP, WDMAW
        Assert ( RAC.State != WRDO ) "WRDO at remote cluster";
        -- conflict or merge
      End;
    Case Locally_Shared:
      Assert ( RAC.State != WINV
             & RAC.State != WRDO
             & RAC.State != WRD
             & RAC.State != RRD
             & RAC.State != WDMAR ) "Inconsistent directory";
    Case Locally_Exmod:
      Assert ( RAC.State != WDMAW
             & RAC.State != WRDO
             & RAC.State != WRD
             & RAC.State != WRDX
             & RAC.State != RRD
             & RAC.State != WDMAR ) "Inconsistent directory";
    End;
  End; -- rule -- DMA Remote Memory Read Request

  --------------------------------------------------
  -- remote CPU issues DMA write request
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  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 := WDMAW;
          RAC.Value := v;
          Send_DMA_W_Req_H(h,n,a,v);
        Case Non_Locally_Cached:
          -- send update request
          RAC.State := WDMAW;
          RAC.Value := v;
          Send_DMA_W_Req_H(h,n,a,v);
      End; --switch;

    Case WINV:
      -- write local cache
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted";
      Cache.Value := v;

    Else
      -- WRD, WRDO, WRDX, RRD, WDMAR, WUP, WDMAW
      Assert ( RAC.State != WRDO ) "WRDO at remote cluster";
      -- conflict or merge
    End;
  End; -- rule -- DMA Remote Memory Write Request
  End; --ruleset; -- v

End; --alias; -- RAC, Cache
End; --ruleset; -- a
End; --ruleset; -- h
End; --ruleset; -- n
```

```
/*
PCPU Ia

PCPU handles basic requests to Home for cache-able memory.

Five sets of rules:
  Rule "handle read request to home"
  Rule "handle read exclusive request to home"
  Rule "handle Sharing writeback request to home"
  Rule "handle dirty transfer request to home"
  Rule "handle writeback request to home"

Handle messages:
        RD_H
        RDX_H
        SHWB
        DXFER
        WB
*/

Ruleset Dst : Proc Do
Ruleset Src : Proc Do
Alias
  ReqChan : ReqNet[Src][Dst];
  Request : ReqNet[Src][Dst].Messages[0].Mtype;
  Addr : ReqNet[Src][Dst].Messages[0].Addr;
  Aux : ReqNet[Src][Dst].Messages[0].Aux
Do

  --------------------------------------------------
  -- PCPU handles Read request to home cluster
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "handle read request to home"
    ReqChan.Count > 0
    & Request = RD_H
  ==>
  Begin
  Alias
    RAC : Procs[Dst].RAC[Dst][Addr];
    Cache : Procs[Dst].Cache[Dst][Addr];
    Dir : Homes[Dst].Dir[Addr];
    Mem : Homes[Dst].Mem[Addr]
  Do
    Switch RAC.State
    Case WINV:
      -- cannot release copy
      Send_NAK(Src, Dst, Dst, Addr);
      Consume_Request(Src, Dst);
    Else
      -- INVAL, WRDO, WRDX, RRD, WDMAR, WUP, WDMAW.
      Assert ( RAC.State != WRD ) "WRD at home cluster";
      Switch Dir.State
      Case Uncached:
        -- no one has a copy. send copy to remote cluster
        If ( Cache.State = Locally_Exmod ) Then
          Cache.State := Locally_Shared;
          Mem := Cache.Value;
        End;
        Dir.State := Shared_Remote;
        Dir.SharedCount := 1;
        Set_Dir_1st_Entry(Dst, Addr, Src);
        Send_R_Reply(Src, Dst, Dst, Addr, Mem);
        Consume_Request(Src, Dst);
      Case Shared_Remote:
        -- some one has a shared copy.
        -- send copy to remote cluster
        Add_to_Dir_Entries(Dst, Addr, Src);
        Send_R_Reply(Src, Dst, Dst, Addr, Mem);
        Consume_Request(Src, Dst);
      Case Dirty_Remote:
        -- some one has a master copy.
        -- forward request to master cluster
        Send_R_Req_RAC(Dir.Entries[0], Dst, Src, Addr);
        Consume_Request(Src, Dst);
      End; --switch;
    End; --switch;
  End; -- alias : RAC, Cache, Dir, Mem
  End; -- rule -- handle read request to home

  --------------------------------------------------
  -- PCPU handles Read exclusive request to home cluster
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "handle read exclusive request to home"
    ReqChan.Count > 0
    & Request = RDX_H
  ==>
  Begin
  Alias
    RAC : Procs[Dst].RAC[Dst][Addr];
    Cache : Procs[Dst].Cache[Dst][Addr];
    Dir : Homes[Dst].Dir[Addr];
    Mem : Homes[Dst].Mem[Addr]
  Do
    Switch RAC.State
    Case WINV:
      -- cannot release copy
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted";
      Send_NAK(Src, Dst, Dst, Addr);
      Consume_Request(Src, Dst);
    Else
      -- INVAL, WRD, WRDO, WRDX, RRD, WDMAR, WUP, WDMAW.
      Assert ( RAC.State != WRD ) "WRD at home cluster";
      Switch Dir.State
      Case Uncached:
        -- no one has a copy. send copy to remote cluster
        If ( Cache.State = Locally_Exmod ) Then
          -- write back dirty copy
          Mem := Cache.Value;
        End;
        Cache.State := Non_Locally_Cached;
        Undefine Cache.Value;
        Dir.State := Dirty_Remote;
        Dir.SharedCount := 1;
        Set_Dir_1st_Entry(Dst, Addr, Src);
        Send_R_Ex_Reply(Src, Dst, Dst, Addr, Mem, 0);
        Consume_Request(Src, Dst);

      Case Shared_Remote:
        -- some one has a shared copy.
        -- send copy to remote cluster
        Cache.State := Non_Locally_Cached;
        Undefine Cache.Value;

        -- invalidate every shared copy
        if (!bug1) then
          -- ############# no 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
            If ( i < Dir.SharedCount ) Then
              Send_Inv_Req(Dir.Entries[i], Dst, Src, Addr);
            End;
          End;
          Send_R_Ex_Reply(Src, Dst, Dst, Addr, Mem,
                          Dir.SharedCount);
        else
          -- ############# bug I
          -- if you want protocol with bug
          -- rediscovered by Murphi which is a subtle bug
          -- that the designer once overlook and only
          -- 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
               & Dir.Entries[i] != Src )
            Then
              Send_Inv_Req(Dir.Entries[i], Dst, Src, Addr);
            End;
          End;
          If ( Exists i : DirIndex Do
               ( i<Dir.SharedCount
               & Dir.Entries[i] = Src )
             End )
        Then
          Send_R_Ex_Reply(Src, Dst, Dst, Addr, Mem,
                          Dir.SharedCount -1);
        Else
          Send_R_Ex_Reply(Src, Dst, Dst, Addr, Mem,
                          Dir.SharedCount);
        End;
      End; -- bug or no bug

      Dir.State := Dirty_Remote;
      Dir.SharedCount := 1;
      Set_Dir_1st_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(Dir.Entries[0], Dst, Src, Addr);
      Consume_Request(Src, Dst);
    End; --switch;
  End; --switch; -- RDX_H
End; -- alias : RAC, Cache, Dir, Mem
End; -- rule -- handle read exclusive request to home

--------------------------------------------------
-- PCPU handles sharing writeback request to home cluster
--------------------------------------------------
Rule "handle Sharing writeback request to home"
  ReqChan.Count > 0
  & Request = SHWB
==>
Begin
Alias
  v : ReqNet[Src][Dst].Messages[0].Value;
  Dir : Homes[Dst].Dir[Addr];
  Mem : Homes[Dst].Mem[Addr]
Do
  Assert ( Dir.State = Dirty_Remote )
         "Writeback to non dirty remote memory";
  Assert ( Dir.Entries[0] = Src ) "Writeback by non owner";
  Mem := v;
  Dir.State := Shared_Remote;
  Add_to_Dir_Entries(Dst, Addr, Aux);
  Consume_Request(Src, Dst);
End; -- alias : v, Dir, Mem
End; -- rule -- handle sharing writeback to home

--------------------------------------------------
-- PCPU handles dirty transfer request to home cluster
--------------------------------------------------
Rule "handle dirty transfer request to home"
  ReqChan.Count > 0
  & Request = DXFER
==>
Begin
Alias
  Dir : Homes[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_1st_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 handles writeback request to home cluster
  --------------------------------------------------
  Rule "handle writeback request to home"
    ReqChan.Count > 0
    & Request = WB
  ==>
  Begin
  Alias
    v : ReqNet[Src][Dst].Messages[0].Value;
    Dir : Homes[Dst].Dir[Addr];
    Mem : Homes[Dst].Mem[Addr]
  Do
    Assert ( Dir.State = Dirty_Remote )
           "Explicit writeback for non dirty remote";
    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

End; --alias; -- ReqChan, Request, Addr, Aux
End; --ruleset; -- Src
End; --ruleset; -- Dst
```

```
/*
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 messages:
        DRD_H
        DWR_H
*/
Ruleset Dst : Proc Do
Ruleset Src : Proc Do
Alias
  ReqChan : ReqNet[Src][Dst];
  Request : ReqNet[Src][Dst].Messages[0].Mtype;
  Addr : ReqNet[Src][Dst].Messages[0].Addr;
  Aux : ReqNet[Src][Dst].Messages[0].Aux
Do

  --------------------------------------------------
  -- PCPU handles DMA read request to home cluster
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "handle DMA read request to home"
    ReqChan.Count > 0
    & Request = DRD_H
  ==>
  Begin
  Alias
    RAC : Procs[Dst].RAC[Dst][Addr];
    Cache : Procs[Dst].Cache[Dst][Addr];
    Dir : Homes[Dst].Dir[Addr];
    Mem : Homes[Dst].Mem[Addr]
  Do
    Switch RAC.State
    Case WINV:
      -- cannot release copy
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted";
      Send_NAK(Src, Dst, Dst, Addr);
      Consume_Request(Src, Dst);
    Else -- INVAL, WRD, WRDO WRDX, RRD WDMAR, WUP, WDMAW.
      Switch Dir.State
      Case Uncached:
        If ( Cache.State = Locally_Exmod ) Then
          Send_DMA_R_Reply(Src, Dst, Dst, Addr, Cache.Value);
        Else
          Send_DMA_R_Reply(Src, Dst, Dst, Addr, Mem);
        End;
        Consume_Request(Src, Dst);
      Case Shared_Remote:
        Send_DMA_R_Reply(Src, Dst, Dst, Addr, Mem);
        Consume_Request(Src, Dst);
      Case Dirty_Remote:
        Send_DMA_R_Req_RAC(Dir.Entries[0], Dst, Src, Addr);
        Consume_Request(Src, Dst);
      End;
    End;
  End; -- alias : RAC, Cache, Dir, Mem
  End; -- rule -- handle DMA read request to home

  --------------------------------------------------
  -- PCPU handles DMA write request to home cluster
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "handle DMA write request to home"
    ReqChan.Count > 0
    & Request = DWR_H
  ==>
  Begin
  Alias
    v : ReqNet[Src][Dst].Messages[0].Value;
    RAC : Procs[Dst].RAC[Dst][Addr];
    Cache : Procs[Dst].Cache[Dst][Addr];
    Dir : Homes[Dst].Dir[Addr];
    Mem : Homes[Dst].Mem[Addr]
  Do
    Switch RAC.State
    Case WINV:
      -- cannot release copy
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted";
      Send_NAK(Src, Dst, Dst, Addr);
      Consume_Request(Src, Dst);
    Else -- INVAL, WRD, WRDO WRDX, RRD WDMAR, WUP, WDMAW.
      Switch Dir.State
      Case Uncached:
        -- update local data
        Switch Cache.State
        Case Locally_Exmod:
          Cache.Value := v;
        Case Locally_Shared:
          Cache.Value := v;
          Mem := v;
        Case Non_Locally_Cached:
          Mem := v;
        End;
        Send_DMA_W_Ack(Src, Dst, Dst, Addr);
        Consume_Request(Src, Dst);

      Case Shared_Remote:
        -- update local data and remote shared caches
        If ( Cache.State = Locally_Shared ) Then
          Cache.Value := v;
        End;
        Mem := v;
        For i:NodeCount Do
          If ( i < Dir.SharedCount ) Then
            Send_DMA_Update_Req(Dir.Entries[i], Dst, Src,
                                Addr, v);
          End;
        End;
        Send_DMA_Update_Count(Src, Dst, Addr, Dir.SharedCount);
        Consume_Request(Src, Dst);

      Case Dirty_Remote:
        -- update remote master copy
        Send_DMA_W_Req_RAC(Dir.Entries[0], Dst, Src, Addr, v);
        Consume_Request(Src, Dst);
      End;
    End; --switch
  End; -- alias : v, RAC, Cache, Dir, Mem
  End; -- rule -- handle DMA write request to home

End; --alias; -- ReqChan, Request, Addr, Aux
End; --ruleset; -- Src
End; --ruleset; -- Dst
```

```
/*
PCPU IIa

PCPU handles basic requests to non-home for cache-able 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
        RDX_RAC
*/

Ruleset Dst: Proc Do
Ruleset Src: Proc Do
Alias
  ReqChan: ReqNet[Src][Dst];
  Request: ReqNet[Src][Dst].Messages[0].Mtype;
  Addr: ReqNet[Src][Dst].Messages[0].Addr;
  Aux: ReqNet[Src][Dst].Messages[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"
    ReqChan.Count > 0
    & Request = RD_RAC
  ==>
  Begin
  Alias
    RAC: Procs[Dst].RAC[Src][Addr];
    Cache: Procs[Dst].Cache[Src][Addr]
  Do
    Switch RAC.State
    Case WINV:
      -- cannot release copy
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted.";
      Send_NAK(Aux, Dst, Src, Addr);
      Consume_Request(Src, Dst);

    Else
      -- INVAL, WRDO, WRD, WRDX, RRD, WDMAR, WUP, WDMAW
      Assert ( RAC.State != WRDO ) "WRDO at remote cluster";
      Switch Cache.State
      Case Locally_Exmod:
        -- has master copy; sharing write back data
        Cache.State := Locally_Shared;
        If ( Src = Aux ) Then
          -- read req from home cluster
          Send_R_Reply(Aux, Dst, Src, Addr, Cache.Value);
        Else
          -- read req from local cluster
          Send_R_Reply(Aux, Dst, Src, Addr,  Cache.Value);
          Send_SH_WB_Req(Src, Dst, Aux, Addr, Cache.Value);
        End;
        Consume_Request(Src, Dst);
      Else
        -- cannot release
        -- possible situation is :
        -- WRDX => still waiting for reply
        -- i.e. request message received before reply
        Send_NAK(Aux, Dst, Src, Addr);
        Consume_Request(Src, Dst);
      End; --switch;
    End; --switch;
  End; -- alias : RAC, Cache
  End; -- rule -- handle read request to remote cluster
```

```
--------------------------------------------------
-- PCPU handles invalidate request
--------------------------------------------------
Rule "handle Invalidate request to remote cluster"
  ReqChan.Count > 0
  & Request = INV
==>
Begin
Alias
  RAC: Procs[Dst].RAC[Src][Addr];
  Cache: Procs[Dst].Cache[Src][Addr]
Do
  Assert ( Dst != Src ) "Invalidation to Local Memory";
  If ( Dst = Aux )
  Then
    --------------------------------------------------
    -- PCPU handles invalidate request to initiating cluster
    --------------------------------------------------
    If ( Cache.State = Locally_Shared ) Then
      Cache.State := Non_Locally_Cached;
      Undefine Cache.Value;
    End;
    If ( RAC.State = WINV )Then
      -- have to wait for 1 less invalidation
      RAC.InvCount := RAC.InvCount -1;
    Else
      -- invalidation acknowledge come back before reply
      -- keep a count of how many acks so far
      RAC.InvCount := RAC.InvCount +1;
    End;
    If ( RAC.InvCount = 0
       & RAC.State = WINV )
    Then
      -- finished collecting all acknowledgments
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
    End;
    Consume_Request(Src, Dst);

  Else
    --------------------------------------------------
    -- PCPU handles invalidate request
    -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
    --                       up_mp.tbl
    --------------------------------------------------
    Switch RAC.State
    Case WINV:
      Error "invalidation cannot be for this copy!";
    Case WRD:
      RAC.State := RRD;
    Case RRD:
      -- nochange
    Else
      -- INVAL, WRDX, WRDO, RRD, WDMAR, WUP, WDMAW.
      Assert ( RAC.State != WRDO )
             "Inconsistent RAC with invalidation";
      Switch Cache.State
      Case Non_Locally_Cached:
        -- already flushed out of the cache
        -- 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 flushing";
      Case Locally_Shared:
        -- invalidate cache
        Cache.State := Non_Locally_Cached;
        Undefine Cache.Value;
      Case Locally_Exmod:
        Error "Invalidate request to master remote block.";
      End;
    End;
    Send_Inv_Ack(Aux, Dst, Src, Addr);
    Consume_Request(Src, Dst);
  End; -- if
End; -- alias : RAC, Cache
End; -- rule -- handle invalidate request
```

```
  --------------------------------------------------
  -- 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: Procs[Dst].RAC[Src][Addr];
    Cache: Procs[Dst].Cache[Src][Addr]
  Do
    Switch RAC.State
    Case WINV:
      -- cannot release copy
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted.";
      Send_NAK(Aux, Dst, Src, Addr);
      Consume_Request(Src, Dst);
    Else
      -- INVAL, WRDO, WRD, WRDX, RRD, WDMAR, WUP, WDMAW.
      Assert ( RAC.State !=  WRDO ) "WRDO in remote cluster";
      Switch Cache.State
      Case Locally_Exmod:
        -- has master copy; dirty transfer data
        If ( Src = Aux ) Then
          -- request from home cluster
          Send_R_Ex_Reply(Aux, Dst, Src, Addr, Cache.Value, 0);
        Else
          -- request from remote cluster
          Send_R_Ex_Reply(Aux, Dst, Src, Addr, Cache.Value, 1);
          Send_Dirty_Transfer_Req(Src, Dst, Aux, Addr);
        End;
        Cache.State := Non_Locally_Cached;
        Undefine Cache.Value;
        Consume_Request(Src, Dst);
      Else
        -- cannot release
        -- possible situation is :
        -- WRDX => still waiting for reply
        -- i.e. request message received before reply
        Send_NAK(Aux, Dst, Src, Addr);
        Consume_Request(Src, Dst);
      End; --switch;
    End; --switch;
  End; -- alias : RAC, Cache
  End; -- rule -- handle read excl. req. to remote cluster

End; --alias; -- ReqChan, Request, Addr, Aux
End; --ruleset; -- Src
End; --ruleset; -- Dst
```

```
/*
PCPU IIb

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"

Handle Messages:
        DUP
        DWR_RAC
        DRD_RAC
*/

Ruleset Dst: Proc Do
Ruleset Src: Proc Do
Alias
  ReqChan: ReqNet[Src][Dst];
  Request: ReqNet[Src][Dst].Messages[0].Mtype;
  Addr: ReqNet[Src][Dst].Messages[0].Addr;
  Aux: ReqNet[Src][Dst].Messages[0].Aux
Do

  --------------------------------------------------
  -- PCPU handles DMA Read request to remote cluster
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                       up_mp.tbl
  --------------------------------------------------
  Rule "handle DMA read request to remote cluster"
    ReqChan.Count > 0
    & Request = DRD_RAC
  ==>
  Begin
  Alias
    RAC: Procs[Dst].RAC[Src][Addr];
    Cache: Procs[Dst].Cache[Src][Addr]
  Do
    Switch RAC.State
    Case WINV:
      Send_NAK(Aux, Dst, Src, Addr);
      Consume_Request(Src, Dst);
    Else -- INVAL, WRD, WRDO, RRD, WRDX, WDMAR, WUP WDMAW.
      Assert ( RAC.State !=  WRDO ) "WRDO 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 :
        -- WRDX => still waiting for reply
        -- i.e. request message received before reply
        Send_NAK(Aux, Dst, Src, Addr);
        Consume_Request(Src, Dst);
      End;
    End;
  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].Messages[0].Value;
    RAC: Procs[Dst].RAC[Src][Addr];
    Cache: Procs[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;
    Case RRD:
      -- no change
    Else -- INVAL, WRDO, WRDX, WDMAR, WUP WDMAW.
      Assert ( RAC.State !=  WRDO ) "WRDO in remote cluster";
      Switch Cache.State
      Case Locally_Shared:
        Cache.Value := v;
      Case Non_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 flushing";
      Case Locally_Exmod:
        -- possible situation is :
        -- arise if the bug I is used...otherwise not.
        -- 1) cluster 1 get shared copy
        -- 2) home try DMA update
        -- 3) cluster 1 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; -- alias : v, RAC, Cache
  End; -- rule -- handle DMA update request to remote cluster

  --------------------------------------------------
  -- PCPU handles DMA write request to remote cluster
  -- confirmed with tables net.tbl, rc_1.tbl, rc_3.tbl,
  --                         up_mp.tbl
  --------------------------------------------------
  Rule "handle DMA write request to remote cluster"
    ReqChan.Count > 0
    & Request = DWR_RAC
  ==>
  Begin
  Alias
    v: ReqNet[Src][Dst].Messages[0].Value;
    RAC: Procs[Dst].RAC[Src][Addr];
    Cache: Procs[Dst].Cache[Src][Addr]
  Do
    Switch RAC.State
    Case WINV:
      -- cannot update copy
      Assert ( Cache.State = Locally_Exmod )
             "WINV with Exmod not asserted.";
      Send_NAK(Aux, Dst, Src, Addr);
      Consume_Request(Src, Dst);

    Else -- INVAL, WRD, WRDO, RRD, WRDX, WDMAR, WUP WDMAW.
      Switch Cache.State
      Case Locally_Exmod:
        Cache.Value := v;
        Send_DMA_W_Ack(Aux, Dst, Src, Addr);
        Consume_Request(Src, Dst);
      Else
        -- cannot update copy
        -- possible situation is :
        -- WRDX => still waiting for reply
        -- i.e. request message received before reply
        Send_NAK(Aux, Dst, Src, Addr);
        Consume_Request(Src, Dst);
      End;
    End;
  End; -- alias : v, RAC, Cache
  End; -- rule -- handle DMA update request to remote cluster

End; --alias; -- ReqChan, Request, Addr, Aux
End; --ruleset; -- Src
End; --ruleset; -- Dst
```

```
/*
RCPU Iab

RCPU handles cache-able and DMA acknowledgments and replies.

Four sets of rules:
  Rule "handle Acknowledgment"
  Rule "handle negative Acknowledgment"
  Rule "handle Indirect Acknowledgment"
  Rule "handle Supplementary Acknowledgment"

Handle messages:
   ACK
   NAK
   IACK
   SACK

-- confirmed with table net_up.tbl
-- except simplified in handling NAK on WDMAW
*/

Ruleset Dst : Proc Do
Ruleset Src : Proc Do
Alias
  ReplyChan : ReplyNet[Src][Dst];
  Reply : ReplyNet[Src][Dst].Messages[0].Mtype;
  Addr : 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
  ==>
  Begin
  Alias
    RAC : Procs[Dst].RAC[Aux][Addr];
    Cache : Procs[Dst].Cache[Aux][Addr]
  Do
    Switch RAC.State
    Case INVAL:
      -- no pending event
      Error "ACK in INVAL RAC state";
    Case WRD:
      -- pending read , i.e. read reply
      Cache.State := Locally_Shared;
      Cache.Value := v;
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
      Consume_Reply(Src, Dst);
    Case WRDO:
      -- pending read , i.e. read reply
      Cache.State := Locally_Shared;
      Cache.Value := v;
      Homes[Dst].Mem[Addr] := v;
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
      Consume_Reply(Src, Dst);
  Case WRDX:
    -- pending exclusive read, i.e. exclusive read reply
    -- no invalidation is required
    Cache.State := Locally_Exmod;
    Cache.Value := v;
    If ( Dst = Aux )
    Then
      Alias
        Dir : Homes[Dst].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 := INVAL;
    RAC.InvCount := 0;
    Consume_Reply(Src, Dst);
  Case RRD:
    -- invalidated pending event, ignore reply
    Undefine RAC;
    RAC.State := INVAL;
    RAC.InvCount := 0;
    Consume_Reply(Src, Dst);
  Case WDMAR:
    -- pending DMA read , i.e. read reply
    -- return data to cpu
    Undefine RAC;
    RAC.State := INVAL;
    RAC.InvCount := 0;
    Consume_Reply(Src, Dst);
  Case WDMAW:
    -- pending DMA write , i.e. write acknowledge
    Undefine RAC;
    RAC.State := INVAL;
    RAC.InvCount := 0;
    Consume_Reply(Src, Dst);
  Else
   -- WINV, WUP
   Error "ACK in funny RAC state";
  End; --switch;
End; -- alias : RAC, Cache, Dir, Mem
End; -- rule -- handle ACK

--------------------------------------------------
Rule "handle negative Acknowledgment"
  ReplyChan.Count > 0
  & Reply = NAK
  -- ANY kind of operation -- negative acknowledge
==>
Begin
Alias
  RAC : Procs[Dst].RAC[Aux][Addr];
  Cache : Procs[Dst].Cache[Aux][Addr];
Do
  Switch RAC.State
    Case INVAL:
      Error "NAK in INVAL RAC state";
    Case WRD:
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
      Consume_Reply(Src, Dst);
    Case WRDO:
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
      Consume_Reply(Src, Dst);
    Case WRDX:
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
      Consume_Reply(Src, Dst);
    Case RRD:
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
      Consume_Reply(Src, Dst);
    Case WDMAR:
      -- invalidated request, no data returned
      -- DMA read retry later
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
      Consume_Reply(Src, Dst);
    Case WDMAW:
      -- issues DMA write on bus again
      If ( Dst = Aux ) Then
        -- home cluster issuing DMA write
        Alias
          Dir : Homes[Dst].Dir[Addr];
          Mem : Homes[Dst].Mem[Addr]
        Do
          If ( Cache.State != Non_Locally_Cached ) Then
            Cache.Value := RAC.Value;
            Undefine RAC;
            RAC.State := INVAL;
            RAC.InvCount := 0;
          End;
          Switch Dir.State
          Case Uncached:
            Mem := RAC.Value;
            Undefine RAC;
            RAC.State := INVAL;
            RAC.InvCount := 0;
          Case Shared_Remote:
            Mem := RAC.Value;
            For i:NodeCount Do
              If ( i < Dir.SharedCount ) Then
                Send_DMA_Update_Req(Dir.Entries[i], Dst,
                                    Dst, Addr, RAC.Value);
              End;
            End;
            RAC.State := WUP;
            RAC.InvCount := Dir.SharedCount;
            Undefine RAC.Value;
            RAC.State := INVAL;
            RAC.InvCount := 0;
          Case Dirty_Remote:
            RAC.State := WDMAW;
            Send_DMA_W_Req_RAC(Dir.Entries[0], Dst,
                               Dst, Addr, RAC.Value);
          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 := INVAL;
          RAC.InvCount := 0;
        Else
          RAC.State := WDMAW;
          Send_DMA_W_Req_H(Aux, Dst, Addr, RAC.Value);
        End;
      End;
      Consume_Reply(Src, Dst);
    Else --switch
      -- WINV, WUP
      Error "NAK in funny RAC state";
  End; --switch;
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         -- acknowledge with update count
==>
Begin
Alias
  RAC : Procs[Dst].RAC[Aux][Addr];
  Cache : Procs[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 : Homes[Dst].Dir[Addr]
      Do
        Error "already sent invalidations to copy ??";
        Dir.State := Uncached;
        Dir.SharedCount := 0;
        Undefine Dir.Entries;
      End; -- alias : Dir
    End;
    RAC.InvCount := ICount - RAC.InvCount;
    RAC.State := WINV;
    If ( RAC.InvCount = 0 ) Then
      -- all invalidation acks received
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
    End;
    Consume_Reply(Src, Dst);
  Case WDMAW:
    RAC.State := WUP;
    RAC.InvCount := ICount - RAC.InvCount;
    If ( RAC.InvCount = 0 ) Then
      Undefine RAC;
      RAC.State := INVAL;
      RAC.InvCount := 0;
    End;
    Consume_Reply(Src, Dst);
  Else
    -- WRD, WRDO, RRD, WINV, 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 : Procs[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 acknowledgments
      RAC.InvCount := RAC.InvCount -1;
      If  ( RAC.InvCount = 0 ) Then
        -- get all invalidation acks
        Undefine RAC;
        RAC.State := INVAL;
        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 := INVAL;
        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;
  End; -- alias : RAC
  End; -- rule -- SACK

End; --alias; -- ReplyChan, Reply, Addr, Aux, v, ICount
End; --ruleset; -- Src
End; --ruleset; -- Dst
```

```
/*
-- 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
    For a : Address Do
      Homes[h].Dir[a].State := Uncached;
      Homes[h].Dir[a].SharedCount := 0;
      Homes[h].Mem[a] := v;
      Undefine Homes[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 := INVAL;
      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
      ReqNet[Src][Dst].Count := 0;
      Undefine ReqNet[Src][Dst].Messages;
      ReplyNet[Src][Dst].Count := 0;
      Undefine ReplyNet[Src][Dst].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 "Condition for existence of master copy of data"
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 != WRDO )
  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
   ! ( n1 != 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
    ( Homes[h].Dir[a].State != Uncached
    | Homes[h].Dir[a].SharedCount = 0 )
    &
    ( Forall i:0..DirMax-1 Do
        i >= Homes[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 = INVAL
      ->
      ( Isundefined(Procs[n].RAC[h][a].Value)
      & Procs[n].RAC[h][a].InvCount = 0 ) )
  End
  End
  End; -- Irrelevant data is set to zero
```

```
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
    & 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 )
  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 = INVAL
    | Procs[n].RAC[h][a].State = WINV )
  End
  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-1 Do
              ( i < ReqNet[h][n].Count
              & ReqNet[h][n].Messages[i].Mtype = INV )
            End )
      & ! ( Exists i:0..ChanMax-1 Do
              ( i < ReqNet[n][h].Count
              & ReqNet[n][h].Messages[i].Mtype = SHWB )
            End
            |
            Exists m : Proc Do
            Exists i : 0..ChanMax-1 Do
              ( i < ReqNet[m][h].Count
              & ReqNet[m][h].Messages[i].Mtype = SHWB
              & ReqNet[m][h].Messages[i].Aux = n)
            End
            End )
      & ! ( Exists i:0..ChanMax-1 Do
              ( i < ReplyNet[n][h].Count
              & ReplyNet[n][h].Messages[i].Mtype = ACK )
            End
            |
            Exists m:Proc Do
            Exists i:0..ChanMax-1 Do
              ( i < ReplyNet[n][h].Count
              & ReplyNet[m][h].Messages[i].Mtype = ACK
              & ReplyNet[m][h].Messages[i].Aux = n)
            End
            End )
      & Procs[n].RAC[h][a].State != WDMAW
      & ! ( Exists i:0..ChanMax-1 Do
                ( i<ReqNet[h][n].Count
                & ReqNet[h][n].Messages[i].Mtype = DUP )
            End ) )
  End
  End
  End; -- Consistency of data
```

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

Bibliography

[ACH$^+$92] R. Alur, C. Couroubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. *3rd International Conference on Concurrency Theory*, pages 340–354, 1992.

[AFdR80] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.

[AK86] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

[AKS83] S. Aggarwal, R.P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. *Protocol Specification, Testing, and Verification, III*, pages 19–34, 1983.

[BBG$^+$93] A. Bouajjani, S. Bensalem, S. Graf, C. Loiseaux, and J. Sifakis. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1993.

[BBLS92] A. Bouajjani, S. Bensalem, C. Loiseaux, and J. Sifakis. Property preserving simulations. *4th Workshop on Computer-Aided Verification*, June 1992.

[BCM$^+$90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *5th IEEE Symposium on Logic in Computer Science*, 1990.

[BD87] S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–24, 1987.

[BFH90] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation. *2nd Workshop on Computer-Aided Verification*, 1990.

[BFH$^+$92] A. Bouajjani, J.C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[Bri86] Ed Brinksma. A tutorial on Lotos. *Protocol Specification, Testing, and Verification V*, pages 171–194, 1986.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BSV94] F. Balarin and A.L. Sangiovanni-Vincentelli. On the automatic computation of network invariants. *6th International Conference on Computer-Aided Verification*, June 1994.

[BWHM86] J. Billing, M.C. Wilbur-Ham, and M.Y.Bearman. Automated protocol verification. *Protocol Specification, Testing, and Verification, V*, 1986.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. *Automatic Verification Methods for Finite State Systems*, 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4th ACM Symposium on Principles of Programming Languages*, pages 269–282, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. Technical report, Ecole Polytechnique, Laboratoire d'Informatique, 1992.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Workshop on Logics of Programs*, pages 52–71, 1981.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, August 1996.

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

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *5th International Conference on Computer-Aided Verification*, June 1993.

[CG70] D.G. Corneil and C.C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the Association for Computing Machinery*, 17(1), January 1970.

[CG87] E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. *6th Annual ACM Symposium on Principle of Distributed Computing*, pages 294–303, 1987.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. *6th International Conference on Concurrency Theory*, 1995.

[CGL91] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Symposium on Principles of Programming Languages*, 1991.

[CK80] D.G. Corneil and D.G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM Journal of Computing*, 9, May 1980.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design — a Foundation.* Addison-Wesley, 1988.

[CMCHG96] E.M. Clarke, K.L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. *8th International Conference on Computer Aided Verification*, pages 419–422, July/August 1996.

[Cou81] P. Cousot. Semantic foundations of program analysis. *Program Flow Analysis: Theory and Applications*, pages 303–342, 1981.

[Cou90] P. Cousot. Methods and logics for proving programs. *Form Models and Semantics*, B:843–993, 1990.

[CRL96] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. *ACM SIGPLAN '96: Programming Language Design and Implementation*, May 1996.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[DGG94] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: abstractions preserving ∀CTL*, ∃CTL* and CTL*. *Programming Concepts, Methods and Calculi (PROCOMET)*, pages 561–581, 1994.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Dij85] E.J. Dijkstra. Invariance and nondeterminacy. In *Mathematical Logic and Programming Languages.* Prentice-Hall, 1985.

[Dil95] David L. Dill. Protocols used in class CS355. *Stanford University*, Spring 1994-1995.

[Dil96] D.L. Dill. The Murφ verification system. *8th International Conference on Computer Aided Verification*, pages 390–393, July/August 1996.

[DPN93] David L. Dill, Seungjoon Park, and Andreas Nowatzyk. Formal specification of abstract memory models. *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 38–52, March 1993.

[Ebe88] Carl Ebeling. GeminiII: A second generation layout validation program. *IEEE/ACM International Conference on Computer-Aided Design*, 1988.

[EH83] E. Allen Emerson and Joseph Y. Halpern. 'sometime' and 'not never' revisited: on branching versus linear time. *10th ACM Symposium on Principles of programming languages*, 1983.

[Eme96] E. Allen Emerson, editor. *Formal Methods in System Design, Special Issue on Symmetry in Automatic Verification*, volume 9(1/2). Kluwer Academic Publishers, August 1996.

[ES93] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *5th International Conference on Computer-Aided Verification*, June 1993.

[ES95] E.A. Emerson and A.P. Sistla. Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. *7th International Conference on Computer-Aided Verification*, 1995.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, August 1996.

[Fer93] J.C. Fernandez. Abstract interpretation and verification of reactive systems. *3rd International Workshop Proceedings on Static Analysis*, pages 60–71, 1993.

[Flo67] R. Floyd. Assigning meaning to programs. *Symposium on Applied Mathematics 19, Mathematical Aspects of Computer Science*, pages 19–32, 1967.

[GL93a] S. Graf and C. Loiseaux. Program verification using compositional abstraction. *TAPSOFT 93 joint conference CAAP/FASE*, 1993.

[GL93b] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. *5th International Conference on Computer-Aided Verification*, April 1993.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. *2nd Workshop on Computer Aided Verification*, pages 176–185, June 1990.

[God95] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-explosion Problem.* PhD thesis, Universite de Liege, Faculté des Sciences Appliquées, 1995.

[Gor85] Mike Gordon. HOL: A machine oriented formulation of higher order logic. Technical report, University of Cambridge Computer Laboratory, 1985.

[Gou88] Ronald Gould. *Graph Theory,* section 9.2, pages 257–267. The Benjamin/Cummings Publishing Company, Inc., 1988.

[Gra94] Susanne Graf. Verification of a distributed cache memory by using abstractions. *6th International Conference on Computer-Aided Verification*, pages 207–219, 1994.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of Association for Computing Machinery*, 39(3):675–735, 1992.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, May 1994.

[HB95] Ramin Hojati and Robert K. Brayton. Automatic datapath abstraction in hardware systems. *7th International Conference on Computer-Aided Verification*, 1995.

[HD93a] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. *5th International Conference on Computer-Aided Verification*, June 1993.

[HD93b] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. *30th Design Automation Conference*, pages 266–271, 1993.

[HGP92] G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. *International Symposium on Protocol Specification, Testing, and Verification*, pages 349–363, June 1992.

[HHK96] R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. *8th International Conference on Computer Aided Verification*, pages 423–427, July/August 1996.

[HJJJ84] Peter Huber, Ame M. Jensen, Leif O. Jepsen, and Kurt Jensen. Towards reachability trees for high-level petri nets. *Advances on Petri Nets*, pages 215–233, 1984.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hof95] Robert D. Hof. Intel takes a bullet — and barely breaks stride. *Business Week*, pages 38–39, January 1995.

[Hol85] G.J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(12):2413–2434, 1985.

[Hol87] Gerard J. Holzmann. *Automated Protocol Validation in Argos, Assertion Proving and Scatter Searching.* Computer Science Press, 1987.

[Hol91a] Gerard J. Holzmann. *Design and Validation of Computer Protocols*, chapter 13. Prentice-Hall, 1991.

[Hol91b] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[HP94] G.J. Holzmann and D. Peled. An improvement in formal verification. *7th International Conference on Formal Description Techniques*, pages 177–194, 1994.

[HP96] G.J. Holzmann and D. Peled. The state of SPIN. *8th International Conference on Computer Aided Verification*, pages 385–389, July/August 1996.

[Hu95] Alan John Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*, chapter 4 on 'BDD Blow-Up Representing Sets of States', pages 41–49. Stanford University, December 1995. Ph.D. Thesis.

[ID93a] C. Norris Ip and David L. Dill. Better verification through symmetry. *11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, April 1993.

[ID93b] C. Norris Ip and David L. Dill. Efficient verification of symmetric concurrent systems. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, October 1993.

[ID96a] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.

[ID96b] C. Norris Ip and David L. Dill. State reduction using reversible rules. *33rd Design Automation Conference*, pages 564–567, June 1996.

[ID96c] C. Norris Ip and David L. Dill. Verifying systems with replicated components in Murφ. *8th International Conference on Computer-Aided Verification*, pages 147–158, 1996.

[Jen96] Kurt Jensen. Condensed state spaces for symmetrical coloured petri nets. *Formal Methods in System Design*, 9(1/2):7–40, August 1996.

[JJ91] C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-fly. *3rd Workshop on Computer-Aided Verification*, July 1991.

[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 Design*, 5:227–244, 1994.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:2:125–143, 1977.

[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 657–668, 1983.

[LG81] G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[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 Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH multiprocessor. *Computer*, 25(3), 1992.

[Lon93] D.E. Long. *Model Checking, Abstraction and Compositional Verification.* PhD thesis, Carnegie Mellon University, July 1993.

[LSU89] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1989.

[Lub84] Boris D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs I. *Acta Informatica*, 21:125–169, 1984.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. *24th Annual ACM Symposium on the Theory of Computing*, pages 264–274, 1992.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1), 1991.

[Mit88] Hari Ballabh Mittal. A fast backtrack algorithm for graph isomorphism. *Information Processing Letters 29*, pages 105–110, 1988.

[MK96] Hillel Miller and Shmuel Katz. Saving space by fully exploiting invisible transitions. *8th International Conference on Computer-Aided Verification*, pages 336–347, 1996.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. *The Correctness Problem in Computer Science, International Lecture Series in Computer Science*, 1981.

[MP83] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. *10th Annual ACM Symposium on Principles of Programming Languages*, pages 141–154, 1983.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4:257–289, 1984.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, 1976.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4:199–223, 1982.

[Ora88] F. Orava. Formal semantics of SDL specifications. *Protocol Specification, Testing, and Verification VIII*, 1988.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. *International Conference on Automated Deduction*, pages 748–752, 1992.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Par94] Seungjoon Park. *Computer assisted analysis of multiprocessor memory systems,* section 4.3 on 'Verification Using A Finite State Method'. PhD thesis, Stanford University, June 1994.

[PD93a] F. Pong and M. Dubois. Correctness of a directory-based cache coherence protocol: Early experience. *5th Symposium on Parallel Distributed Processing*, pages 37–44, 1993.

[PD93b] F. Pong and M. Dubois. The verification of cache coherence protocols. *5th Symposium on Parallel Algorithm and Architecture*, pages 11–20, 1993.

[PD95a] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). *7th ACM Symposium on Parallel Algorithms and Architectures*, pages 34–41, 1995.

[PD95b] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):773–87, 1995.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model checking. *6th International Conference on Computer Aided Verification*, pages 377–390, 1994.

[Pel96] D. Peled. Partial order reduction: Model-checking using representatives. *21st International Symposium on Mathematical Foundations of Computer Science*, 1996.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), 1981.

[PNAD95] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. *First International EURO-PAR Conference on Parallel Processing*, 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pon95] Fong Pong. *Symbolic State Model: A New Approach for the Verification of Cache Coherence Protocols.* PhD thesis, University of Southern California, 1995.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. *5th International Symposium on Programming*, 1981.

[RS93] June-Kyung Rho and Fabio Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. *5th International Conference on Computer-Aided Verification*, June 1993.

[SD95a] U. Stern and D.L. Dill. Improved probabilistic verification by hash compaction. *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.

[SD95b] Ulrich Stern and David L. Dill. Automatic verification of the SCI cache coherence protocol. *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[SG87] A.P. Sistla and S.M. German. Reasoning with many processes. *Symposium on Logic in Computer Science*, pages 138–152, 1987.

[SHTO93] Kenji Shibata, Yutaka Hirakawa, Akira Takura, and Tadashi Ohta. Reachability analysis for specified processes in a behavior description. *IEICE Transaction on Communication*, E76-B(11), November 1993.

[Sta91] P.H. Starke. Reachability analysis of petri nets using symmetries. *Systems Analysis - Modeling - Simulation*, 8(4/5):293–303, 1991.

[Tan76] C.K. Tang. Cache design in the tightly coupled multiprocessor system. *AFIPS Conference Proceedings, National Computer Conference*, pages 749–753, June 1976.

[TM91] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[TSL$^+$90] Herve J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. *IEEE International Conference on Computer-Aided Design*, 1990.

[Tur49] A.M. Turing. On checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949. See also: F.L. Morris and C.B. Jones, *An early program proof by Alan Turing, Annals of the History of Computing 6,* pages 139-143, 1984.

[Val90] A. Valmari. A stubborn attack on state explosion. *2th Workshop on Computer Aided Verification*, pages 156–165, June 1990.

[Val91] A. Valmari. Stubborn sets for reduced state space generation. *Advances in Petri Nets 1990*, pages 491–515, 1991.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. *5th International Conference on Computer Aided Verification*, pages 397–408, June 1993.

[WG94] David Weaver and Tom Germond, editors. *The SPARC Architecture Manual Version 9,* appendix D on 'Formal Specification of the Memory Models'. Prentice Hall, 1994.

[WL89] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80. Springer-Verlag, 1989.

[WL93] P. Wolper and D. Leroy. Reliable hashing without collision detection. *5th International Conference on Computer Aided Verification*, 1993.

[Wol86] P. Wolper. Expressing interesting properties of programs. *13th ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.

[Wol87] Pierre Wolper. On the relation of programs and computations to models of temporal logic. *Colloquium on Temporal Logic in Specification*, 1987.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, 1984.

[YGM$^+$95] Lawrence Yang, David Gao, Jamshid Mostoufi, Raju Joshi, and Paul Loewenstein. System design methodology of UltraSPARC-I. *32nd Design Automation Conference*, pages 7–12, 1995.

[ZWR$^+$80] Pitro Zafiropulo, Colin H. West, Harry Rudin, D.D. Cowan, and Daniel Brand. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, 28(4), April 1980.