FORMAL VERIFICATION OF PARAMETERIZED PROTOCOLS ON BRANCHING NETWORKS

by

Michael D. Jones

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2001

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Michael D. Jones

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Ganesh Gopalakrishnan
	Mark Aagaard
	Erik Brunvand
	Chris Myers
	Gary Lindstrom

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of The University of Utah:

I have read the dissertation of Michael D. Jones in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ganesh Gopalakrishnan
Chair, Supervisory Committee

Approved for the Major Department

Tom Henderson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Verifying the functional correctness of a parameterized protocol on all valid branching networks is a difficult problem that can not be solved using simulation alone because the number and shape of valid branching networks is unbounded. This problem is of particular interest because multibus IO and memory protocols can be modelled as parameterized protocols on branching networks.

The central result of the thesis is a model checking algorithm for verifying functional correctness properties of a protocol on a family of topologically isomorphic networks. The algorithm computes a finite set of abstract states, which over-approximate the reachable states for the protocol. Concrete transitions are applied to concrete state fragments during the computation of abstract state successors to avoid the construction of an abstract protocol representation. The algorithm is supported by a specification language, topological case split and generalization argument.

The algorithm, specification language and case split have been implemented in the Πcasso model checker. Πcasso has been used to verify the producer/consumer transaction ordering property for a corrected version PCI 2.1 multibus networks. Πcasso has also been used to find a configuration dependent violation of a write coherence property in PCI networks. Since PCI was not intended to satisfy this property, the significance of this violation is not that it is present in one class of configurations, but that it is not present in every other class of configurations.

To my Father

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNKOWLEDGMENTS

This dissertation is a natural product of a series of lessons in Computer Science that my father taught me many years ago on Sunday afternoons. Both my father and mother have completed graduate degrees; what I know about precise research and observation I have learned from them.

In addition to typing all of the final corrections, my wife offered needed support and encouragement throughout the entire process.

My advisor, Ganesh Gopalakrishnan, provided many hours of helpful and detailed analysis of the formal models contained in this thesis. In addition to many technical contributions, Ganesh has been a consistent example of wisdom and patience during my graduate career at the University of Utah.

Thanks also to the members of my committee for their insightful comments and suggestions.

Finally, thanks to the front office staff at the University of Utah School of Computing. I literally could not have completed my degree without their assistance.

CHAPTER 1

INTRODUCTION

1.1 Background

The motivation for this dissertation is the difficulty of verifying a distributed IO or memory protocol on all valid branching network configurations. This verification problem is difficult because branching networks include subtle and complex behaviors and simulation is inherently incomplete for protocols on branching networks. Simulation is inherently incomplete because the family of branching networks is unbounded. Whereas simulation may be productively used to find errors in many behaviors of many network configurations, simulation can not be used to find all errors in all configurations.

In this dissertation, mathematically-based, or formal, verification techniques are developed that can be used to exhaustively verify a protocol for all instances of a branching network configuration. The remainder of this section contains a more detailed discussion of the design and verification problems for protocols on branching networks.

Particular emphasis is placed on published standards in which errors have been found using formal verification—including the FutureBus+, in which a configuration dependent deadlock scenario was missed by an earlier formal verification effort, but detected in a later formal verification effort. It should be noted that none of the verification efforts discussed in this section contain complete results for a family of multibus configurations. An algorithm for checking multibus configuration families is the main contribution of this dissertation.

1.1.1 Protocols on Branching Networks

This dissertation considers distributed IO or memory protocols in which the behavior of each node is described by the same transition relation. Such protocols are called *parameterized protocols* because the protocol is parameterized by the number and arrangement of nodes.

The description of a parameterized protocol can be divided into a description of the structure of the network and the behavior of the protocol. The structure of a network includes the number and arrangement of nodes along with the structure of each individual node. A network configuration consists of external and internal nodes. An external node sits at the edge of the network whereas an internal node connects two or more devices inside the network. Nodes are connected by a stateless electrical connection such as a bus or serial line. A sequence of connected nodes without any branches is called a path segment. A sequence of path segments forms a path between every pair of external nodes. Each node in the network has the same internal structure. A node consists of a set of unbounded reordering queues which store and forward messages. Messages may be deleted or reordered within a queue by protocol transitions.

A set of parameterized guarded commands describes the behavior of the protocol. Each guarded command is instantiated at each node. If the guard of a command is satisfied by the current nodes' state, then the command creates a new state by changing the state of several nodes. Guarded commands are restricted to modify the state of a finite set of adjacent nodes. The protocol model is asynchronous so that guarded commands may fire whenever their guards are satisfied.

Several parameterized protocols for branching networks have been released and incorporated into existing products. These protocols include the Encore Gigamax [1], FutureBus+ [2], Firewire [3] and PCI [4]. Each of these protocols contains at least one design error found using formal verification. A brief description of each

protocol is given below to further clarify the kinds of systems that use parameterized protocols on branching networks.

The Encore Gigamax protocol is a multilevel cache coherency protocol in which processors and memories are arranged in clusters. The nodes within a cluster communicate on a common bus. Clusters may be connected with other clusters via UIC nodes. The UIC nodes monitor the traffic on a pair of clusters and forward relevant messages. An arbitrary number of clusters can be connected in a hierarchy.

FutureBus+ and PCI are both hierarchical multibus protocols. FutureBus+ connects several processors and memories whereas PCI connects several IO devices and processors. Both protocols allow split transactions through delayed completions. FutureBus+ includes a protocol for maintaining cache coherence whereas PCI is intended to provide a variant of the producer/consumer property.

The Firewire protocol is a serial communication protocol for a hierarchy of IO devices. A device may reside at the edge of a network or at one of the interior junctions. One novel aspect of Firewire is that devices may be added and removed from the network without bringing the network down. When a new device is added, a tree-identify protocol is used to elect a new root node.

The kinds of errors that occur in these protocols include deadlock, starvation, invalid reorderings and signaling errors. Whereas protocols must allow messages to be reordered to avoid deadlock, a protocol must also provide a reliable message delivery service to the terminal nodes. Reordering errors happen when the reordering rules allow reorderings that violate the communication model. In many cases, the presence of a particular reordering error depends on the network configuration. The focus of this dissertation is detecting such errors. Signaling errors often arise from inconsistencies between the text and figures of the protocol specification. Interesting signaling errors have been found through formal analysis of published protocols. These errors are discussed later, but are beyond the scope of this dissertation.

1.1.2 Verification Techniques

A variety of verification techniques have been applied to the analysis of parameterized protocols. This section describes simulation based verification techniques and several formal verification techniques. This section closes with a discussion of errors that have been found using formal verification in each of the Gigamax, Firewire, FutureBus+ and PCI protocols. The FutureBus+ verification efforts are particularly interesting because a second formal verification effort detected a configuration-dependent deadlock scenario in configuration not verified by a previous effort. A more thorough discussion of related research in parameterized protocol verification is deferred to Section 1.2. The formal verification methods developed in this dissertation rely on a form of model checking. A more detailed introduction to model checking can be found [5].

Simulation based verification techniques apply a sequence of stimuli to a network configuration and observe the response. The stimuli can be generated randomly or designed by a verification engineer to exercise a particular part of the design. Simulation based verification has the advantage of being simple and effective. Unfortunately, simulation is inherently incomplete for parameterized protocols because only a finite number of stimuli can be applied to a finite number of network configurations and parameterized protocols are defined over an unbounded set of network configurations.

Formal verification techniques apply reasoning tools to mathematically well-defined models of protocols or systems. The primary benefit of formal verification is a precise statement of what was and, more importantly, what was not verified. The difficulties of applying formal verification are creating a description of a protocol in a mathematically well-defined language and reasoning about the model.

Many formal specification languages and reasoning techniques have been developed for use in formal verification. Only a few representative techniques are

discussed here. Formal methods for verification may be broadly divided into model checking and theorem proving.

A model checker uses an algorithm to determine if a specification is a model of a property. A specification is a model of a property if all specification behaviors specification are contained in the behaviors described by the property. Model checking algorithms in the SPIN [6] and Murϕ [7] model checkers verify linear temporal logic (LTL) properties by enumerating all of the states of a specification. Model checking algorithms in SMV [1] and VIS [8] check computational tree logic (CTL) properties by building symbolic representations of the states of a specification. In both SMV and VIS, reduced ordered binary decision diagrams (ROBDD) store the symbolic representation. An ROBDD is a compact and canonical representation for boolean formulae.

The primary advantage of model checking is that the reasoning is done automatically. The difficulties are that the specification languages and property languages are unexpressive and that the algorithms are exponential in the size of the specification and property. Since the algorithms enumerate states (either explicitly or symbolically) the specification must include only a finite number of states. Symbolic model checking algorithms for systems with infinite, but regular, state sets have been developed. The exponential complexity of model checking algorithms requires clever user intervention to create verification problems that are within the capacity of available computational resources.

Interactive theorem provers provide assistance to a user proving a property of a specification. For verification, the proof requires showing that the specification implies the desired property. Both the specification and property are usually expressed in a variant of higher-order logic. Higher-order logic is a convenient language for expressing infinite behaviors on infinite systems, but deciding the validity of a higher-order logic formulae is undecidable.

Parameterized protocols are not readily amenable to automated analysis using a model checker because most verification problems for parameterized systems are undecidable [9]. Given this fundamental limitation, reasoning about parameterized protocols is done manually, using a proof assistant, or automatically using an incomplete model checking algorithm. An incomplete algorithm will terminate, but only for a certain instances of the general problem. Other automated solutions are defined for a wider class of problem instances, but are not guaranteed to terminate. Some solutions combine manual reasoning and automated analysis.

Compared to model-checking in the context of parameterized protocol verification, higher-order logic is more expressive but the proofs are prohibitively difficult. In this dissertation, a solution based on model checking is developed, but a case study combining higher-order logic theorem proving and model checking is also discussed.

Formal verification has been used to detect errors in the Gigamax, FutureBus+, PCI and Firewire protocols. Whereas these techniques successfully detected interesting errors, none of these methods yield a complete analysis of a multibus topology family. McMillan used SMV to find a deadlock in a single-bus instance of the Gigamax protocol that occurs after a sequence of 13 transitions [1]. The expected time to find this error using random simulation at 10,000 steps per second is between 2.4 years and 29 millenia.

Clarke et al. found several functional errors in the cache coherence portion of the FutureBus+ protocol using symbolic model checking [10]. Pnueli and Shahar later used a combination of theorem proving and model checking to find another deadlock in the same protocol [11]. The new violation was found in a configuration that had not been verified by Clarke et al.

Sighireanu and Mateescu using a model checker to find a deadlock scenario in the link layer of the Firewire protocol [12]. The deadlock stems from a discrepancy

between the text and diagram description of the standard.

Clarke et al. used symbolic model checking to find an error in the signaling protocol for PCI busses [13]. This error also arises from a discrepancy between the textual and diagrammatic descriptions. Corella identified a violation of the PCI transaction ordering property [14]. The proposed solution is verified for all families of multibus PCI networks later in this dissertation.

1.2 Related Research

This verification method developed in this dissertation is most closely related to other work in parameterized system verification. This dissertation builds on prior work in branching topologies and multibus PCI verification case studies.

Related research in parameterized system verification is divided into four groups: induction on finite instantiations, direct symbolic model checking, state enumeration with symbolic state representations, and nonalgorithmic techniques. The largest body of work involves induction on finite instantiations, so these works are discussed in some detail. These four groups are discussed in sections 1.2.1 through 1.2.4. A brief comparison with the work contained in this dissertation appears at the end of each section.

Section 1.2.5 contains a brief discussion of topological results from mathematics and molecular biological evolution which apply to the generation of acyclic branching networks. Section 1.2.6 describes several attempts to verify a property of multibus PCI networks.

1.2.1 Induction on Finite Instantiations

The early works on verification for parameterized systems use induction on a finite instantiation. Intuitively, the process is to show that a property P of a system with n nodes then the property is true of all systems with n nodes or more. The

advantage of this approach is that the system with n nodes has a finite number of states and can be checked using automatic model checking algorithm. The difficulty is showing that the induction is valid.

In the first of these works, by Browne, Clarke and Grumberg [15], the user is required to show bisimulation [16] between a system with two nodes and a system with n nodes. The bisimulation relation, is sufficient to show that all indexed CTL* properties of the two-node system hold for all n node systems. The two-node system is verified using a CTL model checking algorithm, such as [17].

Unfortunately, showing bisimulation requires building the state graph of the two-node system. The size of the state graph grows exponentially in the size of the transition system. Clarke and Grumberg [18] address this problem through the construction of a *closure process*. In this method, the user creates a closure process $P*$ so that the product of an n node system with $P*$ bisimulates the product of an $n+1$ node system with $P*$. A polynomial algorithm is given for showing bisimulation between closure process products that avoids the construction of the state graph. The product of the n node system and $P*$ is then verified using a model checking algorithm. The difficulty here is finding a suitable closure process.

German and Sistla [19] give a fully automatic method for verifying a propositional temporal logic (PTL) specifications of parameterized synchronous systems using a Vector Addition System With States (VASS). A VASS is a labeled graph in which the vertices represent states and the edges represent transitions between states. A VASS is similar to a Vector Addition System [20] or a Petri Net [21]. Each edge is labeled with a vector. In a VASS representing a parameterized system, the ith entry of a vector-label has value a if a nodes are in state i. A model checking algorithm for PTL formulae on VASS models is developed and extended to include certain liveness properties.

Both [18] and [15] consider only linear network topologies. Shtadler and Grum-

berg [22] give an inductive argument for topologies generated by a context-free *network grammar*. In a network grammar, the terminals represent primitive network elements and the productions combine the primitives to form networks. A property is verified automatically by showing that the property holds for a suitable finite network model generated from the grammar.

Rather than finding a process that is suitable for induction, Wolper and Lovinfosse [23] and Kurshan and McMillan [24] find a property which is suitable for induction. In this method, the user provides a *network invariant* I which is preserved by adding nodes to the network. Because I is unchanged by adding nodes to the network, I is called a *network invariant*. A model checker verifies that I holds on a system with just n nodes. This method moves shifts the burden from showing bisimulation to finding a network invariant.

Clarke et al. [25] eliminate the need for a bisimulation proof or a network invariant for network grammars using an automated abstraction method. The abstraction method generates a network invariant I, which can be used to verify the system. Unfortunately, if the automatically generated invariant of [25] is too weak, then the method fails. Lesens et al. [26] used a widening [27] technique to create a range of invariants, some of which may succeed. This method also applies to a wide class of network topologies.

Emerson and Namjoshi [28] eliminate the bisimulation proof by restricting network topologies to token rings. The behavior of all token passing rings can be modeled by the behavior of a token passing ring with a predetermined number of nodes. The smaller ring can then be verified by a model checker.

The method presented in this dissertation does not use induction on a finite instantiation. Finite instantiation for protocols in which a transition can change the state of more than one node is difficult because the finite instantiation must be large enough to contain the product of all the transitions. In the related work section,

a dynamic state model checker is discussed, which may allow a finite instantiation solution for more complex protocols.

The method presented in this dissertation uses an abstraction argument that requires showing a simulation preorder as used in [25]. However, rather than constructing the abstract transition system (even automatically, as in [25]) the abstract states are constructed by applying the concrete transition relation.

1.2.2 Direct Symbolic Model Checking

In this approach, the transition system is specified using a rich assertional language which is adequate for symbolic model checking. The specification can then be checked directly—without induction or abstraction.

Symbolic model checking with rich assertion languages was first proposed by Kesten et al. [29]. An assertion language L is adequate for symbolic model checking if: the initial states and the property to be verified can be expressed in L, L is closed under boolean operations and there is an algorithm for deciding equivalence of two expressions and there is an algorithm for constructing predicate transformers in L. Given an adequate rich assertion language, the usual symbolic model checking algorithm can be used to verify safety properties of protocols with locally scoped transitions. Kesten et al. then use regular expressions as an assertional language for linear topologies and propose tree regular expressions for branching topologies. Results are given for linear arrangements of FutureBus+ agents.

Abdulla et al. [30] extend the symbolic model checking algorithm for parameterized systems to include transitions with global guards through the use of an *acceleration* operation. The acceleration operation transforms a transition T into an accelerated transition T_a which represents the effects of applying T to many nodes in a single step. Previously, acceleration had been used in the context of communicating finite state automata [31, 32]. Unfortunately, whereas the guard

can depend on the global state, a transition can change the state of only a single node at a time, and accelerated operations must be atomic. Jonsson and Nilsson [33] introduced a *transitive closure* operation that terminates for transitions with finite local scope. Bouajjani et al. [34] give a more general transitive closure operation, but this construction is not guaranteed to terminate.

Three research groups use variants of the weak second-order monadic theory of one successor (WS1S) [35] as an assertional language. Pnueli and Shahar [36] use FS1S (WS1S with an upper-bound on set sizes) and obtain both liveness and acceleration for a wide class of protocols. Bodeveix and Filali [37, 38] also use FS1S, but do not have liveness results. Baukus et al. [39] define an abstraction on WS1S representations that terminates for a wider class of systems than [30], but with an over-approximation.

In this dissertation, an abstraction is combined with state enumeration, rather than symbolic model checking. The key difficulty with symbolic model checking for branching topologies is that all states of all topologies are generated in a single verification run. In this dissertation, each class of isomorphic topologies is considered in one verification run. This topological case split avoids the state explosion problem, but precludes checking a property for *all* unique topologies—as can be done, in theory, with the symbolic approaches.

1.2.3 Enumeration of Canonical States

In this method, an enumeration algorithm generates a set of reachable canonical states. A canonical state represents a set of equivalent concrete states. This method avoids the state space explosion problem by representing many similar states with a single state. The idea of representing many states with a single abstract state can be traced to Lubachevsky [40] and Dijkstra [41].

Pong and Dubois [42] use canonical states to verify parameterized cache coherence protocols. In this representation, the repetition constructors $\{0, 1, +, *\}$ indicate that a system state contains zero, one, one or more or zero or more caches in a given local state. Unfortunately, the canonical state representation can not represent topology information and requires a user-supplied abstract transition relation.

Ip and Dill [43] use the same representation, but eliminate the need for an abstract transition relation. Two concrete states represented by a canonical state are selected and used to construct the next canonical states using the concrete transition relation. The resulting algorithm is implemented in the Murϕ model checker [7].

The algorithm presented in this dissertation is a state enumeration algorithm for canonical states, but it uses an explicit canonical state representation rather than repetition constructors. An explicit representation allows canonical states to contain topology information, but limits canonical states to a finite set of messages.

In this dissertation, fragments of concrete states represented by an abstract state, rather than complete concrete states, are used to construct the canonical state successors. Since global transitions require the entire concrete state, the resulting algorithm is limited to locally scoped transitions. However, the algorithm must use fragments because a canonical states represents an unbounded number of complete concrete states—but only a finite set of size-limited fragments. The algorithm presented in this dissertation is also implemented as an extension of the Murϕ model checker.

1.2.4 Other Nonalgorithmic Methods

Nonalgorithmic methods, such as Higher-order logic theorem proving an logic program transformations, have been applied to parameterized system verification.

Since the problem of parameterized system verification is undecidable in general, nonalgorithmic methods can be applied to a wider class of systems than the algorithmic methods discussed previously.

Parameterized system verification using higher-order logic has been done in the context of circuits containing linear replicated components by Hanna et al. [44] and Windley [45]. Circuits with tree-shaped replicated components were considered by Bunker [46]. These works identify induction schema that simplify verification for a class of circuits created from replicated components. These works use induction on finite instantiations as before, but the proofs are made explicit.

Bhargavan et al. [47] use a manual proof in HOL [48] together with a model checker to verify certain properties of the routing information protocol (RIP). Model checking was used to verify key lemmas required to complete the manual proof of the desired correctness property. Whereas this case study includes a rich class of network topologies, the results do not generalize to other protocols.

Corella et al. [49] wrote a pencil and paper liveness proof in higher-order logic for PCI multibus networks under certain fairness assumptions. Pencil and paper proofs provide the same expressive power as higher-order logic theorem provers, but do not offer the extra assurance of machine-checked proofs. This work was also a one-time case study rather than a general verification method.

Roychoudhurry et al. [50] enhanced the XMC model checker [51] with logic logic program transformations to verify properties of parameterized systems. Given a model of a parameterized system and a model of a property, the parameterized system is transformed into the property. Some steps in the process require manual assistance, whereas others can be completed automatically. The enhanced XMC model checker completes the automated parts of the transformation and requires user intervention for the remaining steps. An extended set of logic program trans-

formations for handling recursion [52] is needed. The general purpose verification method covers a wide class of properties, topologies and protocols.

In this dissertation, nonalgorithmic solutions are avoided, but a case study involving theorem proving and model checking is presented in Chapter 7. Nonalgorithmic solutions are avoided due to the difficulty of identifying and proving inductive invariants required to verify parameterized systems. As expected, the algorithm developed in this dissertation can be applied to a narrower class of problems than the nonalgorithmic methods.

1.2.5 Branching Network Topologies

Branching network topologies are variants of full Steiner topologies. Whereas Steiner topologies have not been used in parameterized system verification, they have been studied in the context of mathematics and molecular biological evolution.

General properties of Steiner topologies are discussed in [53]. A particularly important property of Steiner topologies is the number of full Steiner topologies over n leaf nodes. This bound is derived constructively by Gilbert [54]. The constructive derivation yields a method for enumerating full Steiner topologies used later in the dissertation.

Network topologies represented by variants of full Steiner topologies are identical to intermediate trees generated by the neighbor joining method. The neighbor joining method is used to infer maximally simple evolutionary trees from the gene sequences of present day species given by Saitou [55].

1.2.6 Multibus PCI

The work presented in this dissertation builds four previous attempts to verify properties of multibus PCI networks. The goal in each attempt was to show that all execution sequences of all PCI multibus networks satisfy a message transaction

ordering property. Whereas only one of the four attempts succeeded in proving the desired property, the lessons learned each attempt contributed to the development of this dissertation. Each attempt is summarized below along with the reason for failure and what was learned in the process.

The first attempt [56] used a higher-order logic model of multibus PCI networks in the PVS [57] theorem prover. Using this model, some useful invariants about PCI states were proven and a case split on PCI network topologies was developed. Despite a combined 18 person-months spent by three experienced users, the proof of the ordering property was never completed. The key difficulty was the identification and proof of auxiliary invariants needed to prove the ordering property as an invariant. For example, a proof that a message exists only if it was created previously required 60 hours of effort—after it was determined that this invariant was needed. Although a proof of the ordering property was never completed, the topological case split was seminal for the topological case split given later.

The second attempt [56] used a PROMELA model of multibus PCI networks in the SPIN [6] model-checker. Using this model, only certain execution sequences of certain networks were shown to satisfy the ordering property. Several hundred thousand states of three different PCI networks were enumerated in 24 hours of CPU time. The key difficulty with the PROMELA model is that SPIN is a finite state verification tool and multibus PCI networks are an infinite family of finite state networks. A pencil-and-paper argument was developed to generalize the SPIN results to include all PCI networks and execution sequences. Although this argument was incomplete and unconvincing, it forms the foundation of the monotonicity and well-routed properties used later to generalize the results of NDFS.

The third attempt [58] developed a representation based on quantified predicate abstraction. Predicate abstraction [59] is a technique for using a set of predicates

to characterize abstract states in the computation of an abstract interpretation. Predicate abstraction has been used by Das et al. [60] and Lesens and Saidi [26] to verify parameterized systems. In each case, the key difficulty is the quantification needed to represent collections of nodes in a certain state. In [58], a method for avoiding explicit quantification of the predicates used in the abstraction was proposed. Whereas the method appeared promising, reasoning about quantified expressions was still required. Unable to hide quantification, this representation was never used to reason about multibus PCI. Only explicit state representations were used after this attempt, due to the difficulty of dealing with states encoded using variables encountered in this abortive attempt

The fourth attempt [61, 62] used an abstract model of multibus PCI in the Murϕ model checker with a manual refinement proof built in PVS. All execution sequences of all PCI networks were shown to satisfy the ordering property using this model—modulo a complex axiomatization of the abstraction. In this attempt, an abstract and a concrete model of PCI were developed in PVS. The abstraction was axiomatized, and it was proven that the concrete model refines the abstract model. The concrete model was then translated into a Murϕ model and each topology was verified. The Murϕ verification required approximately 5 seconds of CPU time, and the refinement proof required approximately 1 month of development time. Whereas this attempt achieved the verification goal, the process was inadequate. The key difficulty was the refinement proof. The refinement proof was difficult and applies only to a specific model of PCI. The abstraction used in the PVS model is precisely the abstraction implemented by the NDFS algorithm. This attempt is discussed in more detail in Chapter 7.

This dissertation builds on the success of the fourth effort with an algorithm that computes the reachable states of the abstract model directly from the concrete model. Avoiding construction of the abstract model avoids the need for a

refinement proof. The PCI ordering property property was re-proven using the the NDFS algorithm in approximately 8 minutes of CPU time without a refinement proof. Whereas the NDFS algorithm required 60 times more CPU time than the abstract Murϕ model, use of the NDFS algorithm saved 1 month of refinement proof development time needed for the Murϕ model.

1.3 Contributions

This dissertation identifies a new parameterized verification problem and proposes a solution based on the enumeration of abstract canonical states. The resulting verification technique can be used to exhaustively analyze functional behavior of protocols on an entire family of isomorphic networks. The verification techniques developed in this dissertation support the functional analysis of parameterized protocols modeled at the network level on all valid network configurations.

The new verification problem is to show that a parameterized protocol provides a communication service to a collection of terminal nodes in a family of isomorphic network configurations. In terms of the International Standards Organization Open Systems Interconnection Reference Model (ISO OSI) [63], this dissertation uses network-level models to verify transport-level properties. Prior research in parameterized protocol verification uses presentation-level models to verify properties at the presentation level or above.

The solution to this problem uses a new abstract canonical state enumeration algorithm, called NDFS. The set of states generated by the NDFS algorithm approximates the reachable states of a protocol on a family of isomorphic network configurations. A significant feature of the NDFS algorithm is the use of the partial concretization to compute abstract state transitions. The NDFS algorithm

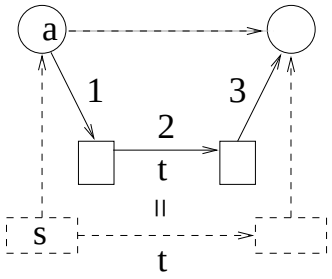

Figure 1.1. Computing abstract state transitions.

is supported by a topological case split and a novel specification language called Newspeak[1].

The NDFS algorithm computes the set of reachable abstract canonical states using a three step process, shown in Figure 1.1. First, fragments of concrete states represented by an abstract state a are generated. In Figure 1.1, circles represent abstract states and small rectangles represent concrete state fragments. Second, every transition t from P is applied to every concrete state fragment. Third, each resulting fragment is merged with a copy of a to create new abstract states. The point is that for every application of t to a state s, NDFS computes the effect of applying t to the abstract state of s. The figure shows this relationship for transition t applied to concrete state s. The dashed lines show the usual commutative diagram and the solid lines show the transitions completed by NDFS.

Since most verification problems for parameterized systems are undecidable, the applicability of NDFS is necessarily limited. The limitations impact three areas: correctness properties, protocol behaviors, and network topologies. Correctness properties may describe only safety properties of systems that send m messages between n terminal nodes. Protocol transitions may modify only a finite set of

[1]Newspeak is named after the "Newspeak" language developed in George Orwell's post-utopian novel Nineteen Eighty-Four [64]. In the novel, Newspeak controlled popular thought by limiting the ideas that could be expressed.

adjacent nodes and may not create states with more than m messages. Network topologies may connect only n terminal nodes. Given these limitations, the correctness statement generated by NDFS is: "all states of protocol P containing at most m messages in any network with topology T satisfy property ϕ."

The NDFS algorithm is supported by the Newspeak specification language and a topological case split. All well-formed Newspeak transitions modify only a finite set of adjacent nodes, as required for NDFS. Consequently, NDFS can be applied to syntactically valid Newspeak specifications which do not create new messages. The topological case split generates all acyclic network configurations over n terminal nodes. Each topology enumerated in the case split is verified by a single run of NDFS.

NDFS, Newspeak and the topological case split have been implemented in the Πcasso model checker. Πcasso has been used to verify the producer/consumer transaction ordering property on Peripheral Component Interconnect (PCI) 2.1 multibus networks in just over 8 minutes of execution time after enumerating 1,448 abstract states. This result improves our previous PCI verification result, which required significantly less execution time, but required a manually derived abstract transition relation system and a lengthy manual refinement proof.

Πcasso has also been used to verify a write-coherence property for PCI. Πcasso found a violation of the coherence property in one of the four four-terminal multibus PCI topologies after checking 103 abstract states in 5.6 seconds. Since PCI was not intended to satisfy this property the significance of this error is not that the error exists in one topology, but that the error does not exist in *any* other topologies.

In some cases, the network and protocol behavior restrictions on NDFS results can be relaxed. The network restrictions can be relaxed when adding an additional i terminals to an n-terminal network does not result in any new behaviors in the original n-terminal network. The protocol behavior restrictions can be relaxed

when states with $x + m$ messages have fewer behaviors than the original state with m messages. Since multibus PCI satisfies both of these properties, the Πcasso verification results for four-node PCI networks can be generalized to all states of all PCI networks.

1.4 Scope of the Dissertation

The scope of this dissertation is limited to detecting topology dependent violations of safety properties by locally scoped protocols on acyclic networks with fixed sets of terminal nodes. A complete and more detailed list of the restrictions on protocols, properties and networks considered in this dissertation is given at the end of Chapter 2. Many open issues regarding the verification of protocols on branching networks are not addressed by this dissertation. Each of the paragraphs that follow discuss an issue that is beyond the scope of the dissertation.

Verification of liveness properties is beyond the scope of dissertation. The NDFS algorithm can be used only to verify safety properties. Liveness properties are beyond the scope of the dissertation. Verification of liveness properties in an abstract model is difficult because infeasible states mask potential deadlock violations. In future work, NDFS may be adapted to checking $\forall CTL$ (CTL limited to universal quantification) properties rather than just safety properties. An algorithm for full CTL is preferable to an algorithm for $\forall CTL$, but the use of an abstraction introduces extra computational paths that can mask violations of existentially quantified CTL properties [65].

Case splits for topologies other than acyclic branching networks are beyond the scope of the dissertation. In general, there are an unbounded number of cyclic topologies for n terminal nodes because a single network on n terminal nodes may contain an arbitrary number of cycles.

Serializability for concurrent transitions is beyond the scope of the dissertation. As will be seen in Chapter 3, NDFS imposes a serialization on concurrent transitions. A brief argument for serializability of multibus IO transitions protocols is given in Chapter 3, but the general problem of showing that a transition relation is serializable is beyond the scope of the dissertation.

Global transitions and properties are beyond the scope of the dissertation. The abstract state model permits only the number of nodes in a network path to be unbounded—every other aspect of the property and protocol must be bounded. More specifically, NDFS can not be used on:

- Transitions that depend on an unbounded portion of the network state. An example of such a transition system is a transition in which a node observes, or modifies, the state of all its neighbors. Whereas a node in a single network configuration has a finite set of neighbors, checking all network configurations requires assuming that a node may be adjacent to an unbounded, but finite, number of nodes. Complete verification for protocols with universal quantification over neighbors requires modeling an unbounded number of neighbors and is beyond the scope of this dissertation.

- Safety properties for an unbounded sets of terminal nodes. The topological coverage of NDFS limits the property class to include properties over a fixed set of terminal nodes.

1.5 Structure of the Dissertation

The dissertation is structured in a problem-solution format. The dissertation begins with a model of parameterized systems followed by two verification problems for the model. Next, one problem is solved using the NDFS algorithm. After giving the NDFS algorithm, an implementation of NDFS in the Πcasso model checker

is discussed along with two verification results for PCI multibus networks. The second problem is solved by identifying protocols for which NDFS results can be generalized. An earlier attempt based on a combination of assisted reasoning and model checking is given and compared to the use NDFS on the same problem. The dissertation concludes with a summary and avenues for future work.

In Chapter 2, the two verification problems addressed in the remainder of the dissertation are defined. The problem statements require a formal model of parameterized protocols that includes network topology. This model is developed and the chapter concludes with two problem statements.

An algorithmic solution to the second problem is given in Chapter 3. Since the algorithm operates on an abstract model, an abstract network model is developed, followed by the pseudo-code for the NDFS algorithm. Chapter 4 contains two proofs about NDFS. In the first proof, NDFS is shown to terminate. In the second proof, NDFS is shown to terminate with an over-approximation.

Chapter 5 contains an implementation of NDFS and experimental results. The implementation is used to verify a simple property of a simple network and two properties of PCI multibus networks. We verify that a corrected version of PCI complies with the intended transaction ordering property given in the specification (the published standard does not). We also verify PCI compliance with a write coherence property. A topology dependent violation of the coherence property is found. These are the first algorithmic verification results for a commercial multibus IO protocol.

Chapter 6 returns to the first (and undecidable) verification problem. In some cases, the results for NDFS can be generalized to solve the first problem. These cases are formally identified and two proofs are given. The first proof generalizes state coverage and the second proof generalizes network coverage.

Chapter 7 contains a case study in which the abstraction computed by NDFS was derived manually. The derivation of the abstract transition relation is given along with a refinement proof. The results in Chapter 7 detail the effort avoided by using the NDFS algorithm. Model checking results for the abstract model are given.

CHAPTER 2

TWO VERIFICATION PROBLEMS FOR NETWORKS OF REPLICATED COMPONENTS

In this chapter, two verification problems for networks of replicated components are defined. Before giving the problem definitions, the computational model used in each verification problem is presented. Solutions to each of these problems are presented in the chapters that follow. A glossary, found in Appendix A, lists all of the symbols defined in this and the next chapters.

Two problem definitions are given since the goal of this thesis is an algorithmic solution to an undecidable problem. The undecidable network parameterized verification problem (NPV) requires checking a property for all states of all network configurations. The decidable limited network parameterized verification problem (LNPV) is a subproblem of NPV which requires checking fewer network configuration classes and network states. In some cases, results for LNPV can be generalized to NPV.

Both problems are defined in the context of a restrictive computational model. The computational model is divided into three parts: network structure, protocol behavior and correctness property. Restrictions on each part of the model are essential to both the decidability of LNPV and the generalization to NPV. The restrictions were chosen not for their formal elegance, but for their pragmatic utility. For both problems it is required that:

- network structure is limited to acyclic branching networks of replicated nodes. Each node contains an identical set of queues,
- protocol behavior is limited to locally-scoped transitions that have address independent behavior,
- correctness properties are limited to safety properties of a finite set of terminal nodes communicating using a finite set of messages (messages will be formally defined in the next section).

Transitions are expressed in specification language called Newspeak. Newspeak is defined such that every Newspeak transition is both locally scoped and location independent.

The computational model for LNPV adds the following restrictions to the above limitations:

- network structure is limited to configurations that connect *only* the terminal nodes in a correctness property,
- protocol behavior is limited to noncreative transitions that delete as many messages as they insert, or more.

After laying the notational groundwork in Section 2.1, we give formal definitions of the network, protocol and property models in Sections 2.2 through 2.4. The NPV and LNPV problems are defined in Section 2.5. An algorithm for LNPV is given Chapter 3 and a limited generalization from LNPV to NPV is given in Chapter 6.

2.1 Preliminaries

The natural numbers are denoted Nat. The notation $a = b[c/d]$ denotes the set a constructed from b by replacing d with c. A relation on A denotes a relation

which is a subset of $A \times A$. Functions are represented as relations over their domain and co-domain. The co-domain of a function is the last element of its relational representation. A function in $A \times B \rightarrow C$ takes a pair from $A \times B$ as an argument and returns a value from C. A function on $A \rightarrow B$ where $f(a) = b$ can also be written as a set containing the pair $(a, b) \in f$. Function f applied i times to argument a, $f(f(...f(a)))$, is denoted $f^i(a)$. Sets are often defined using a set comprehension. A set comprehension is an expression $A = \{a \mid p(a)\}$ which means that set A contains all elements a that satisfy predicate p.

2.2 Network Model

Networks are defined first because protocols are parameterized by networks.

Definition 2.1 (Network) *A network N consists of a finite set of node locations L and a routing function* next.

The routing function next describes the connections between nodes. Given the addresses node n and target node n', next of (n, n') is the address of the next node from n toward n'. Using the nodes in Figure 2.1 as an example, $\mathsf{next}(a, d)$ is b since b is the next node from a to d. Applications of next can be nested to find the *ith* node from n to a target n'. Returning to Figure 2.1, $\mathsf{next}(\mathsf{next}(a, d), d)$ is c since c is the second node from a to d.

Nodes are addressed using a two part addressing scheme based on the node's position within a path segment. Node addresses are called *locations* and the set of

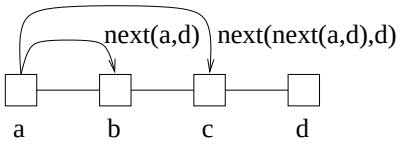

Figure 2.1. Determining the next node.

locations for a network is denoted L. A node at location (n_i, p_j) is the ith node in path segment j.

A *path segment* is a sequence of nodes that does not contain branches. A *path* is a sequence of adjacent path segments. Every node in a path segment p, except the end nodes, has exactly two neighbors. The end nodes of a path segment p may have several neighbors if several path segments are connected to p. In some cases, the end points have one neighbor. Path segment end points with one neighbor are called *terminal nodes* because they sit at the terminal ends of a path segment. The set of terminal nodes in a network is denoted L_E.

The network N_{ex}, shown in Figure 2.2, visually describes the anatomy of a network. Network N_{ex} contains several nodes, shown as rectangles, arranged into a network with five path segments, labeled a, b, c, d and e. For path segment c, the end points are nodes $(0, c)$ and $(3, c)$. Neither of these two end points are terminal nodes since each node has more than one neighbor. The neighbors for $(0, c)$ are $(2, a), (2, b)$ and $(1, c)$. Node $(0, a)$ is a terminal node because $(0, 2)$ has only one neighbor.

The remainder of the thesis focuses on completely connected acyclic networks. In a completely connected acyclic network, exactly one sequence of path segments

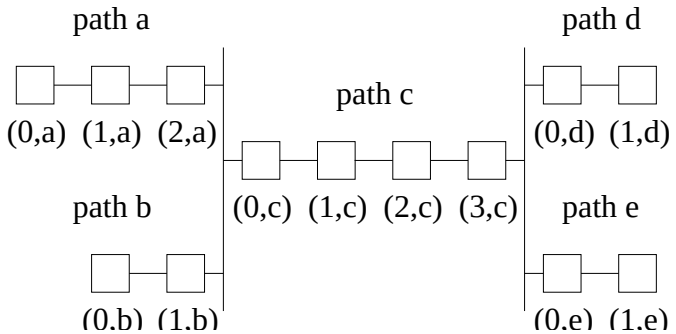

Figure 2.2. Anatomy of a branching network N_{ex}.

connect each pair of nodes and next is uniquely defined for every pair of locations in L. The network N_{ex} in Figure 2.2 is both completely connected and acyclic.

Completely connected acyclic networks are classified by the number of terminal nodes in the network.

Definition 2.2 (Network class) *The network class N_i is the set of all networks $N = (L, \mathsf{next})$ such that $|L_E| = i$.*

The network N_{ex} from Figure 2.2 is a member of N_4 since N_{ex} has four terminal nodes.

Although there are only a finite number of network shapes in a given N_i, each N_i contains an infinite number of unique networks. Two unique networks with the same shape may contain a different number of nodes per path segment. For example, N_{ex} has one of two shapes from N_4, the other shape is a star-network with four branches. An infinite number of unique networks shaped like N_{ex} can be built from N_{ex} by adding nodes to path segment a, or any of the other path segments.

The bound on the number of shapes for acyclic networks with i terminal nodes is necessary to retain decidability for the LNPV problem. Unfortunately, the bound grows super-exponentially in i, as will be seen in Chapter 4.

2.3 Protocol Model

The preceding network model described network structures, the following protocol model describes networks states—and how to change them. Protocols are executed on specific network configurations, but are defined for all network configurations. Henceforth, the formal network parameter for protocol P will be denoted N with locations L and routing function next. The protocol model has three parts: network states (Section 2.3.1), state transition systems (Section 2.3.2) and reachable states (Section 2.3.3).

The elements of are protocol are summarized in the following definition for convenience.

Definition 2.3 (Network parameterized protocol) *Given a network $N = \langle L, \mathsf{next} \rangle$, a network parameterized protocol $P(N)$ is a 5-tuple*

$$P(N) = \langle Q, \Sigma, S, T, T_E \rangle$$

where Q, Σ and S define states of network N and T and T_E define transitions between network states.

2.3.1 Network State Model: Q, Σ, S

The network state model is based on the node state model. The node state model describes messages stored in queues. A *message* Σ is a triple (opc, src, dst) consisting of an opcode opc, taken from a finite alphabet Λ, together with source src and destination dst. Both src and dst are locations taken from the set of locations L for network N. A message is *valid* if its opcode is taken from Λ and its source and destination are taken from L. The empty message is denoted ϵ and is valid by definition.

A *queue* stores an unbounded sequence of messages. Each node in a network contains the same number of queues; each queue is identified by a unique index. The set of *queue indices* is denoted Q. Within a queue, messages are indexed by queue positions which are numbered consecutively starting with 1. The *length* of a queue is the number of nonempty messages stored in the queue. For queue q, the length of q is the position of the last nonempty message in q. The length of q in node n_i of path segment p_j in state s is denoted $\mathsf{Qlength}(q, (n_i, p_j), s)$.

The state of a single node containing two queues is shown in Figure 2.3. For this node, and all other nodes in the network, the set of queue indices Q contains 1 and 2. The length of queue 1 is three and the length of queue 2 is one. The first

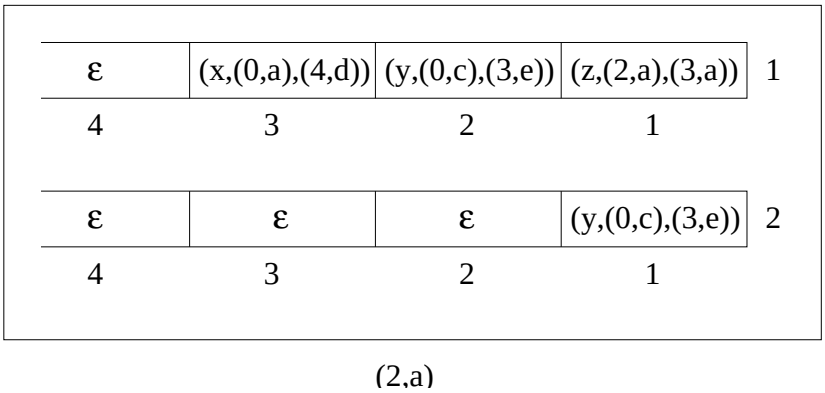

Figure 2.3. State of a single node.

message in queue 1 is a message with opcode z from the node at location $(2, a)$ to location $(3, a)$. This particular node is at location $(2, a)$ which means this is the second node in path segment a.

The state of the entire network is the union of the states of each node. A *network state* is modeled as a set of four-tuples. The set of all network states is denoted S. Tuple (i, q, l, m) in network state $s \in S$ indicates that queue position i of queue q in the node at location l contains message m. If state s does not contain an entry for position i of queue q in node l, then that position is assumed to be empty. Alternatively, the state can be viewed as a function in which $s(i, q, l)$ returns m if (i, q, l, m) is in s and returns ϵ if (i, q, l, m') is not in s for any m'. If node $(2, a)$ is included in state s, then state s contains an entry $(0, 1, (2, a), (z, (2, a), (3, a)))$. Viewed as a set, state s does not contain an entry for $(4, 2, (2, a))$ because position 4 of queue 2 in node $(2, a)$ is empty.

A network state s is *valid* if s satisfies the following two properties. First, every tuple (i, q, l, m) in s consists of a queue index q from Q, a location taken from L and a valid message m. Second, if (i, q, l, m) is in s for a nonempty valid message m, then (j, q, l, m') is also in s for every j less than i and any valid message m'. The first requirement avoids storing messages in nonexistent locations and the second

requirement avoids gaps between messages in queues. Unless otherwise mentioned, all states are assumed to be valid. For example, node $(2, a)$ in Figure 2.3 is in a valid state if queue 1 contains the empty messages for positions greater than 4 and if queue 2 contains the empty message for positions 2 or greater.

2.3.2 State Transition Model: T, T_E

Protocol transitions are network state transformers. More precisely, protocol transitions are adjacent node state transformers because the effects, or scope, of a single transition is limited to a finite set of adjacent nodes. A transition that is limited to a finite set of adjacent nodes is called a *locally scoped* transition. In contrast, a globally-scoped transition can observe or modify the entire network state. The restriction to locally-scoped transitions is also a necessary aspect to retain decidability for the LNPV problem.

Protocol transitions are modeled using a two-part *parameterized transition relation*. A node location parameterizes each transition. Every node in the network uses the same transition relation instantiated with the location of that node.

The transition relation is divided into two parts because some transitions are applied only to terminal nodes and others are applied to every location. The set of *external transitions* applied to terminal nodes only are denoted T_E and the set of *universal transitions* applied to any node are denoted T.

Every transition has two parts: a guard and a command. The guard is a predicate on the local state and the command is an assignment on the local state.

Both the guard and command are expressed in a transition language called *Newspeak*. The syntax of a Newspeak transition is given in Figure 2.4. The top-level production in Newspeak is the *transition* production which creates either an external transition (*exttransition*) in T_E or a universal transition (*univtransition*) in T. The guard may compare an opcode from Λ with the opcode of a message

$$
\begin{array}{rl}
\mathit{transition} & ::= \mathit{exttransition} \mid \mathit{univtransition} \\
\mathit{exttransition} & ::= \langle \mathit{guard}, \mathit{extcmd} \rangle \\
\mathit{univtransition} & ::= \langle \mathit{guard}, \mathit{cmd} \rangle \\
\mathit{extcmd} & ::= (\mathit{qloc}, \mathit{qindex}, \mathit{curloc}) := (\mathit{opc}, \mathit{curloc}, \mathit{extloc}) \\
 & \quad \mid \mathit{extcmd}; \mathit{extcmd} \\
\mathit{cmd} & ::= (\mathit{qloc}, \mathit{qindex}, \mathit{obsloc}) := (\mathit{opc}, \mathit{obsloc}, \mathit{loc}) \mid \mathit{cmd}; \mathit{cmd} \\
\mathit{univloc} & ::= \mathit{loc} \mid \mathit{extloc} \\
\mathit{guard} & ::= \mathit{opc} = \mathit{opc} \mid \mathit{loc} = \mathit{loc} \mid \mathit{guard} \vee \mathit{guard} \mid \neg \mathit{guard} \\
\mathit{opc} & ::= \mathit{anopc} | \mathit{msg.opc} \\
\mathit{msg} & ::= \texttt{msg\_at}(\mathit{qloc}, \mathit{qindex}, \mathit{obsloc}) \\
\mathit{loc} & ::= \mathit{obsloc} \mid \mathit{msg.src} \mid \mathit{msg.dst} \\
\mathit{obsloc} & ::= \mathit{curloc} \mid \texttt{next}(\mathit{obsloc}, \mathit{loc}) \\
\mathit{curloc} & ::= l_p \\
\mathit{keywords} & : \texttt{msg\_at} \mid \texttt{next} \\
 & \\
\text{Primitives} & : \mathit{extloc} : L_E,\ \mathit{qloc} : \mathit{Nat},\ \mathit{qindex} : Q \text{ and } \mathit{anopc} : \Lambda
\end{array}
$$

Figure 2.4. Syntax of Newspeak transitions

contained in an observable location or may compare observable locations and the source or destination of a message. An observable location is either the current location l_p or or the next node along the path toward an observable location or toward a message source or destination. The differences between external and universal transitions are that the command of an external transition may create messages destined to any other terminal node and that external transitions can modify only the local state of the current location.

Newspeak has two keywords: `msg_at` and `next`. The `msg_at` keyword is used to determine the message at a certain location and the `next` keyword is used to identify the next node along the path between two nodes. The primitives in Newspeak and their types are shown at the bottom of Figure 2.4.

Well-formed Newspeak transitions are *location independent* and have finite *branching* and *linear* local scope. Finite branching and linear local scope are required to make the LNPV problem decidable. Location independence and branch-

ing local scope are required to show that NDFS conservatively approximates LNPV. Finite branching local scope is also used to generalize some results for LNPV to NPV. A transition is *location independent* if the transition cannot determine the index of the node at which it is applied. Location independence is achieved by disallowing comparison of l_p with a constant.

A transition has finite *branching local scope* if the transition can observe nodes along a finite set of adjacent path segments. Newspeak transitions have branching local-scope because a transition applied at the current location, *curloc*, can observe or modify only the states of nodes between *curloc* and the source or destination of messages contained in *curloc*. If each node contains a finite number of messages (as is the case in LNPV), then each transition will have branching local scope. Branching local scope precludes transitions that poll the states of *all* adjacent nodes. Such polling must be disallowed because a node may have an unbounded number of adjacent neighbors.

A transition has finite *linear local scope* if the transition can observe only a finite set nodes along any adjacent path segment. Newspeak transitions have linear local scope because Newspeak does not allow iteration. Without iteration, Newspeak transitions can only make finitely nested applications of `next`. The implementation of Newspeak includes loops over finite domains as syntactic sugar.

Since the linear scope of a transition plays an important role in the construction of a solution to the LNPV problem, a scope operator is defined in Figure 2.5 that gives the linear scope of a transition. The scope operator is defined recursively over the structure of Newspeak expressions. Given a transition t, the scope (t) is one plus the depth of the deepest nesting of `next` in either the guard or command of t.

An example of a Newspeak transition is:

$$\begin{aligned} t_{ex} = \quad & \langle Guard : \texttt{msg\_at}(0, 2, l_p).opc = a, \\ & \quad Cmd : \texttt{msg\_at}(3, 2, \texttt{next}(l_p, \texttt{msg\_at}(0, 2, l_p).dst)) := \texttt{msg\_at}(0, 2, l_p)\rangle. \end{aligned}$$

$\mathsf{scope}(< g, c >)$ for guard g and any command c is $1 + max(\mathsf{scope}(g), \mathsf{scope}(c))$
$\mathsf{scope}(a; a')$ for any actions a and a' is $max(\mathsf{scope}(a), \mathsf{scope}(a'))$
$\mathsf{scope}((q, i, l) := (o, l, l'))$ for any location l is $\mathsf{scope}(l)$
$\mathsf{scope}(\neg g)$ for guard g is $\mathsf{scope}(g)$
$\mathsf{scope}(g \vee g')$ for guards g and g' is $max(\mathsf{scope}(g), \mathsf{scope}(g'))$
$\mathsf{scope}(g = g')$ for guards g and g' is $max(\mathsf{scope}(g), \mathsf{scope}(g'))$
$\mathsf{scope}(o)$ for opcode o is 0
$\mathsf{scope}(msg.field)$ for any message msg and any message field selector $field$,equal to opc, src or dst, is $\mathsf{scope}(msg)$
$\mathsf{scope}(\texttt{msg\_at}(q, i, l))$ for any location l is $\mathsf{scope}(l)$
$\mathsf{scope}(\texttt{next}(l, l'))$ for any location l is $1 + \mathsf{scope}(l)$
$\mathsf{scope}(p)$ for any primitive p is 0

Figure 2.5. The scope function.

The labels *Guard* and *Cmd* are added to t_{ex} to improve readability. The guard of transition t_{ex} checks that a message of type a appears at the head of queue 2. If the guard is satisfied, t_{ex} moves the message to the third entry of queue 2 in the next location along the path to the message target. The scope of t_{ex} is one, since next appears once and is nested zero times.

The interpretation of Newspeak transitions is shown in Figure 2.6. A *context* C consists of a state s and a location l_p. The location l_p in the last line of Figure 2.6 is same as the *curloc* primitive of the Newspeak syntax. The interpretation of a guard/command pair $\langle g, c \rangle$, at line 2 in Figure 2.6 is, "if the guard g evaluates to true, then create a new state by interpreting the command c in the context of s. Otherwise, return s unchanged."

Three new operators, opc, src and dst, are introduced in the right side of Figure 2.6. Each of the new operators are accessor functions for messages. As might be expected, opc returns the opcode of a message, src returns the source location and dst returns the destination location. The interpretation of `next` is defined using the next operator from the network N used to instantiate $P(N)$.

$[\![expr]\!]_C = s'$ denotes the interpretation of expression $expr$
in context $C = \langle s, l_p \rangle$ which yields state s'.

$$
\begin{array}{rl}
[\![\langle g, e \rangle]\!]_C \equiv & \text{if } [\![g]\!]_C \wedge l_p \in l_E \text{ then } [\![c]\!]_C \text{ else } s \\
[\![\langle g, c \rangle]\!]_C \equiv & \text{if } [\![g]\!]_C \text{ then } [\![c]\!]_C \text{ else } s \\
[\![a; a']\!]_C \equiv & [\![a']\!]_{\langle [\![a]\!]_C, l_p \rangle} \\
[\![(q, i, l) := m']\!]_C \equiv & s[(q, i, [\![l]\!]_C, [\![m']\!]_C)/(q, i, [\![l]\!]_C, m)] \\
[\![g \vee g']\!]_C \equiv & [\![g]\!]_C \text{ or } [\![g']\!]_C \\
[\![\neg g]\!]_C \equiv & \text{not } [\![g]\!]_C \\
[\![l = l']\!]_C \equiv & [\![l]\!]_C = [\![l']\!]_C \\
[\![o = o']\!]_C \equiv & [\![o]\!]_C = [\![o']\!]_C \\
[\![\sigma]\!]_C \equiv & \sigma \\
[\![m.dst]\!]_C \equiv & \mathsf{dst}([\![m]\!]_C) \\
[\![m.src]\!]_C \equiv & \mathsf{src}([\![m]\!]_C) \\
[\![m.opc]\!]_C \equiv & \mathsf{opc}([\![m]\!]_C) \\
[\![\texttt{msg\_at}(i, q, l)]\!]_C \equiv & s(i, q, [\![l]\!]_C) \\
[\![\mathsf{next}(l, l')]\!]_C \equiv & \mathsf{next}([\![l]\!]_C, [\![l']\!]_C) \\
[\![l_p]\!]_C \equiv & l_p
\end{array}
$$

Figure 2.6. Interpretation of Newspeak transitions

State s from context C is used as a function in the interpretation of the `msg_at` operator. Recall from the discussion of network states given on page 30 that $s(i, q, l)$ returns the (possibly empty) message at position i in queue q of node l.

2.3.3 Computing Reachable States

The final elements of the protocol model are the definitions of reachable states. The network model N is included in each definition because some behaviors of $P(N)$ may depend on the structure of N.

Definition 2.4 ($\rightarrow_M$: One-step reachability) *Given a model $M = P(N)$ for network N, state s' is reachable in one step from state s iff there is a transition t such that the interpretation of t at some location l_p gives s':*

$$s \rightarrow_M s' \text{ iff } \exists t \in T,\ l_p \in L.\ [\![t]\!]_{\langle s, l_p \rangle} = s'.$$

Definition 2.5 ($\rightarrow_M^*$: Multistep reachability) *Given a model $M = P(N)$ for network N, state s' is reachable in several steps from s iff there is a sequence of states reachable in one-step from s that give s':*

$$s \rightarrow_M^* s' \text{ iff } s' = s \text{ or } \exists s''.s \rightarrow s'' \text{ and } s'' \rightarrow_M^* s'.$$

Definition 2.6 (S_M: States reachable from s_{init}) *Given a model $M = P(N)$ for network N, the set of states reachable in model M from the empty initial state s_{init} is the set of states reachable in one or more step from s_{init} and is denoted S_M:*

$$S_M = \{s \mid s_{init} \rightarrow_M^* s\}.$$

2.4 Property Model

The kinds of properties to be verified are described by *stimulus/response (SR) transition systems*. Such a transition system determines if a protocol provides a communication model among a group of n terminal nodes connected by a branching network. The determination is made by sending messages (the stimuli) between the n nodes and observing the resulting states (the response) of the n nodes. If the SR transition system observes an incorrect behavior, the transition system moves to a special error state.

To clarify the meaning of an SR transition system, an example of an SR transition system ϕ is shown in Figure 2.7. The network in Figure 2.7 contains six terminal nodes, labeled A,B,C,D,X and Y, connected by three internal nodes to form a network. The grey box around the network represents ϕ, which can observe $P(N)$ only at through terminal nodes A, B, C and D. Transition system ϕ sends messages between these nodes and observes their resulting states. For example, ϕ might require that messages sent from A to B, then from B to C and then from C to D are received at D in the same order as they were sent from A.

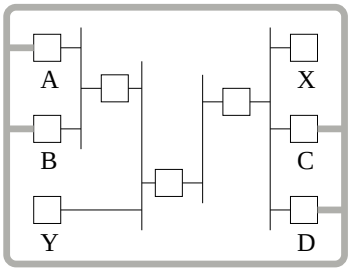

Figure 2.7. Example of an SR transition system.

More formally:

Definition 2.7 (SR transition system) *A SR transition system ϕ is a 6-tuple, parameterized by a protocol model $P(N)$, such that:*

$$\phi(P(N)) = \langle L_{Obs}, \mathsf{addr}, \Sigma_{Obs}, S_\phi, T_{Obs}, T_\phi, \rangle$$

where $P(N) = \langle Q, \Sigma, S, T, T_E \rangle$ with $N = \langle L, \mathsf{next} \rangle$.

Each element of ϕ is explained in more detail below, but briefly summarized here for convenience. The set L_{Obs} and function addr describe the terminal nodes stimulated and observed by ϕ. Message set Σ_{Obs} contains the messages sent and observed by ϕ. State set S_ϕ contains the states of $P(N)$ augmented with a finite auxiliary store. Transition relation T_{Obs} contains transitions that make changes to the observed terminal nodes depending on the state of the auxiliary store and the state of the observed terminal nodes. Transition relation T_ϕ is the union of T_{Obs} with T_E and T from $P(N)$.

The set of *observed terminal node indices*, L_{Obs}, contains a unique index, or label, for each terminal node observed by ϕ. The labels in L_{Obs} identify terminal nodes in the transitions of T_{Obs}.

Observed terminal node indices are paired with terminal nodes by the *address mapping* addr. Given an terminal node index a from L_{Obs}, $\mathsf{addr}(a)$ is the terminal

node l_e corresponding to a in L. Observed terminal nodes are specified using an address mapping and index set so that the transitions in T_{Obs} are defined independently of actual locations in L. T_{Obs} is independent so that the same transition set can be checked on different network shapes and configurations.

The set of *observed messages*, Σ_{Obs}, contains messages that are sent between observed terminal nodes. The set Σ_{Obs} is a subset of Σ such that message (opc, $\mathsf{addr}(a_1), \mathsf{addr}(a_2)$) is in Σ_{Obs} for terminal node indices a_1 and a_2. The SR transitions are limited to messages in Σ_{Obs}. In contrast, the set of nonobserved messages are messages that do not travel between observed terminal nodes.

The set of *augmented states*, S_ϕ, contains the union of S and a finite auxiliary store. The *auxiliary store* is a set of address value pairs that are used to track progress through a multistep protocol. The auxiliary state stores information that determines both the expected response and the next stimulus. Individual states in S_ϕ are identical to states in S with the addition of tuples (a, v) to states in S_ϕ. The presence of (a, v) in state $s \in S_\phi$ indicates that auxiliary address a contains value v.

The set of *SR transitions*, T_{Obs}, describes the stimuli and responses. Transitions in T_{Obs} are described using the SR transition language, which is shown in Figure 2.8. Like Newspeak, the syntax of SR enforces several limitations on transitions on T_{Obs}. Well-formed SR transitions observe and modify only the locations of observed external indices and create messages only to and from the locations of observed external indices. The *obsloc* production in Figure 2.8 guarantees both of these conditions because valid locations are built only from $\mathsf{addr}(a)$.

The interpretation of SR transitions is the same as Newspeak transitions, with four additions. The additions support the interpretation of addr function calls and auxiliary state usage. For addr function calls the interpretations

$$[\![addr(a)]\!]_{\langle s,l_p\rangle} \equiv \mathsf{addr}([\![a]\!]_{\langle s,l_p\rangle}),$$

$$
\begin{array}{rl}
\mathit{transition} & ::= \langle \mathit{guard}, \mathit{cmd} \rangle \\
\mathit{cmd} & ::= (\mathit{qloc}, \mathit{qindex}, \mathit{obsloc}) := (\mathit{opc}, \mathit{obsloc}, \mathit{obsloc}) \mid \mathit{aux} := \mathit{value} \\
 & \mid \mathsf{error} \mid \mathit{cmd}; \mathit{cmd} \\
\mathit{guard} & ::= \mathit{opc} = \mathit{opc} \mid \mathit{val}(\mathit{aux}) = \mathit{val}(\mathit{aux}) \\
 & \mid \mathit{guard} \vee \mathit{guard} \mid \neg \mathit{guard} \\
\mathit{qloc} & ::= i \in N \\
\mathit{qindex} & ::= q \in Q \\
\mathit{obsloc} & ::= \mathsf{addr}(\mathit{locindex}) \\
\mathit{opc} & ::= \sigma \in \Lambda \mid \mathit{msg}.\mathit{opc} \\
\mathit{locindex} & ::= i \in L_{Obs} \\
\mathit{msg} & ::= \mathtt{msg\_at}(\mathit{qloc}, \mathit{qindex}, \mathit{obsloc}) \\
 & \\
\mathit{primitives} & : \mathit{aux},\ \mathit{value},\ \mathtt{msg\_at},\ \mathtt{next},\ \Lambda,\ \mathsf{error} \text{ and } L_{Obs}
\end{array}
$$

Figure 2.8. Syntax of SR transitions

are included. Two interpretation rules are added to allow the access and update of auxillary state values. The first interpretation rule is similar to the interpretation of the `msg_at` command and the second rule modifies the auxillary state:

$$
\begin{array}{l}
[\![\mathit{val}(i)]\!]_{\langle s, l_p \rangle} \equiv s(i) \\
[\![a := v]\!]_{\langle s, l_p \rangle} \equiv \langle s[(a, v)/(a, v')], l_p \rangle
\end{array}
$$

where v' is the value at address a in state s.

Finally, the *combined transition system* T_ϕ is the union of T_{Obs}, T and T_E. The combined transition system represents the behavior of protocol $P(N)$ augmented with the stimuli and responses of ϕ.

An SR transition system instantiated with a protocol forms a model $M = \phi(P(N))$ in which reachable states over S_ϕ can be defined as in Definition 2.4.

2.5 Two Verification Problems

The central verification problem is to show that a protocol $P(N)$ complies with an SR transition system ϕ for all states of all networks. A protocol $P(N)$ complies with ϕ for network N if the error state is not contained in the reachable states

of $\phi(P_N)$. The most challenging aspects of this verification problem are checking compliance for *all* network configurations and modeling creative transitions.

The NPV problem includes all network configurations and all protocol creative transitions. More formally,

> **Definition 2.8 (NPV problem)** *The verification problem* NPV *is: Given a network parameterized protocol P and an SR transition system ϕ, return yes if*
>
> $$\forall N. \mathsf{error} \notin S_{\phi(P(N))}$$
>
> *and return* error *otherwise.*

Unfortunately, NPV is not decidable in general. NPV is not decidable because the halting problem for Turing machines with a tape of size n can be encoded as an instance of the NPV problem. In the encoding, protocol transitions on nodes represent Turing machine transitions on tape cells. The state of a single node represents the state of a single tape cell. The SR transition system transitions to the error state if the Turing machine halts.

The LNPV problem is a decidable subproblem of the NPV problem. The LNPV problem requires a bound on the number of messages in a state, a bound on the number of terminal nodes. The bound on the number of messages per state is obtained by considering only SR transition systems which that a finite number of messages per state and noncreative protocols. The maximum number of nodes per state for an SR transition system is denoted max_ϕ.

A protocol, $P(N)$, is noncreative if the noncreative subset of the transitions T and T_E include all transitions necessary to move messages between source and target. The noncreative subset of T and T_E contains the transitions that delete at least as many messages as they insert. If a noncreative transition, t, is applied to a state s to create a state s', then state s' will contain the same, or fewer, messages

as state s. For example, a transition that copies a message into an adjacent node, but deletes the old copy, is noncreative. A transition that copies a message into an adjacent node, but does not delete the old copy, is creative. The noncreative subset of a protocol is denoted $NC(P(N))$. LNPV is defined over noncreative subsets because the forthcoming NDFS algorithm can represent only a finite number of messages in a single abstract state.

A bound on the number of terminal nodes is obtained by limiting LNPV to networks with $|L_{Obs}|$ terminal nodes, where $|L_{Obs}|$ is the number of observed terminal node indices in ϕ. This set of networks is called the set of ϕ-limited network configurations and is denoted $N_{|L_{Obs}|}$. LNPV is defined over a finite family of network configurations because the forthcoming NDFS algorithm checks only one family of configurations per run.

The second verification problem is stated for $NC(P(N))$ and $N_{|L_{Obs}|}$:

Definition 2.9 (LNPV problem) *The verification problem* LNPV *is: Given an SR transition system ϕ and a noncreative network parameterized protocol P, return* yes *if*

$$\forall N \in N_{|L_{Obs}|}.\mathsf{error} \notin S_{\phi(NC(P(N)))}$$

and return error *otherwise.*

A precise solution for LNPV can be developed using an inductive argument on a finite instantiation, similar to prior work discussed in Section 1.2.1. Given a network parameterized protocol $P(N)$ containing t transitions each with scope s, the finite network N_F is constructed with $t \cdot (2 \cdot s + 1)$ nodes per path segments. Only $t \cdot (2 \cdot s + 1)$ nodes are needed to observe every interaction between every adjacent application of every transition. For an SR-transition system ϕ with at most max_ϕ messages in a state, each queue in each node of N_F is contructed with max_ϕ entries. Only max_ϕ entries are required to store every message in a single queue. The states

of N_F can then be ennumerated. The results can be generalized to any network in LNPV because every behavior of every transition has been observed in the nodes of N_F.

This finite instantiation solution is not pursued in this thesis because this solution suffers from the state-explosion problem. Instead, an abstraction is used that reduces both the size and number of states that must be ennumerated. This approach is similar to prior reseach based on canonical state ennumeration, as discussed in Section 1.2.3. In Chapter 8, the combination of the finite instantiation solution with a dynamic state vector state ennumeration algorithm is considered as an alternative solution the state explosion problem.

2.6 Summary

In the preceding sections, models of networks, protocols and properties were defined. A network consists of nodes that store messages in queues. The protocol model describes network states, transitions between network states and state reachability. Properties are expressed as SR transition systems that stimulate a network and observe an expected response. These models are used to define the NPV and LNPV verification problems. The solution for LNPV given in the next chapter depends on several requirements on the network, protocol and property models. These requirements were developed in the preceding models and are summarized below:

- Networks must not contain cycles. Networks with a deterministic routing function next satisfy this restriction.
- Network nodes store messages only in queues. Network parameterized protocols expressed in Newspeak satisfy this requirement.
- Protocols must be noncreative.

- Transitions must have finite branching and linear local scope. All Newspeak transitions satisfy this requirement.
- Transitions must be location independent. All Newspeak transition also satisfy this requirement.
- Properties must observe a fixed set of terminal nodes. All SR transitions satisfy this requirement.
- Properties must observe a fixed set of messages. All SR transition systems satisfy this requirement.

The set of requirements for LNPV were motivated by the properties of multibus IO protocols modeled at the transport level.

A conservative algorithmic solution for LNPV is given in the next chapter. A discussion of how the algorithm behaves in absence of each of the above requirements is given at the end of Chapter 3. Without the above requirements, the algorithm is correct if it terminates, but is not guaranteed to terminate. The termination and correctness of the algorithm is addressed in Chapter 4 and an implementation of the algorithm is discussed in Chapter 5. The generalization of LNPV to NPV for some protocols is discussed in Chapter 6.

CHAPTER 3

NDFS: A CONSERVATIVE SOLUTION FOR LNPV

The sequence of definitions in this chapter lay out a model for computation of reachable states by the NDFS algorithm wich approximates the reachable status of $\phi(NC(P(N)))$ for a network $N \in N_{|L_{Obs}|}$. An algorithm is then given for constructing all members of $N_{|L_{Obs}|}$. Taken together, both algorithms give an approximate solution to LNPV.

The NDFS algorithm uses an abstract model of network states. An abstract network is like a concrete network except an abstract network contains exactly one node per path segment. The goal of the abstract network state representation is to represent the states of topologically isomorphic concrete networks with a finite set of abstract network states.

The abstract model is followed by the construction of a partial concrete state from an abstract network state. A partial concrete state contains only a finite number of nodes per path, and may consist of only a fraction of the paths and messages contained in the abstract state. The application of Newspeak transitions to partial concrete states is then described.

The next definition describes how to merge next concrete state fragments with abstract states. Merging concrete state fragments with abstract states permits the definition of multistep reachability for abstract networks. This in turn permits the definition of the set of reachable states for an abstract network.

The NDFS algorithm enumerates the reachable states of an abstract network using the merge function and Newspeak transitions applied ot partial concrete states. NDFS terminates when an error state is found, or no new abstract states are found. The correctness and termination proofs for the NDFS algorithm are deferred to the next chapter.

A single run of NDFS generates states for a class of isomorphic networks. The NDFS algorithm achieves complete topological coverage when combined with a method for enumerating network configurations. This chapter concludes with a topology enumeration algorithm based on variants of full Steiner topologies.

3.1 Abstraction

This section contains a definition of the abstraction function and the abstract network model. The main difference between the abstract and concrete models is that path segments in the concrete model contain n nodes whereas path segments in the abstract model contain exactly 1 node. An abstract representation with single node per path is chosen so that networks with the same structure, but different path lengths, have the same abstract representation.

The abstraction function α is applied to a model $M = \phi(P(N))$ and a state s to create an abstract transition system $\hat{M}$ and an abstract state $\hat{s}$. In symbols,

$$\alpha(M, s) = \langle \hat{M}, \hat{s} \rangle.$$

Definition 3.1 (Abstract protocol model) *Abstract protocol model $\hat{M}$ consists of six components:*

$$\hat{M} = \langle \hat{L}, \hat{\mathsf{next}}, L_{Obs}, \hat{\mathsf{addr}}, \hat{S}, T_\phi \rangle$$

where $\hat{L}, \hat{\mathsf{next}}, L_{Obs}, \hat{\mathsf{addr}}, \hat{S}$ define an abstract network and its states and T_ϕ is a set of transitions.

The abstraction changes the network structure and state set, but does not change the transition relation T_ϕ taken from M. The construction of each element of $\hat{M}$ and $\hat{S}$ are given below.

The set of *abstract network locations*, $\hat{L}$, is constructed from the set of network locations L from network N of model M. If (n_i, p_j) is in L, then $\hat{p_j}$ is added to $\hat{L}$. In L, locations were modeled as tuples because a location identifies a node within a path segment. In $\hat{L}$, locations are modeled as singletons because a location identifies only a path segment.

The nondeterministic *abstract routing function* $\hat{\mathsf{next}}$ is constructed from the routing function next of N. Given two abstract locations $\hat{p}$ and $\hat{p_{dst}}$, $\hat{\mathsf{next}}$ returns either $+\hat{p_x}$ or $-\hat{p_x}$ (as explained in a moment) if there exist locations (n, p) and (n_{dst}, p_{dst}) in L such that $\mathsf{next}((n, p), (n_{dst}, p_{dst})) = (n_x, p_x)$ for some location (n_x, p_x).

The "+" and "−" annotations on the output of $\hat{\mathsf{next}}$ indicate the direction to the next location. The direction must be added explicitly to disambiguate cases in which the next location is on the same path segment (i.e., $\hat{\mathsf{next}}(\hat{p}, \hat{p_{dst}}) = \hat{p}$). This case is ambiguous because the next location between $\hat{p}$ and $\hat{p_{dst}}$ could be to either the right or the left along $\hat{p}$ and $\hat{p}$ has no notation of direction. The direction can be determined by comparing node indices for the locations returned by next. If $\mathsf{next}((n, p), (n_{dst}, p_{dst})) = (n_x, p_{dst})$, and $n < n_x$, then annotate $\hat{p_{dst}}$ with "+." Otherwise, if $n \geq n_x$ then annotate $\hat{p_x}$ with "−."

Figure 3.1 illustrates a situation in which the direction is ambiguous in the abstract model. The figure shows a concrete network on the left and an abstract network constructed by α on the right. Suppose entries for $\hat{\mathsf{next}}$ of node $(1, b)$. As shown by the labeled arrows, $\mathsf{next}((1, b), (0, a))$ is $(0, b)$ and $\mathsf{next}((1, b), (0, c))$ is $(2, b)$. Based on these two values for next, add $\hat{\mathsf{next}}(b, a) = -b$ and $\hat{\mathsf{next}}(b, c) = +b$ to $\hat{\mathsf{next}}$. Without the + and − annotations, $\hat{\mathsf{next}}(b, a)$ would be the same as $\hat{\mathsf{next}}(b, c)$ and this is incorrect because a and c lie in different directions from b.

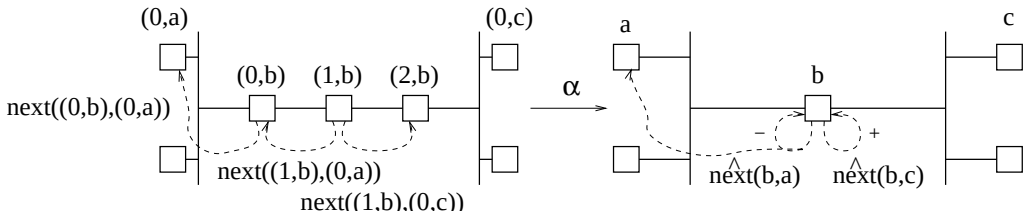

Figure 3.1. Ambiguous routing in the abstract model.

Figure 3.1 also demonstrates why $\hat{\mathsf{next}}$ must be nondeterministic. In the figure, $\mathsf{next}((0,b),(0,a))$ is $(0,a)$ and $\mathsf{next}((1,b),(0,a))$ is $(0,b)$. In the abstract domain, this means that $\hat{\mathsf{next}}(b,a)$ must equal either $-b$ or a.

The creation of $\hat{\mathsf{next}}$ from next is summarized in the following equation:

$$\hat{\mathsf{next}}(\hat{l},\hat{l}_d) = \begin{cases} +\hat{l} & \text{if } \exists n_i, n_d \in Nat. \\ & \quad \mathsf{next}((n_i,\hat{l}),(n_d,\hat{l}_d)) = (n_{i+1},\hat{l}) \\ -\hat{l} & \text{if } \exists n_i, n_d \in Nat. \\ & \quad \mathsf{next}((n_i,\hat{l}),(n_d,\hat{l}_d)) = (n_{i-1},\hat{l}) \\ \hat{l}' & \text{if } \exists n_i, n_d, n_x, \hat{l}' \in Nat. \\ & \quad \mathsf{next}((n_i,\hat{l}),(n_d,\hat{l}_d)) = (n_x,\hat{l}') \wedge \hat{l} \neq \hat{l}'. \end{cases}$$

The directional annotation is not added when $\hat{l} \neq \hat{l}'$ because it is not needed when the next location lies on a different path.

The *abstract address mapping*, $\hat{\mathsf{addr}}$ is constructed from the concrete address mapping addr. Like $\hat{\mathsf{next}}$, $\hat{\mathsf{addr}}$ returns an abstract location and an annotation. The annotation is required to determine which end of a path segment contains the observed location. If $\mathsf{addr}(a)$ for observed location index a is (n_0,p_i), then $\hat{\mathsf{addr}}(a)$ is $(0,\hat{p}_i)$, otherwise, $\hat{\mathsf{addr}}(a)$ is $(1,\hat{p}_i)$.

The final component of $\hat{M}$ is the *abstract state set* $\hat{S}$. A tuple $(i,q,\hat{l},m)$ in $\hat{S}$ indicates that position i of queue q in location l contains message m. The difference between $\hat{S}$ and S is that $\hat{S}$ maps abstract locations to values and S maps concrete locations to values. Values for the auxiliary state are not changed.

In addition to creating an abstract model, α also creates an abstract state $\hat{s}$ from the concrete state s. The abstract state is created by concatenating all messages

from each q_i in each node in path segment p_i, and storing them, in order, in queue q_i of abstract location $\hat{p_i}$. This process is similar to consolidating rush hour traffic traveling on a one-lane road with stoplights. Suppose a one-lane road contains five stoplights, all red, with four cars waiting behind each light. This represents the state of a concrete path segment with five nodes and four messages in each node. Next, suppose all of the lights suddenly disappear—except the first light. Now, all 20 cars are waiting, in the same order, at a single light. Applying α to a concrete state has the same effect as removing the four lights. In the resulting abstract state, all 20 messages are stored in the same order but in a single node.

More formally, the resulting abstract state, $\hat{s}$ contains a tuple $(i, q_j, \hat{p_k}, m)$ if and only if $(i', q_j, (n_a, p_k), m)$ is in s and the following two conditions hold. First, m is not the empty message. Second, the queue indices i and i' are related as shown in Figure 3.2. A sequence of concrete queues in a path segment are shown at the bottom of the figure with a the single abstract queue containing the same messages at the top. The relationship between i and i' is that i is i' plus the sum of the lengths of each q_j in every node in front of (n_a, p_k) in path segment p_k. This relationship holds for i, i' as shown in the figure because the sum of i and the lengths of the queues to the right is seven.

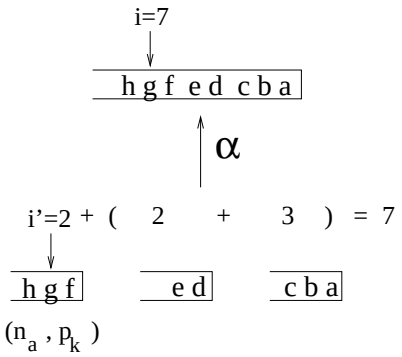

Figure 3.2. Queue positions in abstract state construction.

Neither Newspeak nor SR transitions can be applied directly to abstract states since abstract states do not have the same model of network locations. As a result, the reachable states of an abstract model will be defined using the reachable states of concrete state fragments.

3.2 Partial Concretization

Partial concretization is a key innovation used in the NDFS algorithm. The difference between a partial concrete state and a concrete state is that a partial state contains only a subset of the locations in the full concrete state. Partial concrete states fragments have two important properties: first, there are only a finite number of them represented by an abstract state, and second, they can be made sufficiently large to interpret a concrete transition. The construction of a concrete state fragment from an abstract state is described in this section along with the application of a transition to a partial concrete state.

Partial concrete states are constructed by a *partial concretization function* γ. Given an abstract model $\hat{M}$ and abstract state $\hat{s}$ γ denotes a *set* of partial concrete states. Each partial concrete state is represented by a four-tuple that contains a partial concrete machine $\check{M}$, a partial state $\check{s}$ and two index sets X and Y, in symbols:

$$\gamma\langle\hat{M}, \hat{s}\rangle = \{\langle\check{M}, \check{s}, X, Y\rangle\}.$$

The size of the set returned by γ is a function of the partial state size, which is in turn a function of the scope of the transitions in model M (this function is derived in Theorem 4.2). Given a set of globally, rather than locally, scoped transitions, γ returns an infinite set of partial concrete states. A bound on the size of the set returned by γ is precisely the motivation for restricting M to locally scope transitions.

The definition of $\check{M}$ for a single four-tuple is given below. Other four-tuples are created similarly, but with different choices of locations and messages.

Definition 3.2 (Partial concrete protocol model) *A partial concrete model $\check{M}$ consists of six components:*

$$\check{M} = \langle \check{L}, \check{\mathsf{next}}, L_{Obs}, \check{\mathsf{addr}}, \check{S}, T_\phi \rangle.$$

Where $\check{L}$, $\check{\mathsf{next}}$, L_{Obs}, $\check{\mathsf{addr}}$ and $\check{S}$ define a partial concrete network and state and T_ϕ is a set of transitions.

The construction of each element of $\check{M}$ (except L_{Obs} and T_ϕ which are just copied) is given next. Once again, it should be noted that the transition set T_ϕ from $\hat{M}$ (or M) is not changed by γ.

The *finite set of locations*, $\check{L}$ is built by inventing node indices to go with some of the path segment indices from $\hat{L}$. The resulting set of locations is organized into path segments as described by $\hat{\mathsf{next}}$. Intuitively, $\check{L}$ describes a finite set of adjacent nodes spread across one or more adjacent paths. More precisely, if location (n_i, p) is contained in $\check{L}$ then the following two conditions must be met by other locations contained in $\check{L}$.

1. First, for every $j < i$ location (n_j, p) must also be contained in $\check{L}$. This condition ensures that locations in $\check{L}$ are adjacent.

2. Second, if locations (n_k, p') and (n_i, p) are contained in $\check{L}$ then for any i, $\hat{\mathsf{next}}^i(p, p') = p''$ implies that (n_0, p'') is also contained in $\check{L}$. This condition ensures that path segments in $\check{L}$ are connected by other path segments in $\check{L}$. A bound on the number of nodes and locations in a fragment will be derived later.

The *partial concrete routing function*, $\check{\mathsf{next}}$ is synthesized from $\hat{\mathsf{next}}$. The value of $\check{\mathsf{next}}(l, l_d) = l_n$ is defined when both l_n and l are contained in $\check{L}$. Function $\check{\mathsf{next}}$ is

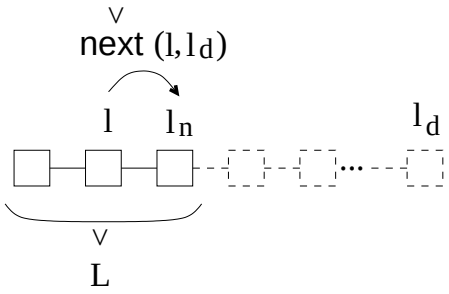

Figure 3.3. An instance in which $\check{\mathsf{next}}$ is well-defined.

defined whether or not the destination l_d is in $\check{L}$ so long as the next location l_n is in $\check{L}$. Figure 3.3 contains an instance of $\check{\mathsf{next}}$ applied to a location l in $\check{L}$ in which l_d is outside of $\check{L}$ but l_n is in $\check{L}$. All values for l_d are allowed because the next location toward l_d can be determined by evaluating $\hat{\mathsf{next}}(l, l_d)$ whether or not l_d is in $\check{L}$. Moreover, all values for l_d *must* be allowed for $\check{\mathsf{next}}$ because Newspeak transitions allow the application of next to message source and destination locations, which may not be contained in $\check{L}$.

The definition of $\check{\mathsf{next}}(l, l_d)$ is split into four cases which depend on the locations in $\check{L}$ and the value of $\hat{\mathsf{next}}$. Intuitively, there are two cases for each direction from l to l_d: $\check{\mathsf{next}}(l, l_d)$ is the next node along path segment p—unless n_i is the last node on path segment p contained in $\check{L}$, in which case $\check{\mathsf{next}}(l, l_d)$ is the next adjacent node in the next adjacent path. Figure 3.4 shows these two cases for the "$-$" direction. An abstract network is shown at the top of Figure 3.4 and a partial concrete network is shown at the bottom. In case 1, $\check{\mathsf{next}}((1, b), (0, a))$ is $(0, b)$ because $(1, b)$ is not the last node in path segment b and $\hat{\mathsf{next}}(b, a)$ is $-b$. In case 2, however, $\check{\mathsf{next}}((0, b), (0, a))$ is $(1, a)$ because $(0, b)$ *is* the last node on path segment b and $\hat{\mathsf{next}}(b, a)$ is a. More formally, let $l = (n_i, p)$ with $l_d = (n_d, p_d)$ and:

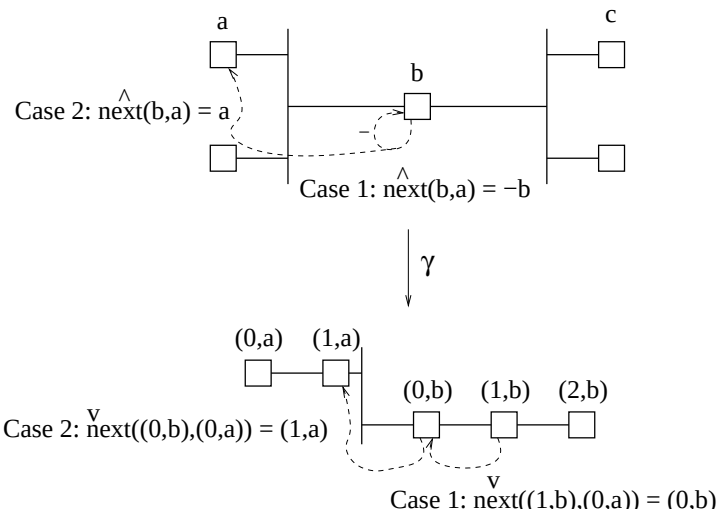

Figure 3.4. Interpreting next in partial concrete states.

$$\check{\mathsf{next}}((n_i,p),(n_d,p_d)) = \begin{cases} (n_{i-1},p) & \text{if } (n_{i-1},p) \in \check{L} \wedge \hat{\mathsf{next}}(p,p_d) = -p \\ (n_k,p'') & \text{if } i = 0 \wedge \hat{\mathsf{next}}(p,p_d) = p'' \wedge p \neq p'' \\ & \wedge (n_k,p'') \in \check{L} \wedge (n_{k+1},p'') \notin \check{L} \\ (n_{i+1},p) & \text{if } (n_{i+1},p) \in \check{L} \wedge \hat{\mathsf{next}}(p,p_d) = +p \\ (n_0,p'') & \text{if } (n_{i+1},p) \notin \check{L} \wedge \hat{\mathsf{next}}(p,p_d) = p'' \\ & \wedge p \neq p'' \wedge (n_0,p'') \in \check{L} \end{cases} \tag{3.1}$$

The two cases shown in Figure 3.4 correspond to the first two cases of Equation 3.1.

Next, the *partial concrete address mapping function* $\check{\mathsf{addr}}$ is defined using $\hat{\mathsf{addr}}$. Suppose a partial concrete state $\check{s}$ contains the end point node n_0 for some path segment p and that the abstract address $\hat{\mathsf{addr}}$ of some observed location index a is $(0,p)$. Even though n_0 is the end of path segment p for $\check{s}$, n_0 does not necessarily represent the end node of path segment p in the full concrete state. Consequently, $\check{\mathsf{addr}}$ maps observed location index a to either (n_0,p) or (n_{-1},p). Similarly, if (n_i,p) is an end point for p in $\check{s}$ and $\hat{\mathsf{addr}}(a) = (1,p)$ then $\check{\mathsf{addr}}$ maps a to either (n_i,p) or (n_{i+1},p)—even if (n_{i+1},p) is not contained in $\check{L}$.

The last element of $\check{M}$ is the *set of concrete state fragments*, $\check{S}$. The set $\check{S}$ is the set of mappings between locations in $\check{L}$ and power sets of the messages in abstract state $\hat{s}$. The states in $\check{S}$ have three important properties. First, $\check{S}$ is always finite

for an abstract state. This will be proven in the next chapter. Second, $\check{S}$ has the same structure as S. Since $\check{S}$ and S have the same structure, transitions from T_ϕ can be applied to $\check{S}$. Third, the auxiliary state tuples are all included in $\check{S}$.

The sets X and Y contain indices that indicate which portion of abstract state $\hat{s}$ was taken to create $\check{s}$. Index $x_{i,j}$ is included in X if the xth message of queue q_i in path segment p_j of abstract state $\hat{s}$ is the first message in node (n_0, p_j) of $\check{s}$. Index $y_{i,j}$ is included in Y if y messages from queue q_i in path segment p_j of abstract state $\hat{s}$ are distributed in the queues labeled q_i in the nodes of path segment p_j in $\check{s}$.

The construction of a partial concrete state is shown in Figure 3.5. An abstract state $\hat{s}$ is shown at the top of the figure with a partial concrete state $\check{s}$ at the bottom. In this partial state, x (shown without subscripts for simplicity) is five because the fifth message from $\hat{s}$ is the first message in $\check{s}$ and y is four because four messages from $\hat{s}$ are included in $\check{s}$.

The final task in defining $\check{s}$ is to distribute the messages in queue locations. The only restriction on message distribution is that messages must appear in the same order, queue and path segment as they appear in $\hat{s}$. The distribution in Figure 3.5 is valid if the first message from $\hat{s}$ appears in the first queue of $\check{s}$ and the remaining three messages appear in order in the second queue.

The second purpose for defining concrete state fragments was to create a model

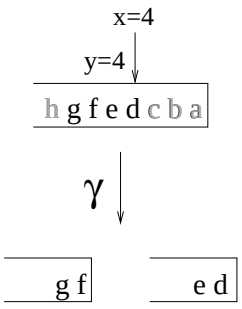

Figure 3.5. Queue positions in partial concrete state construction.

in which concrete transitions can be interpreted. For this purpose, the partial concrete state have the same structure as a concrete state, and the partial concrete state must contain enough nodes to exceed the transition scope. The *depth* of a partial concrete state $\check{s}$ measures the number of nodes along a sequence of one or more paths in $\check{s}$. Returning briefly to Figure 3.4 (page 52), the depth of the partial concrete state at the top of Figure 3.4 is four since locations $(0, a), (1, a)$ and $(0, b)$ through $(2, b)$ form a linear sequence. In general, if $\mathsf{scope}(t) = n$ for transition t, then t can be applied to a partial concrete state with a depth of $2n + 1$. A depth of $2n + 1$ is required because t might apply next n times along a path segment in either direction.

Assuming partial concrete state $\check{s}$ has sufficient depth, the interpretation of t in $\check{s}$ is straighforward. The interpretation of t is exactly as given in Figure 2.6 except $\mathsf{next\ addr}$ and s are replaced by $\check{\mathsf{next}}\ \check{\mathsf{addr}}$ and $\check{s}$.

Transitions are applied to concrete state fragments only during the synthesis of transitions between abstract state traces. Consequently, only single-step reachability is required for partial concrete states.

> **Definition 3.3 (Reachable partial concrete states)** *Given a partial concrete model $\check{M}$, state $\check{s}'$ is reachable in one step from state $\check{s}$ iff there is a transition t such that the interpretation of t at a location in $\check{L}$ gives $\check{s}'$, in symbols:*
>
> $$\check{s} \rightarrow_{\check{M}} \check{s}' \text{ iff } \exists t \in T_\phi,\ \check{l_p} \in \check{L}.\ [\![t]\!]_{\langle \check{s}, \check{l_p} \rangle} = \check{s}'.$$

Definition 3.3 will be incorporated with an abstraction/merge function, given next, to define multistep reachability for traces of abstract states.

3.3 Abstract State Reachability

This section contains a description of how to merge $\check{s}'$ into abstract state $\hat{s}$ to create a new abstract state $\hat{s}'$. The abstraction/merge function μ takes a partial concrete state $\check{s}'$, a pair of index sets X and Y, an abstract model $\hat{M}$ and an abstract state $\hat{s}$. Given these inputs, μ evaluates to an abstract model $\hat{M}$ and a new abstract state $\hat{s}'$, in symbols:

$$\mu(\check{s}', X, Y, \hat{M}, \hat{s}) = \langle \hat{M}, \hat{s}' \rangle.$$

The new abstract state $\hat{s}'$ is created by replacing the contents of $\hat{s}$ between x and $x + y$ with $\check{s}'$. For each queue j and path segment l, the messages before and after $x_{j,l}$ and $y_{j,l}$ are copied directly from $\hat{s}$. The messages between $x_{j,l}$ and $x_{j,l} + y_{j,l}$ are copied from $\check{s}'$.

An example of this construction is shown in Figure 3.6. Abstract state $\hat{s}$ appears at the top left of the figure above a partial concrete state $\check{s}'$. The new abstract state appears at the right. In the figure, messages from $\check{s}'$ are shown using rectangles, messages before x in $\hat{s}$ are shown using downward facing triangles and messages after $x+y$ are shown using upward facing triangles. The circular messages between x and $x+y$ in $\hat{s}$ are not included in $\hat{s}'$ but are replaced with the rectangular messages from $\check{s}'$. Although four circular messages were originally taken from $\hat{s}$ to create $\check{s}$, the new partial concrete state $\check{s}'$ only contains three rectangular messages because the transition that created $\check{s}'$ from $\check{s}$ deleted a message from $\check{s}$. For convenience in the following equation, the expression $\lambda(j, l)$ denotes the sum of the lengths of queues labeled q_j in path segment p_l. In symbols, the values of each tuple in $\hat{s}'$ created by μ of $\hat{s}$ and $\check{s}'$ are:

$$\hat{s}'(i, q_j, p_l) = \begin{cases} \hat{s}(i, q_j, p_l) & \text{if } i < x_{j,l} \\ \check{s}(b, q_j, n_k, p_l) & \text{if } x_{j,l} < i \leq \lambda(j,l) \wedge \\ & \quad i = b + \sum_{a<l} \mathsf{Qlength}(q_j, (n_a, p_l), \check{s}) \\ \check{s}(i + y_{j,l} - & \text{if } x + \lambda(j,k,l) < i \\ \quad \lambda(j,l), q_j, p_k) & \end{cases}$$

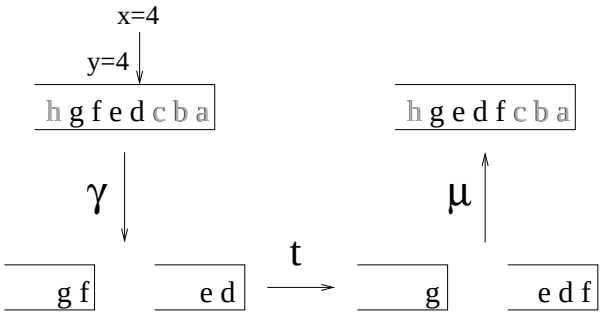

Figure 3.6. Creation of a new abstract state.

Referring to Figure 3.6, the first case corresponds to the downward triangles; the second case corresponds to the squares and the third case corresponds to the upward triangles.

The preceding definitions of α, γ and μ lead to the following definition of n-step reachability for abstract states. First, one-step reachability for abstract states is defined in terms of one-step reachability for concrete state fragments. Then n-step reachability is defined for abstract states.

Definition 3.4 ($\rightarrow_{\hat{M}}$: Abstract one-step reachability) *Given an abstract model $\langle \hat{M}, \hat{s} \rangle = \alpha\langle M, s \rangle$ abstract state $\hat{s'}$ is reachable in one step from $\hat{s}$ iff there is a concrete state fragment $\langle \check{s}, X, Y \rangle$ and transition t such that abstraction/merge of the interpretation of t in $\check{s}$ gives $\check{s'}$, in symbols:*

$$\hat{s} \rightarrow_{\hat{M}} \hat{s'} \text{ iff } \exists \langle \check{M}, \check{s}, X, Y \rangle \in \gamma\ \langle \hat{m}, \hat{s} \rangle. \\ \check{s} \rightarrow_{\check{M}} \check{s'} \wedge \mu(\check{s'}, X, Y, \hat{M}, \hat{s}) = (\hat{M}, \hat{s'}.)$$

Definition 3.5 ($\rightarrow^*_{\hat{M}}$: Abstract multistep reachability) *Given an abstract model $\langle \hat{M}, \hat{s} \rangle = \alpha\langle M, s \rangle$ abstract state $\hat{s'}$ is reachable in several steps from $\hat{s}$ if there is a sequence of reachable states starting with $\hat{s}$ leading to $\hat{s'}$, in symbols:*

$$\hat{s} \rightarrow^*_{\hat{M}} s' \textit{ iff } \hat{s} \rightarrow_{\hat{M}} \hat{s}_1 \rightarrow_{\hat{M}} \hat{s}_2 \ldots \hat{s}'.$$

Definition 3.6 ($\hat{S}_{\hat{M}}$: Set of abstract states reachable from $\hat{s}_{init}$) *The entire set of states reachable in abstract model $\hat{M}$, starting from the empty initial abstract state $\hat{s}_{init}$, is denoted $\hat{S}_{\hat{M}}$*

$$\hat{S}_{\hat{M}} = \{\hat{s} \mid \hat{s}_{init} \rightarrow^*_{\hat{M}} s\}.$$

The abstract machine $\hat{M}$ is unchanged through the entire process of computing the next reachable state. Consequently, $\hat{M}$ is the same for the entire set of states in $S_{\hat{M}}$. $\hat{M}$ is kept explicit in the notation for simplicity, but $\hat{M}$ is not included in the algorithm for computational efficiency.

The process of making transitions between abstract states is summarized in Figure 3.7. In this figure, abstract states $\hat{s}$, $\hat{s}_2$ and $\hat{s}_3$ appear at the top of the figure and several partial concrete states appear at the bottom. The arrow between $\hat{s}$ and $\hat{s}_2$ is drawn using a dashed line because the transition between $\hat{s}$ and $\hat{s}_2$ is computed indirectly by a transition applied to a partial concrete state $\check{s}$. After t is applied to $\check{s}$ to create $\check{s}'$, $\check{s}'$ and $\hat{s}$ are merged by μ to create $\hat{s}_2$. Next, abstract state $\hat{s}_2$ can transition to $\hat{s}_3$ through a similar sequence of steps involving partial concrete state $\check{s}_2$ created from $\hat{s}_2$ by γ. The process may be repeated *ad infinitum*, but, as will be shown later, only n unique states will be generated.

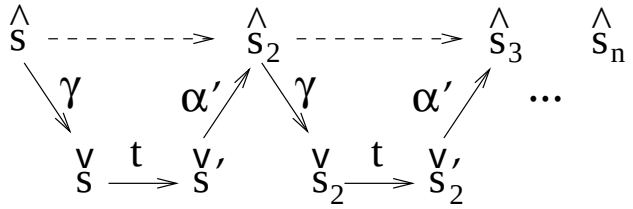

Figure 3.7. Transitions between abstract states.

```
1   Algorithm NDFS (M̂)
2   let ⟨L̂, n̂ext, L_O, âddr, Ŝ, T_φ⟩ = M̂
3     Explored = Empty Set
4     Stack = Empty Stack
5     push ŝ_init onto Stack
6       while (not empty(Stack)) do
7       ŝ = top (Stack)
8       NewStates = Empty Set
9         for each ⟨M̌, š, X, Y⟩ ∈ γ⟨M̂, ŝ⟩ do
10        Explored = Explored ∪ ŝ
11        for each t ∈ T_φ do
12          š′ = [[t]]_⟨š, l̂̌_p⟩
13          if š′ = error dump Stack, report error and quit
14          ŝ′ = μ(š, X, Y, M̂, ŝ)
15          if (ŝ′ ∉ Explored) then NewStates = NewStates ∪ ŝ′
16        endfor
17      endfor
18      if (empty(NewStates)) then
19        pop Stack
20      else
21        for each x̂ ∈ NewStates do
22          push x̂ onto Stack
23        endfor
24    endwhile
25  report no errors found
```

Figure 3.8. NDFS model checking algorithm.

3.4 Verification Algorithm

A suitable context in which to define the NDFS algorithm has now been defined. The pseudo-code for NDFS appears in Figure 3.8. NDFS computes a set of reachable abstract states by considering every transition applied to every partial concrete state represented by every reachable abstract states. NDFS terminates when either an error state is reached, or there are no more new abstract states.

The reachable state computation, at lines 9 through 16 is novel because concrete

transitions are applied to partial concrete states to generate abstract states. Other than lines 9 through 16, NDFS is the standard depth-first state ennumeration algorithm. Using concrete transitions in the abstract state computation avoids the construction of an abstract transition relation. Other algorithms [60, 43] apply transitions to concretized abstract states, but NDFS is the first to apply transitions to *partial* concrete states. Using partial concrete states is essential because a single abstract state represents an unbounded number of full concrete states (since empty nodes can be added anywhere in the newtork).

Each run of NDFS generates the reachable abstract states for a single family of topologically isomorphic network configurations. Coverage for topologically nonisomorphic network topologies requires several runs of NDFS. An algorithm for ennumerating all nonisomorphic topologies on n-node acyclic networks is given in the next section.

Recall from the definition of the LNPV problem (Definition 2.9) that the number of messages in a network state and the number of terminal nodes in network configuration are bounded for LNPV. If this bound did not exist, then the set of reachable abstract states computed at line 15 might grow without bound and NDFS may not terminate (unless a violation is found). Without a bound on the number of terminal nodes, the entire LNPV could not be checked for all network configurations because NDFS considers only one network configuration per run. In the absence of either bound, if NDFS terminates, then NDFS will terminate with an over-approximation.

NDFS imposes a serialization of events on the network state. A network is a concurrent computing environment in which several transitions may change the network state in parallel. Unless the protocol being checked is serializable, the serialization imposed by NDFS may avoid reaching otherwise reachable states. The burden of showing that a protocol is serializable is left to the user. However,

most IO protocols are serializable since they require exclusive access to a shared communication medium to modify the state of a neighboring node. The problem of showing that a protocol is serializable is endemic to using interleaving semantics to model concurrency and is beyond the scope of this thesis.

3.5 Topological Case Split

This section contains the case split which generates all topologically unique acyclic networks shapes for i terminal nodes. Each shape can be paired with a transition set and an observed address mapping to create an abstract model. Running NDFS for each case yields complete coverage for all acyclic networks of i terminal nodes, as required for LNPV.

To generate networks with terminals, the case split ennumerates variants of full Steiner topologies on $n \leq i$ nodes. A Steiner topology [53] is a Steiner tree without the associated geometrical embedding. A *Steiner tree* is a graph (V, E) with vertex set V and edge set E. Usually, Steiner trees are embedded in a two-dimentional Euclidian geometry so that path length is the geometric distance between two points. The vertices are partitioned into terminals and Steiner points. The terminals are connected by a set of edges with minimal length. Steiner points are added between terminals to minimize path lengths. A *Steiner topology* models only the number and degree of vertices, and does not include a geometric notion of length. A full *Steiner topology* [53] includes the maximal number of Steiner points that can be added to the corresponding Steiner tree.

Full Steiner topologies are convenient starting point for ennumerating acylic network topologies for several reasons. First, full Steiner topologies are acylic (as are all Steiner topologies). Second, terminal nodes in full Steiner topologies always have degree 1. Finally, combinatorics for full Steiner topologies are well-studied in other fields, such as phylogenitic systematics.

Since not all acyclic networks are also Steiner topologies, we will need a way to create nonSteiner network topologies. For acyclic networks with i terminal nodes, this is done by ennumerating full Steiner topologies with $i < n$ terminal nodes then connecting each of the remaining $n-i$ terminal nodes to a Steiner point. This gives the following construction:

1. For all n such that $2 < n \leq i$, build all full Steiner topologies on n terminals.

2. For each full Steiner topology on n terminals, connect each of the remaining $i - n$ terminal nodes to any of the $n - 2$ Steiner points.

In step 1, n is greater than two because Steiner topologies for n is less than or equal to two contain no Steiner points. Steiner points are needed in step 2 to connect additional terminals to the network. The case in which $i = 2$ is the topology most commonly studied in parameterized system verification.

The construction of two elements from the set of six-node acyclic networks, N_6, from a full Steiner topology of four terminal nodes is shown in Figure 3.9. The full Steiner topology over four terminals, fS_4, is shown at the top of the figure. Given fS_4, the next step is to add two edges to fS_4 to create four networks from N_6. The four resulting networks, C_1 through C_4, are shown at the bottom of the figure. Although four networks can be built from fS_4, only two are nonisomorphic. Network C_3 is just a rotated copy of network C_1 and network C_4 is just a rotated copy of C_2.

3.6 Summary

The conservative solution to LNPV is based on an abstract state ennumeration algorithm and a case split on network topology. Abstract states are created by an abstraction α that eliminates node boundaries within path segments. Abstract state successors are computed by applying concrete Newspeak transitions to fragments of

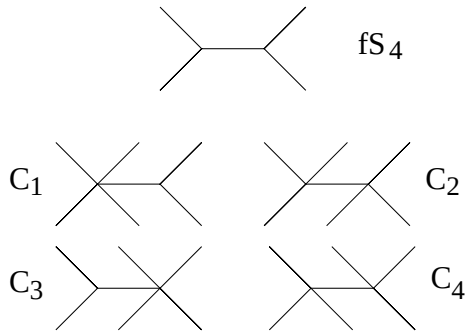

Figure 3.9. Constructing members of N_6.

concrete states represented by an abstract state. The case split on network topology ennumerates variants of full Steiner topologies to generate a complete set of unique topologies over n terminal nodes.

The conservative solution to LNPV depends on the limitations on the protocol, network and property models. The limitations is summarized below along with a brief description of the motivation for each limitation.

- Networks must not contain cycles. Networks of n terminals with cycles can not be divided into a finite set of network topologies since a new cycle can be added to any topology to create a new topology with the same number of terminals. The NDFS algorithm will terminate for networks with a finite set of cycles, but the topology case split will generate incomplete coverage of the topology space.

- Networks must store messages in unbounded queues. Unbounded queues simplify the costruction of partial concrete states. Properties that depend on queue lengths can not be reliably checked with an unbounded queue model. Adding queue lengths to the concrete model would require modification of the γ function used at line 9 of NDFS.

- Protocols must be noncreative. Noncreative protocols are essential to the termination of NDFS. If protocols could create messages arbitrarily, then abstract states could grow without bound. The number of abstract states and the number of concrete state fragments per abstract state would also grow without bound and loops at lines 6 of NDFS and 9 would not terminate. Given a creative model, NDFS will find all errors in the states it ennumertes—but may not ennumerate all of the states.

- Transitions must have finite branching and linear local scope. Finite scope is essential to the termination of NDFS. Without finite local scope, the number of concrete state fragments ennumerated by γ at line 9 of NDFS would be unbounded. As before, NDFS will find all errors in states in generates for a globally scoped protocol, but NDFS may not generate all states.

- Transitions must be location independent. The abstraction and concretization of network states looses the precise location identity of a node. The location of a node can not be reintroduced in a partial concrete state becuase there are an unbounded number of locations reprsented in a single abstract state. The NDFS algorithm will not detect errors that depend on the behavior of protocols with location dependant transitions.

- Properties must observe a fixed set of terminal nodes. Each terminal node and the paths between terminal nodes a represented explicitely in the abstract model. The number of paths is a function of the number of terminal nodes so the number of terminal nodes must be bounded. Each terminal node and path must be explicitely represented in the abstract model to allow the detection of topoplogy dependant errors.

- Properties must observe a fixed set of messages. Messages are represented explicitly in both the abstract and concrete states. Without a bound on the number of messages observed by a property, abstract states could grow without bound. This restriction implies that the SR transition system creates a finite number of messages, the noncreativity restriction implies that the protocol doesn't add any new messages to this set.

The termination and correctness of NDFS are treated more formally in the next chapter.

CHAPTER 4

TERMINATION AND CORRECTNESS OF NDFS

The claims that NDFS terminates and terminates with the correct answer are substantiated in this chapter. This chapter contains no new definitional material required to understand the remainder of the thesis and may be skipped by the uninterested reader.

The termination of NDFS depends on the number and size of abstract states, network topologies and concrete state fragments. Bounds on each these elements depend on the number of messages in a network at a given time (denoted max_{ϕ}), the finite scope of Newspeak transitions and the number of topologically isomorphic acyclic networks over $|L_{Obs}|$ terminals (which is a function of the number of terminal nodes).

The second task is to show that NDFS is correct. The correctness requirement for NDFS is conservative over-approximation. A verification alogirhtm is a convservative over-approximation if the algorithm generates a set of abstract states that represent a super-set of the actual reachable states. The correctness arguement depends on the branching local scope and location independence of Newspeak transitions and the definitions of α, γ and μ. Because NDFS uses an abstraction that looses important information regarding node boundaries, it can not be shown that NDFS is a exact solution for LNPV. This point is highlighted at key places in the correctness proof.

An implementation of NDFS is given in the next chapter. The implementation was used to check two properties of certain multibus PCI networks.

4.1 Termination Proof

The termination proof is split into three termination arguement. Each termination arguement demonstrates that a certain aspect of the NDFS algorithm terminates.

4.1.1 Finite Number of Topologically Uniuqe Networks in N_i

Recall that the set N_i contains all networks with exactly i terminal nodes. Although N_i contains an unbounded number of networks, N_i contains only a finite number of network topologies. There are many instances of each topology since networks may contain an arbitrary number of nodes per path segment. In the abstract network model, however, each path segment contains a single node. Since the abtract network model includes a single node per path segment, abstract networks are distinguishable only by their topology. In this section, the number of unique topologies is shown to be finite for acyclic networks on n nodes.

The algorithm for ennumerating every network shape in N_i for a given i yields a bound on the number of network shapes. Members of N_i are constructed from full Steiner topolgies. Four properties of full Steiner topologies over n terminals (from [53]) are of interest:

- a full Steiner topology contains exactly $n-2$ Steiner points,
- a full Steiner topology contains exactly $2n-3$ edges,
- every terminal in a full Steiner topology has degree 1, and
- for $n > 1$ terminals, there are $f(n)$ unique full Steiner for $f(n)$:

$$f(n) = 2^{-(n-2)}(2n-4)!/(n-2)!. \tag{4.1}$$

Using this construction and $f(n)$ from Equation 4.1, we have the following:

Theorem 4.1 *The number of unique acyclic network shapes in N_i is less than*

$$F(i) = \sum_{n=3}^{i} f(n)(n-2)^{i-n} \tag{4.2}$$

Each term in $F(i)$ is described in detail. The term $f(n)$ gives the number of full Steiner topologies over n terminals. For each topology, there are $n-2$ Steiner points on which to attach the remaining $i-n$ edges needed to create a member of N_i. Since more than one edge may be added at each Steiner point, there are $(n-2)^{i-n}$ variants of each full Steiner topology. The sum is taken for $n = 3...i$ because any full Steiner topology with fewer than i terminals may be used to create networks in N_i.

LNPV is limited to $N_{|L_{Obs}|}$ in order to obtain an upper-bound on i in Equation 4.2. In general, a bound on i does not exist because it is always possible to build a network with $i+1$ terminals for any i. Picking i to be $|L_{Obs}|$ gives a set of network topologies that can be easily extended to all network topologies for some protocols. These topologies are described in Chapter 6.

In later proofs, the maximum number of path segments in a network of i terminals is needed. Suppose network x contains i terminals and is constructed from a full Steiner topology with n terminals where $2 < n \leq i$. The full Steiner topology embedded in x contains $2n-3$ edges. The remaining $i-n$ edges are added to complete the construction of x. This results in a graph with $(2n-3)+(i-n)$ edges. The number of edges is maximized when n equals i, so that:

Corollary 4.1 *The maximum number of path segments in any network with i terminals is:*

$$2i - 3. \tag{4.3}$$

4.1.2 Finite Number of Concrete State Fragments

Next, it is shown that γ produces a finite number of concrete state fragments for an abstract state in an instance of LNPV.

Theorem 4.2 *Given an abstract state $\hat{s} \in \hat{S}_{\hat{M}}$ generated by an abstract model $\hat{M} = \alpha\langle\phi(NC(P)_N)\rangle$, where $N \in N_{|L_{Obs}|}$ the number of partial concrete states containing n locations in the partial concretization of $\hat{s}$ has the following bound:*

$$|\gamma(\hat{M}, \hat{s})| \leq \sum_{i}^{max_\phi} (max_\phi - i + 1) \cdot n^i \tag{4.4}$$

Because $NC(P(N))$ never creates new messages, at most max_ϕ messages can appear in any abstract state. Each concrete state fragment contains up to i adjacent messages taken from the set of max_ϕ messages. There are $max_\phi - i + 1$ ways to chose i adjacent messages from a state with max_ϕ messages. In a concrete state fragment of depth n, each of the i messages are then distributed into n nodes. There are n^i ways to partition i messages in n nodes since any of the i messages may appear in the same node.

The bound on n follows from the limited scope of transitions and the number of paths in a branching network. Recall that γ constructs concrete state fragments with enough locations in which to interpret a locally scoped transition. For a protocol with scope x, applied to a network y, with z terminals, γ generates concrete state fragments with at most $(2z+1)x$ locations. Fragments with $(2z+1)x$ locations are sufficient because a transition may view the network state of at most x locations along at most $2z + 1$ paths (Corollary 4.1) in any direction.

The bound in Equation 4.4 includes all distributions of messages across all path segments—not just the *valid* distributions of messages across path segments. A valid distribution places adjacent messages in adjacent path segments. These properties of valid distributions are not considered in the derivation of Equation 4.4

because counting just the valid distributions produces a lower bound which is not needed for a termination proof.

If max_ϕ does not exist, then Equation 4.4 would not be bounded and NDFS will not terminate unless an error is found. The resulting error state would still represent a potential violation of ϕ. Without max_ϕ, NDFS becomes a semidecision procedure rather than an algorithm.

4.1.3 Finite Number of Abstract Reachable States

The final element of the NDFS termination proof is a bound on the number of unique abstract states.

Theorem 4.3 *For any abstract model* $\langle \hat{M}, \hat{S} \rangle = \alpha(\phi(NC(P(N))))$*, where* $N \in N_{|L_{Obs}|}$ *and* n_{aux} *is the number of auxiliary states in* ϕ*, the number of unique reachable abstract states in* $\hat{M}$ *is bound by the following equation:*

$$|\hat{S}_{\hat{M}}| \leq (\sum_{m=0}^{max_\phi} (2|L_{Obs}| - 3)^m \cdot m!) \cdot n_{aux}. \tag{4.5}$$

Since T is restricted to noncreative transitions in $NC(P)$, each state in $\hat{S}_{\hat{M}}$ contains at most max_ϕ messages. Using the upper bound established in Corollary 4.1, subsets of these messages are partitioned between into at most $2|L_{Obs}| - 3$ path segments, and ordered within each path segment. There are $(2|L_{Obs}| - 3)^m$ ways to partition m messages between $2|L_{Obs}| - 3$ path segments. There are $m!$ ways to order m messages within each path segment. Finally, there are n_{aux} auxiliary states paired with any network state.

Use of the $m!$ term assumes all m messages are places in a single path segment in every abstract state. To be more precise, $m!$ should be replaced by $m_1! + \ldots m_j!$, in which m_i messages are contained in path segment i, and $m_1 + ...m_j = m$. However, since $m_1! + ...m_j! < m!$, the less precise, but simpler, $m!$ term is used.

4.2 Approximation Proof

The abstraction, partial concretization and merge are constructed so that the abstraction of any state created by interpreting transition t in state s is the same as μ applied to the concrete state fragment created by interpreting t in a fragment created by γ applied to the abstraction of s. In symbols (briefly abusing notation for clarity):

$$(\alpha(\ t\ s)) = (\mu(\ t\ (\gamma\ (\alpha\ s)))). \tag{4.6}$$

Equation 4.6 is used in the proof of the following theorem, which asserts that NDFS generates a set of abstract states that contain the abstraction of every reachable concrete state.

Theorem 4.4 *For any model $M = \phi(NC(P(N)))$ where network $N \in N_{|L_{Obs}|}$, let $\langle \hat{M}, \hat{s_{init}} \rangle = \alpha\langle M, s_{init}\rangle$. For any reachable state $s \in S_M$, there exists an abstract reachable state $\hat{s} \in \hat{S}_{\hat{M}}$ such that $\alpha\langle M, s\rangle = \langle \hat{M}, \hat{s_{init}}\rangle$. In symbols:*

$$\begin{aligned} &\forall\ M = \phi(NC(P(N))).\ \langle \hat{M}, \hat{s}_{init}\rangle = \alpha\langle M, s_{init}\rangle \wedge N \in N_{|L_{Obs}|} \Rightarrow \\ &\quad \forall\ s \in S_M.\ \exists\ \hat{s} \in \hat{S}_{\hat{M}}.\ \alpha\langle M, s\rangle = \langle \hat{M}, \hat{s}\rangle. \end{aligned}$$

Since state s and abstract state $\hat{s}$ are reachable, there is a sequence of transitions that lead from s_{init} to s and from $\hat{s_{init}}$ to $\hat{s}$:

$$s_{init} \rightarrow_M^* s \text{ and } \hat{s}_{init} \rightarrow_{\hat{M}}^* \hat{s}.$$

The proof is done by induction on the length of the transition sequence from s_{init} to s and $\hat{s}_{init}$ to $\hat{s}$.

The basis case requires showing that $\alpha\langle M, s_{init}\rangle = \langle \hat{M}, \hat{s}_{init}\rangle$ and $\hat{s}_{init}$ is in $\hat{s}_{\hat{M}}$. This equality holds because both s_{init} and $\hat{s}_{init}$ are defined to be empty states and the abstraction of an empty state is the abstract empty state. The reachable abstract state $s \in \hat{S}_{\hat{M}}$, contain the empty initial state by definition (see Definition 3.4).

The proof of the inductive step requires showing that for any $s \in S_M$ and $\hat{s} \in \hat{S}_{\hat{M}}$ where $\langle \hat{s}, \hat{M} \rangle = \alpha\langle M, s \rangle$:

$$\forall s \in S_M.\ s \rightarrow_M s' \text{ implies } \exists \hat{s} \in \hat{S}_{\hat{M}}.\ \hat{s} \rightarrow_{\hat{M}} \hat{s}' \tag{4.7}$$

Equation 4.7 is proven in four steps:

1. Expand the definitions of reachability for concrete and abstract states,

2. Construct a witness, including a concrete state $\check{s}$, for the resulting existentially quantified variables and show these values are constructed by γ as used in NDFS,

3. Show that the interpretation of any t in both $\check{s}$ and s is the same if $\check{s}$ is equivalent to s within the scope of t modulo node indices, which differ by a constant.

4. Show that μ recombines $\check{s}$ with $\hat{s}$ to create a state which is equal to the abstraction of the next state s' created by t applied to s.

The final step is the most difficult and depends on the finite branching scope and location independence of Newspeak transitions. The proof is given in more detail in the remainder of this chapter and may be skipped on first reading since no new definitional material is contained in the proof.

First, begin by expanding $\rightarrow_M$ and $\rightarrow_{\hat{M}}$ as described in definitions 2.4 and 3.4. If $s \rightarrow_M s'$ then

$$\exists t \in T, l \in L. [\![t]\!]_{\langle s,l \rangle} = s'.$$

If $\hat{s} \rightarrow_{\hat{M}} \hat{s}'$, then

$$\begin{array}{l} \exists \langle \check{M}, \check{s}, X, Y \rangle \in \gamma\ \langle \hat{m}, \hat{s} \rangle. \\ \check{s} \rightarrow_{\check{M}} \check{s}' \wedge \mu(\check{s}', X, Y, \hat{M}, \hat{s}) = (\hat{M}, \hat{s}'). \end{array} \tag{4.8}$$

Expanding $\rightarrow_{\check{M}}$ using Definition 3.3,

$$\begin{array}{l} \exists \langle \check{M}, \check{s}, X, Y \rangle \in \gamma \ \langle \hat{m}, \hat{s} \rangle. \\ \exists t' \in T, \check{l_p} \in \check{L}. \\ [\![t]\!]_{\langle s, \check{l_p} \rangle} = \check{s}' \wedge \mu(\check{s}', X, Y, \hat{M}, \hat{s}) = (\hat{M}, \hat{s}'). \end{array} \tag{4.9}$$

The next step is to provide witnesses for Equation 4.9. First, choose $t' = t$ and denote the scope of t as $|t|$. Next, choose a particular $\langle \check{M}, \check{s}, X, Y \rangle$ from $\gamma \langle \hat{M}, \hat{s} \rangle$ such that the interpretation of t in $\check{s}$ mimics the interpretation of t in s within an offset.

This is done by building $\check{s}, X$, and Y such that the contents of $\check{s}$ are the same as s within the scope of t and the locations of $\check{s}$ are the same as s within an offset. Recall that the X, Y variables denote the size and location of $\hat{s}$ from which the concrete fragment $\check{s}$ was taken. Define the least node referenced in $[\![t]\!]_{\langle s, l_p \rangle}$ for queue q_j as the *offset* $o_{j,l}$. Choose each $x_{j,l}$ so that $x_{j,l}$ is the sum of queue lengths for all nodes up to $o_{j,l}$. In symbols:

$$x_{j,l} = \sum_{b=0}^{o_{j,l}-1} \mathsf{Qlength}(q_j, (n_b, p_l), s).$$

Define *greatest* node referenced in $[\![t]\!]_{\langle s, l_p \rangle}$ as $e_{j,l}$. Pick each $y_{j,l}$ so that $y_{j,l}$ is the sum of each queue q_j for nodes $(o_{j,l}, p_l)$ up to node $(e_{j,l}, p_l)$. In symbols:

$$y_{j,l} = \sum_{b=o_{j,l}}^{e_{j,l}-1} \mathsf{Qlength}(q_j, (n_b, p_l), s).$$

These particular values for $x_{j,l}$ and $y_{j,l}$ are included in $\gamma \langle \hat{M}, \hat{s} \rangle$ because $\alpha \langle M, s \rangle$ includes all messages from s in the same path segment and queue as the appear in s.

Pick the present location $\check{l_p}$ so that every location accessed in $[\![t]\!]_{\langle \check{s}, \check{l_p} \rangle}$ is included in the concrete fragment. Let $\check{l_p}$ equal (n_a, p_l) and pick n_a so that a *preserves the offset* for every node accessed in path segment p_l. The node index offset is preserved if a minus $o_{j,l}$ is equal to k for every node n_k accessed in path segment p_l. In symbols, pick k so that $a - o_{j,l} = k$

Pick $\check{s}$ from $\gamma\langle\hat{M}, \hat{s}\rangle$ so that the contents of $\check{s}$ match the contents of s within X and Y. More precisely, for every tuple $y = (i, q_j, (n_x, p_l), m)$ in s add the tuple $\check{y} = (i, q_j, (n_{x-o_{j,l}}, p_l), m)$ to $\check{s}$. The only difference between y and $\check{y}$ is that the offset $o_{j,l}$ subtracted from node indices in $\check{y}$.

Tuple $\check{y}$ can be added to $\check{s}$ by γ because the presence of y in s forces α to add tuple $\hat{y} = (z, q_j, p_l, m)$ to $\hat{s}$ where z is equal to the number of messages in front of m in q_j of path segment p_l. The presence of $\hat{y}$ in $\hat{s}$ means that γ can add $\check{y}$ to $\check{s}$. In general, such an $\langle\check{M}, X, Y, \check{s}\rangle$ can always be found in $\gamma\langle\hat{M}, \hat{s}\rangle$ since γ considers all divisions of the contents of $\hat{s}$.

An example of the construction for a transition applied to network with one path segment and one queue per node is shown in Figure 4.1. Queue and path segment indices are ommited from the figure for clarity. A concrete state s is shown at the bottom of the figure. Transition t applied at position $l_p = (3, 1)$ in state s. For this particular application of t, $o = 2$ since t does not observe any nodes to the left of node 2 and $e = 4$ since t does not observe any nodes to the right of node 4. In

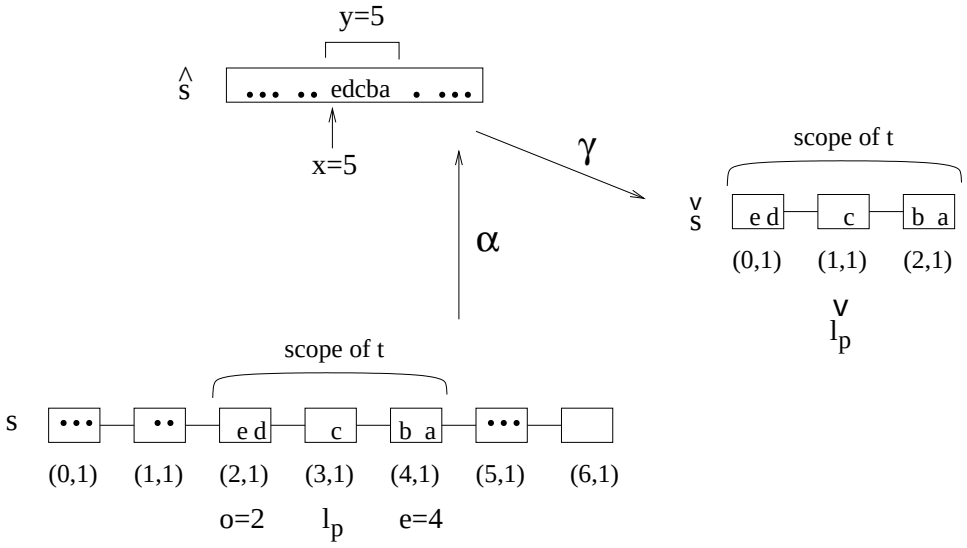

Figure 4.1. Concrete state fragment construction.

the abstract state $\hat{s}$, x and y are chosen so that just the messages within the span of t are included in the partial concrete state $\check{s}$. Value x is chosen to be 5 since there are a total of 5 messages contained in the nodes to the left of node o and the value of y is chosen to be 5 since there are 5 messages contained in the queues within the scope of t in state s. Having chosen x and y, a concrete state fragment of $2(1) + 1 = 3$ nodes is constructed since the span of t is 1. There are 3^6 ways to arrange 6 messages in 3 nodes but only one of them results in a concrete state fragment that is identical to s within the scope of t. This fragment is shown at the top right of Figure 4.1. Finally, $\check{l_p}$ in fragment $\check{s}$ is chosen to be (1,1) since l_p is (3,1), 0 is 2 and 3-2 is 1.

The next step to show that the interpretation of t in $\langle s, l_p \rangle$ is the same as the interpretation of t in $\langle \check{s}, \check{l_p} \rangle$—modulo the offset. Let context C denote $\langle l_p, s \rangle$ and $\check{C}$ denote $\langle \check{l_p}, \check{s} \rangle$. In symbols,

$$(m, i, q_j, n_k, p_l) \in [\![t]\!]_{\check{C}} \text{ if } (m, i, q_j, n_{k+o_{j,l}}, p_l) \in [\![t]\!]_C \wedge k + o_{j,l} \leq e_{j,l}. \tag{4.10}$$

The proof of Equation 4.10 will be done by induction on the interpretation of Newspeak transitions. The scope properties and location independence of Newspeak transitions are the crux of this part of the proof.

Only the interpretations of Newspeak expressions involving next, addr and `msg_at` refer to the context. The interpretation of every other Newspeak expression is trivially equivalent. For next, addr and `msg_at`, the interpretation in the full context C uses next, addr and s whereas the interpretation in the partial context $\check{C}$ uses $\check{\text{next}}$, $\check{\text{addr}}$ and $\check{s}$. If node indices in the scope of t differ by a constant offset, then the interpretations are the same modulo the offset. Each of $\check{\text{next}}$, $\check{\text{addr}}$ and $\check{s}$ leverages properties of Newspeak and SR transition systems to preserve the offset.

For $\check{\text{next}}$, if $\text{next}((n_{x-o}, p_k), l_d)$ is (n_{x-o-1}, p_k) then $\check{\text{next}}((n_x, p_k), l'_d)$ is (n_{x-1}, p_k) because $\check{\text{next}}$ is derived from $\hat{\text{next}}$, and $\hat{\text{next}}$ preserves directional information that

is used in the creation of $\check{\mathsf{next}}$. A similar argument holds when the next node is in the other direction or on an adjacent path segment. A subtle requirement here is that the second arguments to next and $\check{\mathsf{next}}$,l_d and l'_d must have the same path segment index and a node index in the same direction. For this purpose, Newspeak and SR transitions are restricted to apply next to message source or destination addresses. In the noncreative model $\phi(NC(P(N)))$, only messages with the addresses of observed terminal node indices, $\mathsf{addr}(a)$ for some a in L_{Obs}, appear in a network state.

Briefly returning to two necessary properties of Newspeak transitions, if Newspeak did allow next to be applied to arbitrary addresses, then Newspeak transitions would not have finite branching scope and the proof of Theorem 4.4 will fail here. The proof would fail because next applied to arbitrary values allows a transition to observe locations which can not be contained in fragments created by γ. If γ could create such fragments, then γ would have to create an unbounded number of fragments to represent the unbounded number of paths to which a node might be connected.

Also, if Newspeak allowed locations to be compared with constants, then Newspeak transitions would not be location independent and node indices that differ by a constant offset would be distinguishable. As a result, the proof of Theorem 4.4 will fail because interpretation of t in C and $\check{C}$ would be different. Contexts that differ by node indices must be allowed because γ can not create all possible sets of node indices. If γ could create all sets of node indices then γ would create an unbounded set of fragments because node indices are unbounded.

Continuing with the proof, the problem is now reduced to showing that $\mathsf{addr}(a)$ and $\check{\mathsf{addr}}(a)$ have the same path segment index and a node index in the same direction. Once again, since $\check{\mathsf{addr}}$ is derived from $\hat{\mathsf{addr}}$ and $\hat{\mathsf{addr}}$ preserves the direction of the node index for observed terminal node mappings, $\check{\mathsf{addr}}$ also preserves

the path index and node index direction.

Finally, `msg_at` interprets to the same value because s and $\check{s}$ are the same, modulo the offset, within the scope of t. This property follows from the construction of $\check{s}$ outlined previously.

The final step of the proof is show that the new abstract state created by μ applied to the new concrete state fragment is equal to the new abstract state $\hat{s}'$ created by α applied to the new concrete state. Let $\check{s}'$ denote the new fragment and s' denote the new state created by the interpretation of t in $\check{s}$ and s respectively. Function μ applied to $\check{s}'$ and $\hat{s}$ is equal to α applied to s' because μ changes $\hat{s}$ only within the scope of t. The abstract states are equal since $\check{s}'$ and s' are equivalent within the scope of t, modulo node indices, and node indices are not included in the abstract representation.

This concludes the proof of the inductive step of Theorem 4.4.

Although an approximate solution for LNPV is the main result of this thesis, an exact solution would be better. To show that NDFS is an exact solution for LNPV, we would need to show that $\forall \hat{s} \in \hat{S}_{\hat{M}}.s \in S_M$ for $\hat{s}, \hat{M} = \alpha\langle s, M\rangle$. In general, this is not provable for NDFS because the transition computation for an abstract state checks every fragment represented by the abstract state. Some of these fragments may not represent reachable states in S_M. In a proof that NDFS is an exact approximation, unreachable concrete state fragments will foil the proof at an equation similar to Equation 4.9. The implementation of NDFS reduces the number of unreachable concrete state fragments through invariants.

CHAPTER 5

IMPLEMENTATION AND RESULTS

The point of the preceding definitions and proof was to create a useful verification model for certain problems on branching networks. In this chapter, the algorithm is implemented and applied to a published multibus IO standard.

The NDFS algorithm has been implemented in a model checker called "Πcasso." Πcasso is an explicit state model checker derived from the Murϕ model checker [43]. The Πcasso implementation is described in Section 5.1. Πcasso has two parts: a specification compiler and a verification engine. The specification language for Πcasso is a variant of Newspeak that does not allow multiple nestings of next but does include limited support for invariants describing concrete state fragments. The specification compiler translates a Newspeak specification and an SR transition system into an executable verification model. Given a compiled network-parameterized specification, the Πcasso verification engine instantiates the network and checks compliance with an SR transition system for that class of network configurations.

The Πcasso tool has been used to verify a transaction ordering property for corrected multibus PCI 2.1 networks. The transaction ordering property is the producer/consumer property specified in the PCI specification. Unfortunately, the PCI protocol does not satisfy this property for all multibus networks. Corella [14] proposed a modification of PCI that allows compliance with the producer/consumer property. Πcasso was used to show that corrected multibus PCI networks satisfy the producer/consumer property.

Πcasso was also used to find a topology dependent violation of a coherence property in multibus PCI networks. Although PCI was not intended to satisfy this coherence property, the significance of this violation is that *almost all* PCI networks *do* satisfy the coherence property. Results for the alternating bit protocol on a simple lossy network and the multibus PCI results are given in Section 5.2. Both PCI and the lossy network satisfy the requirements to generalize Πcasso results requirements to generalize Πcasso results as described in the next chapter.

5.1 Implementation

The Πcasso tool is derived from the Murϕ model checker [43] by the creation of a new verification engine and specification language. Πcasso provides an implementation of queue and network operations. The interface and implementation of the queue and network models are split. The interfaces are visible in the specification language, and the implementation is contained in the verification engine. This split supports specifications written using concrete queue and network operations, but interpreted using abstract queue and network models.

The verification engine is a breadth-first implementation of the NDFS algorithm (which is described in its depth-first form in Figure 3.8). The verification engine uses the output of the specification compiler to generate an executable C++ model which is then compiled and executed to perform verification.

The Πcasso implementation of NDFS avoids the explicit construction of concrete state fragments through the use of state-vector pointers. The state-vector pointers provide a concrete view of a fragment of the abstract state vector. All queue operations are done in-place directly using the concrete view of the state vector on the abstract state vector. In this manner, the Πcasso implementation avoids copying portions of the state vector into different memory locations then merging the portions back into the abstract state vector.

The Πcasso tool extends NDFS through the inclusion of invariants and certain kinds of creative transitions. Invariants are used at line 9 to identify and eliminate infeasible concrete state fragments. An infeasible concrete state fragment is a fragment that can be generated from an abstract state, but which does not correspond to a reachable state in the concrete model. Invariants on concrete fragment avoid infeasible fragments. This, in turn, prevents infeasible computational paths that might lead to infeasible error states. A future implementation of Πcasso will include a mechanism for more general user-supplied invariants.

Recall that a creative transition is a transition that results in a network state with more messages. Some protocols (such as multibus PCI) support message types that leave a trail of copies in every node through which they pass. The trailing copies guide an acknowledgment back to the source but provide no other functionality. Πcasso extends NDFS by providing support for these message types. In Πcasso only the newest trailing copy in the state vector is kept, other copies are reintroduced nondeterministically as needed. The Πcasso built-in insert and delete operations for queues are modified so that whenever a message of this type is inserted, the trailing copy is deleted. Similarly, whenever such a message is deleted a copy is inserted in the adjacent node.

Although queues are unbounded in the protocol model, Πcasso places a bound on queue length. The fact that such a bound always exists follows from Theorem 4.2 (page 68) which states that all abstract states are finite. The user must provide the queue length bound, if the provided bound is exceeded then execution stops and must be restarted with a larger bound. Another approach to queue lengths is a dynamic, rather than static, state vector. Dynamic state vectors for explicit state model checking will be discussed later in Chapter 8.

The Πcasso compiler reads a specification and creates a network parameterized C++ file from the specification. The specification language includes an interface to

unbounded queues and branching networks. Overloaded functions support queue operations on queues with different element types in the same specification. The Πcasso compiler disambiguates overloaded function and procedure calls using type signatures.

The Πcasso specification language varies from Newspeak in three ways. First, the Πcasso version of Newspeak language allows conditionals, local variable bindings and loops over finite domains. These two constructs simplify the description of a model, but do not change the expressive power of Newspeak.

Second, the Πcasso does not allow nesting of the next operator. Disallowing multiple nestings of next increases the efficiency of the verification algorithm at the cost of expressiveness. The lost expressiveness means that all transitions have a scope of 1 in a Πcasso model. The loss is trivial because an IO protocol that has a transition of scope greater than one has not yet been identified. This model is more efficient due to the reduced number of fragments that must be considered at line 9 of NDFS. In terms of the concrete fragment bound established in Equation 4.4 of Theorem 4.2 (page 68), restricting all transitions to have scope one means that the free variable n can be set to 3.

Finally, Πcasso allows the specification of certain kinds of invariants about relative message positionings. Each message positioning invariant describes the relative position of a pair of messages. As discussed previously, Πcasso uses this extra information to reduce the number of infeasible fragments.

The usage model for Πcasso is shown in Figure 5.1. In the figure, information supplied by the user is shaded grey. Use of Πcasso begins with a Newspeak specification that might include invariants about of concrete state fragments. The message specification includes a type that is used to instantiate the element type for the queue class. The queue class supplies a set of queue operations which are

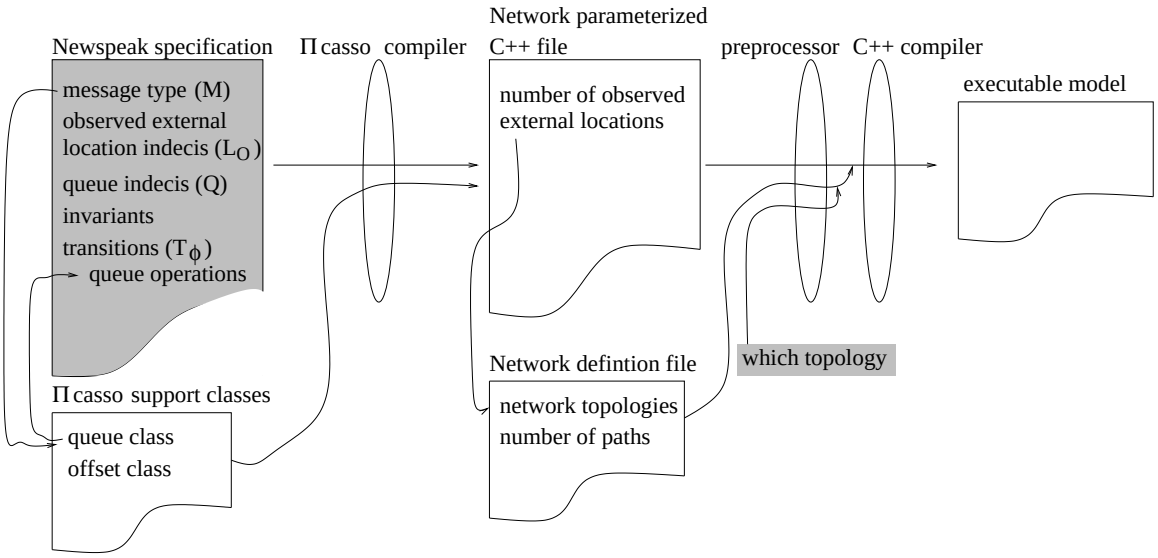

Figure 5.1. Usage model for Πcasso.

used in the Newspeak transition specifications. Both the queue class and offset index class are supplied by Πcasso.

The Πcasso compiler combines the Newspeak specification with the queue and offset classes to create a network parameterized C++ file. In the network parameterized C++ file, all values that depend on the number of paths are defined in terms of a preprocessor variable. The preprocessor variable is instantiated by the network topology selected by the user and number of observed external nodes. The number of observed external locations (taken from the Newspeak specification) together with a user-supplied topology selector determine which topology is used to to instantiate the C++ file with. A C++ compiler generates an executable file from the network instantiated file. The executable file generates states and looks for violations.

5.1.1 Example

A brief example illustrates the interaction between the Πcasso specification language and verification engine. Suppose the following line from a Πcasso specification performs the insertion of a message `m`:

```
Qinsert(next(m1.dst).q1,0,m)
```

The insert operation inserts `m` into queue `q1` of the next node between location `here` and the destination of message `m1`.

Given the above statement, the Πcasso specification compiler generates the following C++ code:

```
q1.Qinsert (here,tm->next(here,m1.dst), 0, m)
```

The C++ statement generated by the compiler is a call to the `Qinsert` method of the `q1` object.

The first two parameters of the `Qinsert` call give the present location and the target of the insert operation. The present location is `here` and the target location is the result of a call to the `next` method of the topology manager, `tm`. The `next` method of `tm` is instantiated by the preprocessor after network topology is chosen. The values of these two locations and the current offsets determine where to insert `m` in the concrete view of the abstract state vector. The final two parameters to `insert` are the queue position and data that are to be inserted.

The evaluation of `insert` makes calls to the offset manager to determine where position 0 of location `next(here, m1.dst)` appears in the state vector. After the state vector location is determined, the state vector is modified to insert `m` at the appropriate location.

5.2 Results

5.2.1 Alternating-Bit Protocol

The first result concerns the verification of the alternating-bit protocol (ABP) on a lossy network (LOSSY) with data transmission errors. The network model for this example allows messages to be dropped by any node and the data bit of a message to be mutated. When a message is dropped, a negative acknowledgment is sent back to the sender.

The Πcasso specification includes an SR transition system that describes the behavior of ABP and a set of transitions that describe the behavior of nodes in a LOSSY network. The transitions that implement ABP use auxiliary state to track the expected value of the alternating bit in the next message. The protocol transition rules move messages between queues in nodes and drop messages. A negative acknowledge is sent to the source of dropped messages. The Πcasso input file for ABP on the LOSSY network can be found in Appendix B.

A single transition specification from this model is shown below:

```
Rule "pass message outq"
!(Qempty (this.outq))
==>
var msg : msg_type ;
begin
      msg := Qpop (this.outq);
      Qappend (next(msg.dst).inq, msg);
end;
```

This transition moves a message from `outq` to the `inq` of the next node. The guard is shown before the "`==>`" and the command is shown after. The guard checks that `outq` is not empty. If `outq` is not empty, then the command pops the first message

and appends the message to `inq` of the next node along the path to the message destination.

In this model, an error state is reached if a message with the wrong alternating-bit value is received. The previous value of the alternating bit is stored in the `old_obs_state` variable, and the current value is stored in the `obs_state` variable. The `old_obs_state` is set to `s0` if the sender sent a message with the alternating bit set to 0, and `obs_state` is set to`r0` if the receiver receives a message with the bit set to 0. The SR transition below moves to the `error` state when the sender or receiver receive an unexpected alternating-bit value.

```
Rule "error transitions for obs_state"
! ((old_obs_state = s0 & obs_state = r0) |
   (old_obs_state = r0 & obs_state = s1) |
   (old_obs_state = s1 & obs_state = r1) |
   (old_obs_state = r1 & obs_state = s0) )
==>
begin
      error "receive or send out of order";
end;
```

Before running the verification engine on the specification, the maximum number of messages that can be present in the network at any given time is determined. For the ABP, this bound is four since the Sender and Receiver can have at most one message and negative acknowledgment in transit in a given state. Using max_ϕ as four in Equation 4.5, and recalling that the ABP uses four bits of auxiliary state and two observed external locations, Πcasso will find at most

$$(2! + 3! + 4!) \cdot 4 = 128$$

abstract states.

The verification engine instantiates the C++ model with a network topology containing two external nodes connected by a single path segment. For ABP on Lossy, the user is not required to select a topology because there is only one topology between two terminals. The computation of the abstract reachable states finds 26 states and in less than 10 seconds of CPU time. Although the ABP on Lossy contains at most 128 abstract states, the actual number of *reachable* abstract states is significantly smaller. Since Πcasso provides answers only for the LNPV, the model checking result demonstrates that the ABP can be implemented on all two-node instances of this simple network in the absence of any other messages. In Chapter 6, it will be shown how to generalize this result to any *pair* of nodes in any lossy network.

5.2.2 Multibus PCI

A PCI multibus network consists of agents, bridges and busses. An agent is an external location, a bridge is an internal location and busses form the connection between agents and bridges. Each agent and bridge stores messages in two queues. The PCI protocol supports three message types: delayed, completion and posted. Delayed messages are acknowledged by completions and posted messages are unacknowledged. There are two kinds of delayed messages: committed and uncommitted. A committed delayed message has been attempted on the bus, but not necessarily received by the target. An uncommitted delayed message has not yet been attempted on the bus. Uncommitted delayed messages and completions can be dropped at any time. A delayed message leaves committed copies of itself in every bridge through which it passes. As the acknowledging completion deletes travels back to the sending agent the committed copies are deleted. Within a queue, all reorderings are allowed—except no message may pass a posted message.

PCI includes a creative transition, the delayed latch, which is essential for the completion of delayed messages. The delayed latch creates a new copied of a delayed message in an adjacent node. The new copy is created only when a copy of the message, or its completion, do not already appear in the adjacent node. All PCI transitions can be expressed as Newspeak transitions in the Πcasso model checker. The kind of creative transition needed for a model of PCI is supported by Πcasso through auxiliary invariants as described previously.

The Newspeak model of PCI includes 12 transitions to move PCI messages between bridges, and 15 transitions to reorder and delete messages within queues. Including supporting functions, the Newspeak specification of PCI is 907 long. The network-independent C++ file generated by Πcasso for PCI contains contains 5,328 lines of code.

Other models of PCI used by Clarke [13] and Shimizu [66] describe PCI at the bus level. The Πcasso models of PCI describe the transitions implemented by the bus-level signals. The Πcasso model is adapted from the model of PCI used by Corella et al. [67]. Similar models of PCI have been developed as PVS theories rather than Πcasso specifications as described later in Chapter 7. The PVS models are more concise because the PVS specification language allows richer quantification.

Each of the following Πcasso results for PCI includes only four families of labeled network topologies and assumes the absence of any other messages. In Chapter 6 it is shown that the results can be generalized to all network topologies in the presence of any other network traffic.

Multibus PCI was tested for compliance with two SR transition systems. The first is the producer/consumer (PC) property. An example of an execution of the transitions in the PC property for one network configuration appears in Figure 5.2.

1. The producer (labeled "P") issues a write to a data address followed by a write to a flag address. These writes are labeled "dw" and "fw" respectively. At the same time, the consumer issues a read to the flag address which is labeled "fr."

2. The producer's writes and the consumer's reads travel to their destinations.

3. When the consumers flag read arrives at the flag address, a completion is created, labeled "cfr" and sent back to the consumer.

4. The flag read completion travels back to the consumer deleting while deleting trailing copies of the flag read.

5. If the producer's flag write arrived at the flag before the consumer's flag read (as is the case in Figure 5.2), then the consumer sees that the flag is set and data is ready to be read. The consumer sends a delayed transaction to read the data (labeled "dr").

6. The consumer's dr travels to the data address and reads the data value written by the producer.

Modeled as an SR transition system, the first five steps of the above example form the stimuli of the PC property and the last step is the expected response.

The PC property was stated as an SR transition system with four observed terminal nodes (Producer, Consumer, Data and Flag) and a maximum of six messages in any given network state. Three Boolean variables comprise the auxiliary state. The auxiliary is used to determine whether or not the flag write completed before the flag read, and whether or not the data read completed before the data write.

The actual number of reachable states for each multibus PCI topology over four terminal nodes is shown in Table 5.1. No violations of the PC property were found in any topologies.

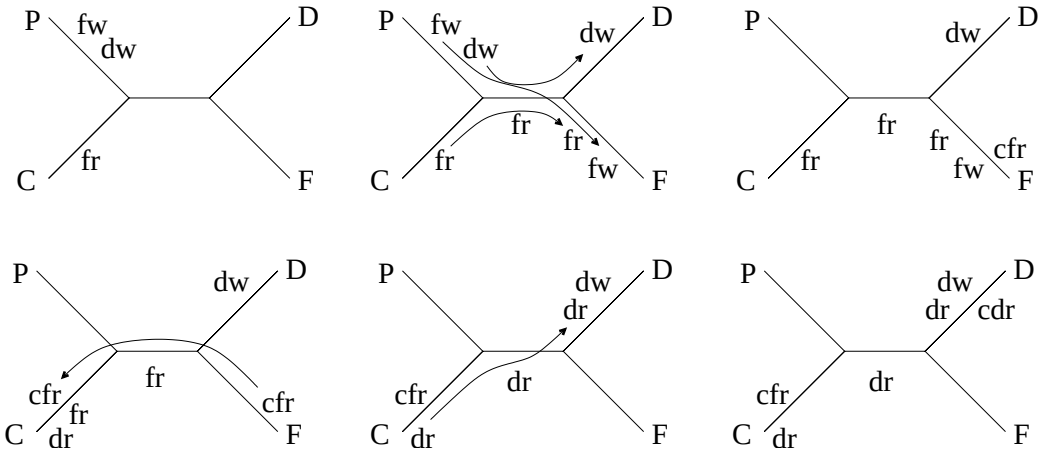

Figure 5.2. Producer/consumer property.

The second result for multibus PCI networks considers compliance with a write coherence property. In this coherence property, two processors each write to a cache and the main memory. One processor reads from the main memory and the other from the cache. Both processors are expected to read the same value. Since PCI was not intended to comply with this memory model, the significance of this result is not that one configuration violates the model but that most configurations do not violate the model. This example highlights the utility of looking for topology dependent violations.

The transition system is introduced using Figure 5.3. Configuration $C1$ is shown on the left and configuration $C2$ is shown on the right. The transition system has

Table 5.1. Πcasso results for PC property on multibus PCI networks.

Network	CPU Time	States	Violation?
C1	167 sec.	393	no
C2	238 sec.	611	no
C3	74 sec.	313	no
C4	17 sec.	131	no
Total	496 sec.	1448	

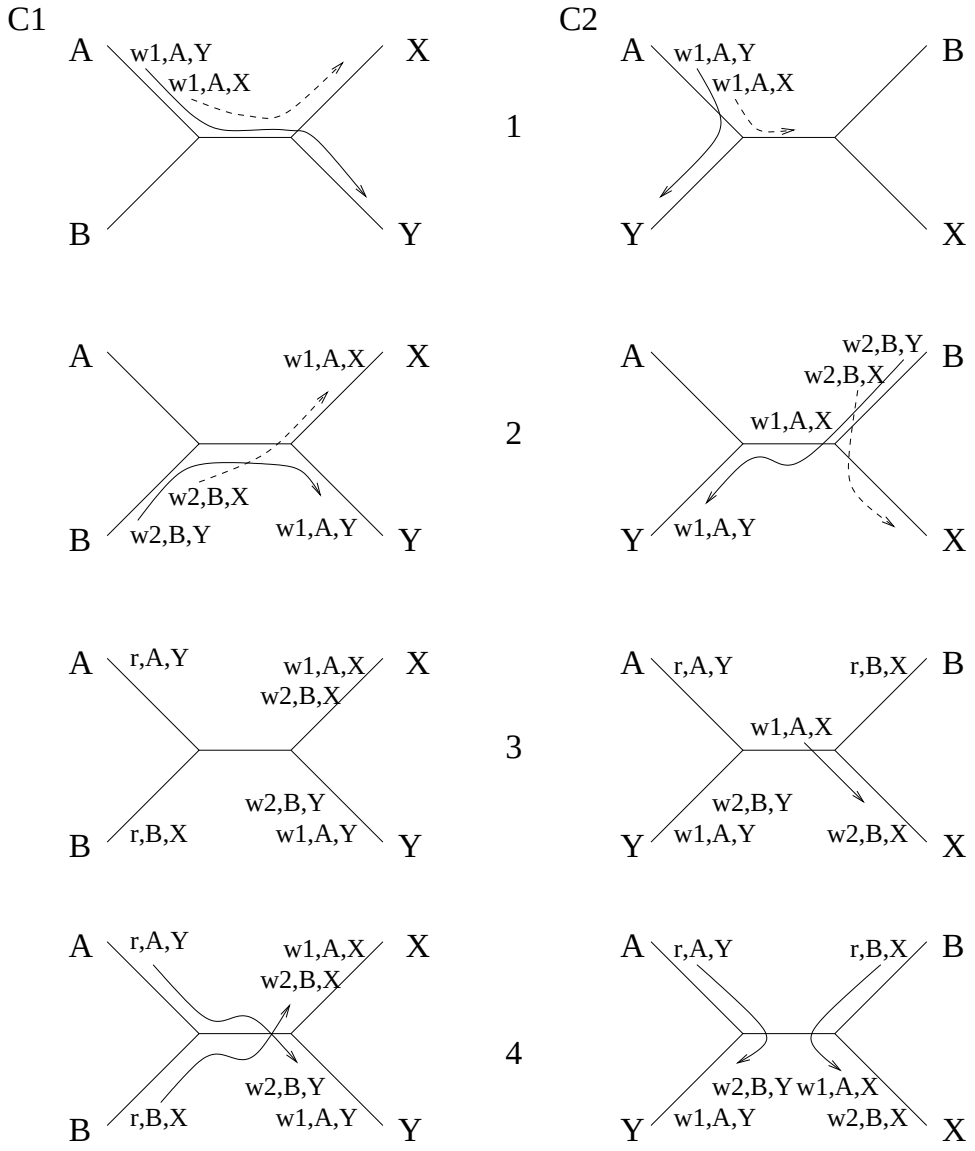

Figure 5.3. Coherence property.

four external locations: two processors A and B and two memories X and Y. The expected response is that if processor A reads a 2 from memory Y then processor B will also read a 2 from memory X. The transition system is divided in four steps:

1. Processor A writes the value 1 to memory X and writes the same value to memory Y. The write to Y pushes the write to X until the paths from A to Y and A to X split. In each configuration, write-pushing is indicated by a

dashed arrow. Note that in configuration $C2$, the write to X is pushed only to the middle path segment.

2. After the writes from A arrive at X and Y, processor B writes the value 2 to memory X then to memory Y. The write to Y pushes the write to X as shown by the dashed arrow.

3. Processor A reads from memory Y and processor B reads from memory X. Meanwhile, the pending write $(w1, A, X)$ in configuration $C2$ might finally complete its journey to memory X.

4. When the reads arrive, processor A will read a 2. Processor B will read a 2 in configuration $C1$, but might read a 1 in configuration $C2$.

The trace in configuration $C1$ conforms with the property but the trace in configuration $C2$ might not.

Πcasso was used to show that all traces of all configurations, other than $C2$, conform with the coherence property. The states and CPU time required to check the coherence model are shown in Table 5.2.

Table 5.2. Πcasso results for Coherence property on multibus PCI networks.

Network	CPU Time	States	Violation?
C1	1.4 sec.	27	no
C2	1.9 sec.	36	yes
C3	1.3 sec.	24	no
C4	0.9 sec.	16	no
Total	5.5 sec.	103	

CHAPTER 6

APPROXIMATE SOLUTION FOR NPV

Knowing where not to look for errors is as useful as knowing where to look for errors. The computational setting of LNPV excludes certain network configurations and protocol behaviors that are included in NPV. For some protocols, we can prove that the excluded configurations and behaviors do not contain any new errors. In these cases, verification results from NDFS for LNPV generalize to NPV.

In this chapter, we formally define this class of protocols and give the generalization proof. The generalization proof is broken in two parts. The first part covers the behavioral generalization from P to $NC(P)$. The crux of this argument is that, for some protocols, adding more messages to a network state containing a set of m messages might disable, but never enable, transitions on the original m messages. The new messages may enable more transitions, but those transitions never change the relative positions of the original m messages. Such protocols are called *monotonic under projection*. This monotonicity property is defined formally and then used in the proof.

The second argument covers generalization from $N_{|\phi|}$ to N_i for any i. Going from $N_{|\phi|}$ to N_i results in complete coverage of branching network configurations. The key to this generalization is that, for some protocols, messages are insensitive to the state of the network outside their path from source to destination. In this case, the network state outside of $N_{|\phi|}$ has no bearing on the behavior of the observed messages. Such protocols are called *well-routed*. The well-routed property is defined formally and used in the proof.

The final section in this chapter discusses the generalization of the Πcasso model checking results obtained in Chapter 5. Results for ABP on the Lossy network and PC and Coherency on multibus PCI networks can be generalized.

6.1 Generalizing from $NC(P)$ to P

Generalizing from $NC(P)$ to P means that the protocol may inject random messages into the network at any time. The problem is that these messages might uncover new violations of a property ϕ. Without the creative transitions in P, the verification of ϕ for $NC(P)$ includes only states that contain just the messages observed by ϕ. Perhaps the new messages added by P will lead to states that now violate ϕ.

For monotonic-under-projection protocols, the new messages will not lead to states that violate ϕ. This monotonicity property describes a relationship between projected states and enabled transitions. Projected states and sets of enabled transitions are defined next, followed by the monotonicity relation.

A projection is a mapping between states that removes all instances of certain messages from a state. The messages that are removed are called the *extra-set* for a projection. A member of the extra-set is called an *extra message*.

Definition 6.1 *A projection π is a function which maps states to states such that $\pi_M s^+ = s$ for some finite removal set of messages M, such that $s = s^+$ without any messages from M.*

$$
\begin{array}{l}
(i, q, (n, p), m) \in s \text{ iff} \\
\quad (j, q, (n, p), m) \in s^+ \wedge m \in M \wedge \\
\quad \forall (k, q, (n, p), m) \in s^+. \\
\qquad m \in \Sigma_{Obs} \wedge k < j \rightarrow (l, q, (n, p), m) \in s \wedge l < i \vee \\
\qquad m \in \Sigma_{Obs} \wedge k = j \rightarrow (l, q, (n, p), m) \in s \wedge l = i \vee \\
\qquad m \in \Sigma_{Obs} \wedge k > j \rightarrow (l, q, (n, p), m) \in s \wedge l > i.
\end{array}
$$

An example of a projection application to a single queue is shown in Figure 6.1. The projection π in Figure 6.1 removes all messages of type b. The only extra

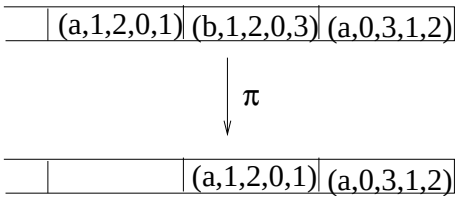

Figure 6.1. Projection of network state.

message in Figure 6.1 is the message of type b between the two messages of type a. The projected state includes both messages of type a, but now the messages are adjacent since the extra message was deleted.

One useful projection is the ϕ-projection defined by an SR transition system ϕ. The *phi*-projection retains only the observed messages Σ_{Obs} required by ϕ. The value of the ϕ-projection is that the ϕ-projection eliminates all of the messages created by P.

The second concept needed for the monotonicity property is the noop-set of a projection for a given state. A transition is included in the noop-set if the projection discards all state changes made by transitions. In the monotonicity property, the size of the noop-set quantifies the number of transitions that do not modify the projection of a state.

Definition 6.2 *The noop set for a state s, transition system T and projection π is denoted $noop(s, T, \pi)$ and is the largest set of transitions from T such that no interpretations of t result in a state different than s under projection π:*

$$t \in noop(s, T, \pi) \text{ iff } \forall l \in L.\ \pi([\![t]\!]_{s,l}) = \pi\ s.$$

The monotonicity property can now be defined in terms of projection and noop-sets. The monotonicity property we want requires that states with more extra messages have larger noop-sets.

Definition 6.3 *A model $M = P(N)$ is monotonic under a projection π if for any reachable state $s \in S_M$,*

$$noop(\pi(s), T, \pi) \subseteq noop(s, T, \pi).$$

A model that satisfies the above monotonicity property for a projection π is denoted $mono(\pi, M)$.

The somewhat unusual expression $noop(\pi(s), T, \pi)$ merits further clarification. The expression $noop(\pi(s), T, \pi)$ denotes the set of transitions which make no modifications of any kind to state $\pi(s)$. Since transitions are applied directly to $\pi(s)$ and $\pi(\pi(s)) = \pi(s)$, the application of π to the interpretation of a transition of $\pi(s)$ does not hide any state modifications made by the transition.

Returning state s in Figure 6.1, three examples of transitions that swap messages will clarify the intuition behind monotonicity. First, a monotonic transition t might swap adjacent messages of type a. Transition t is in the noop-set for s because extra message b sits between both messages of type a. Removing b from s removes t from the noop-set of $\pi(s)$ since both messages of type a are now adjacent. A transition t' which swaps adjacent a and b messages is also monotonic. Although t' may modify state s, the modification is discarded by the projection. Since message b is removed by the projection, the relative position of b in state s does not matter in the projection of s. Finally, a transition t'' which swaps two messages of type a if they are separated by a message of type b is not monotonic. Transition t'' is not monotonic because removing message b from state s adds transition t'' to the noop-set.

In the remainder of this section, we show that the new messages added by P do not cause new violations of ϕ in monotonic protocols.

Theorem 6.1 *Given creative model* $M^+ = \phi(P(N))$ *and noncreative model* $M = \phi(NC(P)_N)$ *with* $mono(\pi_\phi, M^+)$ *and* $N \in N_{|\phi|}$*:*

$$\forall s^+ \in S_{M^+}.\pi_\phi(s^+) \in S_M.$$

Proof. The proof is similar to the trace inclusion proof used for Theorem 4.4. State s^+ was created by a sequence of transition applications. We will show show that each intermediate transition applied to an intermediate state $s^{+\prime}$ in the creation of s^+ is mirrored by a transition from M applied to the projection of $s^{+\prime}$. There are two cases to consider, each transition t could have been in $noop(s^{+\prime}, T, \pi)$ or not. For each case, we must show that an application of t to $\pi(s^{+\prime})$ results in the same state as the projection of t application to $s^{+\prime}$.

First, suppose $t \in noop(s^{+\prime}, T, \pi)$. In this case, $\pi([t]_{s^{+\prime}}) = \pi(s^{+\prime})$ so no transition from M is required to create the same state.

The more interesting case is when $t \notin noop(s^{+\prime}, T, \pi)$. In this case, is must be shown that

$$\pi_\phi([\![t]\!]_{<s^+,l>}) = [\![t]\!]_{<s^+_{\pi_\phi},l>} \tag{6.1}$$

The following Corollary combines the set-inclusions obtained in Theorems 4.4 and 6.1:

> **Corollary 6.1** *Given a model* $M^+ = \phi(P(N))$ *such that* $mono(M^+)$ *and* $N \in N_{|\phi|}$ *and the set of abstract reachable states* $\hat{S}_{\hat{M}}$ *generated by* NDFS*,*
>
> $$\forall s^+ \in S_{M^+}.\ \alpha\ \pi_\phi(s^+) \in \hat{s}_{\hat{M}}.$$

Corollary 6.1 justifies the use of NDFS to approximate a solution to NPV. In the next section Corollary 6.1 is strengthened by widening the class of networks used in the model M^+.

6.2 Generalizing from $N_{|\phi|}$ to N_i

Generalizing from $N_{|\phi|}$ to N_i means that a network configuration may contain any number of terminal nodes. Since each network configuration requires a separate run of NDFS and N_i contains an unbounded number of topology classes, NDFS can not possibly be used to enumerate that state of every topology class. Instead, an argument is made that for certain kinds of protocols, the extra network configuration classes do not contain any error-states—unless a network in $N_{|\phi|}$ contains an error.

This class of protocols includes protocols that always route messages along the path from source to destination. Such a protocol is called a *well-routed* protocol. Before defining well-routed protocols and making the generalization argument the set of reachable states for a set of network configurations is defined.

The generalization argument will relate the set of reachable states for a protocol on a set of network configurations. Given a protocol P and an SR transition system ϕ, P and ϕ determine a set of reachable states on all networks that is the union of the reachable states for each network. The reachable states for one network was defined in Definition 2.4.

Definition 6.4 *The set of reachable states for a protocol P and SR transition system ϕ on all networks of size i is denoted $\mathbf{S^i_{\phi,P}}$ and defined as*

$$\mathbf{S^i_{\phi,P}} = \bigcup_{\mathbf{N \in N_i}} \mathbf{S_{\phi(P(N))}}.$$

Not only is i an unbounded integer, but the set $\mathbf{S^i_{\phi,P}}$ is unbounded for a given i. This is because the number of networks $N \in N_i$ is unbounded since each network may contain an unbounded number of nodes per path segment. Moreover, the number of states in $S_{\phi(P(N))}$ for a given network N is also unbounded since P may introduce an unbounded number of messages into the network.

Next, a well-routed protocol is defined.

Definition 6.5 *A well-routed protocol P is a protocol such that for any message $m = (\sigma, a, b)$, if s is a reachable state of P on network N then $(i, q, l, m) \in s$ implies $\mathsf{next}^i(a, b) = l$ for some integer i. A well-routed protocol is denoted $wr(P)$.*

For well-routed protocols, the extra network configurations beyond $N_{|\phi|}$ do not contain any new violations of ϕ. The statement of this relationship requires the ϕ-projection to eliminate extra messages outside of the observed messages for ϕ. If a protocol is well-routed, all of the messages remaining after the projection will be contained in some network from $N_{|\phi|}$.

Theorem 6.2 *Given a well-routed monotonic protocol P and an SR transition system ϕ with its projection π_ϕ,*

$$\forall s^+ \in \mathbf{S}^{\mathbf{i}}_{\phi,\mathbf{P}}.\pi_\phi \in \mathbf{S}^{|\phi|}_{\phi,\mathbf{P}}.$$

Proof. This proof has two parts. In the first part, observed messages are shown to remain on a subnetwork of N_i contained in $N_{|\phi|}$. In the second part, messages from other parts of N_i are shown to not affect messages on N_ϕ. The second argument requires the monotonicity property.

Recall from definition 6.4 that $\mathbf{S}^{|\phi|}_{\phi,\mathbf{P}}$ contains all reachable states on all networks from $N_{|\phi|}$. For state s^+ in the reachable states of P on network $N^+ \in N_i$, pick network $N =< L, \mathsf{next} >$ from $N_{|\phi|}$ where N is the subnetwork of N^+ that contains only path segments and locations between locations observed by ϕ such that

$$L = \{l \mid \exists a_1, a_2 \in L_{Obs}.\mathsf{next}^j(\mathsf{addr}(a_1), \mathsf{addr}(a_2))\}$$

for L_{Obs} the set of observed location addresses in ϕ. Since P is well-routed and observed messages are always addressed to observed locations, all of the messages observed by ϕ appear in locations contained in N and $\pi_\phi(s)$ defined on N contains all the observed messages.

Next, behaviors outside of N on N^+ are shown to contain reachable states that are not reachable under π_ϕ. Any transition t which causes a change in the projection of s can not observe the state of N^+ outside of N. If t did depend on such a message outside of N, then t would either violate the monotonicity or well-routed properties of P. Since t is expressed in Newspeak, t can only observe the state of locations along the paths of a message contained in the present location. Because P is well-routed and monotonic, then t can not depend on messages outside of Σ_{Obs} or locations outside of N.

Finally, the set inclusions from Theorems 6.2 and 4.4 are combined with Corollary 6.1 give to obtain the following inclusion:

Corollary 6.2 *Given a model $M^+ = \phi(P(N))$ in which mono(M^+), wr(M^+) and $N \in N_i$ for any i and the set of abstract reachable states $\hat{S}_{\hat{M}}$ generated by* NDFS*,*

$$\forall s^+ \in S_{M^+}.\alpha\ \pi_\phi(s^+) \in \hat{s}_{\hat{M}}.$$

Corollary 6.2 justifies the use of NDFS as an approximate solution to NPV for protocols that are monotonic and well-routed.

6.3 Application

The results in this chapter reduce the problem of generalizing NDFS or Πcasso results to showing that a protocol is monotonic under projection and well-routed. In this section, the ABP on LOSSY and both properties (PC and Coherence) on multibus PCI models discussed in Chapter 5 are shown to be monotonic and well-routed.

6.3.1 ABP on Lossy

The first task is to show that ABP on Lossy is monotonic. For the ABP property, the ϕ-projection deletes all messages not addressed between a sender and receiver. Lossy is monotonic under projection because no Lossy transitions depend on the presence of these deleted messages. Even though the Lossy network allows messages to be mutated, the mutations do not affect the source and destination of a message. If the mutations could change a message source or destination, then monotonicity would be lost because a message addressed outside the sender and receiver, and eliminated by the ϕ-projection, could be mutated into a message between the sender and receiver that is kept by the ϕ-projection. The ABP on Lossy is monotonic because ABP does not depend on messages outside the chosen sender and receiver.

The second task is to show that ABP on Lossy is well-routed. Since no ABP or Lossy transitions send messages to another messages' source or target, ABP on Lossy is well-routed.

Given these two properties of ABP on Lossy, the Πcasso results from Chapter 5 can be generalized to all n-node Lossy networks and the presence of any other messages. The generalized result states that two nodes in any Lossy network can implement the ABP in the presence of any other messages traveling in the network.

6.3.2 PC and Coherence on Multibus PCI

Multibus PCI, as corrected by Corella to satisfy the PC property, is monotonic under the ϕ-projections for both PC and Coherency. Multibus PCI is monotonic under these projections because removing messages to and from other agents does not eliminate any behaviors on the remaining messages.

Without Corella's proposed solution to fix the PC violation, multibus PCI networks are not monotonic under the PC projection. Corella's proposed solution

adds a local master ID to each completion to avoid the problem of completion stealing. PCI without local master IDs violates monotonicity because completions are not addressed to their eventual target. Because completions are not addressed to their target, any delayed message to the same source may match a completion intended for another destination. This behavior means that removing completions for other addresses removes reachable states from the projected model. In the case of the PC property, the lost behavior includes a violation.

The statement of the PC in the PCI specification assumes that no other agents write over the data value before it is read by the consumer. Without this assumption, the multibus PCI would not be monotonic under the projection for the PC property because a write from another agent would be eliminated by the projection but modify the state left by the projection. Such a write would not be included in the Πcasso model but would cause a violation of the PC property.

Multibus PCI is well-routed because no PCI transitions send messages toward the source or destination of another message. Given these two properties, the Πcasso results of Chapter 5 can be generalized to include all multibus PCI networks and states. The generalized correctness statement includes all instances of the PC property (or the Coherence property) on all multibus PCI networks in the presence of any other network traffic.

CHAPTER 7

MANUAL ABSTRACTION

The NDFS algorithm computes a set of abstract states using the concrete transition relation. In this chapter, the set of abstract is computed using a manually derived abstract transition relation. Using an abstract transition relation also requires showing that the concrete transition relation is a refinement of the abstract transition relation.

The significance of using NDFS, rather than an abstract transition relation, is that showing noncreativity is easier than showing refinement. This chapter supports this claim by describing the derivation of an abstract transition relation and refinement proof for multibus PCI networks. The manually derived abstraction used in this Chapter is similar to the abstraction used in NDFS.

The cost of using NDFS, rather than an abstract transition relation, is that NDFS requires more CPU time. The P/C property for multibus PCI networks was verified by NDFS in 496 seconds, whereas the same property was verified in 1.8 seconds using the abstract transition relation.

This chapter is adapted from [61]. Section 7.1 contains an overview of the abstract transition relation solution. Section 7.2 describes the application of the abstraction to branching networks and the PCI protocol. Section 7.3 describes the refinement proof and Section 7.4 contains the model checking results for the abstract transition relation. A detailed comparison of the manual abstraction and the NDFS algorithm for PCI multibus PCI verification is given in the final section.

7.1 Solution Overview

The manual solution to the problem of exhaustively verifying the P/C transaction ordering property for multibus PCI networks is outlined in Figure 7.1. The solution can be divided into three steps, each step is labeled 1,2 or 3 in the figure. First, an abstraction on PCI multibus network topologies is used. The abstract network model requires an abstract model of PCI called PCI_A (see Section 7.2). Second, an interactive higher-order logic theorem prover is used to prove that the PCI protocol on PCI networks is a trace inclusion refinement of the PCI_A protocol on the reduced networks (Section 7.3). Third, an explicit state model checker is used to show that all execution traces on the reduced networks satisfy the P/C property (Section 7.4).

The next three sections explain the abstraction process, refinement proof and model checking results in more detail.

7.2 Network and Protocol Abstraction

The abstraction maps instances of the P/C property to one of four reduced networks. The abstraction requires modifications of the PCI protocol to keep the

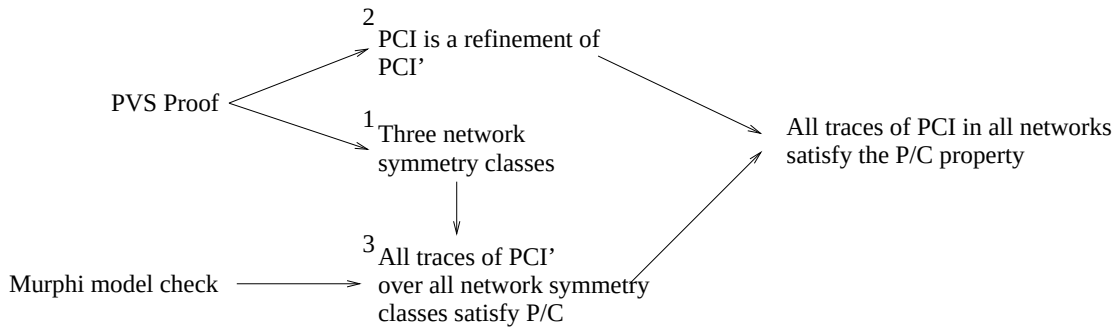

Figure 7.1. Outline of PCI transaction ordering verification.

number of states in each reduced network finite, and to create the same behaviors on abstract networks.

Despite the fact that each network has a (large) finite number of states, there are an unbounded number of networks. One way to reduce the problem is to consider the symmetry classes induced by ignoring agents and paths incidental to the property being verified, and to ignore the lengths of the paths between agents central to the property being verified.

Figure 7.2 contains an example of such a symmetry reduction for an instance of the P/C on a PCI network. The network on the left side of the figure contains an instance of the P/C property defined on a network with seven agents and two bridges. The producer, consumer, data and flag agents are labeled P, C, D and F, and the other agents are all labeled with A's. The parts of the network on the left side of the figure that contribute to the reduced network have been circled. The reduced network appears on the right side of the figure. In the reduced network, all agents incidental to the P/C property are ignored, and the path between bridges 1 and 2 in the center of the left network have been coalesced into path B in the right network. The transactions in the queues of bridges 1 and 2 are concatenated and placed in path B. Arbitrarily many busses, bridges and agents can be added to the network on the left, but the reduction will still result in the same network shown on the right. Unfortunately, the PCI protocol has an unbounded number of states in each of these reduced networks. Although each coalesced bridge

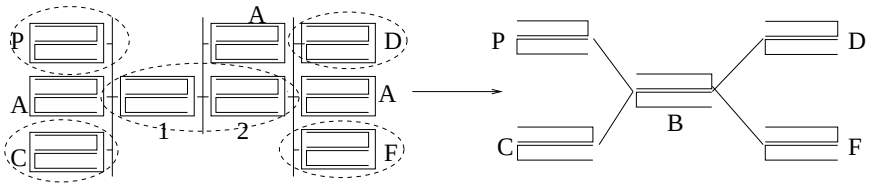

Figure 7.2. Example of a network symmetry reduction

contains a bounded number of transactions, there may be arbitrarily many bridges in a path. This allows arbitrarily many transactions in a coalesced path. The number of transactions in a coalesced path can be reduced by ignoring transactions other than those required by the property being verified. However, in the case of PCI, paths may still contain an unbounded number of transactions. Recall that delayed transactions leave a trail of committed copies in each bridge through which they pass. Each of these committed copies, is then concatenated and placed in the corresponding coalesced path in the reduced network—again leading to an unbounded number of states in the reduced network. This source of unboundedness is eliminated by keeping only the newest copy of a committed delayed transaction in the state of the reduced network. This reduction is justified by a PCI invariant and requires several modifications to the PCI_A protocol.

7.3 PVS Refinement Proof

The purpose of showing that PCI is a trace inclusion refinement of PCI_A is to formally establish that safety properties of PCI_A hold for PCI. This allows model checking results from PCI_A to be generalized to PCI. Proving trace inclusion refinement requires showing that for every concrete trace, there exists an abstract trace such that each state in the abstract trace is equal to the abstraction of the corresponding concrete state [68]. Trace inclusion is *conservative* with respect to safety properties. This means that properties violated by PCI_A are not necessarily violated by PCI. In the context of PCI refinement means that for each concrete trace, σ, there exists an abstract trace, σ_A in PCI_A, such that the following relationship holds:

$$\begin{aligned} \forall \sigma \in PCI \quad & \{\sigma = (A_0, C_0), (A_1, C_1), (A_2, C_2) \ldots (A_n, C_n) \\ & \Rightarrow \exists \sigma_A \in PCI_A \sigma_A = A_0, A_1, A_2 \ldots A_n\} \end{aligned}$$

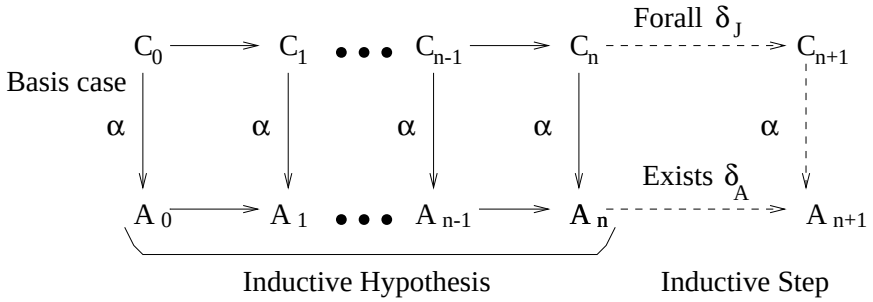

Figure 7.3. Outline of proof that PCI is a refinement of PCI_A.

in which for every A_i in σ_A $A_i = abs(C_i)$ for some abstraction function abs. The proof is outlined in Figure 7.3. As shown in the figure, the relationship will be shown by induction on the length of σ.

The operational semantics of each protocol were defined inductively using a set of inference rules. The set of reachable states is inductively defined, starting from an initial state, as the parallel composition of the inference rules. An inference rule can be applied to a reachable state to create a new reachable state if and only if the rule's preconditions are satisfied by the current reachable state. A state is reachable if and only if it can be derived from the initial state by a sequence of applications of the inference rules.

The inference rules facilitate the inductive refinement proof by suggesting a natural way to perform a case split on the construction of the next state. The inference rules were used in the inductive step of the refinement proof shown in Figure 7.3. For each application of a PCI rule δ_j it was shown that an application of a PCI_A rule $delta_A$ creates the same abstract state.

The refinement proof was completed in the PVS theorem prover using three theories which describe PCI, PCI_A and the abstraction. Each of the three theories is described below followed by the discussion of the refinement proof.

7.3.1 Definitional Theory of PCI

The theory describing the concrete PCI protocol contains ten inference rules and several supporting definitions. Each of the 10 rules describes a PCI transition, and were adapted from [49, 4].

Given a reachable state, a rule constructs a new reachable state if the preconditions are met. The preconditions involve at most a bridge (or agent) and an adjacent bridge (or agent). If the preconditions are met, the next reachable state is constructed by modifying the contents of a bridge (or agent) and an adjacent bridge (or agent). The limited scope of each rule allows each rule to apply to networks with different topologies because each rule depends only on the connection between two entities, rather than on the topology of the entire network.

The theory of PCI contains 551 lines of PVS, including comments. The theory is built on top of a theory of unbounded finite sequences that are used to model the queues. Two rules describe the movement of posted transactions and the remaining eight describe the movement of delayed transactions and their completions. Starting from a network in its initial state, the entire set of reachable states for the network can be computed by repeatedly applying rules whenever and wherever their preconditions are satisfied.

7.3.2 Definitional Theory of PCI$_A$

The PCI$_A$ rules were designed to mimic every effect of the PCI rules on the reduced networks, while preserving as many properties of PCI as possible. If the PCI$_A$ rules mimic every effect of the PCI rules then it can be shown that PCI is a refinement of PCI$_A$. Suppose the PCI$_A$ rules were defined as the CHAOS rule-set, in which all actions are always enabled. In this case, PCI is certainly a refinement of CHAOS, but CHAOS does not have has any useful properties. The PCI$_A$ protocol is refined by PCI but preserves useful properties.

The theory describing the PCI_A protocol contains 15 rules and several supporting definitions. The PCI_A theory contains more rules than the PCI theory to compensate for the network information lost in the abstraction. The lost information about the network structure induces nondeterministic behavior in the PCI_A rules. Each nondeterministic action is modeled by a different rule with the same precondition.

For example, Figure 7.4 illustrates the ambiguity resulting from the loss of node boundaries in the abstract network model. Network fragments, $N1$ and $N2$, on the left contain a single transaction, $T1$ and $T2$. A path boundary is indicated by the dashed line. Despite the fact that $T2$ has reached the final position in the path, both $T1$ and $T2$ map to the head of the paths $P1$ and $P2$ under the abstraction. This is because no transactions appear ahead of $T1$ in $N1$. This is a problem when $T1$ and $T2$ are copied to the next node to the right. In $P1$, the new copy of the transaction would appear in front of $T1'$ and in $P2$ the new transaction would appear at end of the next path after $P2$. In this case, PCI_A requires two versions of certain rules to cover both possibilities. Similar instances of ambiguity involving suppressed committed copies of delayed transactions necessitated additional rules in the PCI_A model.

The PCI_A theory is 507 lines long, including comments and uses the same theory of finite sequences to represent paths. Of the 15 rules, 3 cover the movement of

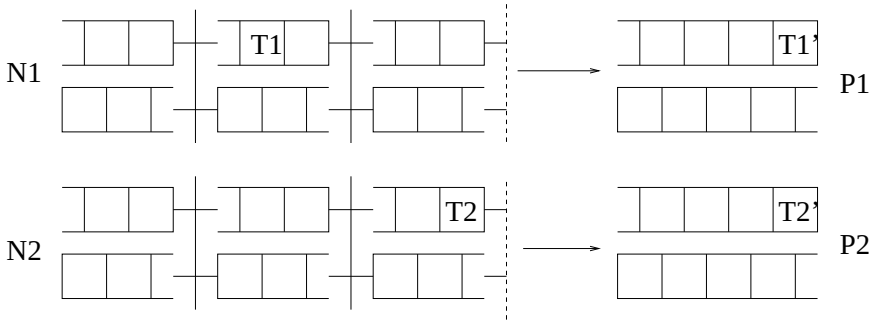

Figure 7.4. Example of ambiguity in reduced network paths.

posted transactions and the other 12 cover the movement of delayed transactions and their completions.

7.3.3 Axiomatic Theory of the Abstraction

The abstraction theory describes the abstraction function that relates states in PCI to states in PCI_A. The abstraction function performs the network symmetry reduction and eliminates transactions (as discussed in Section 7.2) so that each reduced network has a finite number of states. Rather than define the abstraction using a set of definitions and prove lemmas about the abstraction for use in the refinement proof, the required properties of the abstraction were stated as axioms and used directly in the refinement proof. An axiomatic, rather than a definitional theory, was used to save the effort required to prove the abstraction lemmas. A total of 51 axioms about the abstraction were used in the refinement proof. The axioms describe the effects of the network symmetry reduction, predicates which can be inferred about a PCI_A state from a PCI state and the effects of inserting and deleting transactions. Thirty of the axioms have been shown to be noncontradictory. The 21 unvalidated axioms describe the effects of inserting and deleting transactions.

The main source of complexity in the unvalidated axioms is the appearance and disappearance of committed delayed transactions in the states of PCI_A. For example, if a new transaction is inserted into a path, then the significant committed delayed transactions already present in that path may or may not disappear depending on the type and location of the newly inserted transaction. This behavior made defining axioms that describe the effects of insertion and deletion difficult. If PCI did not include a notion of "leaving trails of transactions" the axiomatization of the abstraction would have been smaller and simpler (but not as interesting).

7.3.4 Refinement Proof

The PVS proof was developed by a single experienced PVS user in about one month of effort. The final refinement proof required approximately 1000 proof commands (determined by taking the number of lines in the PVS proof file and dividing by two, to account for the pretty-printing of right parentheses). Whereas the proof required a significant amount of effort, the refinement proof can be reused for other properties of the PCI protocol

7.4 Model Checking

Model checking was used to show that every trace in every reduced network satisfies the P/C property. The inference rules describing the operational semantics of PCI_A were used to define a model of PCI_A for use in the Murϕ model checker [43].

The Murϕ models were an average of 670 lines long (including comments). The Murϕ code for the PCI_A "delayed commit" rule is given below. The rule is part of a larger rule-set in which the subscripts i and j range over queue position i in path j of the network.

```
Rule "d Commit 1"
 (uncommitted (trans)) &
==>
begin delete_trans (path, commit (trans)); -- delete t_t
      network[i][j] := commit (trans);
end;
```

The precondition checks that the transaction is uncommitted. The action, given after the "==>", replaces the transaction with its committed form.

Two pieces of auxiliary state to encode the P/C property as a safety property. The auxiliary state stores the completion order of the read and write transactions

at their respective destination agents. Using the completion orders stored in the auxiliary state, the P/C property is stated as a safety property which reads:

> If the data read transaction has completed at the data address, then either flag write occurred after the flag read, or the flag read has not completed, or the data read occurred after the data write.

This safety property is stated as an invariant of the Murϕ model of PCI_A and checked using explicit state enumeration. The number of states and execution required to check each reduced network is given in Table 7.1. No violations were found in any of the networks. The Murϕ models were developed and verified in about one week of effort by a new Murϕ user (including time to install and learn Murϕ).

7.5 Discussion

Both the manually derived PCI_A abstraction and the NDFS algorithm have been used to verify the P/C property for multibus PCI networks. These methods differ in computational requirements, as can be seen by comparing Tables 7.1 and 5.1. In this section, the methods are compared based on how they model nondeterminism in the abstract state transitions and model trails of delayed PCI transitions.

Nondeterminism is modeled in the PCI_A transition relation by including several transitions with the same guard. In the NDFS algorithm, the same nondeterminism

Network	CPU Time	States
A	0.8 sec.	306
B	0.8 sec.	170
C	0.7 sec.	247
D	0.3 sec.	134
Total	2.6 sec.	857

Table 7.1. Model checking results for P/C property on PCI_A.

is captured in the definition of the $\check{\mathsf{next}}$ function. For the situation shown in Figure 7.4, the PCI_A model includes a transition for each case. The NDFS generates both cases because $\check{\mathsf{next}}$ (p, p') can be either p, which gives the upper case in Figure 7.4 or p', which gives the lower case in Figure 7.4.

In the PCI_A model, trails of delayed transitions must be handled explicitly by the PCI_A transitions. This is done by inserting a copy of a trailing delayed transition when a delayed transition is deleted, and deleting a trailing copy of a delayed transition when a delayed transition is inserted. In the NDFS model, Πcasso has been extended to model such behavior without modifying PCI transitions.

CHAPTER 8

CONCLUSION

8.1 Summary

A major theme of this work is detecting topology dependent violations of communication protocols on nodes connected by branching networks. The strategy advocated in this thesis is to perform a case split on network topologies then enumerate abstract states of each topology separately. The case split reduces the computationally difficult problem of checking all states of all topologies to the less-difficult problems of checking all states in a single family of topologies. While this strategy can find topology dependent errors, a drawback to this approach is that only a finite number of topology families can be checked. Because an abstraction is used, the resulting algorithm is a conservative overapproximation (all errors will be found, but some infeasible errors may be found).

Two algorithms are developed to realize this strategy. One algorithm, NDFS, enumerates abstract states of a family of isomorphic network configurations; and the other algorithm enumerates the a complete set of network topologies over n terminal nodes.

The novel feature of the NDFS algorithm is the use of partial concrete states during the computation of abstract state successors. In NDFS, locally scoped concrete transitions are applied to concrete state fragments represented by an abstract state. The resulting concrete state fragment successors are then incorporated back into an abstract state to create the abstract state successors. The use of partial

concrete states obviates the need to construct and validate an abstract transition relation. Concrete state fragments are sufficient for locally scoped transitions because transitions are limited to a fragment the network state. Concrete state fragments are required for the more detailed network model used in NDFS because each abstract state represents an unbounded number of complete concrete states, but only a finite number of unique fragments (assuming a network state contains a finite number of messages). The NDFS algorithm terminates only for a limited class of transition systems and correctness properties. The NDFS algorithm is supported by the Newspeak specification language in which all noncreative (i.e., only the messages required to check a property are created) protocols expressible in Newspeak can be analyzed used the NDFS algorithm.

The NDFS algorithm was implemented in the Πcasso model checker and used to check a corrected version of PCI. The corrected version of PCI, as proposed by Corella, was shown to satisfy the producer/consumer property (the published PCI specification violates the property). Πcasso was also used to find a topology dependent violation of a coherence property in multibus PCI networks. Since PCI was not intended to satisfy this property, the significance of this error is not its existence in one class of topologies, but its absence in *every other* class of topologies.

8.2 Future Work

8.2.1 Reasoning Support

The usability of the NDFS algorithm would benefit from increased reasoning support. Using NDFS requires finding a noncreative subset of a protocol, and finding a bound on the number of messages in SR transition system. Extending NDFS results from LNPV to NPV further requires showing that a protocol is monotonic under projection and well-routed.

One possibility might be to create support for reasoning directly about Newspeak protocols. Reasoning support for Newspeak specifications would permit the NDFS model checker and reasoning tool to use the same specification language. This would eliminate the need to create a translation between tools.

8.2.2 Parallel Implementation

The NDFS algorithm is a good candidate for parallelization because many independent computations are applied to a small data set. The resulting data set is also small. The core of the NDFS algorithm consists of a lengthy sequence of unrelated computations that apply transitions to fragments of concrete states. In many cases, this large set of computations results in a small set of new abstract states. Parallelizing the core of the NDFS algorithm may result in increased efficiency by performing these computations concurrently.

Historically, explicit state model checkers are poor candidates for parallelization [69]. The NDFS algorithm is different. In contrast to most model checking algorithms, the NDFS algorithm is CPU rather than memory intensive. NDFS is not more memory efficient than other algorithms; rather, NDFS is less CPU efficient. Adding more CPUs to the problem may improve performance.

8.2.3 Dynamic State Vector

The amount of memory used by the Πcasso model checker can be reduced by using a dynamic, rather than a static, state vector. In the current implementation, every state vector must be as large as the largest abstract state. A dynamic state vector would allow the state vector to grow and shrink with the number of messages stored in each queue. Dynamic state vectors might give average case memory usage rather than worst case memory usage.

The cost of a dynamic state vector is information required to decode how many messages are in a state thus facilitating decoding. In a static state vector, every state has the same bit-level layout. In a dynamic state vector, the layout changes with the number of messages in each queue. Extra bits will be needed to determine where messages start and end. For most protocols, it is anticipated that the extra memory per state required to use a dynamic state vector will be offset by the memory saved.

8.2.4 Precise Solution

A precise solution to LNPV can be developed using a finite instantiation and induction argument. This precise solution was not pursued in this thesis because the number and size of states in the finite model grows exponentially in the product of: the number of protocol transitions, network path segments and messages required by an SR transition system.

A promising avenue of future research might combine the finite instantiation solution with a dynamic state vector model checker. The size of the state vector in the finite instantiation solution is determined by the *worst case* size of the state of a single node and network. Since only one queue of one node can contain the maximum number of messages and each queue of each node has space for the maximum number of messages the finite instantiation is overly-pessimistic.. A dynamic state vector representation gives a state vector with *average case* size.

APPENDIX A

LIST OF SYMBOLS

C $(S \times L)$ A context consisting of a state and a present location, l_p, for the interpretation of a Newspeak transition, 34

L **(Nat**$\times$ **Nat)** A finite set of location indices. Each location index consists of a path index and a node index, 26

$L_E \subseteq L$ The set of terminal nodes in a network $N = (L, \mathsf{next})$. A terminal node is a node that sits on the edge of a network, 27

L_{Obs} (Nat) The set of observed terminal node indices defined by an SR transition system. Elements of L_{Obs} are mapped to observed locations by the address mapping addr, 37

M A model consisting of a network parameterized protocol $P(N)$ instantiated on network N, 35

M^+ A model derived from a protocol P which includes creative transitions, 94

$NC(P(N))$ The noncreative subset of a protocol P in which transitions delete at least as many messages as they insert, 41

N_i The set of all networks $N = (L, \mathsf{next})$ such that the number of external locations, L_E, in N is i, 28

$P(N)$ A network parameterized protocol with network argument N, 29

Q The set of queue indices for a protocol, 29

S_M The set of states reachable in zero or more steps in model M from the empty initial state s_{init}, 36

S_ϕ The universe of augmented states consisting of network states S paired with states of the auxiliary store, 38

T The *set* of universal protocol transitions that may be applied to any network location to create a new network state, 31

T_E The *set* of external protocol transitions that may be applied only to external network locations in L_E, 31

T_{Obs} The set of SR transitions in an SR transition system $\phi(P(N))$, 38

T_ϕ $(T \cup T_E \cup T_{Obs})$ The combined transition system for an SR transition system $\phi(P(N))$ with universal transition set T, external transition set T_E and SR transition set T_{Obs}, 39

X $(\{Nat\})$ A set of indices that indicate which portion of an abstract state was used to concrete a concrete state fragment, 53

Y $(\{Nat\})$ A set of indices that indicate how many messages were taken from an abstract state to concrete a concrete state fragment, 53

Λ A finite opcode alphabet, 29

$\Sigma(\Lambda \times L \times L)$ The set of messages consisting of messages with the form (opc, src, dst) with opcode opc, source src and destination dst, 29

α $(M \times S \to (\hat{M} \times \hat{S}))$ An abstraction function that maps a model M and state s to an abstract model $\hat{M}$ and abstract state $\hat{s}$, 45

$\mathbf{S}^{\mathbf{i}}_{\phi,\mathbf{P}}$ The set of states reachable in $\phi(P(N))$ for all networks N with i terminals in N_i, 96

$\check{M}$ A partial concrete protocol model created by the partial concretization γ. The model consists of a partial network model, a universe of partial states and a set of transitions, 49

$\check{S}$ The set of concrete state fragments, 52

γ $\left(\hat{M} \times \hat{s} \to \{(\check{M} \times \check{s} \times X \times Y)\}\right)$ A partial concretization function which maps an abstract protocol model $\hat{M}$ and an abstract state $\hat{s}$ to a partial concrete model M and a *set* of partial concrete states $\check{s}$ contained between the portions for $\hat{s}$ delineated by the index sets X and Y, 49

$\hat{L}$ The set of abstract locations created from the concrete locations L by removing node indices, 46

$\hat{M}$ An abstract protocol model created by the abstraction α consisting of an abstract network structure and state set and concrete transition system, 45

$\hat{S}_{\hat{M}}$ The set of abstract states reachable in abstract model $\hat{M}$ through zero or more steps from the abstract empty initial state $\hat{s}_{init}$, 57

$\hat{s}_{init}$ The abstract empty initial state, 57

$[\![t]\!]_C$ $\left(T \cup T_E \times (S \times L) \to S\right)$ The interpretation of transition t in a context C consisting of a present location and a state, 35

μ $\left(\check{S} \times X \times Y \times \hat{M} \times \hat{S} \to (\hat{M} \times \hat{S})\right)$ The abstraction/merge function that abstracts a concrete state fragment and merges the result back into an abstract state to create a new abstract state, 55

ϕ-projection A projection π which retains only the observed messages observed by ϕ, 93

$\phi(P(N))$ A stimulus/response transition system parameterized by a protocol model $P(NP)$, 37

π_M $(S \to S)$ A projection that maps states to projected states by removing messages in a removal set M, 92

$\to_M$ Single-step reachability in a model M., 35

$\to_M^*$ Multistep reachability in a model M, 36

$\to_{\check{M}}$ Single-step reachability for a partial concrete model $\check{M}$, 54

$\to_{\hat{M}*}$ Multistep reachability for an abstract protocol model $\hat{M}$, 56

$\to_{\hat{M}}$ Single-step reachability for an abstract protocol model $\hat{M}$, 56

l_p (L) The present location in a context, 34

max_ϕ The maximun number of messages in a single reachable state of an SR transition system ϕ, 65

$mono(\pi, M)$ A model $M = P(N)$ that is monotonic under projection π. A model is monotonic under a projection when projecting any state $s \in S_M$ by π results in a state with a smaller noop set, 94

$noop$ $(S \times T \times \pi \to T)$ Denotes the set of transitions that *do not* modify states under a projection π, 93

s^+ A state reachable in a model M^+ derived from a protocol P that contains creative transitions, 94

s_{init} The empty initial state, 36

APPENDIX B

ΠCASSO MODEL OF LOSSY

This appendix contains a Πcasso model used to implement the alternating-bit protocol (ABP) on the LOSSY network of Chatper 5. The model may be divided into three parts: variable and type declarations, auxillary functions, the SR transition system and the transitions for each nodes.

```
Const Qbound   :2;

Type
   agents : terminals {Sender, Receiver};
   state_type : 1..4;
   bit_type : 0..1;
   upper_type : enum {s0,r0,s1,r1};

abp_msg_type : enum {A,N,D};
msg_type : record src: agents ;
      dst: agents ;
      opc: abp_msg_type ;
      bit: bit_type;
           end;

queue_type : queue of msg_type;

node_type : node   q1 : right queue_type;
       q2 : left queue_type;
      end;

Var
   s_state :  state_type;
   r_state :  state_type;
   obs_state : upper_type;
   old_obs_state : upper_type;
```

```
-- ------------------------------------------------
-- Auxillary functions
-- ------------------------------------------------

Function receive_msg (j : agent; o : abp_msg_type) : boolean ;
var msg : msg_type ;
begin
   if (left_side(j)) then
   if (Qempty (j.q2)) then return false endif;
   msg := Qhead (j.q2);
      endif;
      if (right_side(j)) then
   if (Qempty (j.q1)) then return false endif;
   msg := Qhead (j.q1);
      endif;
   return (msg.opc = o & msg.dst = j);
end;

Function not_a_nack_q1 (j : nindex) : boolean ;
var msg : msg_type;
begin
      if (Qempty (j.q1)) then return false; endif;
      msg := Qhead (j.q1) ;
      return (! msg.opc = N);
end;

Function not_a_nack_q2 (j : nindex) : boolean ;
var msg : msg_type;
begin
      if (Qempty (j.q2)) then return false; endif;
      msg := Qhead (j.q2) ;
      return (! msg.opc = N);
end;

-- ------------------------------------------------
-- SR Transition System
-- ------------------------------------------------

Rule "sender recvs ack in state  2 or 4 "
(s_state = 4 | s_state = 2) &
receive_msg (Sender, A)
==>
var msg : msg_type;
begin
```

```
  msg := Qpop (Sender.q2);
  if (s_state = 2) then
    if (msg.bit = 1) then
      s_state := 3; old_obs_state := obs_state ; obs_state := s0;
      else s_state := 1 endif;
  endif;
  if (s_state = 4) then
     if (msg.bit = 0) then
       s_state := 1; old_obs_state := obs_state ;
       obs_state := s1; else s_state := 3 endif;
  endif;
end;

Rule "sender recvs nack in state 2 or 4"
(s_state = 2 | s_state = 4) &
receive_msg (Sender,N)
==>
var msg : msg_type;
begin
  msg := Qpop (Sender.q2);
  s_state := s_state - 1;
end;

Rule "sender send d w/ bit = 1 or 0"
s_state = 1 | s_state = 3
==>
var msg : msg_type;
begin
  msg.opc := D;
  msg.src := Sender;
  msg.dst := Receiver;
  if (s_state = 1) then msg.bit := 1 endif;
  if (s_state = 3) then msg.bit := 0 endif;
  Qappend (Sender.q1, msg);
  s_state := s_state +1;
end;

Rule "receiver receive data  w/ bit = 1 or 0"
(r_state = 1 | r_state = 3) & receive_msg (Receiver, D)
==>
var msg : msg_type ;
begin
  msg := Qpop (Receiver.q1);
  if (msg.bit = 1 & r_state = 1) then
```

```
    r_state := 2; old_obs_state := obs_state ; obs_state := r1;
  endif;
  if (msg.bit = 0 & r_state = 1) then r_state := 4 endif;
  if (msg.bit = 0 & r_state = 3) then
    r_state := 4; old_obs_state := obs_state ; obs_state := r0;
  endif;
  if (msg.bit = 1 & r_state = 3) then r_state := 2 endif;
end;

Rule "receiver receive nack in state 1 or 3"
(r_state = 1 | r_state = 3) & receive_msg (Receiver , N)
==>
var msg : msg_type ;
begin
  msg := Qpop (Receiver.q1);
  if (r_state = 1 & msg.bit = 1) then r_state := 4 endif;
  if (r_state = 3 & msg.bit = 0) then r_state := 2 endif;
end;

Rule "receiver send ack from state 2 or 4"
r_state = 2 | r_state = 4
==>
var msg : msg_type ;
begin
      msg.opc := A;
      msg.src := Receiver;
      msg.dst := Sender;
      if (r_state = 2) then msg.bit := 1 else msg.bit := 0 endif;
      Qappend (Receiver.q2, msg);
      if (r_state = 2) then r_state := 3 else r_state := 1 endif;
end;

Rule "error transitions for obs_state"
! ((old_obs_state = s0 & obs_state = r0) |
   (old_obs_state = r0 & obs_state = s1) |
   (old_obs_state = s1 & obs_state = r1) |
   (old_obs_state = r1 & obs_state = s0))
==>
begin
  error "receive or send out of order";
end;

Startstate
  s_state := 1;
```

```
  r_state := 1;
  old_obs_state := r0;
  obs_state := s1;
end;

end; -- SR Transition System

-- ------------------------------------------------
-- Newspeak rules for each node.
-- ------------------------------------------------

Ruleset this : nindex
do

Rule "pass message q1"
!(Qempty (this.q1))
==>
var msg : msg_type;
begin
      msg := Qpop (this.q1);
      Qappend (next(msg.dst).q1,
         msg);
end;

Rule "mutate message q1"
!(Qempty (this.q1)) & not_a_nack_q1 (this)
==>
var msg : msg_type;
begin
      msg := Qpop (this.q1);
      if (msg.bit = 1) then msg.bit := 0; else msg.bit := 1 endif;
      Qinsert (this.q1, 0, msg);
end;

Rule "drop message q1"
!(Qempty (this.q1)) &
not_a_nack_q1 (this)
==>
var msg : msg_type;
begin
      msg := Qpop (this.q1);
      msg.opc := N;
      msg.dst := msg.src;
      Qappend (this.q2, msg);
```

```
end;

Rule "pass message q2"
!(Qempty (this.q2))
==>
var msg : msg_type ;
begin
      msg := Qpop (this.q2);
      Qappend (next(msg.dst).q2, msg);
end;

Rule "drop message q2"
!(Qempty (this.q2)) & not_a_nack_q2 (this)
==>
var msg : msg_type;
begin
  msg := Qpop (this.q2);
  msg.opc := N;
  msg.dst := msg.src;
  Qappend (this.q1,msg);
end;

Rule "mutate message q2"
!(Qempty (this.q2)) & not_a_nack_q1 (this)
==>
var msg : msg_type;
begin
      msg := Qpop (this.q2);
      if (msg.bit = 1) then msg.bit := 0; else msg.bit := 1 endif;
      Qinsert (this.q2, 0,msg);
end;

end;
```

REFERENCES

[1] K. L. McMillan, *Symbolic Model Checking: An approach to the state explosion problem*, Ph.D. thesis, Carnegie Mellon University, 1992.

[2] IEEE Computer Society, *IEEE Standard for Futurebus+—Logical Protocol Specification*, IEEE Computer Society, March 1992, IEEE Standard 896.1-1991.

[3] IEEE Computer Society, *IEEE Standard for a Higher Performance Serial Bus*, IEEE Computer Society, 1996, IEEE Standard 1394-1995.

[4] PCISIG, "PCI Special Interest Group–PCI Local Bus Specification, Revision 2.1," June 1995.

[5] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2000.

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

[7] David L. Dill, "The Murϕ verification system," In Alur and Henzinger [70], pp. 390–393.

[8] The VIS Group, "VIS: A system for verification and synthesis," In Alur and Henzinger [70], pp. 428–432.

[9] Krzysztof R. Apt and Dexter C. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Information Processing Letters*, vol. 22, no. 6, pp. 307–309, May 1986.

[10] E. M. Clarke, O. Grumberg, H. Haraishi, S. Jha, D. Long, K. L. McMillan, and L. Ness, "Verification of the Futurebus+ cache coherence protocol," *Formal Methods in System Design*, vol. 6, no. 2, pp. 217–232, 1995.

[11] A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," In Alur and Henzinger [70], pp. 184–195.

[12] M. Sighireanu and R. Mateescu, "Verication of the link layer protocol of the IEEE-1394 serial bus (FireWire): an experiment with E-LOTOS," *Springer International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 1, pp. 68–88, December 1998.

[13] Edmund Clarke, Somesh Jha, Yuan Lu, and Dong Wang, "Abstract BDDs: A technique for using abstraction in model checking," in *Correct Hardware Design and Verification Methods, CHARME '99*, Laurence Pierre and Thomas Kropf, Eds., Bad Herrenalb, Germany, Sept. 1999, vol. 1703 of *Lecture Notes in Computer Science*, Springer-Verlag.

[14] Francisco Corella, "Proposal to fix ordering problem in PCI 2.1," 1996, Accessed June 2001 www.pcisig.com/reflector/thrd8.html#00704.

[15] M.C. Browne, E.M. Clarke, and O. Grumberg, "Reasoning about networks with many identical finite state processes," *Information and Computation*, vol. 81, pp. 13–31, April 1989.

[16] R. Milner, *A Calculus of Communicating Systems*, Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic," *ACM Trans. on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, April 1986.

[18] E.M. Clarke and O. Grümberg, "Avoiding the state explosion problem in temporal logic model checking algorithms," in *ACM Symposium on Principles of Distributed Computing*. ACM, August 1989, pp. 294–303.

[19] A.P. Sistla and S. M. German, "Reasoning with many processes," in *Annual IEEE Symposium on Logic in Computer Science*, New York, June 1987, pp. 138–152, IEEE.

[20] R. Karp and R. Miller, "Parallel program schemata," *Journal of Computer and System Science*, vol. 3, no. 4, pp. 302–311, May 1969.

[21] C. Petri, "Fundamentals of a theory of asynchronous information flow," in *Information Processing 62: Proceedings of the 1962 IFIP Congress*, Amsterdam, The Netherlands, 1962, pp. 386–390, North-Holland.

[22] Z. Shtadler and O. Grumberg, "Network grammars, communication behaviors and automatic verification," in *Workshop on Automatic Verification Methods for Finite-State Systems*, J. Sifakis, Ed., Grenoble, France, June 1989, vol. 407 of *Lecture Notes in Computer Science*, pp. 151–165, Springer-Verlag.

[23] Pierre Wolper and Vinciane Lovinfosse, "Verifying properties of large sets of processes with network invariants," in *Automatic Verification Methods for Finite State Systems*, J. Sifakis, Ed., June 1989, vol. 407 of *Lecture Notes in Computer Science*, pp. 68–80.

[24] R.P. Kurshan and K.L McMillan, "A structural induction theorem for processes," in *Proc. of the 8th ACM Symp. Principles of Distributed Computing*, 1989, pp. 239–247.

[25] E. M. Clarke, O. Grumberg, and S. Jha, "Verifying parameterized networks using abstraction and regular languages," in *6th International Conference on Concurrency Theory (CONCUR'95)*, I. Lee and S. Smolka, Eds., Philadelphia, PA, August 1995, vol. 962 of *Lecture Notes in Computer Science*, pp. 395–407.

[26] David Lesens and Hassen Saïdi, "Automatic verification of parameterized networks of processes by abstraction," in *2nd International Workshop on the Verification of Infinite State Systems: INFINITY'97*, July 1997.

[27] P. Cousot and R. Cousot, "Comparing the Galois connection and widening/narrowing approaches to abstract interpretation," in *Programming Language Implementation and Logic Programming, 4th International Symposium*, M. Bruynooghe and M. Wirsing, Eds. August 1992, number 631 in Lecture Notes in Computer Science, pp. 269–295, Springer-Verlag.

[28] E. Allen Emerson and Kedar S. Namjoshi, "Reasoning about rings," in *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, Jan. 1995, pp. 85–94.

[29] Y. Kesten, O. Maler, M. Marcus, A Pnueli, and E. Shahar, "Symbolic model checking with rich assertional languages," In Grumburg [71], pp. 424–435.

[30] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson, "Handling global conditions in parameterized system verification," In Halbwachs and Peled [72], pp. 134–145.

[31] A. Bouajjani and P. Habermehl, "Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations," in *Proceedings of the International Colloquium on Automata, Languages and Programming*. 1997, number 1256 in Lecture Notes in Computer Science, pp. 560–570, Springer-Verlag.

[32] P. A. Abdulla, A. Bouajjani, and B. Jonsson, "On-the-fly analysis of systems with unbounded, lossy FIFO chanels," in *Computer-Aided Verification, CAV '98*, Alan J. Hu and Moshe Y. Vardi, Eds., Vancouver, BC, Canada, June/July 1998, vol. 1427 of *Lecture Notes in Computer Science*, pp. 305–318, Springer-Verlag.

[33] B. Jonsson and M. Nilsson, "Transitive closures of regular relations for verifying infinite-state systems," In Graf and Schwartzbach [73], pp. 220–234.

[34] Ahmed Bouajjani, Bengt Johnsson, Marcus Nilsson, and Tayssir Touilli, "Regular model checking," In Emerson and Sistla [74], pp. 403–418.

[35] Wolfgang Thomas, *Handbook of Theoretical Computer Science*, vol. B, chapter Automata on Infinite Objects, pp. 133–191, Elsevier Science Publishers, 1990.

[36] Amir Pnueli and Elad Shahar, "Liveness and acceleration in parameterized verification," In Emerson and Sistla [74], pp. 328–343.

[37] J.-P. Bodeveix and M. Filali, "FMona: A tool for expressing validation techniques over infinite state systems," In Graf and Schwartzbach [73], pp. 204–219.

[38] J-P. Bodeveix and M. Filali, "Experimenting acceleration methods for the validation of infinite state systems," in *Intl Workshop on Distributed System VAlidation and Verification (DSVV'2000)*, Pao-Ann Hsiung, Ed., Taipei, Taiwan, April 2000.

[39] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl, "Abstracting WS1S systems to verify parameterized networks," In Graf and Schwartzbach [73], pp. 188–203.

[40] B. D. Lubachevsky, "An approach to automating the verification of compact parallel coordination programs I," *Acta Informatica*, vol. 21, pp. 125–169, 1984.

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

[42] F. Pong and M. Dubois, "A new approach for the verication of cache coherence protocols," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 8, pp. 773–787, August 1995.

[43] C. Norris Ip and David L. Dill, "Verifying systems with replicated components in Murϕ," In Alur and Henzinger [70], pp. 147–158.

[44] Mark Longley F. Keith Hanna, Neil Daeche, "Specification and verification using dependant types," *IEEE Transactions on Software Engineering*, vol. SE-16, no. 9, pp. 949–964, September 1990.

[45] Phillip J. Windley, "Correctness properties for iterated hardware structures," in *Proceedings of the Fifth Annual IEEE/NASA Symposium on VLSI Design*, November 1993.

[46] Annette Bunker, "A hardware combinator for tree-shaped circuits," M.S. thesis, Brigham Young University, September 1998.

[47] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter, "Routing information protocol in HOL/SPIN," in *Theorem Provers in Higher-Order Logics 2000: TPHOLs00*, August 2000, pp. 53–72.

[48] Michael J. C. Gordon and Thomas F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, Cambridge University Press, 1993.

[49] Francisco Corella, Robert Shaw, and Cui Zhang, "A formal proof of absence of deadlock for any acyclic network of PCI buses," in *Computer Hardware Description Languages, CHDL '97*, Toledo, Spain, Apr. 1997, pp. 134–156.

[50] A. Roychoudhurry, K. N. Kumar, C. R. Ramakrishnan, I.V. Ramakrishnan, and S. A. Smolka, "Verification of parameterized systems using logic program transformations," In Graf and Schwartzbach [73], pp. 172–187.

[51] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren, "Efficient model checking using tabled resolution," In Grumburg [71], pp. 143–154.

[52] A. Roychoudhurry, K. N. Kumar, C. R. Ramakrishnan, and I.V. Ramakrishnan, "A parameterized unfold/fold transformation framework for definite logic programs," in *PPDP*. 1999, number 1702 in Lecture Notes in Computer Science, pp. 396–413, Springer-Verlag.

[53] F. K. Hwang, D. S. Richards, and P. Winter, *The Steiner Tree Problem*, vol. 53 of *Annals of Discrete Mathematics*, North-Holland, Amsterdam, Netherlands, 1992.

[54] E. N. Gilbert and H. O. Pollak, "Steiner minimal trees," *SIAM J. Appl. Math.*, vol. 32, pp. 835–859, 1968.

[55] N. Saitou and M. Nei, "The neighbor-joining method: A new method for reconstructing phylogenetic trees," *Molecular Biological Evolution*, vol. 4, pp. 406–425, 1987.

[56] Abdel Mokkedem, Ravi Hosabettu, Michael D. Jones, and Ganesh Gopalakrishnan, "Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem," *Formal Methods in Systems Design*, vol. 16, no. 1, pp. 93–119, January 2000.

[57] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, Deepak Kapur, Ed. 1992, vol. 607 of *Lecture Notes in Artificial Intelligence*, pp. 748– 752, Springer-Verlag.

[58] Michael D. Jones and Ganesh Gopalakrishnan, "Toward automated abstraction for protocols over branching networks," in *IEEE International High Level Design Validation and Test Workshop (HLDVT'00)*, November 2000, pp. 147–152.

[59] Susanne Graf and Hassen Saidi, "Construction of abstract state graphs with PVS," In Grumburg [71], pp. 72–83.

[60] Satyaki Das, David L. Dill, and Seungjoon Park, "Experience with predicate abstraction," In Halbwachs and Peled [72], pp. 160–172.

[61] Michael Jones and Ganesh Gopalakrishnan, "Verifying transaction ordering properties in unbounded bus networks through combined deductive/algorithmic methods," in *Formal Methods in Computer-Aided Design: FMCAD'00*, Warren A. Hunt Jr. and Steven D. Johnson, Eds., Austin, Texas, November 2000, number 1954 in LNCS, pp. 505–519.

[62] Michael D. Jones and Ganesh Gopalakrishnan, "A PCI formalization and refinement," in *TPHOLs 2000 Supplemental Proceedings*, Mark Aagaard, John Harrison, and Tom Schubert, Eds. 2000, number CSE 00-009, OGI Technical Report.

[63] J. D. Day and H. Zimmerman, "The OSI reference model," *Proceedings of the IEEE*, vol. 71, pp. 1334–1340, December 1983.

[64] George Orwell, *Nineteen Eighty-Four*, Harcourt, Brace and Co., 1949.

[65] Edmund M. Clarke, Orna Grumberg, and David E. Long, "Model checking and abstraction," *ACM Trans. on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, September 1994.

[66] Kanna Shimizu, David L. Dill, and Alan J. Hu, "Monitor-based formal specification of PCI," in *Formal Methods in Computer-Aided Design: FMCAD'00*, Warran A. Hunt Jr. and Steven D. Johnson, Eds., Austin, Texas, November 2000, number 1954 in LNCS, pp. 335–353.

[67] Francisco Corella, "Verifying memory ordering model of I/O systems," in *Computer Hardware Description Languages, CHDL '97*, Toledo, Spain, Apr. 1997, Invited Talk.

[68] Martín Abadi and Leslie Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.

[69] Ulrich Stern and David L. Dill, "Parallelizing the Murϕ verifier," In Grumburg [71], pp. 256–267.

[70] Rajeev Alur and Thomas A. Henzinger, Eds., *Computer-Aided Verification, CAV '96*, vol. 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[71] Orna Grumburg, Ed., *Computer-Aided Verification, CAV '97*, vol. 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[72] Nicolas Halbwachs and Doron Peled, Eds., *Computer-Aided Verification, CAV '99*, vol. 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[73] S. Graf and M. Schwartzbach, Eds., *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[74] E. Allen Emerson and A. Prasad Sistla, Eds., *Computer-Aided Verification, CAV '00*, vol. 1855 of *Lecture Notes in Computer Science*, Chicago, IL, July 2000. Springer-Verlag.