TRANSFORMATIONS ON DEPENDENCY GRAPHS: FORMAL SPECIFICATION AND EFFICIENT MECHANICAL VERIFICATION

by

Sreeranga Prasannakumar Rajan

B.Tech., Indian Institute of Technology (Madras, India), 1986
MS., University of Southern California (Los Angeles), 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1995

©Sreeranga Prasannakumar Rajan, 1995

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date October 2, 1995

Abstract

Dependency graphs are used to model data and control flow in hardware and software design. In a transformational design approach, optimization and refinement transformations are used to transform dependency-graph-based specifications at higher abstraction levels to those at lower abstraction levels. In this dissertation, we investigate the formal specification and mechanical verification of transformations on dependency graphs. Among formal methods, the axiomatic method provides a mechanism to specify an object by asserting properties it should satisfy. We show that an axiomatic specification coupled with an efficient mechanical verification is the most suitable formal approach to address the verification of transformations on dependency graphs.

We have provided a formal specification of dependency graphs, and verified the correctness of a variety of transformations used in an industrial synthesis framework. Errors have been discovered in the transformations, and modifications have been proposed and incorporated. Further, the formal specification has permitted us to examine the generalization and composition of transformations. In the process, we have discovered new transformations that could be used for further optimization and refinement than were possible before. We have devised an efficient verification scheme that integrates model-checking and theorem-proving, the two major techniques for formal verification, in a seamless manner.

First, we focus on the dependency graph formalism used in the high-level synthesis system part of the SPRITE project at Philips Research Labs. The transformations in the synthesis system are used for refinement and optimization of descriptions specified in a dependency graph language called SPRITE Input Language (SIL). SIL is an intermediate language used during the synthesis of hardware described using languages such as VHDL, SILAGE and ELLA. Besides being an intermediate language, it forms the backbone of the TRADES synthesis system of the University of Twente. SIL has been used in the design of hardware for audio and video applications.

Next, we present schemes for seamless integration of theorem-proving and model-checking for efficient verification. We use the Prototype Verification System (PVS) to specify and verify the correctness of the transformations. The PVS specification language, based on typed higher order logic allows us to investigate the correctness using a convenient level of abstraction. The PVS verifier features automatic procedures and interactive verification rules to check properties of specifications. We have integrated efficient simplifiers and model-checkers with PVS to facilitate verification.

Finally, we show how our method can be applied in the study of formalisms for hybrid/real-time systems, optimizing compilers, data-flow languages, and software engineering. Based on the applications of our method on such off-the-shelf for-

malisms, we substantiate our claim - that an axiomatic specification coupled with an efficient mechanical verification is the most suitable approach to specify and verify transformations on dependency graphs independent of underlying behavior models.

Contents

List of Tables

List of Figures

Acknowledgment

I am indebted to Jeff Joyce for initiating me into mechanical verification. Jeff has been an excellent advisor by consistently providing directions for research and giving the freedom to boldly explore the ideas on my own. I am thankful to Paul Gilmore, Alan Mackworth, and Mabo Ito for providing enlightening comments and discussions. I thank Carolyn Talcott of Stanford University for initiating me into formal methods and semantics of programming languages. The members of the Integrated Systems Laboratory at UBC provided a stimulating environment for research. A major part of the work reported here was done while I was at Philips Research Laboratories, Eindhoven, The Netherlands from September 1993 till April 1994. The author wishes to thank Ton Kostelijk for the invitation to work on this project, and for providing illuminating suggestions, support and a homely environment. I am grateful to Corrie Huijs, Wim Kloosterhuis, T. Krol, Jaap Hofstede, Peter Middelhoek, and Wim Smits for the cooperation, review and corrections. Thanks to group leader G. Beenker and CAD group members for a pleasant stay in Eindhoven.

I am grateful to SRI International for hosting me and providing support during the last year. The idea of combining theorem-proving and model-checking would not have matured without the excellent guidance of N. Shankar, M. Srivas, and J. Rushby of SRI International. Thanks to David Cyrluk, Pat Lincoln, and Sam Owre of SRI International, J.U. Skakkebaek of TU Denmark, and J. Hooman, G. Janssen of TU Eindhoven, and D. Stringer-Calvert of York University (UK) for discussions.

I would not have embarked on pursuing this scientific enquiry without the constant support and encouragement of my illustrious parents: Sri K. Prasanna Kumar and Smt. T. Nagarathnamma, and my celebrated brothers: Sri P. Venkat Rangan and Sri P. Srihari Sampath Kumar, and the guiding light of my venerable Gurus Swami Chinmayanandaji, Paramahamsa Yoganandaji, and Matha Amrithanandamayi. Finally, I am indebted to my wife, Suneetha, who has been an immense source of inspiration and support for a successful culmination of this dissertation on the auspicious occasion of *Vijaya Dashami* in the month of *Dasara* celebrations in India.

Dedicated to my parents
Sri K. Prasanna Kumar and Smt. T. Nagarathnamma

To Know That by Knowing Which everything else becomes Known.
– from the ***Upanishads***, the fountain head of ***Sanathana Dharma*** of India.

Among creations, I am the beginning, the middle and also the end, O Arjuna; among sciences I am the Science of the Self and I am the logic in all arguments.
– from the ***Srimad Bhagawad Geeta***, Chapter 10, Verse 32.

Chapter 1

Introduction

Dependency graphs[1] are graph-based specifications of data and control flow in a system. They are used to model systems at a high level of abstraction in both hardware and software design. Typically, dependency graphs are represented pictorially as graph structures with an associated behavior. A transformation transforms one graph structure into another by removing or adding nodes and edges. In high-level synthesis of hardware, a sequence of transformations is used for refinement of dependency-graph-based specifications at an abstract behavior level into dependency-graph-based implementations at the register-transfer level. Further, register-transfer-level implementations could be converted to concrete hardware de-

[1] In the literature, they are also known as control-flow/data-flow graphs and signal-flow graphs.

signs by low-level logic synthesis. An informal representation of dependency graphs would lead to subtle errors, making it difficult to verify the correctness of the transformations. The problem we have addressed in this work is, how the correctness of transformations on dependency graphs can be formally specified and verified.

The ***behavior***[2] of a dependency graph is the set of all tuples, where each tuple has input data values and corresponding output data values of the dependency graph. In a transformational design approach, a sequence of transformations is used for refinement of specifications into concrete implementations. A transformation is correct if the sequence of behaviors allowed by the implementation is a subsequence of the behaviors permitted by the specification. Trivial implementations that allow an empty sequence of behaviors can be ruled out by showing either, that at least one behavior is allowed by the implementation, or that the implementation is equivalent to its specification with respect to behavior. The solution to the problem of verifying the correctness of transformations we have sought in this work, is independent of models of behavior underlying dependency graphs.

A typical transformation employed in optimizing compilers is ***cross-jumping tail-merging***[EMH+93], shown in Figure 1.1. In this transformation, two identical nodes on dependency paths that are never active at the same time are merged into one node. However, as we found out using our approach explained in this paper, the

[2]Usually known as input/output behavior.

transformation does not preserve behavior. Informally, the reason is as follows. The conditions True/False on open/filled bullets in Figure 1.1 on the bottom/top of the nodes control the execution of the nodes. Further, we stipulate[3] that the value of p2 is determined by the edge that transmits a well-defined value to it - i.e. when c is *true*, q20 determines the value of p2, while q21 determines its value otherwise. In graph G1, when c is *false*, the value of q10 is arbitrary, and so is the value of p10. If we choose the value of r1 to be that of p10, the value of r1 is also arbitrary. In graph G2, when c is *false*, the value of p2 is that of q21. In this case, the value of r2 is (y1 * y2). Thus, the set of outputs for a given y1 and y2 is the set of all values allowed by the type of the output port in graph G1. Whereas in graph G2, the set of outputs is a singleton set containing the element (y1 * y2). Since, with identical inputs the corresponding sets of possible outputs are unequal, the behaviors of the graphs are not equivalent. A corrected and generalized cross-jumping tail-merging transformation is presented in Chapter 7.

The main contributions of this work are the following:

- A formal specification of dependency graphs, independent of underlying behavior models, has been achieved.

- A set of optimization and refinement transformations on dependency graphs

[3]Section 2.2 explains this stipulation on how data conflicts at ports called *joins*, such as p2, are resolved.

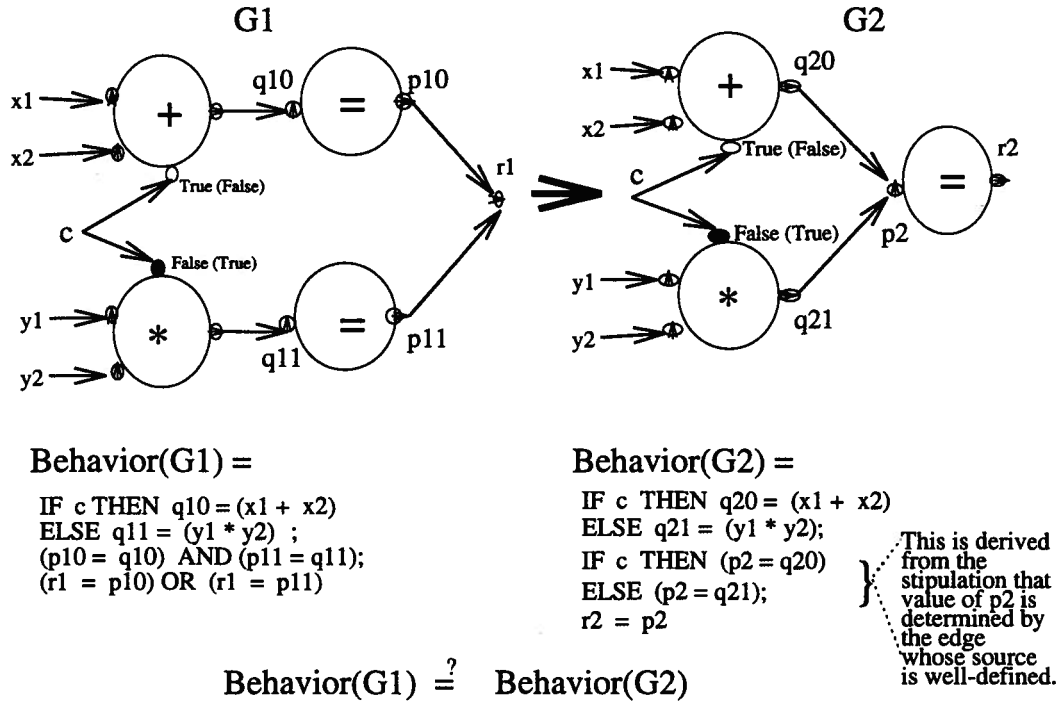


Figure 1.1: Cross jumping tail merging: incorrectly specified in informal document.

have been verified. Generalization of transformations have also been proposed.

- Errors have been discovered in the transformations used in industrial strength hardware design. Modifications for the erroneous transformations have been proposed and incorporated.

- New transformations have been devised that could be used for further optimization and refinement than was possible before.

- An efficient verification scheme that provides a seamless integration of theorem-proving and model-checking - the two major formal verification techniques - has been achieved.

- Application of our work in other domains such as modeling hybrid/real-time systems, optimizing compilers, data-flow languages, and software engineering has provided rigor for formalisms used in those domains.

Formal methods could be divided into two main categories: property-oriented methods and model-oriented methods [Win90]. In a property oriented method, the system under consideration is specified by asserting properties of the system, minimizing the details of how the system is constructed. While, in a model-oriented method, the specification describes the construction of the system from its components. An axiomatic approach is a property-oriented method. Typically, a small set of properties, called *axiom*s, are asserted to be true, while other properties, called *theorem*s, are derived. In this work, we have chosen a property oriented method. We propose an axiomatic specification coupled with an efficient verification method to study the correctness of transformations on dependency graphs [Raj94a]. As we discuss later in Chapter 9, an axiomatic approach does not require us to develop a concrete behavioral model for dependency graphs, thus enabling it to be simpler and more general than other formal approaches.

A *dependency graph*[4] is a graph-based representation of the behavior of a system. It consists of nodes representing operations or processes, and directed edges representing data dependencies and data flow through the system. In addition, control

[4]In this dissertation, the term *dependency graph* includes control-flow/data-flow graphs and signal-flow graphs.

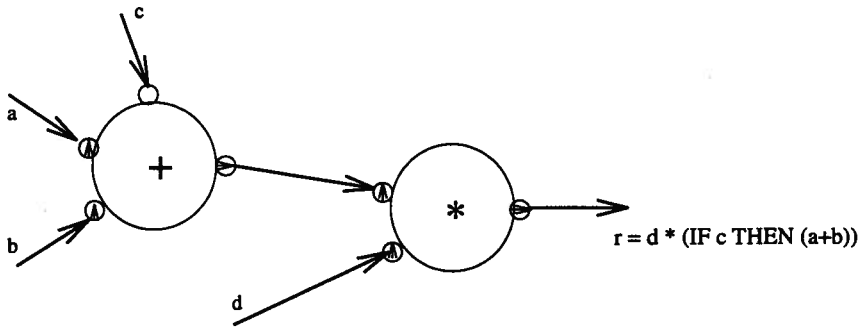


Figure 1.2: Example of a dependency graph with control specification.

flow could also be represented in a dependency graph in several ways. We show an example of such a graph in Figure 1.2.

In order to present our work in a concrete context, we consider a transformational design approach used in the ***high-level behavioral synthesis*** system as part of the SPRITE project at Philips Research Labs (PRL). In this approach, transformations are used for optimization and refinement of descriptions specified using the SPRITE Input Language (SIL). Descriptions in SIL at a ***register-transfer level*** could eventually be converted to ***gate-level*** hardware designs by a ***logic synthesis*** application such as PHIDEO at PRL.

SIL is an intermediate language used during the synthesis of hardware described using hardware description languages such as VHDL [oEE88], SILAGE [Hil85], and ELLA [Compu90]. It also forms the backbone of the TRADES synthesis system at the University of Twente. Important features of SIL include hierarchy and design freedom. Design freedom is provided by permitting several implementation

choices for a SIL description. Implementation choices are constrained by allowing an implementation suggestion in a SIL description. The implementation suggestion may be tailored by using refinement and optimization transformations. SIL has been used in the design of hardware for audio and video signal processing applications such as a direction detector for the progressive scan conversion algorithm [vdWvMM+94, Mid94a]. In one of the applications [Mid94b], a reduction of power consumption by 50% has been achieved.

Many of the optimization transformations used in SIL are inspired by those used in compiler optimization, such as dead-code elimination and common subexpression elimination. An optimized SIL graph has to satisfy the original graph with respect to behavior. This satisfaction can be guaranteed by showing the correctness of the optimization transformations. Correctness means that every behavior allowed by an optimized SIL graph implementation is required to be one of the behaviors allowed by its SIL graph specification. An informal specification of SIL has been presented and documented as part of the SPRITE project [KeH+92, KMN+92]. A detailed denotational semantics of SIL for showing the correctness of transformations has been worked out earlier [HHK92, HK93]. The optimization and refinement transformations have been specified informally as part of the SPRITE project [EMH+93, Mid93, Mid94b].

We use the Prototype Verification System (PVS) [OSR93b], an environment for

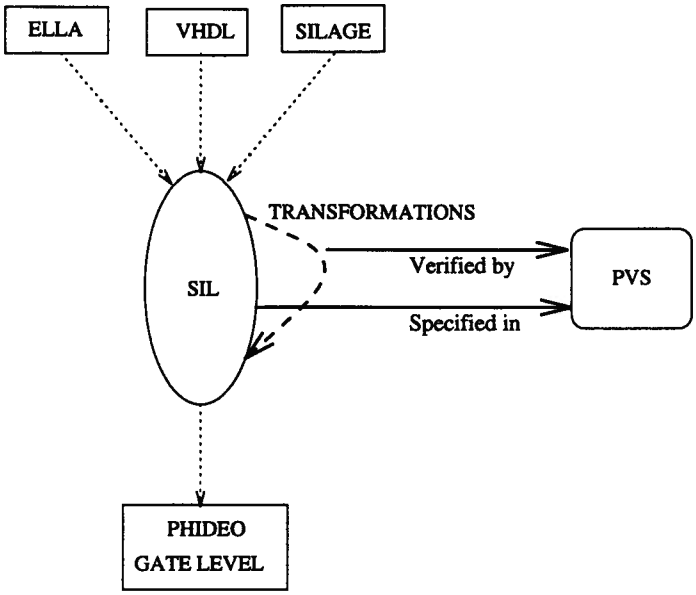


Figure 1.3: SIL transformations and verification in PVS

formal specification and verification. The PVS specification language, based on typed higher order logic, permits an axiomatic method to develop specifications. This entails expressing properties of a system at a convenient level of abstraction. The choice of a high level of abstraction obviates the need to provide a detailed definition of the behavior of data flow graphs. Thus, without specifying a detailed behavioral model of a SIL description, we can still compare descriptions with respect to their behavior, thus establishing the correctness of transformations [Raj94b]. However, we stress that this work addresses the transformations as intended in their informal specification, and not verification of the software implementations of transformations. We show SIL and our work in the context of the synthesis system in Figure 1.3.

The rest of this dissertation is organized as follows: Chapter 2 gives an overview of SIL. In Chapter 3, we give a brief description of the PVS formal system. We discuss in Chapter 4 our work on integration of model-checking to PVS for efficient verification. This work has enabled verification of many classes of hardware designs to be fully automatic [KK94]. In Chapter 5, we describe the specification of structure of SIL graphs, while in Chapter 6, we describe the specification of behavior, refinement, and equivalence of SIL graphs. We present the specification and verification of transformations in Chapter 7. In Chapter 8, we show how our formalism could be applied in other domains such as, modeling of real-time systems, compiler optimization, data-flow languages, and structured methods in software design. Finally, following a general discussion, conclusions are summarized in Chapter 9. A compilation of the specification of SIL and its verified properties in PVS is given in Appendix B. Transcripts of the verification in PVS for two transformations discussed in detail in this paper are listed in Appendix C. The theories and proof transcripts given in the appendices were automatically generated from the corresponding PVS specifications and proof runs. In the remainder of this chapter, we discuss related work done in the past.

1.1 Related Work

There have been some efforts in analysis and verification of refinement transformations in the past. One of the earliest efforts on formalizing the correctness of optimization transformations is by Aho, Sethi and Ullman [ASU71]. They formalized a restricted class of transformations known as Finite Church-Rosser (FCR) transformations, and provided simple tests to check properties of such transformations. However, none of the past work has dealt with transformations on dependency graphs in general. Most of the efforts have concentrated on specialized hardware description languages and programming languages. Many other efforts have provided a formal operational and denotational semantics for variations of dependency graph formalisms. But, the operational and denotational semantics, unlike axiomatic semantics, are tied to a specific behavior model. This makes them unsuitable to verify transformations on dependency graphs of arbitrary size. Further, it would not be possible to extend such concrete operational or denotational models to dependency graphs used in a multitude of formalisms such as in software engineering and real-time systems modeling.

A formal model was proposed for verifying correctness of high-level transformations by McFarland and Parker [MP83]. Transformations used in YIF (Yorktown Internal Form) [BCD+88] have been proved to be behavior preserving [Cam89]. In this work, a strong notion of behavior equivalence based on an operational seman-

tics tied to a particular model of representation is used. A formal system using transformations for hardware synthesis has been discussed by Fourman [Fou90]. We briefly discuss this work in Section 1.1.1. A synthesis system for a language based on an algebraic formalism has been presented by Jones and Sheeran [JS90], and its formalization has been presented by Rossen [Ros90]. This effort is explained briefly in Section 1.1.2. Another algebraic approach to transformational design of hardware has been worked out by Johnson [Joh84]. A short discussion on this approach is presented in Section 1.1.3. In the work on tying formal verification to silicon compilation [JLR+91], a preliminary study with an emphasis on the use of formal verification at higher levels of VLSI design was presented. Correctness of register-transfer-level transformations for scheduling and allocation has been dealt with in [Vem90]. An automatic method for functional verification of retiming, pipelining and buffering optimization has been presented by Kostelijk [KvdW93]. It has been implemented in a CAD tool called RetLab as part of PHIDEO. A formal analysis of transformations used in Systems Architect Workbench (SAW) high-level synthesis was studied by McFarland [McF93]. This work is discussed briefly in Section 1.1.4. A ***post-facto*** verification method for comparing logic level designs against a restricted class of data flow graphs in SILAGE was presented by Aelten and others [AAD93, AAD94]. A formalization of SILAGE transformations in HOL was studied by Angelo [Ang94]. A concise description of this work appears in Section 1.1.5. An approach based on the execution model for representation languages in

BEDROC high-level synthesis system [CBL92] has been used to verify the correctness of optimization transformations. A formal verification of an implementation of a logic synthesis system has been reported by Aagard and Leeser [AL94], but it does not provide a mechanical verification for optimization and refinement transformations in high-level behavioral synthesis. In Section 1.1.6, we briefly discuss the work on formal specification and verification of refinement transformation in software design.

1.1.1 LAMBDA

LAMBDA [Fou90] is formal system based on higher order logic for designing hardware from high level specifications. In this formalism, a design state is represented as an inference rule derived within the framework of higher order logic. A refinement is a rule derived within this logic that can be applied to an abstract design state to arrive at a concrete design state. The different kinds of refinements that are applied are temporal, data and behavioral. However, a definite set of refinement and optimization transformations have not been presented. ELLA, a hardware description language has been formalized in LAMBDA.

1.1.2 Formal Ruby

In this work, an algorithmic specification of sequential and combinational circuits is specified in a language called Ruby [JS90], based on an algebraic formalism. The algebraic formalism consists of relations and operations on relations such as composition, inversion and conjugation. Types are defined as equivalence relations. Data structures such as lists and tuples are used to represent larger hardware structures. A parallel composition operator allows specification of hardware composed of independent modules. Other operators such as row and column are introduced for succinct specification of regular structures such as systolic arrays.

Ruby has been formalized [Ros90] in a proof checking system called ISABELLE. ISABELLE, based on type theory, allows syntactic embedding of other logics. A fragment of Ruby corresponding to combinational circuits, delay elements, serial composition and parallel composition called Pure Ruby is specified as a type. Properties and proof rules such as induction on Ruby terms are then derived on the type definition. The rest of the language is then specified using this type.

The axiomatization specifies signals as functions of time and properties of relations on signals. General properties of Ruby relations have been formalized. However, in order to derive properties, the semantic embedding involves signals corresponding to a circuit implementation. A Ruby specification itself, and hence its formalization even at a high level is geared to be directly translatable to a circuit

realization having a regular structure. Thus, this formalism is at a lower level of abstraction than our formalization of SIL. A general concept of refinement is not formalized. The formalism does not present a well-defined set of transformations, to be used to refine and optimize Ruby programs, other than retiming.

1.1.3 Digital Design Derivation

This is an algebraic approach to transformational design of hardware [Joh84]. In this formalism, a functional specification is translated into a representation of a Deterministic Finite State Machine specification called behavior tables [RTJ93]. The behavior tables are transformed into a digital design. In a behavior table, rows represent state transitions and columns represent both control and data flow. Some examples of transformations are column merging, deletion and renaming. The transformations are not formally verified.

1.1.4 Transformations in SAW

In this work, a formal analysis of transformations [McF93] used in System Architect's Workbench (SAW) [TDW+88] is carried out. In this system, hardware described at the register-transfer level or higher using ISPB [Bar81] is translated into behavior expressions. Behavior expressions use sequences and relations on sequences to

represent the input/output behavior of the specified hardware. Optimization transformations are carried out on the behavior expressions representations. A number of transformations such as constant folding and loop unwinding have been analyzed revealing a few conceptual errors.

1.1.5 Verification of Transformations in SILAGE

SILAGE [Hil85] is an applicative hardware description language. This is used to describe hardware represented as data flow graphs. Transformations such as commutativity and retiming are used to optimize and refine SILAGE descriptions. In this work [Ang94], the syntax and semantics of SILAGE programs have been formalized as predicates in HOL [GM93] The denotational semantics of SILAGE have been formalized in HOL. The equivalence of SILAGE programs is specified with respect to this denotational semantics. The transformations are then specified as functions from one formal SILAGE program to another. The correctness of transformations are thus verified with respect to the denotational semantic notion of equivalence.

1.1.6 Transformations in Software Design

There have been several efforts in specification and verification of refinements used in program development from high level specifications Most of the efforts choose a

specification formalism and develop a notion of correctness, and an associated set of transformations based on the semantics of the formalism.

The refinement calculus [Bac88] for specifications based on Dijkstra's guarded command language and weakest precondition semantics has been formalized in HOL [vWS91]. Transformations such as data refinement and superposition have been verified to be correct. A formalization of incremental development of programs from specifications for distributed real-time systems has been worked out in PVS [Hoo94]. In this formalism, an assertional method based on a compositional framework of classical Hoare triples is developed for step-wise refinement of specifications into programs.

The KIDS [Smi90]system is a program derivation system. High level specifications written in a language called Refine are transformed by data type refinements and optimization transformations such as partial evaluation, finite differencing, into a Refine program.

Chapter 2

Overview of SIL

The descriptions in SIL are characterized as graphs. They are used to describe synchronous systems. A denotational semantics of SIL has been worked out by Huijs [HK93]. An operational semantics of data-flow graphs has been worked out by de Jong [dJ93]. The behavior of a SIL graph is derived from the behaviors of structural building blocks of the graph. We briefly explain the structural aspects in section 2.1, the behavioral aspects in Section 2.2, and the transformational approach in Section 2.3. Finally, we provide a brief overview of formal operational and denotational semantics in Section 2.4.

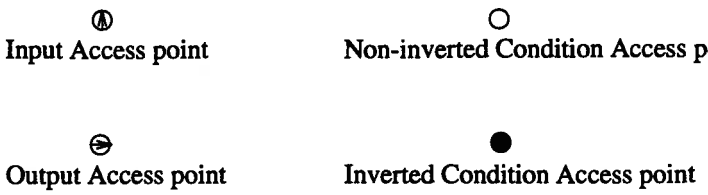


Figure 2.1: Different kinds of SIL ports.

2.1 Structural Aspects of SIL

The basic building blocks of a SIL graph are the nodes for operations such as addition, multiplication, and multiplexing. The nodes have ports (also known as access points) for input, output, and an optional condition input. Every port is associated with a type, which specifies the set of data values that the port can hold. We show the different kinds of port in Figure 2.1.

While input and output ports can be of any type, a condition input port is always Boolean. A node with condition input port is known as a conditional node to stress the presence of the condition inputs.

The ports of the nodes are connected by edges. SIL has different kinds of edges, of which, we address *sequence edge* and *data-flow edge*:

- A *data-flow edge* is used to specify the direction of communication of data values from a *source* port to a *sink* port. Each data-flow edge has exactly one port at its head and exactly one port at its tail. A source port can be the tail of more than one data-flow edge, in which case it is called a distribute, and a

sink port the head of more than edge, in which case it is called a join.

- A *sequence edge* specifies an ordering between two ports. The ordering is used to indicate that one of the ports has the overriding influence on the value of the sink port, to which the two ports are connected by data-flow edges. Each sequence edge has exactly one port as its tail and one port as its head. Sequence edges are primarily used to resolve potential conflicts at joins. All source ports that are tails of data-flow edges with a join as a head must be linearly ordered by sequence edges.

- The nodes and edges form a SIL graph. A SIL graph itself can be viewed as one single node, and used to construct another SIL graph in a hierarchical manner. Figure 2.2 is an example of a SIL graph.

2.2 Behavioral Aspects of SIL

The behavior of a SIL graph is determined by the behavior of individual nodes and their connectivity, which determines the data flow. By behavior, we mean the set of tuples, where each tuple has input data values and corresponding data values of internal and output ports. The values of internal and output ports are constrained by the data relations of the nodes and the connectivity of the ports in the graph. When the ports of interest are the outermost input / output (I/O) ports of the SIL

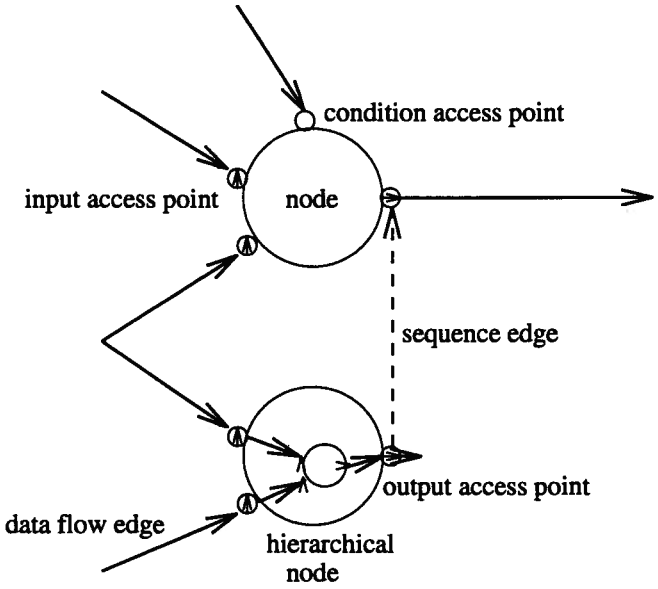


Figure 2.2: An example of a SIL graph description.

graph, then it is called external or I/O behavior.

Each node is associated with a data relation and an order relation. The data relation of a node constrains the outputs of the node according to the inputs of the node. That this is a ***relation***, and not a ***function***, implies nondeterminism allowing several implementation choices for the nodes. This contributes to design freedom. Any state information implicit in the node is incorporated into its data relation. In the case of a conditional node, the output is constrained by the data relation only when the condition input of the node is true. When the condition input is false, the output is not defined. The order relation specifies constraints such as, the output port of a node assumes a value after the value of its input ports have been asserted. This is particularly important in a hierarchically built node. We illustrate these

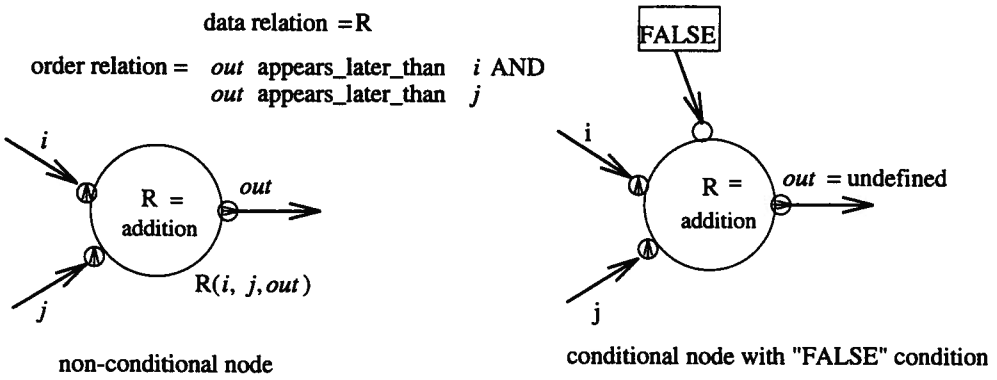


Figure 2.3: SIL node: informal description.

concepts in Figure 2.3.

The communication of data values in a SIL graph is modeled by a ***single token flow concept***, similar to the concept in Signal Flow Graphs (SFG) [Hil85]. A ***token*** is an atomic symbol denoting data. A token generated at an output port (source) is transmitted through a data-flow edge, emanating from the source, exactly once. The token is consumed at an input port (sink) to which the edge is connected. The action of communicating a token through a data-flow edge makes the sequence of values that the sink can assume equal to the sequence of values that the source can assume. However, there is one exception to this when a token communicated to the conditional port of a conditional node denotes a data value that is ***false***. In this case, the output port, unconstrained by the data relation of the conditional node, is not defined. When such an output is a source of a data-flow edge, we force the sink of such a data-flow edge to assume some well-defined arbitrary value. If we do not make this exception, the sink data values would also not be well-defined. Since a

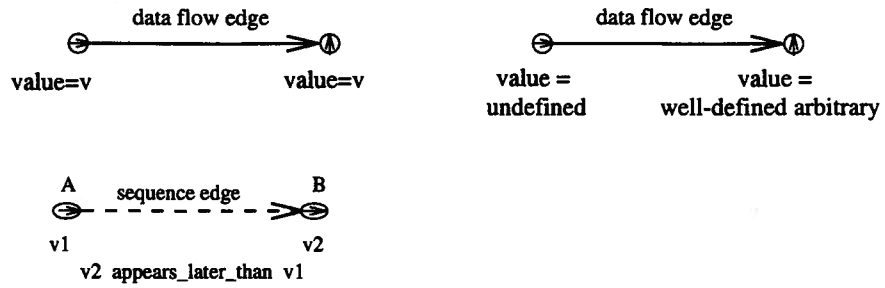


Figure 2.4: SIL edges: informal description.

sink is an input port, it is undesirable to have undefined inputs in practice. In terms of the token flow concept, a sequence edge from port A to port B describes that the token fired from B determines the value of a sink port C connected to A and B by data-flow edges, overriding the effect on the value of C due to the token fired from A. In such a case, we say that the sequence edge orders port A less than port B. A data-flow edge has an implicit sequence edge from its source to its sink. We depict these ideas in Figure 2.4. It should be noted that the token flow concept is an abstract model of the behavior of a SIL graph. The sequence edge is an artifact used to resolve conflicts at joins. A sequence edge does not indicate temporal ordering of the data values that ports would assume when a SIL graph is executed.

The ordering of token communication plays an important part in resolving conflicts at ports. One such conflict occurs when multiple data-flow edges from different sources connect into a single sink. Such a sink port is called a ***join***, as shown in Figure 2.5. To resolve the conflict at a join, first all the data-flow edges that have sources that have assumed well-defined data values are selected. Then, among those

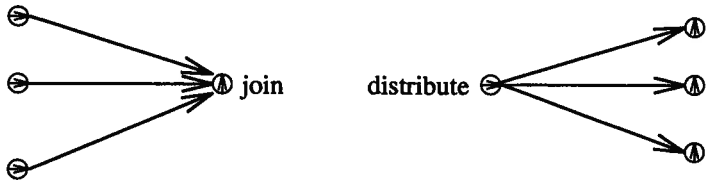


Figure 2.5: SIL Join and Distribute: informal description.

selected data-flow edges, the edge that is responsible for communicating the last token determines the behavior of the join. With the definition of SIL, there will be exactly one such data-flow edge. Thus, the source ports are linearly ordered, so that the last of the well-defined data values arriving at the sink is always specified. If all the data-flow edges to the join originate from sources whose data values are undefined, then the data value that can appear at the join is arbitrary.

The counterpart of a join is a source from which multiple data-flow edges originate. Such a port, known as a ***distribute***, is shown in Figure 2.5. If a distribute is a source that assumes well-defined data values, then the sink to which it is connected by a data-flow edge, will assume a sequence of data values identical to the distribute. Otherwise, if the data values that may appear at the distribute are not defined, the sequence of data values that may appear at the corresponding sink ports are arbitrary.

A SIL graph models the behavior of a system during a single clock cycle. There is no explicit notion of state in a SIL graph. The repetition of a SIL graph, called ***unfold***ing over multiple clock cycles gives the behavior of the system across clock

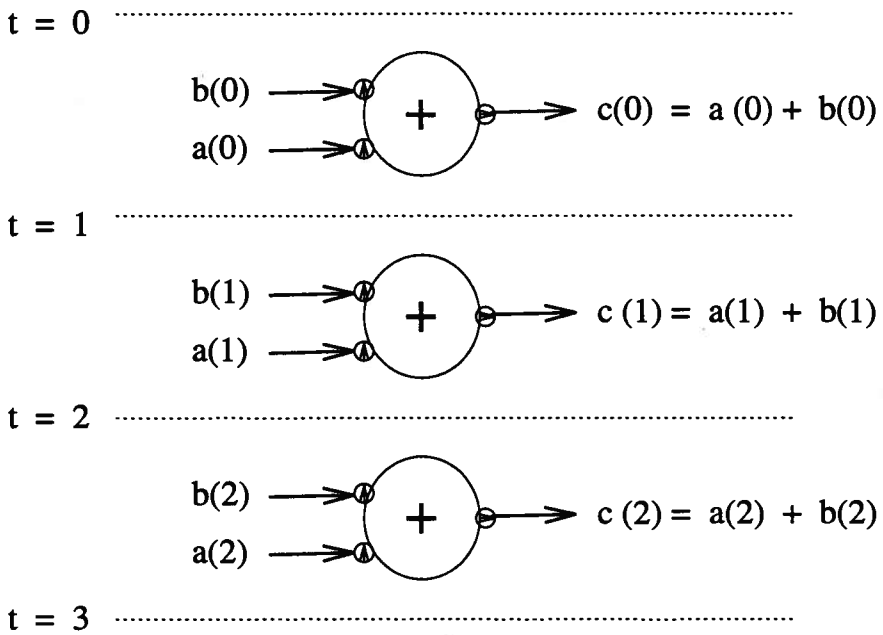


Figure 2.6: Combinational adder: SIL graph repeated over clock cycles.

cycles. We depict an example of a combinational adder in Figure 2.6 unfolded over three clock cycles. The **DELAY** node, one of the primitive nodes in SIL is used to model data flow between clock cycles, and thus encapsulates state information. We can unfold the SIL graph shown in Figure 2.7 over multiple clock cycles to result in a SIL graph without the DELAY node. The ***cumulative*** adder example in Figure 2.8 illustrates the unfolding of a SIL graph with a DELAY node. It should be noted that comparing two graphs with respect to behavior would not involve the state information encapsulated in a **DELAY** node - since the behavior of a SIL graph would be a snapshot of the execution of the SIL graph in a single clock cycle. In contrast, the execution histories would have to be taken into account for comparing two state

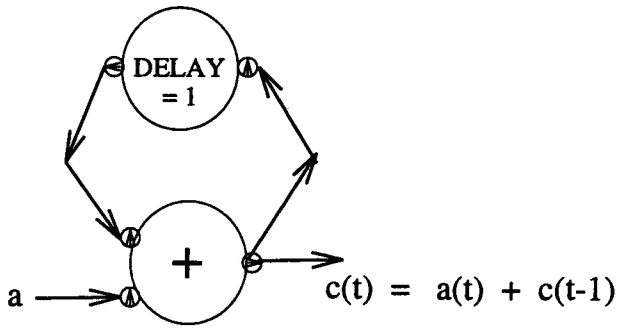
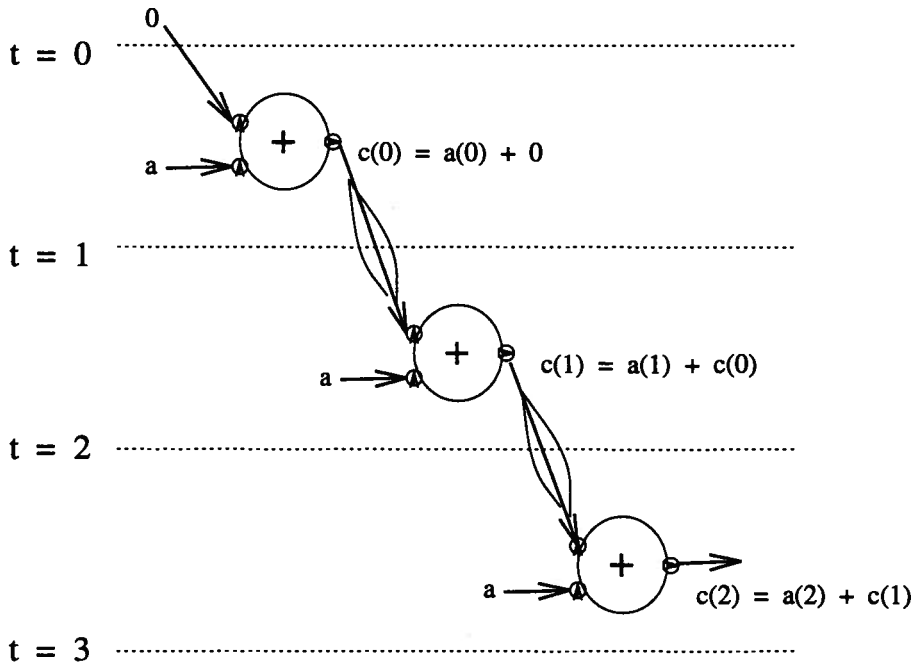


Figure 2.7: Cumulative adder: SIL graph with DELAY node.

machine models.

The ordering imposed by sequence edges reduce non-determinism. This leads to a restriction on implementation choices allowed by its corresponding specification. We illustrate the implementation of a simple multiplexor in Figure 2.10 by reducing non-determinism in a specification shown in Figure 2.9 using a sequence edge. When `c` is ***true***, the value of `d` is `a` if the order is such that value of port `p1` is communicated rather than that of port `p2`. If the order is such that `p2` has the overriding influence, then the value of `d` is `b`. While, when `c` is ***false*** the value of `b` is determined by the port `p2`, due to the behavior of the conditional port and join discussed earlier in section 2.2. The sequence edge in the multiplexor implementation as given in Figure 2.10, imposes that the value communicated to `b` is that of port `p1` when `c` is ***true***. Again, when `c` is ***false***, port `p2` determines the value of `b`.



$$c(-1) = 0$$

$$c(t) = a(t) + c(t\text{-}1)$$

Figure 2.8: Cumulative adder: unfolded SIL graph.

2.3 Transformations in SIL

A transformation is viewed as modifying the structure of a graph into another graph. The modification is done by removing and/or adding nodes and edges. Such modifications should not violate the behavior of the original graph.

In SIL, there are a number of optimization and refinement transformations [EMH+93]. Many of the optimization transformations are inspired by optimizing compiler transformation techniques such as *Common Subexpression Elimination*,

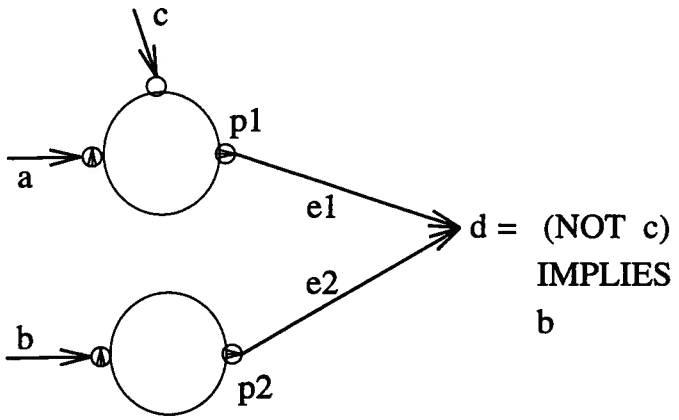


Figure 2.9: Partial specification of a multiplexor.

Cross-Jumping Tail-Merging and algebraic transformations involving ***commutativity***, ***associativity***, and ***distributivity***. Other optimization transformations include ***retiming***. Refinement transformations include type transformations such as ***real to integer***, ***integer to Boolean***, and implementing data relations of the nodes by concrete operators [Mid94b]. We show a retiming transformation example in Figure 2.11

2.4 Operational and Denotational Semantics of Dependency Graphs

In general, the formal semantics of a specification language can be provided by the following three methods:

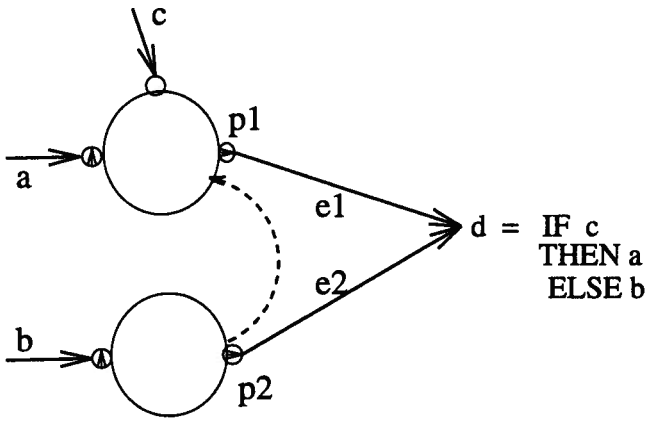


Figure 2.10: Implementation specification of a multiplexor.

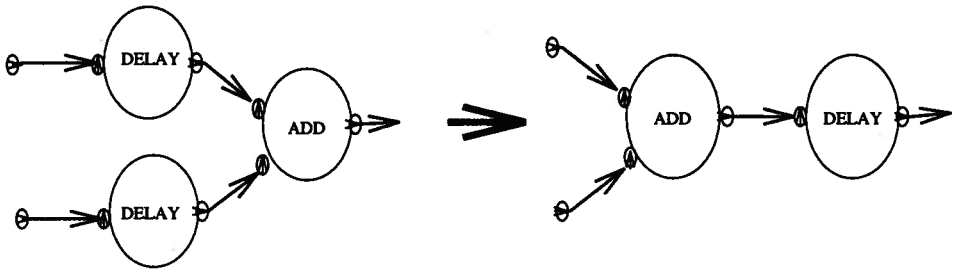


Figure 2.11: Example SIL transformation: retiming.

- Axiomatic semantics: the properties corresponding to the basic syntax of the language are asserted as axioms, and the method of deriving the semantics of other language constructs is given by a set of inference rules. The axiomatic specification of dependency graphs provided in this dissertation is an axiomatic semantics.

- Operational semantics: the syntactic elements of the language are associated with states, and the execution of the syntax is modeled as state transitions.

Thus, it provides a computation model for the specification language - i.e., *how* computation is performed in the specification.

- Denotational semantics: the syntactic elements of the language are mapped to an abstract domain of values by a semantic function. Thus, it describes *what* a specification computes.

An axiomatic semantics is suited to reason about specification language in general, as well as particular instances of specifications written in the specification language. An operational semantics of a specification language provides a mechanism to an interpret a specification. The operational semantics described in an inferential style is called structured operational semantics [Gun92]. Denotational semantics provides a compositional semantic function for a specification language. The overview of operational and denotational semantics of dependency graphs given here is based on earlier work [dJ93, HK93].

We first define a dependency graph G as a 7-tuple $(N, P_{in}, P_{out}, E, I, O, C)$, where

- N is the set of nodes.
- P_{in} is the set of input ports.
- P_{out} is the set of output ports.
- P_{cond} is the set of condition ports.

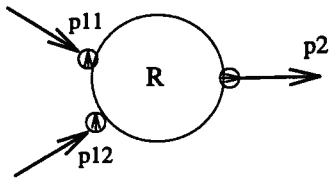


Figure 2.12: A simple node used as an example for operational semantics

- E: $P_{out} \times P_{in} \cup P_{cond}$ is the set of edges.
- I: V $\rightarrow$ Powerset(P_{in}) is a mapping from nodes to input ports.
- O: V $\rightarrow$ Powerset(P_{out}) is a mapping from nodes to output ports.
- C: V $\rightarrow$ Powerset(P_{cond}) is a mapping from nodes to condition ports.

2.4.1 Operational Semantics of Dependency Graphs

An operational semantics of dependency graphs describes *how* a computation in the dependency graph takes place. We briefly outline here the structured operational semantics of dependency graphs [dJ93]. In this style, the computation in a dependency graph is viewed as a sequence of state transitions, where each state transition takes place by executing one node in the dependency graph. State transitions are given in the form of inference rules. A state is represented by a mapping from pairs of node, input ports to a domain of streams:

$$S : N \times P_{in} \rightarrow D_{Stream}$$

where D_{Stream} is the domain on streams.

$$\frac{S(R,p11) \rightarrow S'(R,p11); S(R,p12) \rightarrow S'(R,p12)}{S(R1,p2) \rightarrow S'(R1,p2)}$$

Table 2.1: Example of a structured operational semantics rule. See Figure 2.12

Every state transition rule is based on an update rule that specifies the state update on an execution of a node. The state update would cause the output stream to be updated. We give an example of a semantic rule for the graph given in Figure 2.12 where `S'(R,p)` on a port `p` of node `R` denotes the stream obtained by appending an element to the stream `S(R,p)`. The expression in Table 2.1, above the horizontal line describes the state transition of the input ports of the node given in Figure 2.12. The expression below the horizontal line describes the state transition of the output port of the node. Thus, the structured operational semantics rule means that, if the state transition above the horizontal line is executed, then the state transition below the horizontal line gets executed.

For verification, a comparison of two dependency graphs would be based on the comparison of the output streams, obtained by the computation of the two graphs with identical input streams for each of the graphs. A containment of streams would indicate that the graph whose stream is contained in that of the other graph is a refinement of that graph. However, because the operational model is at a lower level of abstraction than the axiomatic model, as seen in the following chapters of this dissertation, one would not be able to generalize the results for dependency flow

graphs of arbitrary structure and application domain. For example, a node with an arbitrary or indefinite number of input/output ports would not be handled by the operational semantics.

2.4.2 Denotational Semantics of Dependency Graphs

A denotational model specifies *what* a dependency graph computes. It is compositional. In a denotational semantic model, a meaning function maps syntactic objects such as ports, edges and nodes to an abstract domain of semantic objects. Thus, for example, if we define the meaning of a port p as:

$$[\![p]\!] = D_{stream}$$

$$[\![R]\!] \subset [\![(p \times p \rightarrow p)]\!]$$

$$\Rightarrow [\![R]\!] \subset ([\![p]\!] \times [\![p]\!] \rightarrow [\![p]\!])$$

where D_{stream}, the domain of streams, is associated with a partial order $\leq$ and a least element $\perp$. The denotational semantics of the graph shown in Figure 2.12 is obtained by composing the denotations of the ports and the node:

$$[\![p2]\!] = [\![R]\!]([\![p11]\!], [\![p12]\!])$$

Chapter 3

Specification and Verification in PVS

The Prototype Verification System (PVS) [OSR93b, SOR93a] is an environment for specifying entities such as hardware/software models and algorithms, and verifying properties associated with the entities. An entity is usually specified by asserting a small number of general properties that are known to be true. These known properties are then used to derive other desired properties. The process of verification involves checking relationships that are supposed to hold among entities. The checking is done by comparing the specified properties of the entities. For example,

one can compare if a register-transfer-level implementation of hardware satisfies the properties expressed by its high-level specification.

PVS has been used for reasoning in many domains, such as in hardware verification [Cyr93, KK94], protocol verification, algorithm verification [LOR+93, ORSvH95], and multimedia [RRV95]. We briefly give the features of the PVS specification language in Section 3.1, the PVS verification features in Section 3.2, and highlight the syntax of the PVS specification language in Section 3.3. Finally, in Section 3.4 we give some example specifications and verification sessions in PVS.

3.1 PVS Specification Language

The specification language [OSR93b] features common programming language constructs such as arrays, functions, and records. It has built-in types for reals, integers, naturals, and lists. A type is interpreted as a set of values. One can introduce new types by explicitly defining the set of values, or indicating the set of values, by providing properties that have to be satisfied by the values. The language also allows hierarchical structuring of specifications. Besides other features, it permits overloading of operators, as in some programming languages and hardware description languages such as VHDL.

3.2 PVS Verification Features

The PVS verifier [SOR93a] is used to determine if the desired properties hold in the specification of the model. The user interacts with the verifier by a small set of commands. The verifier contains procedures for boolean reasoning, arithmetic and (conditional) rewriting. In particular, Binary Decision Diagram (BDD) [BRB90, Jan93a] based simplification may be invoked for Boolean reasoning. It also features a variety of general induction schemes to tackle large-scale verification. Moreover, different verification schemes can be combined into general-purpose strategies for similar classes of problems, such as verification of microprocessors [Cyr93, KK94].

A PVS specification is first parsed and type-checked. At this stage, the type of every term in the specification is unambiguously known. The verification is done in the following style: we start with the property to be checked and repeatedly apply rules on the property. Every such rule application is meant to obtain another property that is simpler to check. The property holds if such a series of applications of rules eventually leads to a property that is already known to hold. Examples illustrating the specification and verification in PVS are described in Section 3.4.

3.3 Notes on Specification Notation

In PVS specifications[1], an object followed by a colon and a type indicates that the object is a constant belonging to that type. If the colon is followed by the key word *VAR* and a type, then the object is a variable belonging to that type.

For example,

```
x: integer

y: VAR integer
```

describes x as a constant of type integer, and y as a variable of type integer[2].

Sets are denoted by {...}: they can be introduced by explicitly defining the elements of the set, or implicitly by a characteristic function.

For example,

```
{0,1,2}

{x: integer | even(x) AND x /= 2}
```

The symbol | is read as ***such that***, and the symbol **/=** stands for ***not equal to*** in general. Thus, the latter example above should be read as "set of all integers x, such that x is an even number and x is not equal to 2".

[1] PVS specifications in this dissertation are enclosed in framed boxes.
[2] In C, they would be declared as *const int x; int y.*

New types are introduced by a key word *TYPE* followed by its description as a set of values. If the key word *TYPE* is not followed by any description, then it is taken as an uninterpreted type.

Some illustrations are:

```
even_time: TYPE = {x: natural| even(x)}

unspecified_type: TYPE
```

One kind of type that is used widely in this work is the ***record type***. A record type is like the ***struct*** type in the C programming language. It is used to package objects of different types in one type. We can then treat an object of such a type as one single object externally, but with an internal structure corresponding to the various fields in the record.

The following operators have their corresponding meanings:

```
FORALL x: p(x)
```

means ***for every*** x, predicate[3] p(x) is ***true***

[3]A predicate is a function returning a Boolean type: {*true, false*}.

```
EXISTS x: p(x)
```

means ***for at least a single*** x, predicate p(x) is ***true***

We can impose constraints on the set of values for variables inside **FORALL** and **EXISTS** as in the following example:

```
FORALL x, (y| y = 3*x): p(x,y)
```

which should be read as

for every x and y such that y is 3 times x, p(x,y) is ***true.***

A property that is already known to hold without checking is labeled by a name followed by a colon and the keyword **AXIOM**. A property that is checked using the rules available in the verifier is labeled by a name followed by a colon and the keyword **THEOREM**. The text followed by a **%** in any line is a comment in PVS.
We illustrate the syntax as follows:

```
ax1: AXIOM  % This is a simple axiom
FORALL (x:nat): even(x) = x divisible_by 2


th1: THEOREM  % This is a simple theorem
FORALL (x:nat):  prime(x) AND x /= 2 IMPLIES NOT even(x)
```

We also use the terms *axiom* and *theorem* in our own explanation with the same meanings. A *proof* is a sequence of deduction steps that leads us from a set of axioms or theorems to a theorem.

3.4 Specification and Verification Examples in PVS

We illustrate here three examples from arithmetic. The first two examples are taken from the tutorial [SOR93b]. The last example illustrates the use of a general purpose strategy to automatically prove a theorem of arithmetic. The first example is the sum of natural numbers up to some arbitrary finite number n is equal to $n*(n+1)/2$. The specification is encapsulated in the **sum THEORY**. Following introduction of **n** as a natural number **nat**, **sum(n)** is defined as a recursive function with a termination **MEASURE** as an identity function on **n**. Finally, the **THEOREM** labeled **closed_form** is stated to be proved.

```
sum: THEORY
 BEGIN

 n: VAR nat

 sum(n): RECURSIVE nat =
  (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
  MEASURE (LAMBDA n: n)

 closed_form: THEOREM sum(n) = (n * (n + 1))/2

 END sum
```

The **THEORY** is first parsed and type checked, and then the prover is invoked on the **closed_form THEOREM**. The proof is automatic by applying induction and rewriting. The proof session is as follows:

```
closed_form :

  |-------
{1}   (FORALL (n: nat): (sum(n) = (n * (n + 1)) / 2))
```

```
Running step: (INDUCT "n")
```

Inducting on n, this yields 2 subgoals:

```
closed_form.1 :


  |-------

{1}   sum(0) = (0 * (0 + 1)) / 2


Running step: (EXPAND "sum")
```

Expanding the definition of sum, this simplifies to:

```
closed_form.1 :


  |-------

{1}   0 = 0 / 2


Rerunning step: (ASSERT)
```

Invoking decision procedures, this completes the proof of closed_form.1.

```
closed_form.2 :


  |-------

{1}   (FORALL (j: nat):
```

```
        (sum(j) = (j * (j + 1)) / 2
           IMPLIES sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2))

Running step: (SKOLEM 1 ("j!1"))
```

For the top quantifier in 1, we introduce Skolem constants: (j!1), this simplifies to:

```
closed_form.2 :


  |-------
{1}   sum(j!1) = (j!1 * (j!1 + 1)) / 2
        IMPLIES sum((j!1 + 1)) = ((j!1 + 1) * ((j!1 + 1) + 1)) / 2

Running step: (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
closed_form.2 :

{-1}   sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   sum((j!1 + 1)) = ((j!1 + 1) * ((j!1 + 1) + 1)) / 2

Running step: (EXPAND "sum" +)
```

```
Expanding the definition of sum,

this simplifies to:

closed_form.2 :


[-1]   sum(j!1) = (j!1 * (j!1 + 1)) / 2

  |-------

{1}   (j!1 + 1) + sum(j!1) = (j!1 * j!1 + 2 * j!1 + (j!1 + 2)) / 2


Running step: (ASSERT)
```

Invoking decision procedures, this completes the proof of closed'form.2.

```
Q.E.D.



Run time  = 8.09 secs.

Real time = 9.89 secs.

NIL

>
```

The next example illustrates that decision procedures solve the steps involving arithmetic and equality reasoning automatically. While, in the creative step of

supplying the proper instantiation for an existential quantification, the user has to interact with the prover. We first present the following PVS **THEORY** specifying that 3 cent stamps and 5 cent stamps can be used in combination in place of any stamp whose value is at least 8 cents.

```
stamps : THEORY
   BEGIN
  i, j, k: VAR nat
  stamps: LEMMA (FORALL i: (EXISTS j, k: i+8 = 3*j + 5*k))

  END stamps

stamps :
```

The proof follows by induction:

```
  |-------
{1}   (FORALL i: (EXISTS j, k: i + 8 = 3 * j + 5 * k))

Running step: (INDUCT "i")
```

Inducting on i, this yields 2 subgoals:

```
stamps.1 :
```

```
  |-------
{1}   (EXISTS (j: nat), (k: nat): (0 + 8 = 3 * j + 5 * k))
```

Here we have to supply an instantiation interactively.

```
Running step: (QUANT 1 ("1" "1"))
```

Instantiating the top quantifier in 1 with the terms: (1 1), this simplifies to:

```
stamps.1 :

  |-------
{1}   0 + 8 = 3 * 1 + 5 * 1

Running step: (ASSERT)
```

Invoking decision procedures, this completes the proof of stamps.1.

```
stamps.2 :

  |-------
{1}   (FORALL (j: nat):
        ((EXISTS (j_0: nat), (k: nat): (j + 8 = 3 * j_0 + 5 * k))
           IMPLIES (EXISTS (j_1: nat), (k: nat):
                      (j + 1 + 8 = 3 * j_1 + 5 * k))))
```

```
Running step: (SKOLEM 1 ("j!1"))
```

For the top quantifier in 1, we introduce Skolem constants: (j!1), this simplifies to:

```
stamps.2 :


  |-------
{1}   (EXISTS (j_0: nat), (k: nat): (j!1 + 8 = 3 * j_0 + 5 * k))
        IMPLIES (EXISTS (j_1: nat), (k: nat): (j!1 + 1 + 8 = 3 * j_1 + 5 * k))
```

```
Running step: (FLATTEN)
```

Applying disjunctive simplification to flatten sequent, this simplifies to:

```
stamps.2 :


{-1}   (EXISTS (j_0: nat), (k: nat): (j!1 + 8 = 3 * j_0 + 5 * k))
  |-------
{1}   (EXISTS (j_1: nat), (k: nat): (j!1 + 1 + 8 = 3 * j_1 + 5 * k))
```

```
Running step: (SKOLEM -1 ("j!2" "k!1"))
```

For the top quantifier in -1, we introduce Skolem constants: (j!2k!1), this simplifies to:

```
stamps.2 :
```

```
{-1}   j!1 + 8 = 3 * j!2 + 5 * k!1

  |-------

[1]    (EXISTS (j_1: nat), (k: nat): (j!1 + 1 + 8 = 3 * j_1 + 5 * k))
```

The following steps require user interaction:

```
Running step: (CASE "k!1=0")
```

Case splitting on k!1=0, this yields 2 subgoals:

```
stamps.2.1 :


{-1}   k!1 = 0

[-2]   j!1 + 8 = 3 * j!2 + 5 * k!1

  |-------

[1]    (EXISTS (j_1: nat), (k: nat): (j!1 + 1 + 8 = 3 * j_1 + 5 * k))


Running step: (QUANT 1 ("j!2-3" "2"))
```

Instantiating the top quantifier in 1 with the terms: (j!2-3 2), this yields 2 subgoals:

```
stamps.2.1.1 :


[-1]   k!1 = 0

[-2]   j!1 + 8 = 3 * j!2 + 5 * k!1
```

```
  |-------
{1}   j!1 + 1 + 8 = 3 * (j!2 - 3) + 5 * 2
```

Running step: (ASSERT)

Invoking decision procedures, this completes the proof of stamps.2.1.1.

stamps.2.1.2 (TCC):

```
[-1]   k!1 = 0
[-2]   j!1 + 8 = 3 * j!2 + 5 * k!1
  |-------
{1}   j!2 - 3 >= 0
```

Running step: (QUANT 2 ("j!2+2" "k!1-1"))

No suitable (+ve EXISTS/-ve FORALL) quantified formula found.

No change on: (QUANT 2 (j!2+2 k!1-1))

stamps.2.1.2 (TCC):

```
[-1]   k!1 = 0
[-2]   j!1 + 8 = 3 * j!2 + 5 * k!1
  |-------
```

```
{1}   j!2 - 3 >= 0
```

Running step: (ASSERT)

Invoking decision procedures, this completes the proof of stamps.2.1.2. this completes the proof of stamps.2.1.

stamps.2.2 :

```
[-1]   j!1 + 8 = 3 * j!2 + 5 * k!1
  |-------
{1}   k!1 = 0
[2]   (EXISTS (j_1: nat), (k: nat): (j!1 + 1 + 8 = 3 * j_1 + 5 * k))
```

Running step: (ASSERT)

Invoking decision procedures, this simplifies to:

stamps.2.2 :

```
[-1]   j!1 + 8 = 3 * j!2 + 5 * k!1
  |-------
[1]   k!1 = 0
{2}   (EXISTS (j_1: nat), (k: nat): (j!1 + 9 = 3 * j_1 + 5 * k))
```

```
Running step: (QUANT 2 ("j!2+2" "k!1-1"))
```

Instantiating the top quantifier in 2 with the terms: (j!2+2 k!1-1), this simplifies to:

```
stamps.2.2 :

[-1]   j!1 + 8 = 3 * j!2 + 5 * k!1
  |-------
[1]   k!1 = 0
{2}   j!1 + 9 = 3 * (j!2 + 2) + 5 * (k!1 - 1)

Running step: (ASSERT)
```

Invoking decision procedures, this completes the proof of stamps.2.2.

This completes the proof of stamps.2.

```
Q.E.D.

Run time  = 10.67 secs.
Real time = 11.65 secs.
```

```
NIL

>
```

Finally, the following example illustrates the use of a general purpose strategy **induct-rewrite-bddsimp**, that involves induction, rewriting and propositional simplification. The theorem is based on the property of a Fibonacci sequence: 1, 1, 2, 3, 5, ... Here, an element, except the first two, is the sum of the its two immediate predecessors. If we denote the sum of **n** (**n > 0**) elements in the sequence by **fibsum(n)**, then we are required to prove the property that the sum is equal to **fib(n+2) + 1**. The PVS specification can be given as follows:

```
fib: THEORY
  BEGIN

  n: VAR nat

  fib(n): RECURSIVE nat =
        IF n = 0 THEN 1
        ELSIF n = 1 THEN 1
        ELSE fib(n - 2) + fib(n - 1)
        ENDIF
       MEASURE LAMBDA n: n

  fibsum(n): RECURSIVE nat =
        IF n = 0 THEN 3
        ELSE fib(n) + fibsum(n - 1)
        ENDIF
       MEASURE LAMBDA n: n

  FibSumThm: THEOREM
       fibsum(n) = fib(n + 2) + 1

  END fib
```

The verification proceeds automatically by using a strategy based on induction, rewriting and propositional simplification as follows:

```
FibSumThm :

  |-------

{1}   (FORALL (n: nat): fibsum(n) = fib(n + 2) + 1)

Rule? (auto-rewrite-theory "fib")
```

Adding rewrites from theory fib

Adding rewrite rule fib

Adding rewrite rule fibsum

Auto-rewritten theory fib

Rewriting relative to the theory: fib,

this simplifies to:

```
FibSumThm :

  |-------

[1]   (FORALL (n: nat): fibsum(n) = fib(n + 2) + 1)

Rule? (induct-rewrite-bddsimp "n")

fibsum  rewrites fibsum(0)
```

```
   to 3
fib  rewrites fib(0)
   to 1
fib  rewrites fib(1)
   to 1
fib  rewrites fib(2)
   to 2
fib  rewrites fib(j!1 + 1)
   to IF j!1 + 1 = 1 THEN 1 ELSE fib(j!1 + 1 - 2) + fib(j!1 + 1 - 1) ENDIF
fib  rewrites fib(j!1 + 2)
   to fib(j!1)
        + IF j!1 + 1 = 1 THEN 1
          ELSE fib(j!1 + 1 - 2) + fib(j!1 + 1 - 1)
          ENDIF
fibsum  rewrites fibsum(j!1 + 1)
   to IF j!1 + 1 = 1 THEN 1 ELSE fib(j!1 + 1 - 2) + fib(j!1 + 1 - 1) ENDIF
        + fibsum(j!1)
fib  rewrites fib(j!1)
   to IF j!1 = 1 THEN 1 ELSE fib(j!1 - 2) + fib(j!1 - 1) ENDIF
fib  rewrites fib(j!1 + 3)
   to IF j!1 + 1 = 1 THEN 1 ELSE fib(j!1 + 1 - 2) + fib(j!1 + 1 - 1) ENDIF
```

```
        + fib(j!1)
        + IF j!1 = 0 THEN 1
          ELSE fib(j!1 - 1)
              + IF j!1 = 1 THEN 1
                ELSE fib(j!1 - 2) + fib(j!1 - 1)
                ENDIF
          ENDIF
```

fib rewrites fib(j!1)

to IF j!1 = 1 THEN 1 ELSE fib(j!1 - 2) + fib(j!1 - 1) ENDIF

By induction on n and rewriting,

Q.E.D.

Run time = 10.43 secs.

Real time = 30.62 secs.

Chapter 4

Efficient Mechanical Verification

4.1 Introduction

In this chapter, we describe the seamless integration of decision procedures for efficient model-checking and propositional simplification within a generic theorem-proving environment. The decision procedures are based on Binary Decision Diagram (BDD)[1] [Bry92]. A survey by Gupta [Gup92] provides an overview of various

[1]By BDD, we mean Reduced Ordered BDD (ROBDD).

formal verification techniques including theorem-proving and model-checking.

In model-checking [McM93], a typical implementation specification is a state-machine. The verification that the implementation satisfies a property is carried out by reachability analysis. The relationship that a model I satisfies a property S is written as:

$$I \models S$$

In generic theorem-proving, the specification could be of any form belonging to the logical language[2] of the theorem-prover. The verification of a property proceeds by a series of application of deduction rules such as induction. The relationship that an implementation I satisfies a property specification S is written as:

$$\vdash I \Longrightarrow S$$

Our seamless integration allows the model-checker to be invoked just like another decision procedure inside the theorem-prover. We have applied our integrated system to verification of safety and liveness properties of a variety of hardware and software examples, in which the entities may be of arbitrary size.

The rest of the chapter is organized as follows. We first give the motivation for our work on integration in Section 4.2. Next, we give a brief account of terminol-

[2]A typical logical language is based on typed higher-order logic.

ogy in Section 4.3. In Section 4.4 we describe, the integration scheme used in the decision procedure. In Section 4.5, we recapitulate the PVS specification logic and verification architecture. The algorithm used to implement the integration of the decision procedure is described in Section 4.6, with a brief mention of supplying contextual information and a proof of correctness. An integration of model-checking as a decision procedure within theorem proving is described in Section 4.7. In section 4.8 we give a small illustration of using the decision procedure. Finally, discussion and conclusions are summarized in section 4.10.

4.2 Motivation

Whereas generic theorem-proving provides powerful abstraction mechanisms, model-checking based on BDDs provides efficient propositional simplification, and verification of properties of state-machine specifications. For example, model-checking could be used to verify a typical cache-coherence protocol that has a small number (typically 5) of processors, it suffers from the state-explosion problem [McM93, BCM+90b] for a larger number of processors. Further, if we have an unspecified arbitrary number of processors model-checking would not be able to handle the verification problem. But, generic theorem-proving would be capable of handling a verification problem that involves arbitrary structure. However, theorem-proving does not provide efficient automatic procedures for propositional simplification and

state-machine verification.

The BDD-based decision procedures are used as efficient simplifiers for logical formulas in the generic proof checker. This, essentially involves abstracting logical formulas into well-formed expressions of boolean type, which are suitable for simplification by an external BDD-based simplifier, and getting back simplified boolean expressions. The simplified boolean expressions are then mapped back into corresponding well-formed logical formulas for further proof-checking. Additionally, the BDD-based simplifier is supplied with contextual information associated with terms of the logical formulas in the proof checker.

There has been earlier work on linking VOSS, a symbolic model-checker with HOL, another proof checker [JS93]. Their work, in hardware verification, was the first attempt to use a combined approach of theorem proving and BDD-based model checking. However, the focus in that work was on using VOSS to eventually determine the truth of a conjecture in hardware specification that has been sufficiently reduced to a concrete specification using HOL. The model-checker was used only in the final step of a proof. The form of logical formulas sent to the model-checker was restricted due to a syntax-based interface scheme. One such restriction was that the atomic terms of the formula had to be Boolean. There was no mechanism for supplying contextual information from the proof checker to the model checker. Also, a general facility to map back results obtained from VOSS into HOL was not

present.

Our work, on the other hand, does not impose any restrictions on the structure of logical formulas. The components of a non-atomic Boolean term could be of any arbitrary type. Further, it allows contextual information to be supplied to the BDD-based simplifier. We use the BDD-based procedure as if, it is one of several decision procedures available in the proof checker, by admitting results from the simplifier to be mapped up into the logic domain. The simplifier uses Reduced Ordered BDD (ROBDD), a canonical representation of boolean expressions [BRB90], to compute greatest (ν) and least (μ) fixpoints [Eme90, BCM$^+$90b]. Properties expressed as temporal logic formulas are transformed into computation of greatest and least fixpoints. Due to the canonicity property of ROBDDs, we can interpret our scheme as using execution or computation to speed up proof checking. We have implemented the system in Prototype Verification Systems (PVS) from SRI International [OSR93b] using the BDD-based simplifier from TUE [Jan93a, vEJ94].

4.3 Terminology

Here, we briefly account the terms we use to describe the objects in the logic domain. We follow the terminology employed in Gentzen's sequent calculus succinctly summarized by Rushby [Rus93].

- A *term* is a constant symbol, a variable symbol, or a function application.
- An *atomic formula* is a predicate symbol applied to a term.
- A *propositional connective* is one of $\sim$, $\wedge$, $\vee$, $\Longrightarrow$, $\Longleftrightarrow$ and IF-THEN-ELSE.
- A *quantifier* is either $\forall$ or $\exists$.
- A *well-formed formula* (*wff*) is an atomic formula, a quantified *wff* or a series of *wff*s combined using propositional connectives.
- A *sentence* is a *wff* that does not contain free (unbound) variables. The *wff* is then said to be *closed.*
- A *sequent* is of the form $\Gamma \vdash \Delta$ where Γ and Δ are each a set of sentences. The sentences in Γ are assumptions, while those in Δ are conclusions. The meaning of a *sequent*, intuitively, is that the conjunction of assumptions implies the disjunction of conclusions.

4.4 Integration Scheme

A sentence in a sequent is mapped to a tree structure, whose nodes are closed *wff*s that are related by any one of the propositional connectives. The terminal leaves of this tree are *wff*s which cannot be further mapped to such a tree structure (i.e.

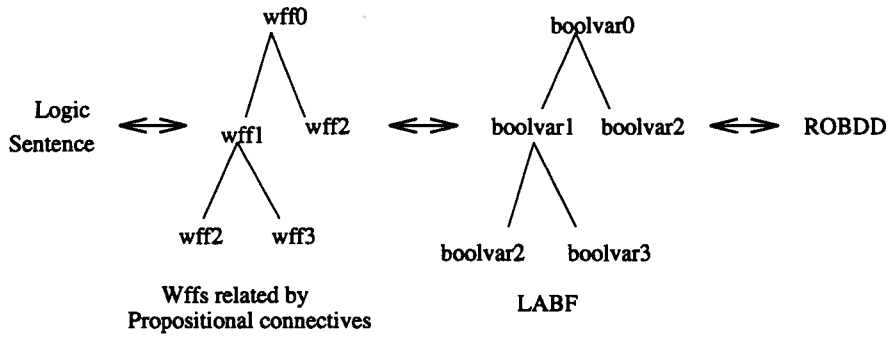


Figure 4.1: BDD Integration Scheme

the terminal *wff*s do not have any more top-level propositional connectives). The *wff*s are then substituted by variables of boolean type, with syntactically equal *wff*s being replaced by a single variable. A new formula is reconstructed back after substitution. We call this a ***Least Abstract Boolean Formula*** (LABF). Furthermore, for each *wff* in the original tree, the global context in the logical domain is checked for any other propositional relation (i.e. if it appears with a *wff* combined by a propositional connective) with other *wff*s in that tree. Such contextual assertions are also similarly mapped up to an LABF and contextual formulas formed. The new formula and the contextual formulas are then sent to the BDD-based simplifier for simplification[3]. A simplified formula is received back, and the variables in it are substituted by their corresponding *wff*s. This simplified formula is introduced in the original sequent. We illustrate this scheme in Figure 4.1.

[3]By simplification, we mean obtaining the simplest sum-of-cubes.

4.5 PVS Specification Logic and Verification Architecture: Review

We have described the PVS specification language and verification features in Chapter 3. In this section, we briefly review specification and verification in PVS described in detail in Chapter 3. PVS specification logic is based on typed higher-order logic [OSR93b]. It supports reals, Abstract Data Type (ADT) definition, subtyping and dependent typing. It also features common programming language constructs such as arrays, tuples and records. The specification language also permits overloading. Further, it allows parametric and hierarchical structuring of specifications.

The verification architecture consists of a type checker and a proof checker associated with decision procedures for propositional reasoning, arithmetic, (conditional) rewriting, beta reduction, and data type simplification among others. The user interacts with the system by a small set of proof commands. Besides other facilities, one can combine proof commands into strategies (tactics). It provides internal hooks to attach foreign decision procedures.

A PVS specification is first parsed and type-checked. The type-checker produces type correctness conditions (tccs) that assures the well-formedness of the specification. The tcc obligations are discharged using, in part, the proof-checker. At this stage, the type of every term in the specification is unambiguously known. The

proof checking is conducted using Gentzen's style ***sequent calculus*** [OSR93b]. In this ***backwards style***, we start with the conjecture to be proved and repeatedly apply deductive rules, which might use decision procedures. The conjecture is a theorem if such a series of applications of proof rules eventually leads to an axiom or another theorem.

4.6 Integration Algorithm

In our implementation of the BDD-based decision procedure integration, we use the available internal PVS hooks for attaching the BDD-based simplifier to the proof checker. A sentence in a sequent under consideration in a proof is recursively checked for *wff*s combined by propositional connectives. Terms whose types are finite sets are efficiently encoded in terms of boolean values. Such terms are generally used to describe finite state machines, and properties of the state machines.

4.6.1 Supplying Contextual Information

. The terms in a sentence could be associated with facts in the global context of the proof checking environment. This information has to be used, in general, by decision procedures. A typical situation in which contextual information is used, occurs due to the introduction of abstract data types (ADTs). The proof checker

generates axioms for enumerated type definitions, a kind of ADTs. Two kinds of axioms are generated:

- Exclusivity axiom: this states that a variable declared as an enumerated type, can only be exactly one member of the type at a time.
- Inclusivity axiom: this states that a variable declared as an enumerated type is at least one of the members of the type.

The contextual information such as the above axioms are supplied to the BDD simplifier with the ***restrict***ion operator. The ***restrict***ion operation in the BDD simplifier evaluates a boolean expression under the assumption that the other boolean formula holds.

4.7 Model-Checking within Theorem Proving

The integration of a model-checker or any finite state enumeration tool with a theorem prover can be done using either a "shallow embedding" or a "deep embedding" of the interface to the model-checker in the theorem prover. For example, in the HOL/VOSS implementation [JS93], which uses a shallow embedding, only the language (a very restricted temporal logic) used to specify the properties to be proved in VOSS was embedded in HOL. The model over which the validity of the proper-

ties are checked exists only in the model-checker and is not defined in the theorem prover. In a deep embedding, the approach used here, the semantics of the temporal logic and the model are represented in the theorem prover. An advantage of deep embedding is that it is possible to reason about the embedding and the model in the theorem prover also. For example, the process of abstracting a model before it is sent to the model-checker can be formalized in the theorem prover.

In our integration scheme [RSS95], Computation Tree Logic (CTL) [McM93] operators that are parameterized with respect to a binary next-state relation N are defined in PVS in terms of mu-calculus, which is itself embedded in the logic of PVS. Temporal logic formulas involving a specific interpretation for N, also defined in PVS, are transformed into mu-calculus formulas involving greatest and least fixpoints [BCM+90a, Eme90] by rewriting in PVS. For a class of temporal formulas, where N is known to be finite, the simplified mu-calculus formulas are translated and sent to an external BDD-based mu-calculus simplifier to compute greatest and least fixpoints. If the simplifier returns true then the original temporal formula is considered to be proved. The rewriting step, the translation process and the invocation of the simplifier is encapsulated as an automatic proof strategy that can be invoked as a primitive PVS command called `model-check`.

4.7.1 Propositional mu-calculus and Temporal Logic: Overview

Propositional mu-calculus is an extension of propositional calculus that includes universal and existential quantification on propositional variables[4]. In addition to atomic propositional terms and predicates on propositional terms, propositional terms may contain predicate variables bound by greatest (ν) and least (μ) fixpoint operators. It is strictly more expressive than CTL, and provides a framework to express fairness (fairCTL) and extended temporal modalities [EL85].

There have been several variations of mu-calculus proposed in the past [Cle89, EL85, Koz83, Par89, BCM+90a]. We closely follow the formal definition of the syntax of propositional mu-calculus from Clarke and others [BCM+90a] that forms the basis of the model-checker [Jan93b] used in this work. Let Σ be a finite signature, in which every symbol is a propositional variable or a predicate variable with a positive arity. The two syntactic categories ***formulas*** and ***relational terms*** are defined in the following manner. A formula is either a conventional propositional expression or a relational term. A relational term is one of the following:

- Z, a predicate variable in Σ.
- $\lambda z_1, z_2, \ldots, z_n.f$, where f is a formula and $z_1, z_2, \ldots, z_n$ are propositional variables in Σ.

[4]Quantification on propositional variables serves to succinctly express conjunction and disjunction, and does not enhance expressive power.

- $\mu Z.P[Z]$ is the least fixpoint of P, where Z is a predicate variable in Σ and P is a relational term formally monotone in Z. Similarly, $\nu Z.P$ is the greatest fixpoint of P, and is equivalent to the negation of the least fixpoint of $\neg P[\neg Z]$.

Temporal logics have proved to be feasible in specification and verification of properties of concurrent programs and state-machines models [Eme90, McM93]. Temporal logics such as CTL with extensions of fairness (fairCTL) and other temporal modalities, and PLTL can be succinctly expressed using the mu-calculus [EL85, BCM+90a] defined above. Additionally, it has been shown that LTL model-checking can be reduced to fairCTL model-checking [CGH94].

4.7.2 mu-Calculus and fairCTL in PVS

We develop a fixpoint theory based on an ordering by logical implication as follows. The membership of an element in a set is denoted by its characteristic predicate. An ordering on the predicates corresponding to set inclusion is introduced by logical implication on the predicates, i.e., $p_1 \leq p_2 \equiv \forall$ s: $p_1(s) \Rightarrow p_2(s)$. Monotonicity is defined as a relation that preserves such an ordering. A fixpoint of a functional `pp`, when applied to a predicate argument equals the predicate itself. The Greatest Lower Bound `glb` of a set `set_of_pred` is the intersection of all members of the set. The Least Upper Bound `lub` of a set `set_of_pred` is the union of all members of the set. The least fixpoint (`mu`) of a monotonic functional `pp` is the `glb` of the

set of functions satisfying the property that the application of the functional to a predicate p is less than or equal to the predicate p itself. The greatest fixpoint (nu) of a monotonic functional pp is the LUB of the set of functions satisfying the property that the application of the functional to a predicate p is greater than p itself:

```
mu (pp) = glb (LAMBDA p: pp(p) <= p)

nu (pp) = lub (LAMBDA p: pp(p) <= p)
```

We have mechanically verified in PVS, the property that predicates ordered by logical implication constitute a complete lattice.

The mu-calculus theory thus developed is used to develop the PVS theory (shown in Table 4.1 of the Appendix) for fairCTL. The fixpoint and fairCTL theories are parametrized by an uninterpreted type t. In a state-machine model specification, the type t is instantiated to the type of the state. In fairCTL theory, first the next state relation N is defined as a binary predicate on a pair of states. The temporal operators EX, EG, EU, and EF with existential path quantifiers are defined in a standard manner, and the corresponding operators AX, AF, AG and AU with universal path quantifiers are defined by introducing appropriate negations in the corresponding existential versions. An example definition for EG is the following:

```
EG(N,f):pred[t] = nu (LAMBDA Q: (f AND EX(N,Q)))
```

Fairness

A fairness constraint expresses a condition that should hold sufficiently often along a path [CG87]. Fairness constraints cannot be directly expressed in CTL. We use the more expressive mu-calculus to specify the fair temporal logic operators as follows. We first define `fairEG(N,f)(Ff)`:

```
fairEG(N,f)(Ff):pred[t] =
      nu(LAMBDA P: EU(N, f, f AND Ff AND EX(N, P)))
```

"there exists a computation path in the state machine model defined by the next state relation `N`, in which a fair formula `Ff` holds infinitely often and formula `f` holds globally along the path", where `N` is the next state relation, `Ff` is the formula that has to hold infinitely often, and `f` is the formula that has to hold globally on all such fair paths. The definition of fairness we have used above is based on the work by Emerson and Lei [EL85], which also allows other notions of fairness. The other fairCTL operators are defined by using the `fairEG` operator in the same manner as Burch et al. [McM93, BCL$^+$94]. The advantage of having an explicit formalization of fairness in a verification system is that it allows one to check if there exists at least one fair path in a given model. Without such an explicit formalization, there is a danger of imposing fairness constraints that might lead to empty models in which every property holds trivially.

4.7.3 Translation from PVS to mu-calculus

The PVS specification language [OSR93a] features basic types such as reals, integers, naturals, and lists, and a rich set of common programming language constructs such as scalars, arrays, functions, and records for structuring specifications. However, since the low-level BDD-based mu-calculus model-checker accepts only the language of propositional mu-calculus as discussed in Section 4.7.1, an automatic translation is provided from a fragment of the PVS language to propositional mu-calculus.

The fragment of the PVS language that is translated into mu-calculus consists of expressions whose types are hereditarily finite: i.e., types which are either themselves finite or constructed from expressions which are of finite types. However, boolean expressions in which subexpression types are not hereditarily finite are translated into a single boolean variable. For example, the equality `x = 1`, where `x` is a variable of type **natural**, is translated into a simple boolean variable **bvar_x1**. This scheme ensures that the output of translation is always a legal mu-calculus expression even when the input contains a subterm of non-finite type.

Let the category **t-expr** denote the set consisting of propostional expressions, scalars, records, tuples, arrays of a specified finite length, equalities on **t-expr**s, conditional expressions on **t-expr**s, **t-expr**s with quantifications on boolean variables, and **t-expr**s quantified by μ or ν. In the following, we give an abstract description of the translation in terms of the function **pvs-to-mu**. The input to **pvs-to-mu** is

either a **t-expr** belonging to the PVS language, or a list whose elements are either **t-exprs** or nested lists of **t-exprs**. The output of **pvs-to-mu** is a mu-calculus formula defined in Section 4.7.1.

1. **pvs-to-mu(prop-expr) = prop-expr**, where **prop-expr** is a propositional expression. Thus, **true**, **false**, boolean variables, and boolean expressions consisting of only simple boolean variables are translated without change.

2. **pvs-to-mu(scalar) = binary-encoding(scalar)**, where a **scalar** is either a variable or a constant of enumerated type, and **binary-encoding** is the corresponding boolean representation of the scalar in a binary encoding scheme. For example, let **instr: TYPE = {jmp,mov,add}** be a scalar type, and **x** be a scalar variable of **instr** type. Scalar constants **jmp**, **mov**, and **add** are translated as lists **[true,true]**, **[true,false]**, and **[false,false]**, respectively. The scalar variable **x** is translated as a list **[bvar1,bvar2]**.

3. **pvs-to-mu(nested_list_of_t-expr) = flattened_tuple_of_t-expr**
   For example, **[[bvar1,bvar2],bvar3]** is translated as **bvar1,bvar2,bvar3**. Such a translated tuple will eventually be incorporate as part of arguments to predicates, and as variables bound by λ and quantifiers. Thus **P(x,x)** is translated as **P(bvar1,bvar2,bvar1,bvar2)**, where **x = instr**.

4. **pvs-to-mu(record) = [pvs-to-mu(field1),...,pvs-to-mu(fieldN)]**,

where `record = [# field1,...,fieldN #]` and `field1,...,fieldN` are `t-exprs`. A `tuple` is translated in a similar manner. For example, let `state: TYPE = [# x1,x2: instr #]` be a record type, and `s1: VAR state` be a record variable. The translation output gives a list `[bvar-x1,bvar-x1,bvar-x2, bvar-x2]`. This will eventually be translated into a mu-calculus expression via rules 7,9, and 10. arguments to predicates

5. `pvs-to-mu(array([0...N]→fType))`

   = `[ pvs-to-mu(el1),...,pvs-to-mu(elN)]` where the range type `fType` is hereditarily finite, `el1,...,elN` are the elements of the array, and `N` is specified and finite.

6. `pvs-to-mu(t-expr1 = t-expr2)`

   = `(pvs-to-mu(t-expr1) = pvs-to-mu(t-expr2))`.

7. `pvs-to-mu(nested_list_of_t-expr_1 = nested_list_of_t-expr_2)` = $\bigwedge_i$ `pvs-to-mu(t-expr1[i] = t-expr2[i])`, i.e., equality on lists is translated to the conjunction of the equalities on the elements of the flattened lists.

8. `pvs-to-mu(t-exprs = IF test THEN then-t-exprs ELSE else-t-exprs)`

   ```
   =        IF test THEN pvs-to-mu(t-expr = then-t-exprs)
                    ELSE pvs-to-mu(t-expr = else-t-exprs).
   ```

   i.e., the inner IF expressions are lifted to the top.

9. `pvs-to-mu`$(\forall x.$`t-expr`$) = \forall$ `pvs-to-mu(x)`. `pvs-to-mu(t-expr)`. This translation also holds for $\exists$ and λ binders in place of $\forall$.

10. `pvs-to-mu`$(\mu.P(e_1, e_2, \ldots, e_n)) =$

    $\mu.P($`pvs-to-mu`$(e_1),$`pvs-to-mu`$(e_2), \ldots,$ `pvs-to-mu`$(e_n))$, where $e_1, e_2, \ldots, e_n$ are `t-exprs`. This translation also holds for ν binder in place of μ.

Other constructions such as ***functions*** and ***subtypes*** of hereditarily finite types could also be translated The fragment of the PVS language given above is rich enough to express specifications and properties of state-machine models in a structured manner. In comparison to language front-ends for other model-checkers such as SMV [McM93], the PVS language is more expressive, and has the significant advantage of a proof system for the language.

4.8 Example: BDD-based Propositional Simplification in PVS

We give here a small example to illustrate a PVS specification for a part of a tiny microprocessor. The simplicity of the example allows us to explain our approach without digressing into many specification details. In this example, we define a theory named ***tinymicro*** and, declare ***time*** as type ***real*** and instructions as an

```
mucalculus[t:TYPE]: THEORY
BEGIN

IMPORTING orders[pred[t]]
<=(p1,p2) = FORALL s: p1(s) IMPLIES p2(s)

ismono(pp) = (FORALL p1,p2: p1 <= p2 IMPLIES pp(p1) <= pp(p2))
isfix (pp,p) = (pp(p) = p)
glb(set_of_pred)(s) = (FORALL p: member(p,set_of_pred) IMPLIES p(s))

% least fixpoint
mu (pp) = glb (LAMBDA p: pp(p) <= p)
islfp (pp,p1) =
isfix (pp,p1) AND
FORALL p2: isfix (pp,p2) IMPLIES p1 <= p2
lub (setofpred)(s):bool = EXISTS p: member(p,setofpred) AND p(s)

% greatest fixpoint
nu (pp) = lub (LAMBDA p: pp(p) <= p)

ff: VAR [pred[t]->pred[t]]
lfp_is_lfp: THEOREM
ismono(ff) IMPLIES islfp(ff,lfp(ff))
END mucalculus
```

Table 4.1: Parts of μ-calculus theories in PVS

```
ctlops[t:TYPE]: THEORY

BEGIN

IMPORTING mucalculus

u,v,w: VAR t
f,g,Q,P,p1,p2: VAR pred[t]
N: VAR [t, t->bool]

% Higher order propositional connectives
AND(f,g)(u):bool =  f(u) AND g(u);
OR(f,g)(u):bool =  f(u) OR g(u)
NOT(f)(u):bool =  NOT f(u);
IMPLIES(f,g)(u):bool = f(u) IMPLIES g(u);
IFF(f,g)(u):bool = f(u) IFF g(u)

% CTL operators
EX(N,f)(u):bool =  (EXISTS v: (f(v) AND N(u, v)))
EG(N,f):pred[t] = nu (LAMBDA Q: (f AND EX(N,Q)))
EU(N,f,g):pred[t] = mu (LAMBDA Q: (g OR (f AND EX(N,Q))))
EF(N,f):pred[t] = EU(N,(LAMBDA u: TRUE),f)
AX(N,f):pred[t] = NOT EX(N, NOT f)
AF(N,f):pred[t] = NOT EG(N, NOT f)
AG(N,f):pred[t] = NOT EF(N, NOT f)
AU(N,f,g):pred[t] = NOT EU(N, NOT g, (NOT f AND NOT g))
    AND AF(N,g)

END ctlops
```

Table 4.2: PVS theory for CTL operator definitions in terms of greatest and least fixpoints.

```
fairctlops[t:TYPE]: THEORY

BEGIN
IMPORTING ctlops
u,v,w: VAR t
f,g,Q,P,p1,p2: VAR pred[t]
N: VAR [t, t->bool]

% fairCTL operators: CTL extended with fairness
fairEG(N,f)(Ff):pred[t] =
  nu(LAMBDA Q: f AND mu(LAMBDA P: EX(N, f AND ((Ff AND Q) OR P))))

fairEXlist_for_fairEG(N,f,P,Q)(Fflist)(u): RECURSIVE bool =
  CASES Fflist OF
    null: TRUE,
    cons(x,y): EX(N, f AND ((x AND Q) OR P))(u) AND
               fairEXlist_for_fairEG(N,f,P,Q)(y)(u)
  ENDCASES
  MEASURE (LAMBDA N,f,P,Q: (LAMBDA Fflist: length(Fflist)))

fairEG(N,f)(Fflist):pred[t] =
  nu(LAMBDA P: EU(N, f, f AND Ff AND EX(N, P)))

Fair?(N,Ff):pred[t] = fairEG(N,LAMBDA u: TRUE)(Ff)
Fair?(N,Fflist):pred[t] = fairEG(N,LAMBDA u:TRUE)(Fflist)

fairEX(N,f)(Ff):pred[t] = EX(N,f AND Fair?(N,Ff))
fairEX(N,f)(Fflist):pred[t] = EX(N,f AND Fair?(N,Fflist))
fairEU(N,f,g)(Ff):pred[t] = EU(N,f,g AND Fair?(N,Ff))
fairEU(N,f,g)(Fflist):pred[t] = EU(N,f,g AND Fair?(N,Fflist))
fairEF(N,f)(Ff):pred[t] = EF(N,f AND Fair?(N,Ff))
fairEF(N,f)(Fflist):pred[t] = EF(N,f AND Fair?(N,Fflist))
fairAX(N,f)(Ff):pred[t] = NOT fairEX(N, NOT f)(Ff)
fairAX(N,f)(Fflist):pred[t] = NOT fairEX(N, NOT f)(Fflist)
fairAF(N,f)(Ff):pred[t] = NOT fairEG(N,NOT f)(Ff)
fairAF(N,f)(Fflist):pred[t] = NOT fairEG(N,NOT f)(Fflist)
fairAG(N,f)(Ff):pred[t] = NOT fairEF(N,NOT f)(Ff)
fairAG(N,f)(Fflist):pred[t] = NOT fairEF(N,NOT f)(Fflist)
fairAU(N,f,g)(Ff):pred[t] = NOT fairEU(N,NOT g,NOT f AND NOT g)(Ff)
                          AND fairAF(N,g)(Ff)
END fairctlops
```

Table 4.3: PVS theory for fairCTL operator definitions in terms of greatest and least fixpoints.

enumerated type of 3 elements. We then, declare variables *t* and *instr*. The function symbol *tinydef* is declared as a predicate on the tuple [*time*,*instructions*]. The definition of *tinydef* is given using the IF-THEN-ELSE construct. The theorem *tinytheorem* is to be proved.

```
tinymicro: THEORY
  BEGIN
  time: TYPE = real
  instructions: TYPE = JMP, MOVE, NOOP
  t: VAR time
  instr: VAR instructions
  tinydef: pred[[time,instructions]]
  tinydef(t, instr)
      =
        IF t < 2 THEN instr = NOOP
        ELSE instr = JMP OR instr = MOVE OR instr = NOOP
        ENDIF
  tinytheorem: THEOREM
  FORALL t, instr:
      t > 2 IMPLIES tinydef(t, instr)
END tinymicro
```

We now elucidate the verification using ROBDD Decision Procedure in the context of the above example in PVS. A PVS goal appears in the following form: the formulas above the dashed line (turnstile) are assumptions, and those below the line are conclusions. The proof checking is done in a ***backwards style*** described in section 4.5.

We first set up the PVS goal and remove the universal quantifiers:

```
tinytheorem :


  |-------

{1}    FORALL t, instr: t > 2 IMPLIES tinydef(t, instr)
```

For the top quantifier in 1, we introduce Skolem constants: (t!1 instr!1) this simplifies to:

```
tinytheorem :


  |-------

{1}    t!1 > 2 IMPLIES tinydef(t!1, instr!1)
```

Applying disjunctive simplification to flatten sequent, this simplifies to:

```
tinytheorem :
```

```
{-1}    t!1 > 2
  |-------
{1}     tinydef(t!1, instr!1)
```

Expanding the definition of tinydef this simplifies to:

```
tinytheorem :

[-1]    t!1 > 2
  |-------
{1}     IF t!1 < 2 THEN instr!1 = NOOP
        ELSE instr!1 = JMP OR instr!1 = MOVE OR instr!1 = NOOP
        ENDIF
```

Here, we choose to use our ROBDD decision procedure on formula 1 in the above sequent. The integration algorithm, first constructs an LABF by assigning boolean variables to the expression $t!1 < 2$ and the equalities in THEN-branch and ELSE-branch of the IF-THEN-ELSE expression. Then the contextual information is extracted corresponding to ***instruction*** enumerated type defined in the ***tinymicro*** theory above. Here the contextual assertions are:

((instr!1 = JMP) OR (instr!1 = MOVE) OR (instr!1 = NOOP) AND

(NOT ((instr!1 = JMP) AND (instr!1 = MOVE))) AND...

The BDD simplifier, then evaluates the IF-THEN-ELSE expression under the restriction that the enumerated-type exclusive and inclusive axioms hold. In this example, only the inclusive axiom is useful. We then get back a simplified expression as below:

```
tinytheorem :

[-1]    t!1 > 2
  |-------
{1}    instr!1 = NOOP OR NOT t!1 < 2
```

We then invoke built-in arithmetic and propositional decision procedures to finally arrive at:

```
tinytheorem :
```

```
[-1]    t!1 > 2
  |-------
{1}    TRUE
```

which is trivially true.

4.9 Examples: BDD-based Model-Checking in PVS

We illustrate here, a typical mutual-exclusion protocol due to Peterson [Pet81]. In this protocol, processors might be in one of the four states: **idle, entering, critical, exiting**. However, no two processors might be in the critical section at the same time. Using PVS, we can reduce an N-processor mutual-exclusion protocol into 2-processor mutual-exclusion by successively running a tournament competition for N/2 processors at each competition stage, until we arrive at a 2-processor com-

petition [Sha94]. At this stage, the BDD-based model-checking decision procedure is called to verify the mutual-exclusion property of the reduced problem. We give the PVS specification of the 2-processor mutual-exclusion protocol in Table 4.4. The verification proceeds by rewriting temporal operators into ν and μ, and calling the BDD-based simplifier to compute fixpoints and map back the results into PVS. The automatic proof transcript is given in Appendix A [5].

4.10 Discussion and Conclusions

Reasoning can be done by a series of applications of syntactic inference rules. However, we can choose a representation for the formulas and the simplification done at the representation level more efficiently. We can view this representation as a semantic model of the formulas. The PVS architecture of decision procedures can

[5]The proofs were run on a SUN/Sparc-2 with 32 Meg memory

```
Peterson: THEORY
  BEGIN
  IMPORTING ctlops
% four states for each processor
  stateenum: TYPE ={idle, entering, critical, exiting}
% global state for the 2 processors "pc1" and "pc2", a shared "try"
% variable and "sem1","sem2": two semaphores each for processors "pc{1,2}"
  staterec: TYPE = [# pc1, pc2: stateenum, try, sem1, sem2:boolean #]
  s,s1,s2: VAR staterec
  state, state1, state2: VAR staterec
  % next state relation for processor pc1
  nextp1(state1,state2) =
   IF idle?(pc1(state1)) THEN
   idle?(pc1(state2)) OR (entering?(pc1(state2)) AND try(state2))
   ELSIF
   entering?(pc1(state1)) AND try(state1) AND NOT sem2(state1) THEN
   critical?(pc1(state2))
   ELSIF critical?(pc1(state1)) THEN
   (critical?(pc1(state2)) AND try(state2)) OR (exiting?(pc1(state2)) AND
   NOT try(state2)) ELSE idle?(pc1(state2)) ENDIF
  % similarly for nextp2 and nextsem; composite next state relation
  N(state1,state2) = nextp1(state1,state2) AND nextp2(state1,state2) AND
                                  nextsem(state1,state2)
  % initial state for "pc1", "pc2" and semaphores
  init(state): boolean = idle?(pc1(state)) AND idle?(pc2(state))
  initsem(state): boolean = NOT sem1(state) AND NOT sem2(state)
 % No two processors are in the critical section at the same time
 safe: THEOREM
 FORALL state:
 (init(state) AND initsem(state)) IMPLIES
 AG(N, LAMBDA state: NOT (critical?(pc1(state)) AND
                          critical?(pc2(state))))(state)
 % There is at least one path from all states that lead eventually
 % to a state in which a process enters a critical section
live: THEOREM
AG(N,(LAMBDA s:entering?(pc1(s)))=>EF(N, LAMBDA s: critical?(pc1(s))))(s)

 % Some paths could lead to deadlock (starvation)
not_live: THEOREM
AG(N,(LAMBDA s:entering?(pc1(s)))=>AF(N, LAMBDA s: critical?(pc1(s))))(s)

 % There is at least one fair departing path from the initial state
 % Starvation (deadlock) freedom
  fair: THEOREM
  (init(s) AND initsem(s)) => Fair?(N, LAMBDA s: critical?(pc1(s)))(s)
END Peterson
```

Table 4.4: State machines and their properties in PVS: Mutual-Exclusion Protocol Specification

be viewed, in part, allowing the simplification to be done at a representation level. Here, in our work, ROBDD is the representation level. Since, ROBDDs are canonical representations of boolean expressions, we can view the reduction of a sentence to ROBDD representation as performing computation or execution. From this perspective, this work is an outgrowth of the work on studying executability of higher order logic specifications [Raj92, RJS93]. We have used our integrated system in a variety of examples with dramatic improvements in speed up of propositional reasoning in PVS. This has led to fully automatic proof procedures for many classes of hardware designs including n-bit ALUs and pipelined microprocessors [KK94, SM95]. We have used the combination of theorem-proving and model-checking for automatically verifying hardware and software specifications, that would not have been possible by either one of them.

The simplified form, we have chosen is the simplest sum-of-products form. However, further minimization can be carried out using boolean optimization methods. It should be noted, however, ROBDDs are sensitive to ordering of the variables appearing in the ROBDD. In our implementation, we allow the ROBDD simplifier to choose an arbitrary ordering, and appropriately trigger heuristic dynamic ordering if the BDD graph grows large.

Another possible approach to using BDD-based simplification in proof checking is to build deductive mechanisms starting from a BDD basis. However, such a data structure based approach would pose limitations on the proof checker. The HOL-VOSS approach [JS93] and our approach propose that proof checker form the foundational basis. The deductive machinery would then have the flexibility to use a variety of data structures and decision procedures apart from BDD-based extensions.

In a few cases such as multipliers, the BDDs could grow unreasonably large. We envisage that, in such cases, BDD-based simplification can be augmented with other proof checking tools available in PVS. We also plan to explore the use of ROBDDs in a wider context of proof checking and verification. Finally, our work has opened avenues for integrating other specialized decision procedures (such as for hardware verification) with other built-in powerful decision procedures available in a generic verifier such as PVS. For this reason, our seamless integration that allows a model-checker to be used just as another decision procedures plays a key role in verification in-the-large. The system has been used in verification of a core implementation of the Fibre Channel Protocol [NRP95].

Chapter 5

Specification of SIL Graph Structure in PVS

A specification of the structure of SIL graphs is developed step by step in this Chapter. We introduce an entity in a SIL graph, and give its specification in PVS. We repeat some of the definitional concepts reviewed in Chapter 2 to put them in the context of our specification. We explain the specification of ports in Section 5.1, followed by the specification of edges in Section 5.2 and nodes and SIL graphs in Section 5.3. Finally, in Section 5.4 we establish the properties that need to hold for a SIL graph to be well-formed, and thus have a proper behavior.

5.1 Port and Port Array

A *port* is a placeholder for data values. The set of data values that it can hold can be restricted, and such a set is denoted by a *type*. For example, a *port* that is allowed to hold only *true* and *false* is of *Boolean type*. We would like to model a SIL graph and associated transformations for any desired set of data values. We define a *port* as a placeholder for an arbitrary set of data values, by defining it as an *uninterpreted type*:

```
port: TYPE
```

We can create various ports by introducing names such as *p0, p1, p2*, and declaring them as variables (*VAR*) of type *port* :

```
p0, p1, p2: VAR port
```

An array of *port*s is defined as a record *type* containing two type fields. The first field *size* of type *nat* – the set of natural numbers $\{0, 1, 2, \ldots\}$ – specifies the size of the array. The second field is the array of *port*s, whose size is equal to that specified by the first field. Such a typing, in which the type of one field depends on another field is known as *dependent typing*. The *ARRAY* is specified as a function that takes a member from the set of natural numbers less than *size* and gives a member of type *port*:

```
parray: TYPE = [# size:nat,
                  port_array: ARRAY[{i:nat|i<size} -> port]
               #]
```

5.2 Edges

An *edge* is a directed line connecting two *ports*. Mathematically, it is a relation on two *ports*. For convenience, we will call the *port* from which the edge is directed the *source*, and the *port* to which the *edge* is directed to the *sink*. There are two kinds of edges in SIL: *data-flow edge* and *sequence edge*. A data-flow edge between two ports indicates the flow of a token from the source to the sink. A sequence edge between two ports specifies the ordering between them: we will say that a port A is less than a port B if and only if, the token fired at B determines the value of a sink port C connected to A and B, rather than the token fired at A. A data-flow edge between two ports enforces an implicit ordering between the source and sink. The source is strictly less than the sink. There is no token flow through a sequence edge.

We specify both kinds of edges as relations on ports. They modify the behavior of a SIL graph in different ways. We postpone the discussion of the properties of these relations to the next chapter, and just specify the types of the relations as predicates – *pred* – on pairs of ports. A *true* value of the predicate indicates the

```
dfe: pred[[port,port]]
sqe: pred[[port,port]]
```

Table 5.1: .
PVS types for data-flow edge and sequence edge; see Figure 5.1.

presence of an edge between the ports, while a ***false*** value indicates the absence of an edge between the ports. The predicate ***dfe*** is the data-flow edge relation, and ***sqe*** is the sequence edge relation as shown in Table 5.1.

We can explicitly define corresponding relations between arrays of ports. For example, we define the data-flow edges between arrays of ports as:

Figure 5.1: SIL data-flow and sequence edges; see Table 5.1.

```
par, par0: parray


same_size(par,par0) =

       size(par) = size(par0)


dfear(par0,(par1:{par|same_size(par,par0)})) =

      FORALL (i|i<size(par0)):

        dfe(port_array(par0)(i),port_array(par1)(i))
```

The direction of the edges is from the first port to the second port. We illustrate this in Figure 5.1.

5.3 Node, Conditional Node and Graph

A *node* is a structure that takes inputs and gives outputs, satisfying a *data relation* associated with the node. Some of the typical nodes are adders and multiplexers associated with corresponding addition and multiplexing data relations. We also associate an *order relation*, which imposes an order on the inputs and outputs. Externally, a node receives inputs at *input ports*, and delivers outputs at *output ports*. Since a port is a placeholder for a definite set of data values – of a definite *type* – the input and output values should belong to the type of the input and output ports.

A *conditional node* is a node having special Boolean inputs, which control whether the data relation between the inputs and outputs holds. Such inputs are known as *conditions*. The conditions could appear either inverted or noninverted. If all of the noninverted conditions on a node are true, and all the inverted conditions are false, then the outputs and inputs of the node satisfy its data relation. But, if any one of the noninverted conditions is false or any one of the inverted conditions is true, then the output has an arbitrary value. In such a case, the output value is restricted

only by the type of the output port. Effectively, we can replace all the *condition ports* of a conditional node by just one condition port, which takes the conjunction of the condition inputs with appropriate inversions [EMH+93].

A *graph* is a structure constructed by using ports, edges, nodes, and conditional nodes. However, we can hide the structure of a graph, and externally view it as a node with input and output ports, data and order relations. We can then specify graphs as nodes with internal structure and internal relations. This allows for hierarchical construction of smaller graphs into larger graphs.

In our specification, we first introduce a conditional node in PVS as a *record type* as shown in Table 5.2, where

- ***inports*** are the input ports declared as ***parray*** type – that is, they are taken together as one array of an unspecified size.
- ***outport*** is an output port declared just as a port. In this work we consider a

```
cnode: TYPE =
        [#
   inports: parray,
   outport: port,
   intports: parray,
   condport: port,
   cond:pred[port],
   datarel: pred[[{p:parray|size(p)=size(inports)},port]],
   orderrel:pred[[{p:parray|size(p)=size(inports)},port]],
   intrel: pred[[parray,parray]]
         #]
```

Table 5.2: PVS specification of conditional node as a record type; see Figure 5.2.

single output port for convenience in specification. However, in general, output should also be declared as an array of ports, as is the case for hierarchically built graphs and for primitive nodes such as *SPLIT*.

- ***intports*** are the internal ports declared as a ***parray*** type to specify the internal ports the conditional node might have. Such a conditional node would be a hierarchically built graph.

- ***condport*** is a single port providing access for the condition input.

- ***cond*** is a condition function giving the value of the condition on the condition port: this can be either true or false. This is declared as a type ***pred[port]*** – that is, a predicate on port.

- ***datarel*** is the data relation governing the output value based on the inputs. This is declared as a predicate or relation ***pred*** on a tuple. The first type in

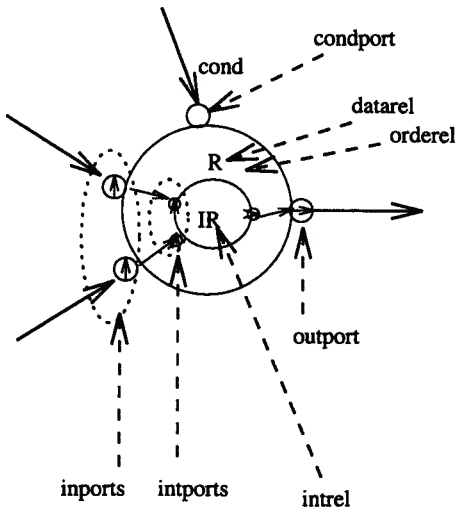


Figure 5.2: SIL conditional node; see Table 5.2.

the tuple is a subset of port arrays, whose size is the same as the inports, and the second type is a port corresponding to the outport.

- *orderrel* is declared as exactly the same type as datarel. The difference lies only in that, it governs the order of output and input values. This is not seen in the structural type definition here.

- *intrel* is the internal relation corresponding to the internal structure and connectivity of the conditional node. This is derived from the internal ports and the edges connecting the internal ports.

The conditional node is shown in Figure 5.2.

We introduce predicates to compare the structures of conditional nodes based on their number of input ports:

```
cn0, cn1: cnode


same_size(cn0,cn1) =

   size(inports(cn0)) = size(inports(cn1))
```

A node without a conditional port is modeled as a conditional node with the condition on its conditional port being always true. The advantage of such a modeling is that it captures both an unconditional node and a conditional node whose conditional port is always set to true. Since they have identical behavior, it minimizes our model by having just one structure for both. In PVS, a feature known as ***subtyping*** allows one to define a type, which denotes a subset of values of an already defined type. We specify the ***node*** type in Table 5.3 by using this PVS subtyping feature. The Figure 5.3 illustrates this specification.

```
node: TYPE = {n:cnode| cond(n) = LAMBDA (p:port):TRUE}
```

Table 5.3: Node as a subtype of a conditional node; see Figure 5.3.

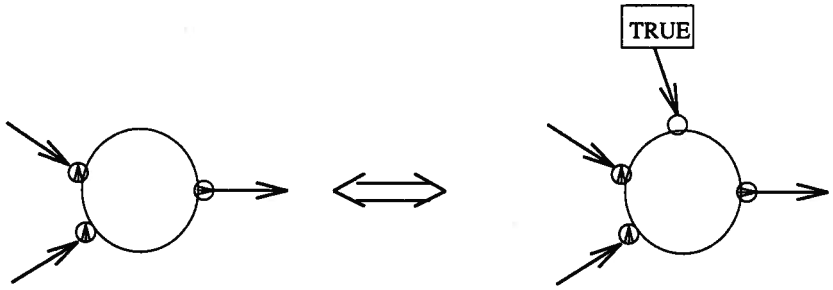


Figure 5.3: Node as a subtype of a conditional node; see Table 5.3.

We model a graph exactly the same as a conditional node, since we have constructed a conditional node to have internal structure and internal relation. This allows for viewing a graph as another node, and thus allows for a hierarchical construction of larger graphs. We specify a graph as a type equal to a conditional node type:

```
graph: TYPE = cnode
```

5.4 Well-formedness of a SIL Graph

A SIL graph has to satisfy certain structural rules governing the connectivity of ports. Only then can the behavior of a SIL graph be well-defined. For example, we cannot connect two input ports by a data-flow edge: a source has to be an output port, while a sink has to be either an input port or a conditional port. The structural rules are stated as axioms in PVS.

Every port has to be exactly one of an input port, output port, and conditional port: no port can be left dangling. Even the terminal I/O ports at the SIL graph boundary are associated with special I/O nodes. We express this as two axioms – inclusivity and exclusivity – as follows:

```
port_inclusive_ax: AXIOM
FORALL (p:port): is_inport(p) OR is_outport(p) OR is_condport(p)

port_exclusive_ax: AXIOM
FORALL (p:port): is_inport(p) IMPLIES
                 NOT (is_outport(p) OR is_condport(p)) AND
                 is_outport(p) IMPLIES
                 NOT (is_inport(p) OR is_condport(p)) AND
                 is_condport(p) IMPLIES
                 NOT (is_inport(p) OR is_condport(p))
```

where is_inport, is_outport, and is_condport are appropriately defined, asserting the existence of a (conditional) node whose input/output/condition is the port being considered, as indicated in the following PVS specification:

```
is_inport(p) = (EXISTS cn, (i:{j:nat|j<size(inports(cn))}):
                p=inport(cn,i))
is_outport(p) = (EXISTS cn: p=outport(cn))
is_condport(p)= (EXISTS cn: p=condport(cn))
```

That a port can be one of the internal ports of a conditional node is consistent with the properties defined here, because even internal ports should be one of the three types of ports.

A data-flow edge is legal only if it connects an input port to an output or a conditional port:

```
dfe_port_ax: AXIOM
FORALL p1,p2:
dfe(p1,p2) IMPLIES (is_outport(p1) AND
                    (is_inport(p2) OR is_condport(p2)))
```

We can derive that self data-flow edges are forbidden by the properties of ports and the data-flow edge from the above property. If we make `p1 = p2` in the above axiom, and use the port exclusivity axiom (given earlier) that any port can be exactly one of input, output and condition, we get the corresponding theorem for preventing self data-flow edges:

```
self_edge_not_th: THEOREM
FORALL (p:port): NOT dfe(p,p)
```

It should be noted that data-flow edges between output ports of a node and the

input ports of the same node are not prohibited.

Self sequence edges are also prohibited, since sequence edges impose strict ordering on ports. This has to be asserted as an axiom, as we have not imposed any restrictive property on the sequence edge:

```
self_seq_edge_not_ax: AXIOM
FORALL (p:port) NOT sqe(p,p)
```

Since sequence edge introduces ordering on ports, we expect **sqe** to be transitive. But, in order to have a clear separation of structure and behavior, we do not impose the property on **sqe** here. However, as we will see in Chapter 6, we formalize the ordering due to the sequence edges, and due to the behavior of a condition node when the condition port has a false value, by introducing weights on pairs of ports. The transitivity property is then imposed on the ordering of weights.

Chapter 6

Specification of SIL Graph Behavior and Refinement

We informally discussed in Chapter 2, the behavior of a SIL graph. We recall that the behavior is the set of ordered tuples of data values that the ports of the graph can assume, and an external or I/O behavior is the set of ordered tuples of values at the I/O ports of the SIL graph. The behavior of a SIL graph is determined by the data relations and order relations of the nodes, connectivity due to the data-flow edges, and ordering imposed by sequence edges. Any implicit state information in a SIL graph is contained in the data relations of the nodes. Thus, a comparison of behaviors in any given clock cycle would not require comparing execution histories due to possible implicit states in a SIL graph. We discuss behavior in Section 6.1, followed by a presentation of refinement and equivalence in Section 6.2. Finally, in Section 6.3, we give a brief synopsis of the axioms and theorems developed and explained in earlier sections of this Chapter.

6.1 Behavior

A detailed definition of behavior would require establishing a concrete formal semantics of SIL, since the data values and ordering can be arbitrary. A denotational semantics of SIL has been discussed by Huijs [HK93]. However, at the level of abstraction we have chosen to specify, we bring about high-level properties of dependency graphs, refinement and equivalence that should hold independent of a detailed behavior model. We thus obviate the need to specify a concrete behavioral model of dependency graphs. Such mechanisms for specification (by defining the properties that have to hold) constitute our axiomatic approach. As we will see in the next chapter, we compare two SIL graphs by asserting the properties that need to be satisfied by the graphs with respect to their behavior. We can thus establish the correctness of transformations. A modification in the concrete behavioral model faithful to the properties on which we have based our approach would not change our specification and verification results. Further discussion of the advantages of our approach is postponed to Chapter 9.

The behavior associated with an access point or a port is described by the same

uninterpreted type, as we used in the introduction of the structural specification of a port:

```
port: TYPE
```

This is the stage where the specification of structure and behavior coincide. The type denoting the set of values being unspecified gives us the freedom to model the behavior (as with the structure) irrespective of the value type.

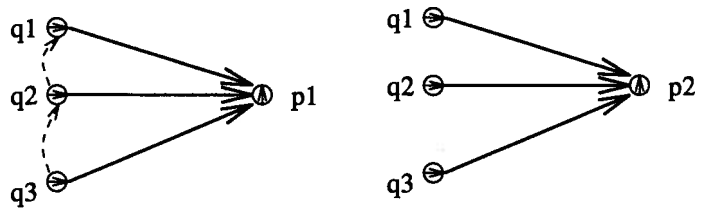
6.2 Refinement and Equivalence

We have developed specification techniques to describe concepts comparing SIL graphs with respect to behavior. A SIL graph SG2 is a refinement of another SIL graph SG1, if the behavior exhibited by SG2 is allowed by SG1. SG2 can then be an implementation of its specification SG1. In order to define graph refinement, we first describe port refinement, and derive graph refinement from the structural connectivity of a SIL graph.

We introduce an abstract refinement relation on ports:

```
silimp: pred[[port,port]]
```

The refinement relation on ports could be interpreted as follows. A port `p1` is



$$value(p1) = \{value \mid value = value(q1)\}$$

$$value(p2) = \{value \mid value = value(q1) \text{ OR } value = value(q2) \text{ OR } value = value(q3)\}$$

$$value(p1) \subset value(p2)$$

silimp(p1,p2): p1 is a refinement of p2

Figure 6.1: Example: refinement of ports due to non-deterministic choice being made deterministic.

a refinement of a port **p2**, if the set of data values allowed by **p1** is a subset of values allowed by **p2**. An instance of such a relation comes about due to the non-deterministic choice as illustrated in Figure 6.1. Another kind of refinement could be a ***data type*** refinement: when one port is a subtype of another. The refinement relation has to be reflexive and transitive. We do not impose antisymmetry to allow the definition of equivalence as a special case of refinement:

```
silimp(p1,p1)


silimp_trans_ax: AXIOM
silimp(p1,p2) AND silimp(p2,p3) IMPLIES
                                    silimp(p1,p3)
```

The refinement relation between arrays of ports is introduced by a property stating that a refinement relation between all corresponding ports of the port arrays implies a refinement relation between the port arrays.

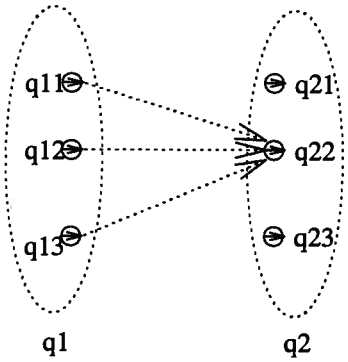
```
par1, par2: parray


silimpar(par1,par2)


silimpar_def_ax: AXIOM
FORALL par1,(par2|same_size(par1,par2):
  (FORALL (i| i< size(par1)):
    EXISTS j: silimp(port_array(par1)(i),port_array(par2)(j))) IFF
   silimpar(par1,par2)
```

It should be noted that the refinement between port arrays does not necessarily imply the refinement relation between corresponding individual ports of the port arrays. We illustrate this notion with an example in Figure 6.2. The reason for underconstraining the definition of port array refinement is to allow refinements for graphs which might have different numbers of input and output ports. We can thus allow behavioral refinement without overconstraining the structures of the graphs.

The properties of reflexivity and transitivity that have to be satisfied by the



silimp(q11,q22) AND silimp(q12,q22) AND silimp(q13,q22)
silimpar(q1,q2)

Figure 6.2: Example: array refinement does not imply every individual port refinement.

refinement relation on port arrays are similar to those satisfied by the refinement relation on ports:

```
silimpar_refl_th: THEOREM

silimpar(par,par)


silimpar_trans_ax: AXIOM

silimpar(par1,par2) AND silimpar(par2,par3) IMPLIES

silimpar(par1,par3)
```

The equivalence of SIL graphs *sileq* is defined by introducing the symmetry property in the refinement relations defined above:

```
sileq(p1,p2) = silimp(p1,p2) AND
               silimp(p2,p1)

sileqar(par1,par2) = silimp(par1,par2) AND
                     silimp(par2,par1)
```

A data-flow edge connecting two ports modifies the behavior of the sink in accordance with other data-flow edges connecting the same edge output. If a port is the sink of multiple data-flow edges, then the behavior of the sink port is determined by an ordering of the source ports. Such a port is called a *join*. The sequence edges in a SIL graph indicate such an ordering. However, since the ordering could be affected by the behavior of a conditional node, we need a general mechanism to specify the ordering. We model this ordering by associating weights with the data-flow edges, rather than source ports. Introducing weights to represent sequence edges also, permits a clear separation of structure from and behavior: whereas a sequence edge is

a structural entity, weight is a behavioral entity that could be derived not only from sequence edges, but also due the behavior of a conditional node. We first introduce *weight* as an uninterpreted type. A function *w* on ports would return a weight, while a function *war* on arrays of ports would return a weight:

```
weight: TYPE

w: [port,port -> weight]

war: [parray,parray -> weight]
```

The ordering is used to determine the behavior of a join. This means that we need to compare the weights on the data-flow edges that form a join. The weights on data-flow edges that do not form a join need not be compared. However, the definition of SIL specifies that no two data-flow edges communicate tokens simultaneously into a join, and no two weights on the edges forming a join can be equal. This suggests that we need a reflexive, transitive, and antisymmetric ordering relation on weights: such a relation is called ***partial order***. We define a partial ordering relation <= on weights, and assert the fact that the weights on a set of data-flow edges are linearly ordered if and only if the associated data-flow edges form a join. Since

```
<=: pred[[weight,weight]]
partial_order(<=)

dfe_w_ax: AXIOM
FORALL (p0:port),(p1:port),(p2:port):
NOT (p0 = p1)
IMPLIES
   dfe(p0,p2) AND dfe(p1,p2)
            IMPLIES
   ((w(p0,p2) <= w(p1,p2) OR
     w(p1,p2) <= w(p0,p2)) AND
     w(p0,p2) /= w(p1,p2))
```

Table 6.1: Using weights to enforce linear ordering of data-flow edges forming a join: PVS; see Figure 6.3.

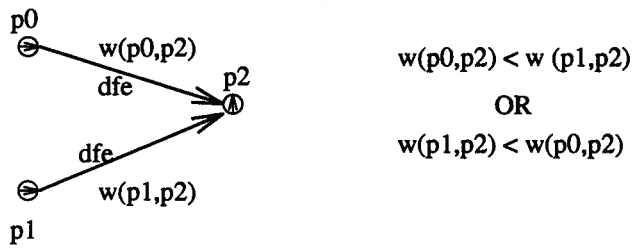


Figure 6.3: Using weights for ordering data-flow edges; see Table 6.1.

we do not compare the weights on edges that do not form a join, the weights are partially ordered on the set of all data-flow edges in the graph. We give the PVS specification of this property in Table 6.1 and illustrate it in Figure 6.3.

We describe the property that the behavior of a ***join*** depends on the ordering of the data-flow edges, by comparing weights on the edges flowing into the join port.

```
join_ax: AXIOM
FORALL p1,p2:
 (FORALL p:
   w(p,p2) <= w(p1,p2)) IMPLIES
 silimp(p1,p2)
```

Table 6.2: Using weights to determine *join* behavior; see Figure 6.4.

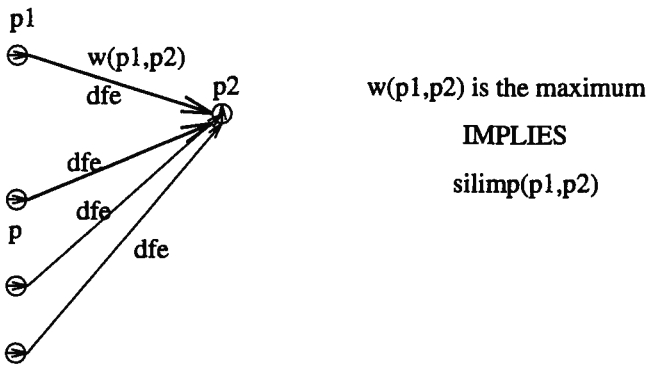


Figure 6.4: Using weights to determine *join* behavior; see Table 6.2.

The greater the weight on a data-flow edge, the later the token is communicated through it. We state the property that the join port is a refinement (an implementation) of the source whose associated data-flow edge has the maximum weight in the axiom shown in Table 6.2. It should be noted that we do not impose equivalence `sileq`, a relation stronger than refinement `silimp`. This would give the freedom to connect a port `p1` to `p2`, when the set of data values allowed by `p1` is always a subset of the set of data values allowed by `p2`. The property is shown in Figure 6.4.

```
cond_bottom_ax: AXIOM
NOT cond(cn)(condport(cn)) IMPLIES
     FORALL p:
      dfe(outport(cn),p) IMPLIES
      FORALL (n:node): dfe(outport(n),p) IMPLIES
                        w(outport(cn),p) <= w(outport(n),p)
```

Table 6.3: Weight when the condition on a conditional node is false; see Figure 6.5.

We still have to capture the notion of behavior of ports connected to the output port of a conditional node. The behavior of the output port of a conditional node, when the condition port holds a ***false*** value, is not defined. In the case where a join port is connected to a conditional node, the behavior of the join is solely determined by edges that propagate well-defined values. This situation is specified by making the associated weight of the data-flow edge emanating out of a conditional node the least of all the weights associated with other data-flow edges. The other data-flow edges, with which the comparison is performed should be connecting the join port to output ports of nodes or conditional nodes whose condition is never false. However, this does not preclude a join port to have an arbitrary value - because, it does not prohibit a graph construction where the join port is connected exclusively to a single conditional node or multiple conditional nodes whose conditions are false, and whose output ports are connected to the join port. The property is specified as an axiom in Table 6.3, and illustrated in Figure 6.5.

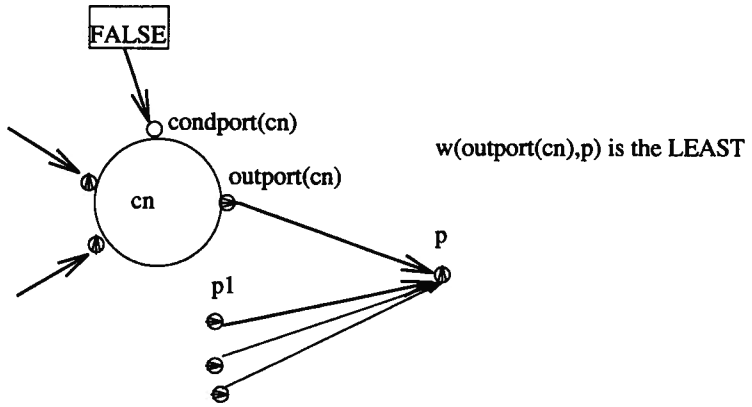


Figure 6.5: Weight when the condition on a conditional node is false; see Table 6.3.

```
xdfe(p1,p2) = dfe(p1,p2) AND
              FORALL p:
               (p /= p1) IMPLIES NOT dfe(p,p2)
```

Table 6.4: Absence of join: exclusive data-flow edge; see Figure 6.6.

We can derive the behavior due to a data-flow edge whose sink is not the output of any other data-flow edge. We will call such an edge an exclusive data-flow edge – *xdfe* defined in Table 6.4 and shown in Figure 6.6.

We can explicitly define an exclusive data-flow edge relation for arrays of ports as in Table 6.5. We can prove the property that an exclusive data-flow edge provides a refinement relation between the source and the sink. However, for this property to hold, we have to impose a restriction on the source port – that it has to be

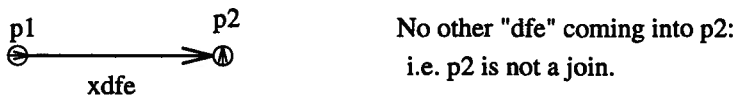


Figure 6.6: Absence of *join*: exclusive data-flow edge; see Table 6.4.

```
par, par0: parray

xdfear(par0,(par1:{par|same_size(par,par0)})) =
        FORALL (i|i<size(par0)):
                xdfe(port_array(par0)(i),port_array(par1)(i))
```

Table 6.5: Array version of exclusive data-flow edge

an output port of a nonconditional node. If the source is an output port of a conditional node, then the value false on the condition port will produce an undefined value on its output port. However, a data-flow edge transforms the undefined value into an arbitrary value of an appropriate type of sink. The undefined value is not communicated by a data-flow edge because the sink could be an input to another node. In practice, as we have pointed out in Chapter 2, an input is required to be a well-defined value, while an output generated could be an undefined value:

```
xdfe_silimp_th: THEOREM

FORALL (n1:node),(p2:port):

xdfe(outport(n1),p2) IMPLIES silimp(outport(n1),p2)
```

We again point out that the relation between the source and the sink is a refinement rather than equivalence. This weaker relation leads to more optimization than if it were equivalence. This issue is discussed further in Chapter 7 as part of generalizations of transformations.

A useful theorem involving a join of exactly two data-flow edges, shown in Table 6.6, states that the behavior of a join associated with exactly two data-flow edges is equal to the behavior of the port from which the edge with a greater weight emanates.

We postulate that the ordering on edges is preserved by behavioral refinement (and therefore also equivalence). We express the property in PVS as an axiom in

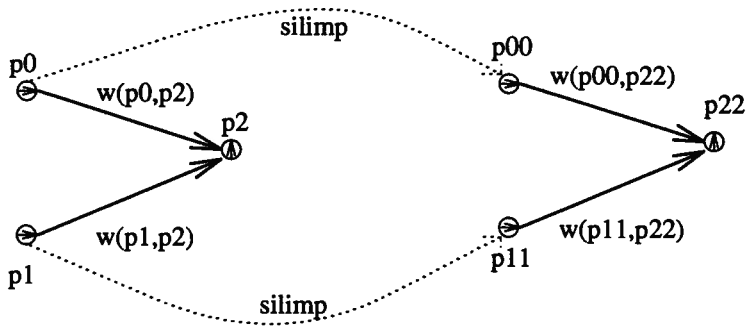
```
dfe2_join_th: THEOREM
(dfe(p1,p3) AND dfe(p2,p3) AND
 (FORALL p0:
   dfe(p0,p3) IMPLIES ((p0 = p1) OR (p0 = p2))))
IMPLIES
  IF w(p1,p3) <= w(p2,p3) THEN
     sileq(p2,p3)
  ELSE sileq(p1,p3)
  ENDIF
```

Table 6.6: A theorem on join of exactly two data-flow edges

```
po_preserve_ax: AXIOM
w(p0,p2) <= w(p1,p2) AND
       silimp(p0,p00) AND
       silimp(p2,p22) AND
       dfe(p00,p22) AND
       dfe(p11,p22)
IMPLIES
w(p00,p22) <= w(p11,p22)
```

Table 6.7: Order preserved by refinement and optimization; see Figure 6.7.

Table 6.7 and show it in Figure 6.7. We can then derive useful extensions of this property of preserving order by behavioral refinement. One useful extension for comparing SIL graphs expresses that the order is preserved with an introduction of an exclusive data-flow edge between an output port of a node and another port. This is shown in Figure 6.8. The statement of the property is the theorem in Table 6.8.



w(p0,p1) < w(p1,p2) IMPLIES w(p00,p22) < w(p11,p22)

Figure 6.7: Order preserved by refinement and optimization; see Table 6.7.

Similarly, we have corresponding postulates and theorems for arrays of ports instead of individual ports. However, we have to make a slight modification on comparing port arrays for inequality – that is, we will interpret the inequality operator `/=` to mean that the port arrays do not have any port in common. We have such a facility of overloading operators and functions in PVS. In comparing behaviors of SIL graphs, we find that the properties expressed using arrays of ports instead of individual ports, make specifications more succinct and economical.

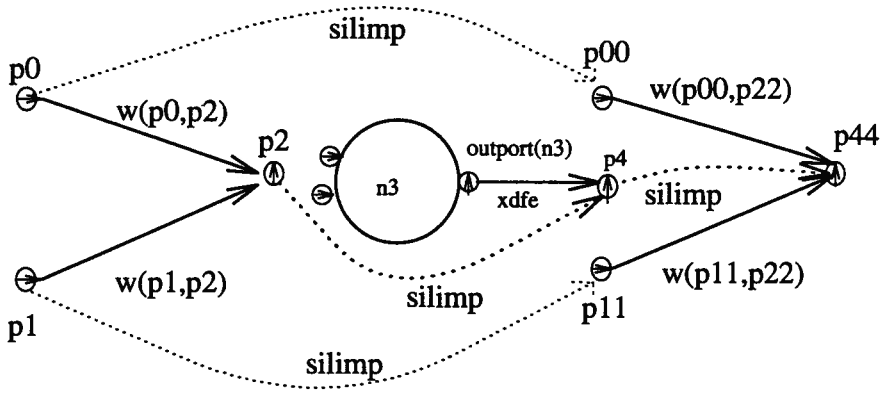
Finally, we need a refinement relation for graphs. A graph **`refines`** or implements another graph, when the data relation of the implementing node is contained in the data relation of the specification node. We call the implementing graph the ***refinement*** of the specification node. Instead of describing the graph refinement by

```
po_preserve_xdfe_th: THEOREM
w(p0,p2) <= w(p1,p2) AND
silimp(p2,outport(n3)) AND
 xdfe(outport(n3),p4) AND
         dfe(p00,p44) AND
         dfe(p11,p44) AND
        silimp(p0,p00) AND
        silimp(p1,p11) AND
        silimp(p4,p44)
IMPLIES
w(p00,p44) <= w(p11,p44)
```

Table 6.8: Order preserved by refinement and exclusive data-flow edge; see Figure 6.8.

describing containment of their data relations, we specify the relationship by using a higher level property. It is the property that, when the inputs of the implementation graph are a refinement of the inputs of the specification graph, then the outputs of the implementation graph have to be refinements of the specification graph. It should be noted that any state information implicit in a SIL graph is encapsulated in the data relations, thus obviating the need to consider behavior histories, rather than a single clock cycle behavior. Furthermore, to incorporate hierarchically structured graphs, we extend the nodes to have multiple output ports. The PVS specification in Table 6.9 illustrates the property in Figure 6.9.

This allows us to compare output ports, given a relationship among the input



w(p0,p2) < w(p1,p2) IMPLIES w(p00,p44) < w(p11,p44)

Figure 6.8: Order preserved by refinement and exclusive data-flow edge; see Table 6.8.

```
refinement_ax: AXIOM
FORALL (n0:node),(n1:node|same_size(n0,n1)):
 refines(n0,n1) AND silimpar(inports(n0),inports(n1)) IMPLIES
 silimpar(outports(n0),outports(n1))
```

Table 6.9: Graph refinement: property expressing relation between outputs and inputs of graphs independent of underlying behavior; see Figure 6.9.

ports and the relationship between the nodes. It should be noted that this represents a typical example of how we express a property for comparing ports, without a detailed representation of the input/output ports and data relations of the nodes. We also introduce convenient predicates in Table 6.10 to express that two nodes, having the same number of input ports (i.e., they are of the same_size), have an *equates* relationship if they are refinements of each other. In model-theoretic terms, the data relations of two nodes having *equates* relationship are identical. However,

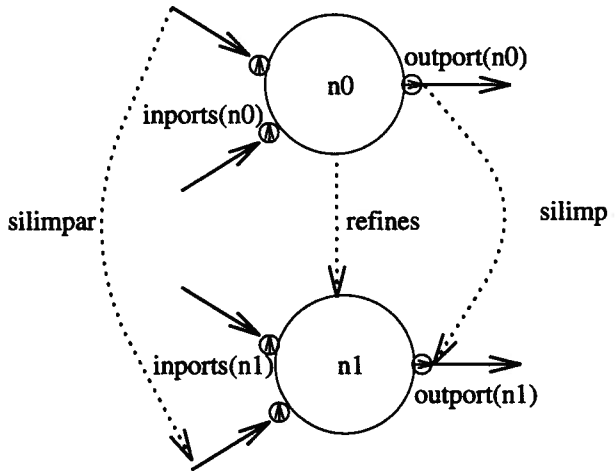


Figure 6.9: Graph refinement: property expressing relation between outputs and inputs of graphs independent of underlying behavior; see Table 6.9.

```
equates(cn0,(cn1|same_size(cn1,cn0))) =
            refines(cn0,cn1) AND refines(cn1,cn0)

sks(cn0,cn1) =
    same_size(cn0,cn1) AND equates(cn0,cn1)
```

Table 6.10: Predicates for expressing the sameness of nodes

in our axiomatic approach, we obviate the need to refer to data relations.

6.2.1 Compositionality

The refinement axiom presented in Table 6.9 allows us to provide a compositional proof of correctness of transformations: i.e., the refinement of component subgraphs of a graph G ensures the refinement of G. This allows us to assert that if a transformation is applied locally to a subgraph g of a graph G, leaving the rest of the graph unchanged, then the graph G undergoes a global refinement transformation.

Thus, we can state the following theorem:

Theorem (Compositionality of Refinement):

Let g_1, g_2,...,g_N be the component subgraphs of G, and g'_1, g'_2,...,g'_N the subgraphs of G´. If every g'_i is a refinement of g_i, then G´ is a refinement of G. i.e.,

(∀ i. **refines**(g'_i,g_i)) ⟹ **refines**(G',G)

Proof:

We carry out the proof by induction on the graph structure. The base case is straightforward by considering any one of the subgraph components g_i, whose refinement is given as g'_i. Thus, the statement that every subgraph component g'_i is a refinement of g_i holds trivially by the precondition given in the statement of the theorem:

$\forall$ i. **refines**(g'_i, g_i)

The induction step as shown in Figure 6.10 consists of showing that given a graph G' consisting of subgraphs G'_i, whose inputs are connected to the outputs of G'_h and outputs are connected to the inputs of G'_j, is a refinement of a graph G consisting of subgraphs G_h, G_i and G_j, whose corresponding refinements are G'_h, G'_i and G'_j. Assuming that the inputs to the graphs are in **silimpar** relationship, i.e. **silimpar**(**inputs**(G'),**inputs**(G)), we shall show that the outputs of the graphs are in **silimpar** relationship: i.e.,

silimpar(**outputs**(G'),**outputs**(G)).

Due to the refinement relationship G'_h **refines** G_h:

refines(G'_h, G_h),

and since the inputs to G, G' are the same as the inputs to G_h and G'_h, by refinement axiom of Table 6.9, every output of G'_h is also an output of G_h:

$$\texttt{silimpar(outputs(}G'_h\texttt{),outputs(}G_h\texttt{))}.$$

Since the outputs of G'_h and G_h are connected to the inputs of G'_i and G_i, therefore every input of G'_i is also an input of G_i:

$$\texttt{silimpar(inputs(}G'_i\texttt{),inputs(}G_i\texttt{))},$$

which together with the fact that G'_i **refines** G_i:

$$\texttt{refines(}G'_i\texttt{,}G_i\texttt{)},$$

ensures that the every output of G_i' is also an output of G_i:

$$\texttt{silimpar(outputs(}G_i'\texttt{),outputs(}G_i\texttt{))}.$$

By a similar argument, we can deduce that every output of G_j' is also an output of G_j:

$$\texttt{silimpar(outputs(}G_j'\texttt{),outputs(}G_j\texttt{))},$$

which in turn is equivalent to the statement that every output of G' is also an output of G:

$$\texttt{silimpar(outputs(}G'\texttt{),outputs(}G\texttt{))}.$$

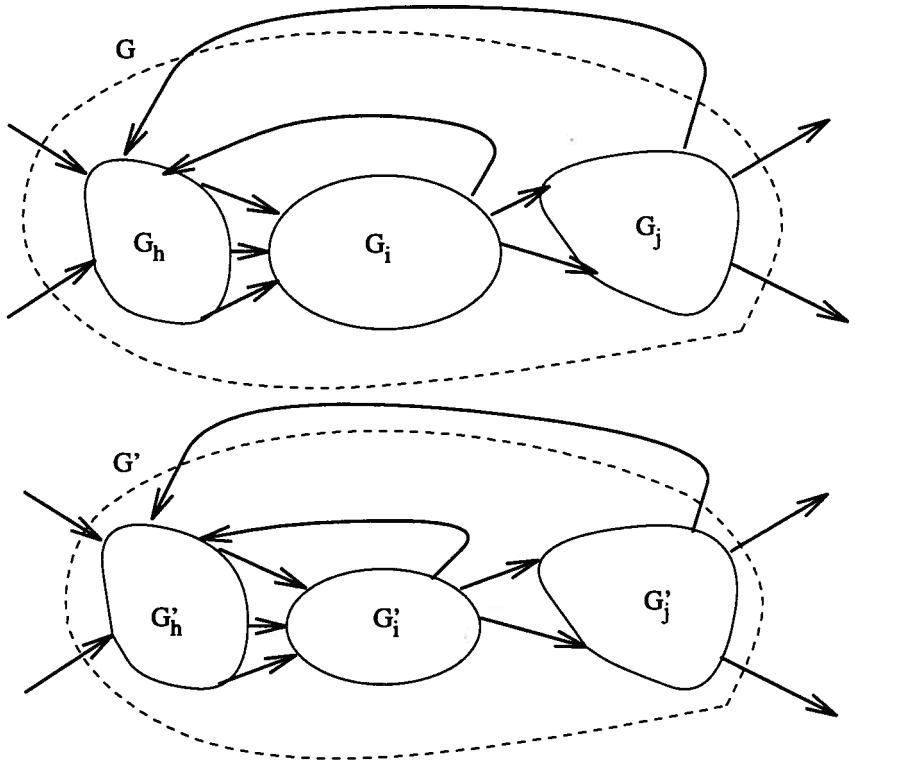


Figure 6.10: Compositionality of refinement.

Thus, by the refinement axiom, G' **refines** G:

$$\texttt{refines}(G', G).$$

Q.E.D.

6.3 Axiomatic Specification of Behavior and Refinement: A Synopsis

In this section, we provide a synopsis of the key axioms and theorems presented in this chapter. We first provide a set of notations and definitions, following which we give the axiomatic specification. Let π be the set of ports, and Π be the power set of π. Also, let $\mathcal{B}$ = {TRUE, FALSE} indicate the Boolean domain. We define a dependency graph Γ as a tuple: <I,O,Int,C,Cond,Drel,Orel,Intrel>

where

I $\in \Pi$ is the set of input ports

O $\in \Pi$ is the set of output ports

Int $\in \Pi$ is the set of internal ports - these are the ports of internal nodes that hierarchically constitute the top-level conditional node

C $\in \Pi$ is the set of conditional ports

Cond: $\pi \rightarrow \mathcal{B}$ is the Boolean condition on a port

Drel: [I, O] $\rightarrow \mathcal{B}$ is the data relation relating inputs and outputs.

Orel: [I, O] $\rightarrow \mathcal{B}$ is the order relation relating inputs and outputs.

Intrel: [Int,Int] $\rightarrow \mathcal{B}$ is the internal data/order relation relating internal ports.

A ***simple*** dependency graph Γ_s, without any boolean conditional ports is equivalent to a dependency graph with all the conditions on the conditional ports always being **TRUE**: i.e., $\forall p \in$ C: `Cond(p) = TRUE`.

Let $\preceq$ denote refinement on ports: $p_1 \preceq p_2$ indicates that that the set of values port p_1 can assume is a subset of the values port p_2 can assume. The relation $\preceq$ is a partial order, i.e., a reflexive, antisymmetric, and transitive relation. We shall denote equivalence between ports by the notation $\preceq\succeq$. Let $\trianglelefteq$ denote refinement on dependency graphs: $\gamma_1 \trianglelefteq \gamma_2$ indicates that the behavior of γ_1 is contained in that of γ_2.

The data flow edge between two ports is formalized by `dfe`, a relation on ports - i.e., $\texttt{dfe}(p_1, p_2)$ indicates a data flow edge from p_1 to p_2. A sequence edge is formalized by introducing weights on data flow edges, and providing a partial ordering on the weights. However, weights on the data flow edges forming a ***join*** are linearly ordered. Let $\texttt{w}(p_1, p_2)$ indicate the weight associated with the data flow edge connecting p_1 to p_2. We are now ready to provide the axioms.

Axiom 6.3.1 (Outputs and Inputs are constrained by datarel)

$$\forall \gamma \in \Gamma.\ \mathtt{Cond}(\mathtt{C}(\gamma)) \supset \mathtt{Drel}(\gamma)(\mathtt{I}(\gamma), \mathtt{O}(\gamma))$$

Axiom 6.3.2 (Weights on data flow edges are linearly ordered)

$$\forall p_0, p_1, p_2 \in \pi.$$

$$p_0 \neq p_1$$

$$\supset \quad (\mathtt{dfe}(p_0, p_2) \wedge \mathtt{dfe}(p_1, p_2))$$

$$\supset ((\mathtt{w}(p_0, p_2) \leq \mathtt{w}(p_1, p_2) \vee \mathtt{w}(p_1, p_2) \leq \mathtt{w}(p_0, p_2)) \wedge \mathtt{w}(p_0, p_2) \neq \mathtt{w}(p_1, p_2))$$

Axiom 6.3.3 (Ordering of weights is preserved by $\preceq$)

$$\forall p_0, p_{00}, p_1, p_{11}, p_2, p_{22} \in \pi.$$

$$(\mathtt{w}(p_0, p_2) \leq \mathtt{w}(p_1, p_2) \wedge p_0 \preceq p_{00} \wedge p_2 \preceq p_{22} \wedge \mathtt{dfe}(p_{00}, p_{22}) \ \wedge \ \mathtt{dfe}(p_{11}, p_{22}))$$

$$\supset$$

$$\mathtt{w}(p_{00}, p_{22}) \leq \mathtt{w}(p_{11}, p_{22})$$

Axiom 6.3.4 (Effect of FALSE value on the condition port)

$\forall\ \gamma\ \in\ \Gamma,\ p_1\ \in\ \mathtt{O}(\gamma\ ).$

$\mathtt{Cond}(\mathtt{C}(\gamma))$

$\supset\ \forall\ p\ \in\ \pi.$

$\mathtt{dfe}(p_1, p)$

$\supset\ \forall\ \gamma_s\ \in\ \Gamma_s,\ p_2\ \in\ \mathtt{O}(\gamma_s\ ).$

$\mathtt{dfe}(p_2, p)\ \supset\ \mathtt{w}(p_1, p)\ \leq\ \mathtt{w}(p_2, p)$

Axiom 6.3.5 (Behavior of a join)

$$\forall\ p_1, p_2\ \in\ \pi.$$

$$\mathtt{dfe}(p_1, p_2)$$

$$\wedge\ \forall\ p\ \in\ \pi.$$

$$\mathtt{dfe}(p, p_2)$$

$$\supset\ \mathtt{w(p}, p_2)\ \leq\ \mathtt{w}(p_1, p_2))$$

$$\supset\ \ p_1\ \preceq\ p_2$$

Axiom 6.3.6 (Refinement of Dependency Graphs)

$$\forall \gamma_0, \gamma_1 \in \Gamma_s, p_{i0} \in \mathtt{I}(\gamma_0), p_{i1} \in \mathtt{I}(\gamma_1), p_{o0} \in \mathtt{O}(\gamma_0), p_{o1} \in \mathtt{O}(\gamma_1).$$

$$\gamma_0 \trianglelefteq \gamma_1 \wedge p_{i0} \preceq p_{i1}$$

$$\supset p_{o0} \preceq p_{o1}$$

Definition 6.3.1 (Exclusive data flow edge)

$$\mathtt{xdfe}(p_1, p_2) \stackrel{def}{=} \mathtt{dfe}(p_1, p_2) \wedge \forall p.(p \neq p_1) \supset \neg\, \mathtt{dfe}(p, p_2))$$

Theorem 6.3.1 (An xdfe enforces equivalence)

$$\forall \gamma \in \Gamma_s, p_1, p_2 \in \pi, p_3 \in \mathtt{O}(\gamma).\ \mathtt{xdfe}(\mathtt{O}(\gamma), p_2) \supset (\mathtt{O}(\gamma) \preceq\succeq p_2)$$

Proof: Mechanically verified in PVS.

Theorem 6.3.2 (Behavior of a *join* of two data flow edges)

$$\begin{aligned}
&\forall\ p_1, p_2, p3\ \in\ \pi. \\
&\quad \mathtt{dfe}(p_1, p_3)\ \wedge\ \mathtt{dfe}(p_2, p_3) \\
&\wedge\ \forall\ p_0\ \in\ \pi. \\
&\qquad ((p_0\ \neq\ p_1)\ \vee\ (p_0\ \neq\ p_2)) \\
&\qquad\ \supset\ \neg\ \mathtt{dfe}(p_0, p3)) \\
&\qquad\qquad \supset\ \mathtt{IF}\ \mathtt{w}(p_1, p3)\ \leq\ \mathtt{w}(p_2, p_3)\ \mathtt{THEN}
\end{aligned}$$

$$p_2 \preceq p_3$$

ELSE $\quad p_1 \preceq p3$

Proof: Mechanically verified in PVS.

Theorem 6.3.3 (Compositionality of Refinement)

Let γ_1, γ_2,...,γ_N be the component subgraphs of Γ,

and γ'_1, γ'_2,...,γ'_N be the subgraphs of Γ'.

If every γ'_i is a refinement of γ_i, then Γ' is a

refinement of Γ.

i.e.,

$(\forall\ i.\ (\gamma'_i \trianglelefteq \gamma_i)) \supset \Gamma' \trianglelefteq \Gamma)$

Proof:

Proof follows by induction on the graph structure, as presented in Section 6.2.1.

Chapter 7

Specification and Verification of Transformations

The formal model of the SIL graph structure and behavior can be used to specify and verify the correctness of transformations. Here, we present optimization transformations, such as *Common Subexpression Elimination* and *Cross-Jumping Tail-Merging*. We have verified the correctness of other optimization transformations, and a similar technique can be adopted for verifying the correctness of refinement transformations. We present an overview of specification and verification of transformations in Section 7.1. We explain in detail *common subexpression elimination* in Section 7.2 and *cross-jumping tail-merging* in Section 7.3. We briefly mention specification and verification of other transformations and proofs in Section 7.4, and generalization and composition of transformations in Section 7.5.1. In Section 7.5.2, we illustrate with an example the usefulness of the axiomatic specification in investigating "what-if" scenarios. Finally, in Section 7.5, we illustrate a new transformation devised in the process of generalization and "what-if" analysis. This transformation can be used

for further optimization and refinement. This could not have been achieved by the existing transformations defined in the current synthesis framework.

7.1 Overview

The general method we employ to specify and verify transformations consists of the following steps:

1. Specify the structure of SIL graph on which the transformation is to be applied. The structure specification could be of graph templates or classes of SIL graphs rather than a particular concrete graph.

2. Assert that the structure of the SIL graph satisfies the preconditions imposed on its structure for applying the transformation. The preconditions would

consist of constraints imposed on structural connectivity and ordering through sequence edges.

3. Specify the structure of the SIL graph expected after the transformation is applied.

4. In the case of verifying refinement, we impose the constraint that the corresponding inputs of the SIL graphs before and after transformation are **silimpar** – that is, the set of input values to the SIL graph after transformation is a subset of the set of input values to the SIL graph before the transformation. For behavioral equivalence, the constraint is imposed as **sileqar**: the sets of input values to both graphs are identical.

5. Verify the property that the outputs of the SIL graph before transformation are **silimpar** – that is, the outputs of SIL graph after transformation are refinements of corresponding outputs of the SIL graph before transformation. In the case of behavior preserving transformation, the corresponding outputs are verified to be **sileqar**.

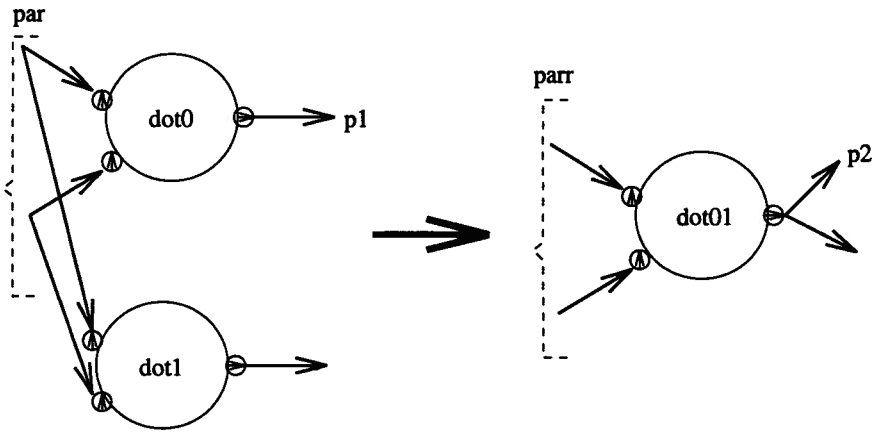


Figure 7.1: Common subexpression elimination; see Table 7.1.

7.2 Common Subexpression Elimination

In this transformation, two nodes of the same kind, which take identical inputs, are merged into one node as shown in the Figure 7.1.

We first specify the preconditions imposed on the nodes and the input ports connected to the nodes:

- The nodes must be of the same kind
- The ports connected to the input ports of one node must be identical to those connected to the input ports of the other node.
- The input ports should not be left dangling: they are required to have an incoming data-flow edge.

For convenience, we will assume that the joins at the input ports of the nodes have been resolved. Such a resolution of the joins would leave exactly one data-flow edge connecting each input port of the nodes. Relaxing that assumption would not change our verification of correctness of the transformation, except for an additional step of resolving the joins before the transformation is applied:

```
preconds(dot0,(dot1:{dot1|equates(dot0,dot1)})):boolean =
% input ports of dot0 and dot1 are connected to identical ports,
% and there exists at least one such set of ports
(FORALL (par|is_outportar(par) AND same_size(par,inports(dot0))):
 xdfear(par,inports(dot0)) IFF xdfear(par,inports(dot1))) AND
(EXISTS (par|is_outportar(par) AND same_size(par,inports(dot0))):
 xdfear(par,inports(dot0)))
```

We then specify the structure of the graphs before and after applying the transformation. The statement of correctness is asserted as a theorem that, if the inputs for the graph are ***sileq*** then the outputs of the graph are ***sileq***. The theorem is stated in Table 7.1.

```
CSubE: THEOREM
FORALL dot0,(dot1|equates(dot1,dot0)),
    (dot01|equates(dot01,dot0)):
((
 preconds(dot0,dot1) AND

% structure before transformation
 (FORALL (par|is_outportar(par) AND same_size(par,inports(dot0))):
        xdfear(par,inports(dot0))  IFF
 (EXISTS (parr|is_outportar(parr) AND
           same_size(parr,inports(dot0))):
          % ports connecting to dot0 and dot01 are equivalent
(sileqar(par,parr) AND xdfear(parr,inports(dot01)))))
 )
IMPLIES

% corresponding output ports of graphs before and after transformation are
% equivalent
(FORALL p1,p2:
((xdfe(outport(dot0),p1) OR
  xdfe(outport(dot1),p1)) AND
    xdfe(outport(dot01),p2)) IMPLIES
        sileq(p1,p2))
)
```

Table 7.1: Correctness of common subexpression elimination; see Figure 7.1.

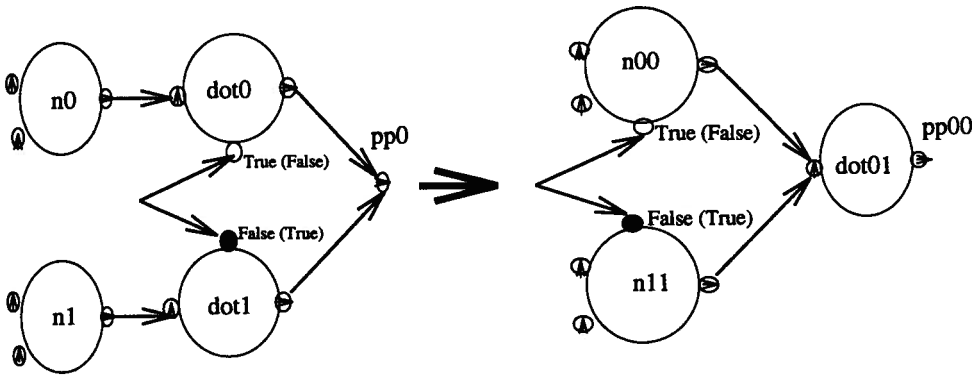


Figure 7.2: Cross-jumping tail-merging: corrected.

7.3 Cross-Jumping Tail-Merging

In the cross-jumping tail-merging transformation, two conditional nodes whose output ports connect to the same sink are checked for being mutually exclusive – that is, if the conditions on both of the conditional ports are not true (or false) at the same time (when exactly one of them is true at any time). In such a case, the two nodes can be merged into one unconditional node of the same kind, and the conditions moved to the nodes of the subgraph connecting it. We show this transformation in Figure 7.2.

In the course of our specification in PVS, we found a mistake in the informal

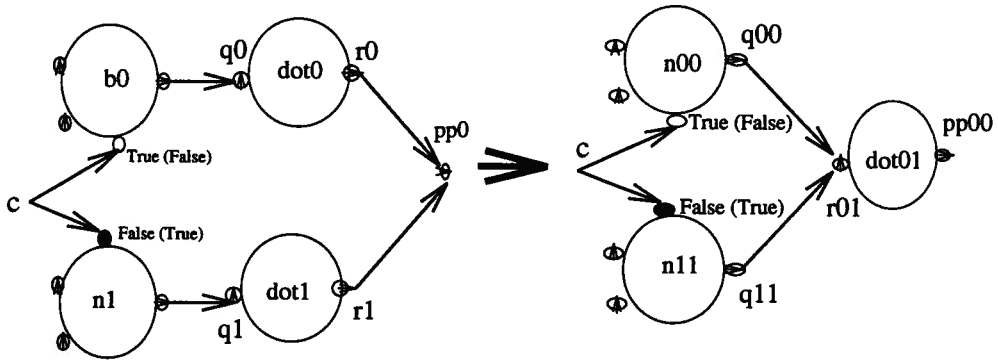


Figure 7.3: Cross-jumping tail-merging: incorrectly specified in informal document.

specification of the transformation. We show the erroneous transformation that was given in the original informal specification in Figure 7.3.

However, the same mistake was discovered later by inspection of the informal specification [Klo94] independently, without the aid of our formalization. The error that occurred in the original informal specification was the incorrect placing of the conditions on the nodes. With such a placing, the correctness of the transformation depends on the *ordering* of the output ports of *dot0* and *dot1*. When condition `c` is *true*, the values at `q1` and so `r1` are arbitrary, while the values at `q0` and `r0` are well-defined. Thus if an ordering is imposed such that the port `pp0` gets the value at `r1`, then that value would be arbitrary. However, in the transformed figure, the condition `c` being *true* results in an ordering such that `r01` gets the value of `q00`, and vice-versa when `c` is *false*. Thus, the transformation would not be correctness

preserving.

The placing of the conditions as given in Figure 7.3 is leads to violation of preconditions - because it prohibits comparing two ports joined exclusively to conditional nodes – that is, ***xdfe(p1,p2) AND is_outport_of_conditionalnode(p1)*** does not ensure ***sileq(p1,p2)***. We found this violation at the very early stage of stating the theorem corresponding to the transformation. Further, we could relax the mutual exclusiveness constraint. We introduce a weak assumption that the ordering of the data-flow edges coming out of the nodes *dot*0 and *dot*1 in the original graph is the same as the ordering of the data-flow edges coming into the node *dot*01 in the optimized graph. We have suitably modified, generalized, and verified the transformation. The generalized transformation is shown in Figure 7.4. The PVS specification of the preconditions is shown in Table 7.2, and the theorem statement is shown in Table 7.3.

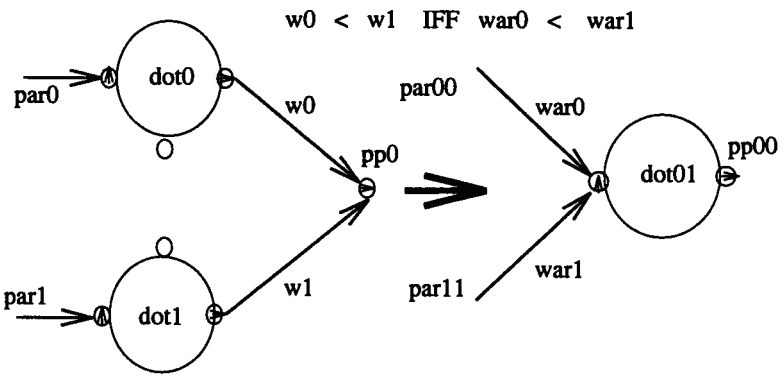


Figure 7.4: Cross-jumping tail-merging: generalized and verified; see Table 7.3.

7.4 Other Transformations and Proofs

We have specified and verified other transformations, such as copy propagation, constant propagation, common subexpression insertion, commutativity, associativity, distributivity and strength reduction described by Engelen and others [EMH+93].

In general, the proofs of transformations, proceed by rewriting, using axioms and proved theorems, and finally simplifying to a set of Boolean expressions containing only relations between ports and port arrays. At this final stage the BDD simplifier in PVS is used to determine that the conjunction of Boolean expressions is indeed true. We show the experimental results for verifying the various trans-

formations in Table 7.4. The proofs are semi-automatic. A few intermediate steps such as instantiation of existentially quantified variables need manual interaction, while other steps are automatic. Examples of inference rules [SOR93a] used in the proofs are **skolem!** for removing universal quantifiers, **assert** to apply arithmetic decision procedures and rewriting, **bddsimp** for Boolean reasoning using BDD, and **inst?** for heuristic instantiation of existential quantifiers. The PVS decision procedures for rewriting, and arithmetic and Boolean reasoning use a number of lower level inference rules that are hidden from the user. Examples of proof transcripts for common subexpression elimination and cross-jumping tail-merging are given in Appendix C.

7.5 Devising New Transformations

Having a mechanized formal approach such as ours, as opposed to approaches that are informal or formal approaches not mechanized has an advantage in the aspect of modifying specifications - the experiments of modifying specifications could be performed in a framework, that allows one to rapidly verify that the modifications do not violate the correctness properties. In this section, we show how we could develop new transformations using our mechanized formal approach.

7.5.1 Generalization and Composition of Transformations

We have seen earlier, in Chapter 7.3, that the specification has assisted in generalizing the transformation. In addition, we can make other observations on using

our work to generalize many transformations. For example, by replacing the equivalence relation ***sileq*** by ***silimp***, we find that the optimization transformations can be generalized as refinement transformations, and the preconditions imposed by the transformations could be relaxed.

The general technique to investigate composition of transformations is to determine that the preconditions imposed by one transformation are satisfied by another transformation. This also applies in the case where a transformation could be applied on one subgraph, while another could be applied on a disjoint subgraph, without having to take into account the effect of one transformation on the preconditions imposed by another. For example, common subexpression elimination (CsubE) produces a subgraph with an output port that is a distribute. Whereas, copy propagation (Copy Prop) [EMH+93] can be applied only to a subgraph that does not have a distribute output port. We can determine in our specification that if we perform CsubE, the conjunction of the subgraph relation thus obtained and the preconditions for performing Copy Prop on the same subgraph are false.

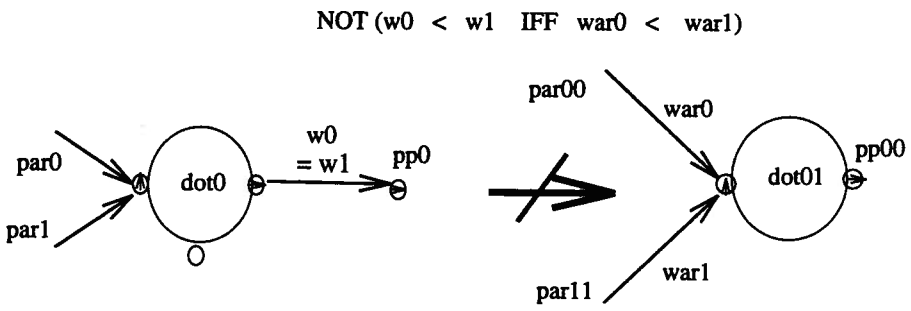


Figure 7.5: Cross-jumping tail-merging: inapplicable when two nodes are merged into one.

7.5.2 Investigations into "What-if?" Scenarios

One of the benefits of our formalism is that it allows us to provide answers to questions on the applicability of transformations, and provide formal justifications that support the answer. A question that comes up quite often in a transformational design process is whether a transformation that has been applied on a graph could still be applied with small changes in the graph. We illustrate this point in the context of a situation that resulted during the transformational design of a direction detector [Mid94a]. It involved a variation of the cross-jumping tail-merging transformation. In Figure 7.4, if we merge the nodes nodes `dot0` and `dot1` in the graph before applying the transformation, the precondition for the transformation would no longer be true. This is shown in Figure 7.5.

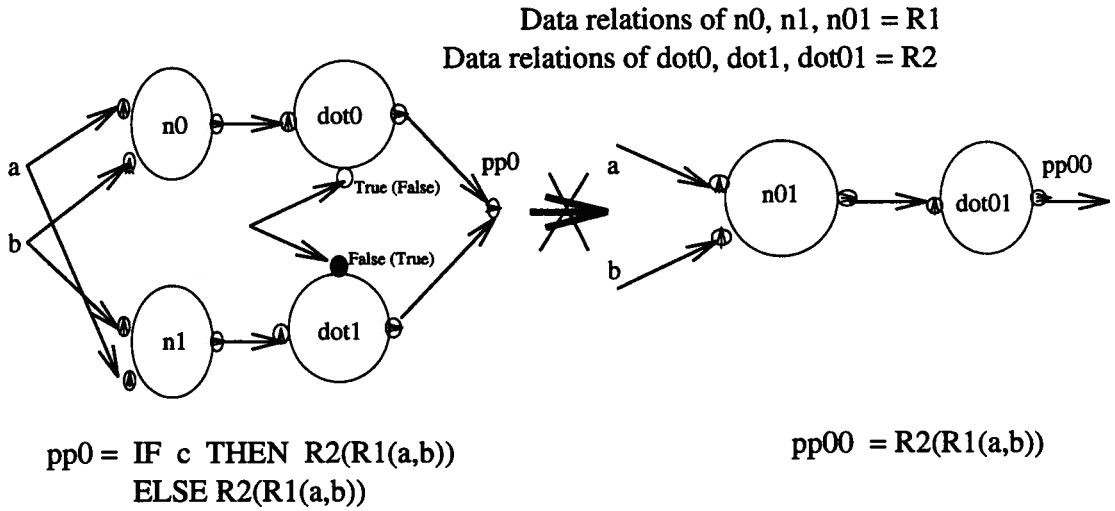


Figure 7.6: Further optimization impossible using existing transformations.

Since the nodes are merged, *w0 = w1*. While, due the ordering imposed by *join*, either *war0* < *war1* or war1 < war0. Thus the equivalence relation *w0 < w1 IFF war0 < war1* no longer holds, and so the precondition for the application of the transformation is violated. This precludes the application of the transformation on the modified graph.

In Section 7.5.2, we argued that cross-jumping tail-merging could not be applied in cases as shown in Figure 7.5. However, we would like to have such a transformation for further optimization in cases as shown in Figure 7.6. We can view this as a transformation derived from the process of generalizing cross-jumping tail-merging

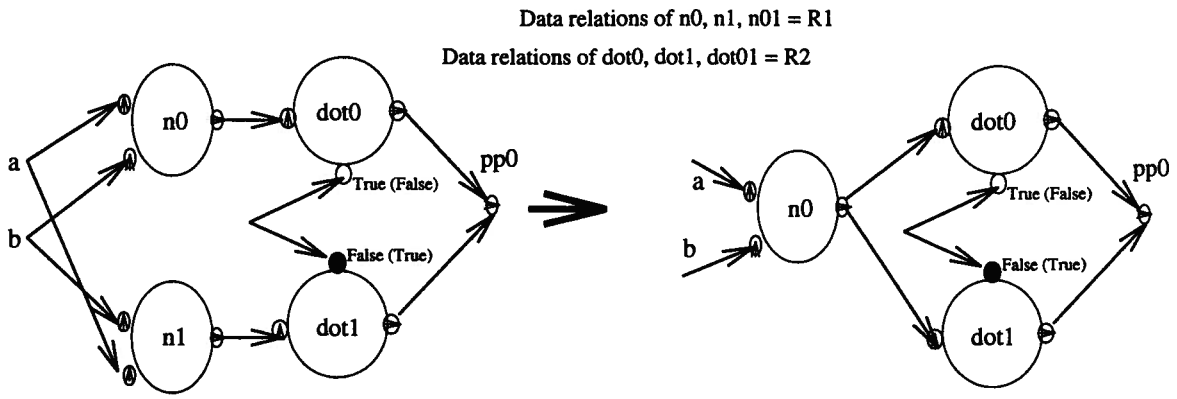


Figure 7.7: Inapplicability of cross-jumping tail-merging after common subexpression elimination: due to precondition restrictions.

and common subexpression elimination. In this transformation, two identical nodes with mutually exclusive conditions (i.e exactly one node will be active at any time) have inputs from identical nodes, which in turn have identical inputs. At first, it appears that we could apply a combination of common subexpression elimination and cross-jumping tail-merging. If we apply common subexpression elimination first, to obtain a single node whose output is connected to the mutually exclusive nodes, then we cannot apply cross-jumping tail-merging as shown in Figure 7.7. On the other hand, if we apply cross-jumping tail-merging first, the outputs of the other pair of identical nodes form a join at the input of the single node obtained. In this case, we cannot apply common subexpression elimination as shown in Figure 7.8.

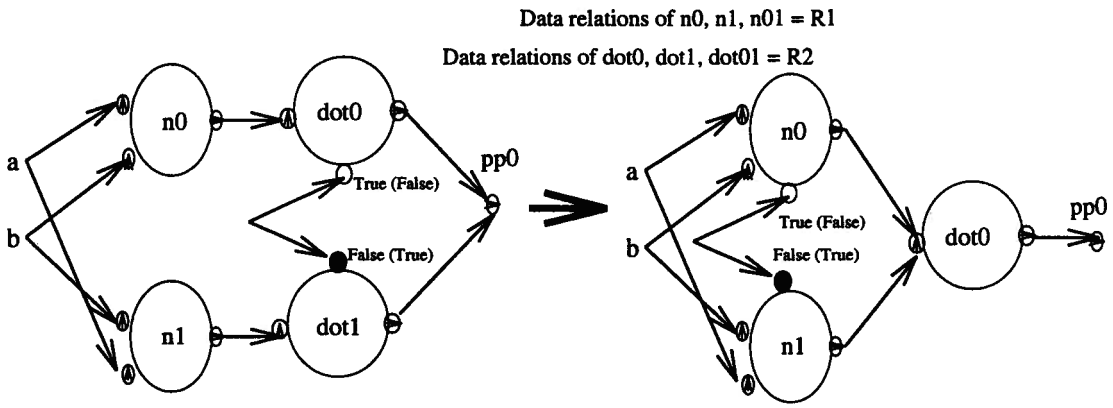
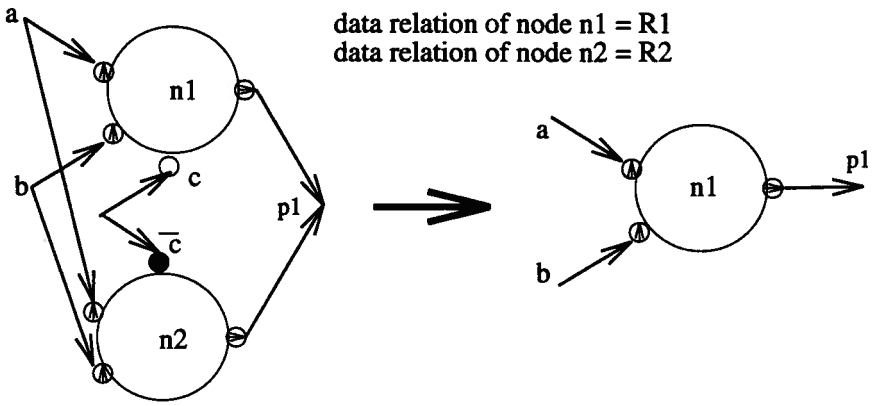


Figure 7.8: Inapplicability of common subexpression elimination after cross-jumping tail-merging: due to precondition restrictions.

The problem can be solved by devising a new and simple transformation as follows. In the description of common subexpression elimination shown in Figure 7.1, the outputs of nodes `dot01` and `dot1` were required to be not connected to join ports. However, we can relax this constraint, and provide a new and simple transformation that can be used to optimize a dependency graph. We show the new transformation in Figure 7.9. We could have arrived at the transformation in an ***ad hoc*** manner simply by examining the semantics of a conditional expression. However, we devised the transformation after examining by doing a "what-if" analysis formally in the problem of composing two transformations. This suggests that our formal model can be used to devise new transformations in a methodical manner.



IF c THEN p1 = R1(a,b)
ELSE p1 = R2(a,b) ;
refines(n2,n1) AND
refines(n2,n1) IMPLIES R2 $\subseteq$ R1

p1 = R1(a,b)

Figure 7.9: A simple new transformation: obvious, post-facto.

```
sks(cn1:cnode,cn2:cnode) = equates(cn1,cn2) AND same_size(cn1,cn2)
preconds
  (dot0,
  (dot1:{dot|sks(dot,dot0)}),
  (dot01:{dot|sks(dot,dot0)}),
  (par0:{par|is_outportar(par)&same_size(par,inports(dot0))}),
  (par1:{par|is_outportar(par)&same_size(par,inports(dot0))}),
  (par00:{par|is_outportar(par)&same_size(par,inports(dot0))}),
  (par11:{par|is_outportar(par)&same_size(par,inports(dot0))}),
  pp0,pp00)
      =
% connectivity at the input ports of SIL graph before transformation
xdfear(par0,inports(dot0)) AND xdfear(par1,inports(dot1)) AND
(w(outport(dot0),pp0) < w(outport(dot1),pp0) IFF
 war(par00,inports(dot01)) < war(par11,inports(dot01))) AND

% connectivity at the output ports of SIL graph before transformation
dfe(outport(dot0),pp0) AND dfe(outport(dot1),pp0) AND
(FORALL pp: ((pp /= outport(dot0)) OR (pp /= outport(dot1)))
IMPLIES NOT dfe(pp,pp0)) AND

% connectivity at the input ports of SIL graph after transformation
dfear(par00,inports(dot01)) AND dfear(par11,inports(dot01)) AND
(FORALL (par|size(par)=size(par00)):
(par /= par00 AND par /= par11)
IMPLIES NOT dfear(par,inports(dot01))) AND

% connectivity at the output ports of SIL graph after transformation
xdfe(outport(dot01),pp00) AND

% corresponding input ports of graph before and after transformation
% are equivalent
sileqar(par0,par00) AND sileqar(par1,par11)
```

Table 7.2: PVS specification of preconditions for cross-jumping tail-merging

```
CjtM: THEOREM
FORALL (dot0:cnode):
LET
 sks = LAMBDA (cn0:cnode),(cn1:cnode):
              same_size(cn0,cn1) AND equates(cn0,cn1),
 sk = LAMBDA (n:cnode):sks(n,dot0),
 ios = LAMBDA par:is_outportar(par) & same_size(par,inports(dot0))
 IN
 FORALL (dot1|sk(dot1)),
        (dot01|sk(dot01)),
        (par0|ios(par0)),
        (par1|ios(par1)),
        (par00|ios(par00)),
        (par11|ios(par11)):

% structure and preconditions on graphs before and after transformation
preconds(dot0,dot1,dot01,par0,par1,par00,par11,pp0,pp00)
IMPLIES

% corresponding output ports are equivalent
sileq(pp0,pp00)
```

Table 7.3: Correctness of cross-jumping tail-merging; see Figure 7.4.

Transformation	Run time in Seconds
Common subexpression elimination	30
Common subexpression insertion	25
Cross-jumping tail-merging	56
Copy propagation	10
Constant propagation	2
Strength reduction	2
Commutativity	3
Associativity	3
Distributivity	3
Retiming	3
Self-inverse	1

Table 7.4: Experimental results for proofs of various transformations on a Sparc-20 with 32 Mb memory

Chapter 8

Applications to Other Domains

In this chapter we discuss how we could apply our formal approach to investigate formalisms used in modeling real-time systems, optimizing compilers, data-flow languages, and Structured Analysis and Design (SA/SD). We give a high-level general methodology of application in each of the formalisms. Section 8.1 describes the graph-based formalism used in optimizing compilers. Data-flow languages that have data-flow graphs as the foundational basis are the subject of Section 8.2. In Section 8.3, we show how we can capture the data-flow diagrams used in SA/SD approach in our formalism. In Section 8.4, we discuss briefly the constraint net formalism used in modeling real-time systems. Finally, in Section 8.5, we give a brief discussion of separation of control-flow and data-flow.

```
        X := 1
        Y := 2
IF (X ?= 1) THEN Y := 3
```

Table 8.1: A typical program involving a branch. See Figure 8.1

```
        MVI #1,X
       MVI #2, Y
        CMP X,#1
        JZ Label
Label:  MVI #3, Y
```

Table 8.2: A typical program involving a branch: assembly language version of program in Figure 8.1.

8.1 Optimizing Compiler Transformations

There are a variety of transformations described in literature [AC72, EMH+93] for optimization and refinement in optimizing compilers. A notion similar to dependency graphs known as dependence graphs [PBJ+91], are used as an intermediate representation of a program during the process of compilation. Like dependency graphs, dependence flow graphs also combine control-flow and data-flow into one graph. We consider a typical program [PBJ+91] given in Table 8.1, whose low-level assembly program is given in Table 8.2. The associated dependence flow graph representation is given in Figure 8.1.

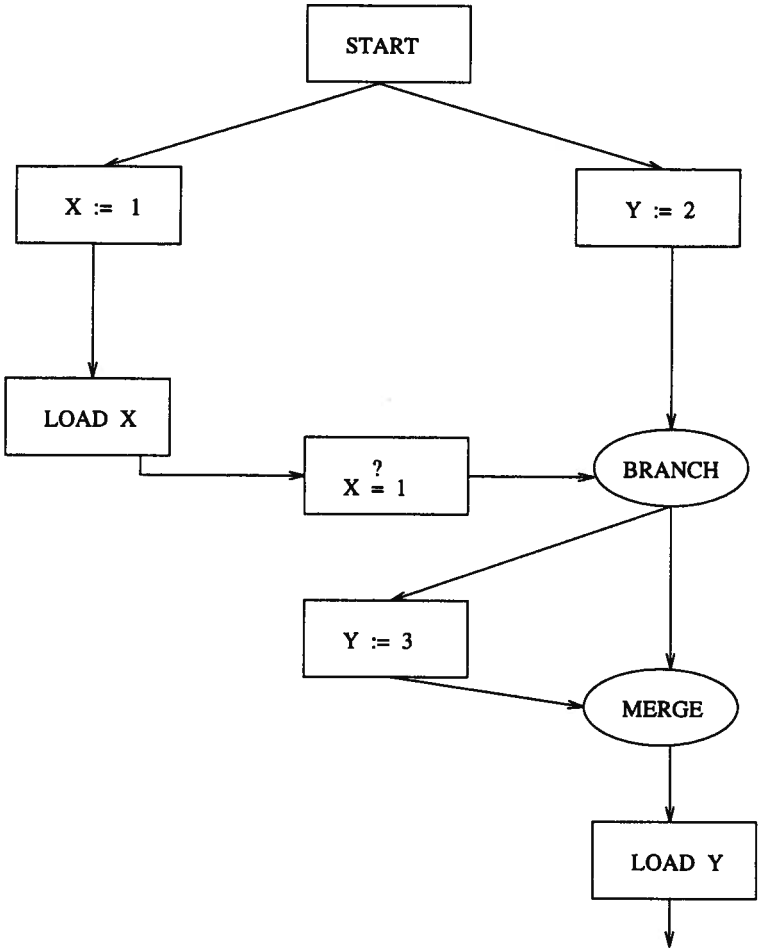


Figure 8.1: A typical dependence flow graph involving a branch. See Table 8.1

In Figure 8.1, the load operator reads the contents of a storage location and outputs the value as a token. While, the store operator is the inverse of the load operator receives a value on a token and stores it into a memory location. Dependence arcs that sequence operations on location y go into switch and merge operators, which implement flow of control. These operators serve to combine control information with data dependencies. We provide a simple transformation of the dependence flow graph of Figure 8.1 to a dependency graph such as SIL. The corresponding dependency graph is shown in Figure 8.2.

The straight-forward transformation from dependence flow graphs to dependency graphs suggests that the formal model developed for dependency graphs such as SIL could be applied for specification and verification of transformations used in optimizing compilers [Raj95b]. A number of transformations such as common-subexpression-elimination and cross-jumping tail-merging, whose specification and verification have been discussed in Chapter 7 have direct origins in optimizing compilers.

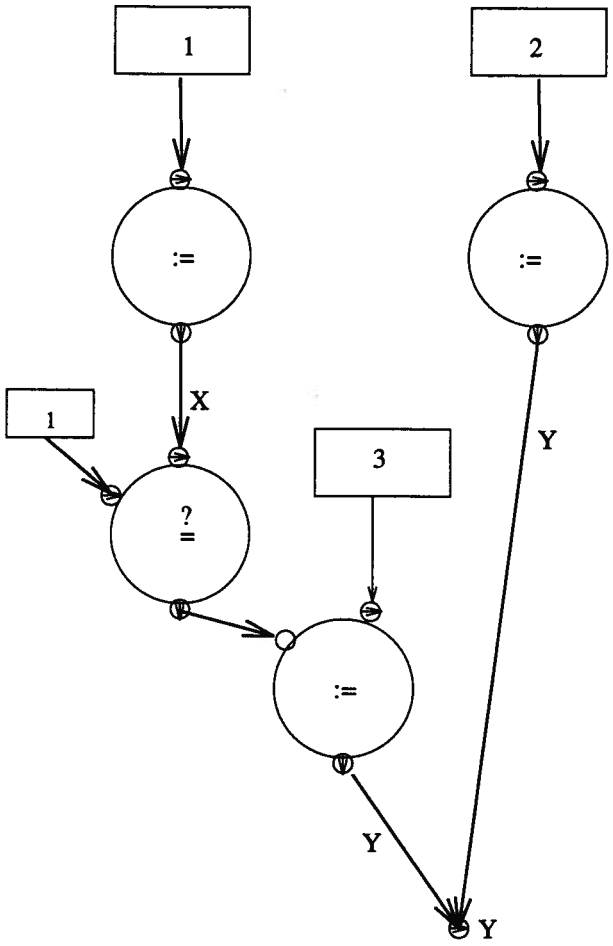


Figure 8.2: SIL dependency graph for program. See Table 8.1

8.2 Data-flow Languages

Data-flow languages [WP94] are based on the data-flow graph paradigm. In this paradigm, a program is modeled as a set of operator nodes, with input and output ports. The ports are connected by edges that carry data. Because of the absence of control specification, every operator node executes whenever the data is available on its input ports. Thus, it models concurrent execution implicitly. The data-flow paradigm is based on a ***single-assignment*** property [Cha79, WP94]. The single-assignment property means that the variables are assigned exactly once. Because the execution of statements need not be in the lexical order of the written program, a statement gets executed as soon as all the variables in the statement are well-defined. Thus, the single-assignment property ensures that the execution of the program is well-defined.

The main characteristics of data-flow languages are the following:

- Absence of side effects.
- Single assignment semantics.
- Inherent concurrency.
- Absence of sequencing.

Because of implicit concurrency and transparent control flow, deadlocks happen less often. Lucid [AW77, WP94] is one of the early data-flow languages designed to be amenable to program transformations and formal verification. Although it developed as a simple non-procedural language with temporal logic operators to prove properties of programs, it has grown to be an intensional multidimensional programming language with applications in parallel computing [AFJWed]. Lustre [HFB93], a synchronous declarative language based on the data-flow paradigm has been used to specify synchronous hardware and real-time systems. Programs written in Lustre are compiled into a sequential code. Its control structure is a finite automaton obtained by exhaustive simulation of Boolean variables appearing in the program. Id [WP94], another data-flow language developed for programming a data-flow computer, has been designed as a block-structured, expression-oriented, single-assignment language avoiding notions of sequential control. SISAL [WP94] is

a stream-oriented single-assignment applicative language. An intermediate dependency graph called IF1 [WP94] is used to represent SISAL programs between parsing and machine code generation. IF1 is an acyclic graph representation with four components: operator nodes, edges denoting data-flow, types denoting data types on edges, and graph boundaries surrounding groups of nodes and edges. Optimization and refinement transformations such as common-subexpression-elimination and introduction of explicit memory management operations are performed to obtain another IF1 graph at a lower abstraction level.

8.3 Structured Analysis and Design

Structured methods in software design are used to capture and analyze the requirements of the system under consideration. A majority of the methods use informal graphical notations for requirements capture. One of the most widely used meth-

ods is Structured Analysis/Design (SA/SD) approach [DeM79, Won94] for software analysis. In this approach, a combination of data flow diagrams (DFD) and control specifications such as state transition diagrams (STD) are used as graphical notations among others. Thus, there is a clear separation of data flow and control flow. There is also an associated data dictionary that contains more information on the data described in the data flow diagram. There have been some efforts in providing a formal sematics for DFD in VDM [LPT93]. However, they do not address control specifications and transformations on DFDs.

We first give a brief overview of DFD and STD in Section 8.3.1. In Section 8.3.2, we present a transformation of DFD to SIL. The SIL specification could then be represented in PVS using the formalism described in Chapter 5. Section 8.3.3 describes this process with an example. We present a brief discussion on the separation of control from data in Section 8.5

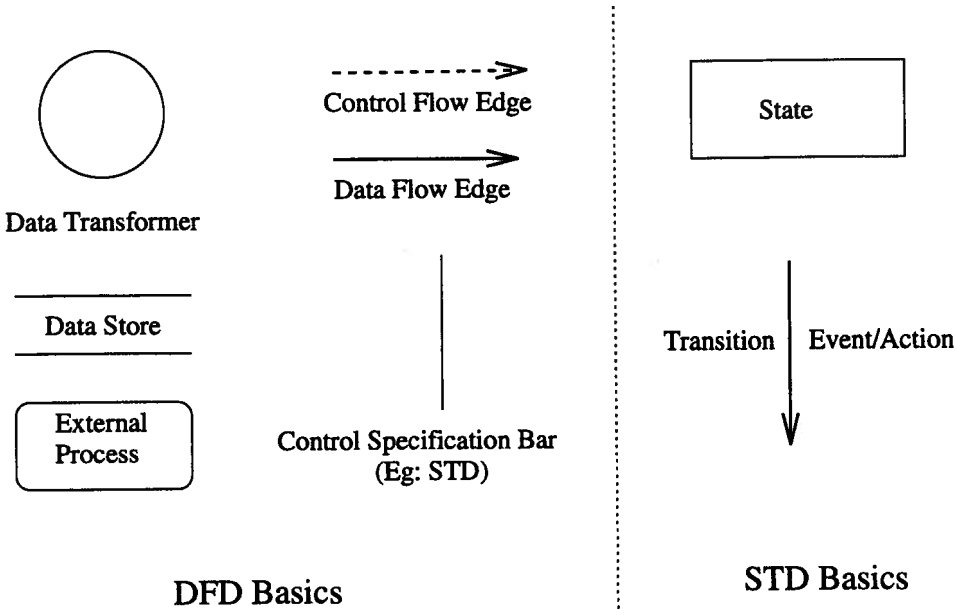


Figure 8.3: Basic building blocks of a DFD

8.3.1 DFD and STD

DFDs are used to describe flow of data through a network of processes that act on the data. The building blocks which form a DFD shown in Figure 8.3 are the following:

- Data transformers: system processes represented as nodes perform an action on inputs and produce outputs.
- Data flow edges: directed edges between data transformers representing flow of data.
- Control specification bar: a vertical bar representing a control specification such as an STD. It is used as a link between data specification and control specification.
- Control flow edges: directed edges carrying di screte valued data into data transformers. Control flow data are generated by control specifications such as STDs.
- Data stores: used as memory elements for storage of data.
- External process: processes part of the environment to indicate inputs to the system.

STDs represent states and state transitions of the system. A transition is labelled by an event that enables it, and the action triggered by it. A transition could trigger an action that can either be setting the value of a control signal in the DFD or activating a process. We show a simple STD in Figure 8.3. The control signals

generated by STD are communicated to the control flow edges of the DFD through a vertical bar that represents the STD.

8.3.2 Transformation of DFD to SIL

We present a transformation from DFDs into SIL as shown in Figure 8.4. In SIL, unlike in SA/SD, control flow is integrated with data flow. So, we can represent both control information present in the STD specification and data flow description of DFDs in SIL. We represent both data transformers and data stores as (conditional) nodes in SIL. Data stores are represented as nodes whose operation is to produce an output equal to the input it was given sometime in the past. Such a node is called a delay node in SIL. External processes are represented by input nodes. Every node except the input nodes have specific input, output and conditional ports. Input nodes have only output ports. Data flow edges of DFDs are transformed into data flow edges in SIL except that they connect output ports to input ports of

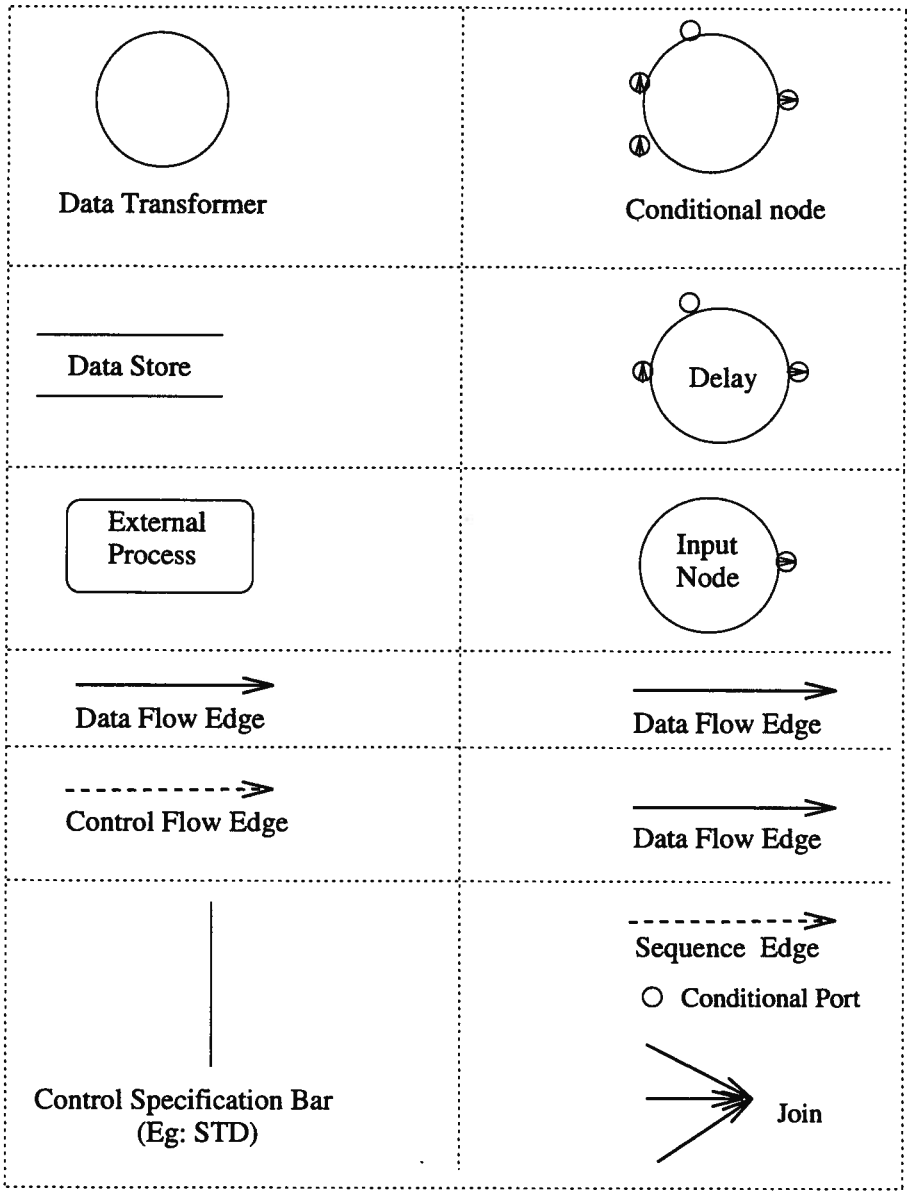


Figure 8.4: DFD to SIL Transformation

nodes rather than nodes in DFDs. Control flow edges are data flow edges in SIL that connect output ports to conditional ports. They communicate only binary data values. A discrete valued data could be encoded in terms of binary data values. Additional sequentiality and additional control information present in an STD are captured by the data flow edges and the notion of join in SIL. Once this transformation is done, we can use our formalization of SIL in PVS to specify systems that have been specified and analyzed using SA/SD approach.

8.3.3 Example Illustration

In order to describe how our formalism can address the specification problem in SA/SD, we consider an example often used to illustrate the SA/SD approach [Tea91]. The example we have chosen is based on the cruise control of an automobile. A specification in the SA/SD approach is as shown using DFD in Figure 8.5 and STD shown in Figure 8.6.

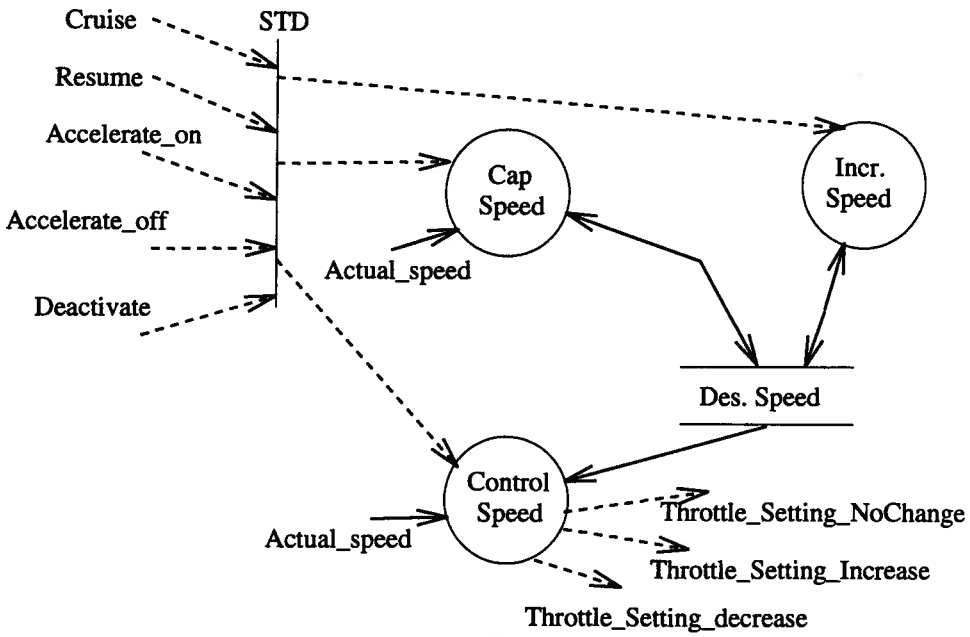


Figure 8.5: DFD for cruise control of an automobile

The capture_speed process captures the current value of the actual speed, and stores it as the desired speed in the data store. The control_speed process computes the difference between the actual speed and desired speed, and increases/decreases or does not change the throttle_setting. The increas_speed process retrieves the desired speed, and updates it by increasing it according to a predetermined amount. The control signals are generated as given in the STD.

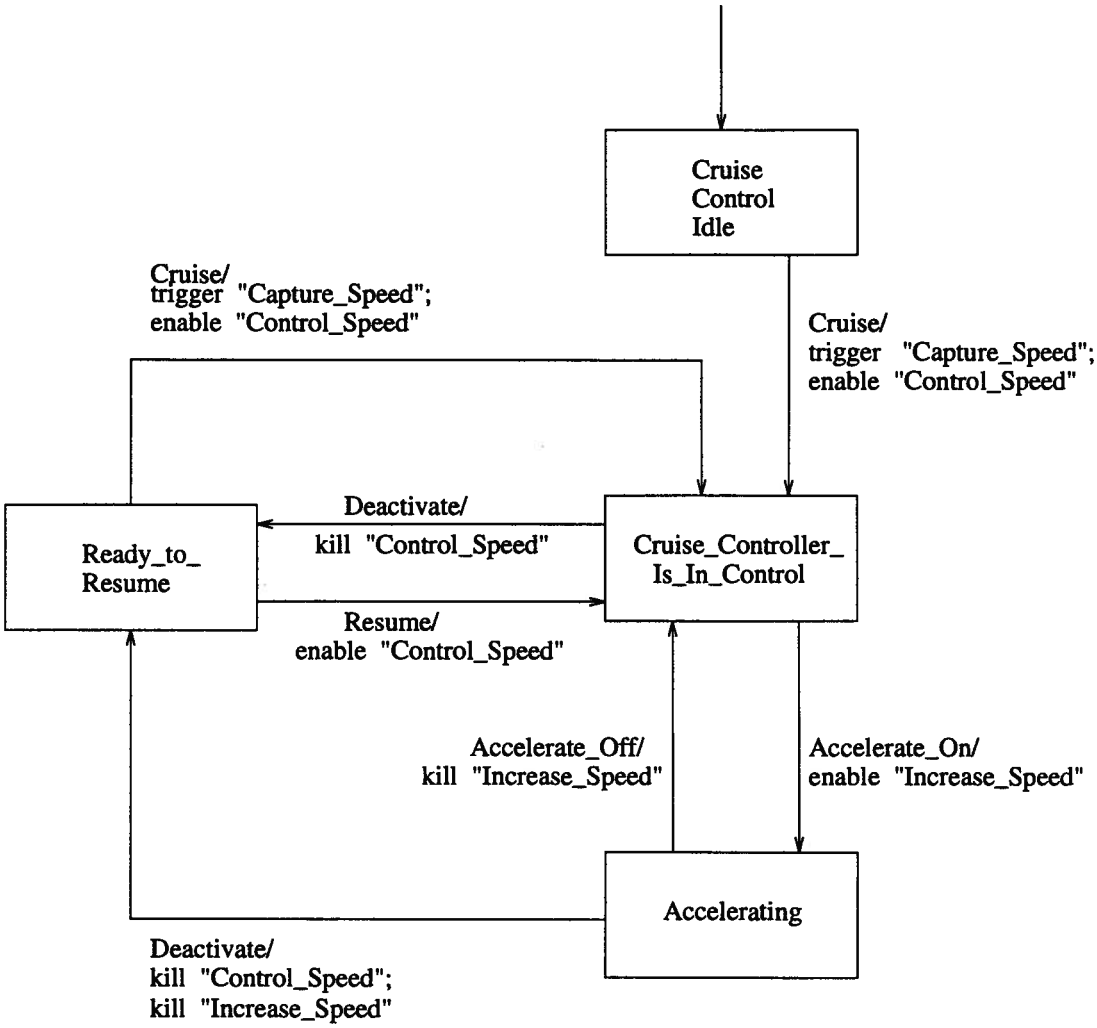


Figure 8.6: STD for cruise control of an automobile

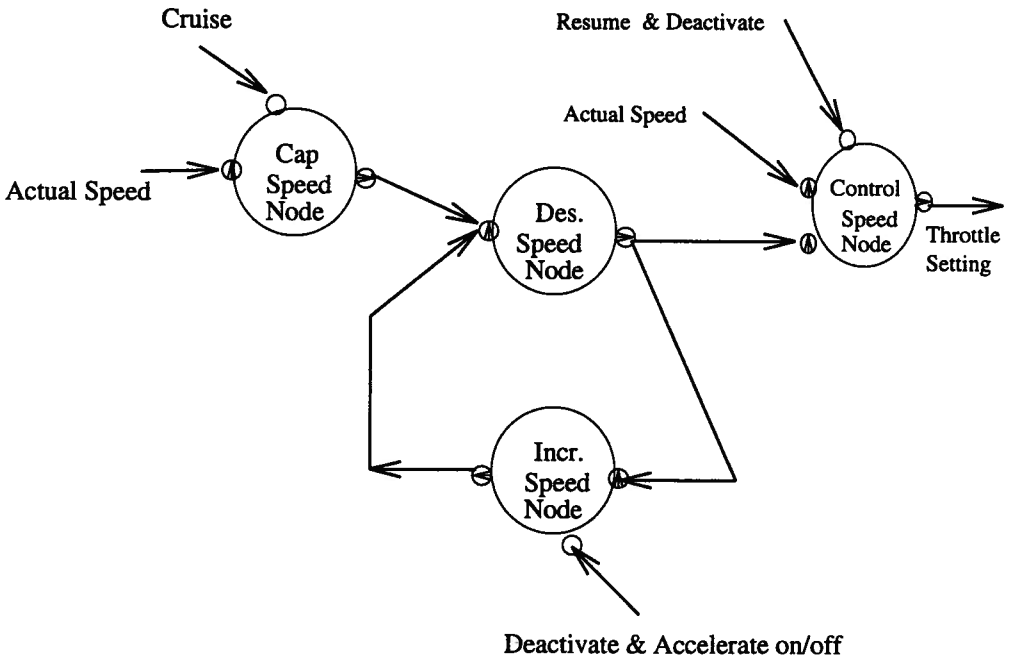


Figure 8.7: SIL for cruise control of an automobile

8.3.4 Transformation to SIL

We transform the SA/SD specification into the data flow graph formalism of SIL as shown in Figure 8.7. An important point that should be noted is the join at the input port of the Desired Speed node. Depending on which one of the control signals: cruise control or the deactivate & accelerate on/off is true, the input ports get the value from the output of the node whose condition is true. This incorporates the specification expressed by the STD.

8.3.5 Specification in PVS

We use the axiomatic specification of SIL presented in Chapter 5 to represent the SIL description of cruise control in PVS. We first specify the data relations of each of the nodes as uninterpreted constants of proper types as follows:

```
control_speed: [port, port -> port]

capture_speed, desired_speed, increase_speed: [port -> port]
```

We specify each of the nodes by instantiating a general conditional node with the required number of input/output/conditional ports, a data relation and an order relation.

```
capture_speed_node(cn:cnode):cnode =
cn WITH [inports:= inports(cn) WITH [size := 1]]
   WITH [outport:= outport(cn)]
   WITH [condport:= condport(cn)]
   WITH [datarel:= LAMBDA (par:parray),(p:port):
                  p= capture_speed(port_array(par)(0))]
   WITH [orderrel:= LAMBDA (par:parray),(p:port):
                   p < port_array(par)(0)]
```

```
control_speed_node(cn:cnode):cnode =
cn WITH [inports:= inports(cn) WITH [size := 2]]
   WITH [outport:= outport(cn)]
   WITH [condport:= condport(cn)]
   WITH [datarel:= LAMBDA (par:parray),(p:port):
                  p= control_speed(port_array(par)(0),
                                   port_array(par)(1))]
   WITH [orderrel:= LAMBDA (par:parray),(p:port):
                   p < port_array(par)(0)]
```

```
increase_speed_node(cn:cnode):cnode =
cn WITH [inports:= inports(cn) WITH [size := 1]]
   WITH [outport:= outport(cn)]
   WITH [condport:= condport(cn)]
   WITH [datarel:= LAMBDA (par:parray),(p:port):
                  p= control_speed(port_array(par)(0))]
   WITH [orderrel:= LAMBDA (par:parray),(p:port):
                   p < port_array(par)(0)]
```

```
desired_speed_node(cn:cnode):cnode =
cn WITH [inports:= inports(cn) WITH [size := 1]]
   WITH [outport:= outport(cn)]
   WITH [condport:= condport(cn)]
   WITH [datarel:= LAMBDA (par:parray),(p:port):
                  p= desired_speed(port_array(par)(0))]
   WITH [orderrel:= LAMBDA (par:parray),(p:port):
                   p < port_array(par)(0)]
```

The connectivity of the graph is specified using data flow edges expressed by the PVS relation *sileq*.

```
cruise_control(cruise,actual_speed,deactivate,accelerator,
               resume,throttlesettling,
               cn1,cn2,cn3) =
sileq(inports(capture_speed(cn1)(0)),actual_speed) AND
sileq(condport(capture_speed(cn1)),cruise) AND
sileq(condport(increase_speed(cn3)), deactivate&accelerate) AND
sileq(inport(control_speed(cn4))(0),actual_speed) AND
sileq(outport(control_speed(cn4)),throttle_setting) AND
sileq(outport(capture_speed(cn1)),inport(desired_speed(cn2))) AND
sileq(outport(desired_speed(cn2)),inport(increase_speed(cn3))) AND
sileq(outport(increase_speed(cn3)),inport(desired_speed(cn2))) AND
sileq(outport(desired_speed(cn2)), inport(control_speed(cn4))(1))
```

8.4 Constraint nets

A constraint net [ZM92, Zha94, ZM93, ZM94, ZM95] is a network of nodes representing functions and locations, and edges representing flow of data between input/output ports of the nodes. Thus, a constraint network represents a set of constraints on input/output traces. Constraint nets are used to model both discrete and continuous behavior of hybrid/real-time systems. In order to relate constraint nets to dependency graph formalism, we depart slightly from the description given by Zhang and Mackworth [ZM93, ZM95]. A constraint net is a five-tuple $CN = < L, I, O, T, E >$, where

- L is a finite set of locations or stores.
- I is a finite set of input ports.
- O is a finite set of output ports.
- T: Powerset(I) $\rightarrow$ O is a function label with a special name called *transduction* label.

- E:L $\times$ I $\cup$O is a set of edges between locations and ports.

where Powerset(I) is the set of all subsets of I.

A transduction associated with a node is causal function that maps traces on input ports to a trace on an output port. The traces could be continuous or discrete data streams. A constraint net can be viewed as a set of equations whose least solution gives its meaning. An constraint net example that models the computation of the distance as an integration of velocity over time is shown in Figure 8.8. The nodes drawn as boxes indicate transductions, while circles indicate locations. The value `x0` is the initial value of the integral (i.e distance at the initial time). The transduction v_t is the velocity at time t.

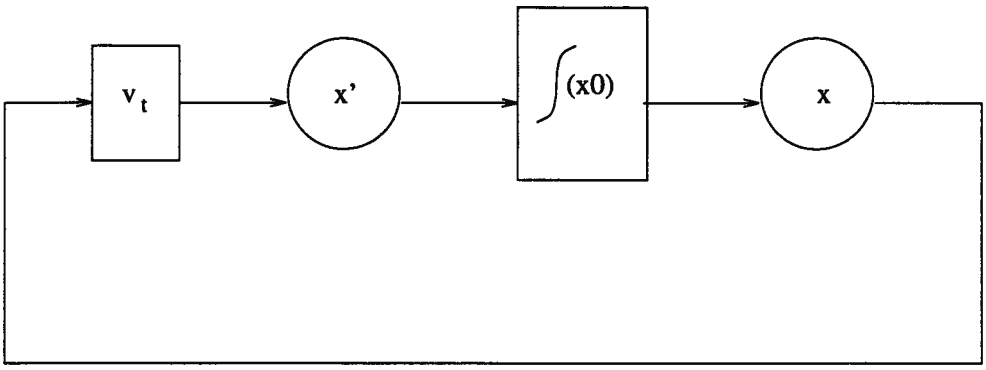


Figure 8.8: Constraint net for modeling integral computation over continuous time

8.4.1 Axiomatization of Constraint Nets

A constraint net, with transductions restricted to simple time-independent functions on traces can be viewed as a typical dependency graph without non-determinism. However, our formalization of dependency graphs allows general transductions to be specified by instantiating uninterpreted type for input port to be a trace of **data**, and having the type of output port as **data**, while **data** itself is an uninterpreted type:

```
data: TYPE % uninterpreted

trace: sequence[data]

input_port: TYPE = trace

output_port: TYPE = data

time: TYPE = real

transduction: TYPE = [[input_port,time] -> output_port]
```

The causality in dependency graphs is specified by sequence edges, and control specification in dependency graphs. The locations have to be viewed as a special kind of a node called store node. Thus, a constraint net can be viewed as a dependency graph, where the transductions would be associated with data relations that could be functions involving time as an additional parameter. Since, we have left the data type uninterpreted in our axiomatization of refinement (**silimp**), we have the freedom to extend it to cover higher refinement on higher order relations and thus, refinements for constraint nets. Such extensions would be difficult with an operational or denotational semantic framework. There are however, some restrictions to be imposed to make further use of our axiomatization of dependency graphs. For example, we have to restrict the number of output ports to one, for every node, and joins should not be allowed. Thus, the axiomatic specification we have provided

for dependency graphs could be used with minor modifications to specify constraint nets.

8.5 Separation of Control Flow from Data Flow

As we have observed in earlier sections, control flow description is separated from data flow description in SA/SD, while control and data form an integrated specification in SIL. Both approaches have some advantages and disadvantages. As an advantage, if the control flow is decoupled from data flow, a data flow could then be associated with a choice of control flow specification. The disadvantage with this approach would be the difficulty of applying transformations: one would have to have separate sets of transformations for each, and a precise definition of the interface between control flow and data flow. The advantage with having an integrated control and data flow specification, as in SIL, is the ability to precisely define

an integrated set of optimization and refinement transformations that involve both control flow and data flow.

Chapter 9

Discussion and Conclusions

One of the goals of a transformational design approach is to achieve designs that are *correct by construction*. We recall from Chapter 1 that a transformation is correct if the set of behaviors allowed by the implementation derived from the transformation is a subset of the behaviors permitted by the original specification. In this work, we have attempted to help accomplish the goal of correctness by construction in verifying the correctness of transformations used in dependency graph formalisms [Raj95a]. However, we have to note the distinction between the transformations as documented and intended by the informal specification and the transformations actually implemented in software. We explain this distinction in Section 9.1. In Section 9.2, we briefly present our experience in developing a formal specification from an informal document. We highlight the advantages of an axiomatic approach in Section 9.3. Finally, Section 9.4 summarizes the conclusions.

9.1 Intent versus Implementation

Our verification has addressed the transformations as documented and intended by the informal specification, and not the transformations actually implemented in software. One has to determine manually if the implemented transformations do, in fact, carry out the intended transformations that have been verified. In general, there is no practical mechanized method to check if software programs (such as those implemented in C) satisfy their specifications. But, in order to check the correctness of the implemented transformations, one has to first ensure that the intended transformations as documented are correct.

The correctness problem of the implemented transformations could be partly tackled in another manner. We can compare the dependency graph that is taken as the input by the software for transformation with the dependency graph that is the output of the software after applying the transformation. However, this would entail developing concrete behavioral models of the dependency graphs. But, a concrete behavior model basis would make the applicability of the formalization more restricted.

9.2 From Informal to Formal Specification

The most difficult part in this investigation has been developing a proper formal specification from informal specifications. Even though the informal specifications were well-documented, creating a formal specification required expressing informal ideas such as ***behavior*** and ***mutual exclusiveness*** in mathematically precise terms. One particular detail in this respect is the following: the informal document describes a value of a conditional node as ***undefined*** when the condition on its condition port is false. Introducing a notion of ***undefined*** value would need a special entity to be introduced for every data type. Further, we would also have to associate a meaning with such special entities. To avoid specification difficulties in stating what ***undefined*** means, we chose to specify how an ***undefined value*** affects the ***overall behavior*** of a subgraph in which such a node is embedded. Such choices have to be made with care towards specification and verification ease.

One of the first tasks that aids the specification process is the choice of abstraction level: how much of the detail present in the informal document should the specification represent? The choice could be based on how the formal specification

has to be verified. For example, we chose not to represent behavior at all: we could express behavioral equivalence (refinement) by an equivalence (refinement) relation, and express the properties that needed to be satisfied by the SIL graphs.

Another important issue in developing a formal specification from an informal document is deciding on data structures to represent entities specified informally. It is desirable to have a formal specification that very closely resembles the informal document. This is essential to map a formal specification back to its informal document. It is essential also for understanding a formal specification, and for tracing errors that have been found in the specification back to its informal representation. We can highlight one such data structure that PVS allows us to use: the *record* type. As we have seen in Table 5.2, it permits us to package all the fields of a conditional node `cn`, and then access the individual fields such as *inports* of the `cn` by `inports(cn)`. This syntax closely resembles the informal specification. Besides providing a simple syntax, the record type also allows making the type of one field depend on the type of another field. We have seen such *dependent typing* in our definition of arrays of ports `parray` in Chapter 5. Alternatively, we could have used Abstract Data Types (ADT) in our formal specification. This would have an advantage of encapsulating well-formedness of the structure of dependency graphs within

the behavior specification. However, this would mean imposing an abstract syntax structure for the behavior. Since our investigation primarily involves transformations which transform structure, it would be difficult to work with a specification that has an integrated structure and behavior.

The properties we have tabled in our formalism could form the basis of studying how we could formulate a composite behavior from smaller behavioral relations. In an earlier work at the register-transfer level [KvdW93], an automatic procedure for functional verification of retiming, pipelining and buffering optimization has been implemented in RetLab as part of the PHIDEO tool at PRL. We have arrived at proofs of properties that could form the basis of a semiautomatic procedure for checking refinement and equivalence at higher levels.

9.3 Axiomatic Approach vs Other Formal Approaches

The advantage in an axiomatic framework is that we could assert properties of SIL graphs that have to hold, without having to specify in detail the behavioral relations or their composition and equivalence. We could therefore embed off-the-shelf data-flow diagrams used in the Structured Analysis/Design approach [DeM79, Won94] in our formalism. One particular example of the advantage of our approach is establishing refinement and equivalence, without expressing the concrete relation between outputs and inputs of nodes. This property, expressed in Table 6.9 and Figure 6.9, does not use any information on the concrete data and order relations of the nodes. Moreover, the automatic verification procedures, simple interactive commands, and many features such as editing and rerunning proofs in PVS made the task of checking properties and correctness much easier than anticipated.

In contrast to an axiomatic approach, a model-oriented approach would compare two dependency graph models with respect to behavior. Such a model-comparison method would involve verifying that the behavior of the transformed model satisfies the behavior of the original model. However, this entails developing concrete behav-

ioral models of the dependency graphs, and formulating the meaning of behavioral refinement, and equivalence. Such concrete modeling of behavior, refinement and equivalence would impose restrictions on the domains where the formalization could be applied. For instance, we laid down the definition of `refines` between two graphs in Section 6.2 axiomatically instead of proposing a subset relationship between the data relations of the graphs. Furthermore, such a modeling would make it inconvenient to study the correctness of transformations on graphs with arbitrary structure. For example, in our approach, we could handle nodes with an unspecified number of ports in studying the correctness problem. On the other hand, an operational or denotational model of a language relates closely to a computation model of the language. This implies that it would be difficult to reason about the properties of the language without reference to the undelying behavior model. For example, we have pointed out earlier in Section 8.4 that, we would not be able to use the operational/denotational semantics developed for dependency graphs [dJ93, HK93] to reason about constraint nets without major modifications to the semantics. However, in our axiomatic specification, we could interpret a port as representing a trace, and data relations as higher-order functions on traces with an optional time parameter. Such a simple interpretation suggested that the axiomatic specification developed for dependency graphs in this dissertation could be used to specify and reason about constraint nets. However, denotational and operational models worked out by de Jong and Huijs [dJ93, HK93] could be used as a concrete model

that satisfies the axiomatic specification.

As a typical example, we are given the behavioral relations of the nodes in a SIL graph and the structural connectivity of the graph. There is no general way to compose these relations into a single behavioral relation for comparison with that obtained from another SIL graph. Moreover, from the behavioral description in SIL, it is not possible in general to extract a state machine or a finite automaton model, and use state machine or automata comparison techniques. This is due to the generality of the dependency graph behavior. In addition, since many synthesis transformations are applied to descriptions of behavior within a single clock cycle, there is no explicit notion of state in such a description. This reinforces the judgment that state machine or automata comparison techniques are not suitable.

9.4 Conclusions and Directions for Further Research

In this work, we have provided an axiomatic specification for a general dependency graph specification language. We have given a small set of axioms that capture a general notion of refinement and equivalence of dependency graphs. We have specified and verified about a dozen of the optimization and refinement transformations. We found errors in this process, and suggested corrections. We have also generalized the transformations by weakening the preconditions for applying the transformations, and verified their correctness. In this process, we have devised new transformations for further optimization and refinement than would have been possible before.

The preconditions for transformations that existed in the SPRITE and TRADES transformational design prior to our specification and verification task were stronger than necessary in order to make them purely syntactic checks made at run-time. However, precondition weakening resulting from our formal approach, involves comparing weights on edges as illustrated in Figure 7.4. If the weights on the edges are the result of an execution of a subgraph in SIL, then run-time checks for preconditions would have to be enhanced to compute the weights, and then compare the

resulting weights. The tool used for designing systems in SIL could communicate with PVS by sending a SIL graph with the computed weights on the edges, and receive a possible set of transformed graphs. Our work has also aided investigating interactions between the transformations, and thus the importance of the order of applying the transformations. The transformations we have verified are being used in industry to design hardware from high level specifications.

We have enhanced existing theorem proving technqiues by integrating efficient simplifiers such as model-checkers based on BDD. This enabled an efficient mechanical verification of the correctness of transformations on dependency graphs. Our enhancement to theorem proving techniques has facilitated automatic techniques to solve large verification problems in hardware and software that could not have been tackled by either model-checking or theorem proving alone. The transformations we have verified are being used in industry to design hardware from high level specifications.

An investigation into the correctness of other transformations such as those involving scheduling and resource allocation in dependency graphs could form a part

of further research. Another direction for extending the research is to study how verification techniques and results could be seamlessly integrated with Very Large Scale Integration (VLSI) Computer-Aided Design (CAD) tools, software/hardware co-synthesis/co-design tools, and Computer-Aided Software Engineering (CASE) tools. The approach we have used, based on expressing properties at a high level, does not depend on the underlying model of behavior. This enabled us to use our formalism for dependency graph specifications in other areas such as structured analysis in software design. Thus, the ability to capture an off-the-shelf formalism underpins our thesis that an axiomatic specification coupled with an efficient mechanical verification is the most suitable approach to investigate the correctness of transformations on generic dependency graphs, independent of the underlying behavior models.

Bibliography

[AAD93] F.V. Aelten, J. Allen, and S. Devadas. Verification of relations between synchronous machines. *IEEE Transactions on Computer-Aided Design*, 12(12), December 1993.

[AAD94] F.V. Aelten, J. Allen, and S. Devadas. Even-based verification of synchronous globally controlled logic designs against signal flow graphs. *IEEE Transactions on Computer-Aided Design*, 13(1), January 1994.

[AC72] Francis E. Allen and John Cocke. A catalogue of optimization transformations. In Randall Rustin, editor, *Design and Optimization of Compilers, Courant Computer Symposium 5*. Prentice Hall, Inc., Englewood Cliffs, NJ, March 1972.

[AFJWed] E.A. Aschroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Declarative Programming.* Oxford University Press, Oxford, U.K., to be published.

[AL94] M. Aagaard and M. Leeser. PBS: Proven boolean algorithm. *IEEE Transactions on Computer-Aided Design*, 13(4), January 1994.

[Ang94] C. Angelo. *Formal Hardware Verification in a Silicon Compilation Environment by means of theorem proving.* PhD thesis, K.U. Leuven/IMEC, Belgium, February 1994.

[ASU71] A.V. Aho, Ravi Sethi, and J. D. Ullman. Code optimization and finite church-rosser systems. In Randall Rustin, editor, *Design and Optimization of Compilers, Courant Computer Symposium 5.* Prentice Hall, Inc., Englewood Cliffs, NJ, March 1971.

[AW77] E.A. Aschroft and W.W. Wadge. Lucid: A nonprocedural language with iteration. *CACM*, 20(7):519–526, 1977.

[Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bar81] M. R. Barbacci. Instruction set processor specifications (isps): The notation and applications. *ieeetc*, C-30(1):24–40, 1981.

[BCD+88] R.K. Brayton, R. Camposano, G. DeMicheli, R.H.J.M. Otten, and J.T.J. van Eijndhoven. The yorktown silicon compiler system. In D. Gajski, editor, *The Yorktown Silicon Compiler System.* Addison-Wesley, 1988.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[BCM+90a] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[BCM+90b] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings*

of the Fifth Annual Symposium on Logic in Computer Science. Association for Computing Machinery, July 1990.

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45. Association for Computing Machinery, 1990.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Cam89] R. Camposano. Behavior preserving transformations in high level synthesis. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 106–128, Ithaca, NY, July 1989. Volume 408 of *Lecture Notes in Computer Science*, Springer-Verlag.

[CBL92] R. Chapman, G. Brown, and M. Leeser. Verified high-level synthesis in BEDROC. In *Proceedings of the 1992 European Design Automation Conference.* IEEE Press, March 1992.

[CG87] E.M. Clarke and Ö. Grumberg, editors. *Annual Review of Computer Science, Volume 2.* Annual Reviews, Inc., Palo Alto, CA, 1987.

[CGH94] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David Dill, editor, *Computer-Aided Verification, CAV '94*, pages 415–427, Stanford, CA, June 1994. Volume 818 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Cha79] Donald D. Chamberlin. The "single-assignment" approach to parallel processing. *Selected papers on Data Flow Architecture, Part 1*, May 1979.

[Cle89] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. Technical Report 2/89, University of Sussex, March

1989.

[Compu90] Computer General Electronic Design U.K. *The ELLA Language Reference Manual.* Computer General Electronic Design The New Church Henry St. Bath BA1 1JR U.K., 1990.

[Cyr93] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical report, SRI International, December 1993. Report CSL-93-12.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New Jersey, 1979.

[dJ93] G.G de Jong. *Generalized data flow graphs: theory and applications.* PhD thesis, Eindhoven University of Technology, The Netherlands, October 1993.

[EL85] E.A. Emerson and C.L Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[EMH+93] W.J.A Engelen, P.F.A. Middelhoek, C. Huijs, J. Hofstede, and Th. Krol. Applying software transformations to SIL. Technical Report SPRITE deliverable Ls.a.5.2/UT/Y5/M6/1A, Philips Research Laboratories, Eindhoven The Netherlands, April 1993.

[Fou90] M. P. Fourman. Formal system design. In J. Staunstrup, editor, *Formal Methods for VLSI Design.* IFIP, North-Holland, 1990.

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

[Gun92] Carl A. Gunter. *Semantics of programming languages : structures and techniques.* MIT Press, Cambridge, MA, 1992.

[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in Systems Design*, 1(2/3):151–238, October 1992.

[HFB93] N. Halbwachs, J-C. Fernandez, and A. Bouajjani. An executable temporal logic to express safety properties and its connection with the language lustre. In *Sixth International Symposium on Lucid and Intensional Programming.* Universite Laval, April 1993.

[HHK92] C. Huijs, J. Hofstede, and Th. Krol. Transformations and semantical checks for SIL-1. Technical Report SPRITE deliverable LS.a.5.1/UT/Y4/M6/1, Philips Research Laboratories, Eindhoven The Netherlands, November 1992.

[Hil85] P. N. Hilfinger. Silage: a high-level language and silicon compiler for digital signal processing. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 213–216, Portland, OR, May 1985. IEEE.

[HK93] C. Huijs and Th. Krol. A formal semantic model to fit SIL for transformational design. In *Proceedings of Euromicro Microprocessing and Microprogramming 39*, liverpool, September 1993.

[Hoo94] Jozef Hooman. Correctness of real time systems by construction. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40, Lübeck, Germany, September 1994. Volume 863 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Jan93a] G. L. J. M. Janssen. *ROBDD Software.* Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[Jan93b] G. L. J. M. Janssen. *ROBDD Software.* Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[JLR+91] J. Joyce, E. Liu, J. Rushby, N. Shankar, R. Suaya, and F. von Henke. From formal verification to silicon compilation. In *IEEE Compcon*, pages 450–455, San Francisco, CA, February 1991.

[Joh84] S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, Cambridge MA, 1984.

[Joh95] Steven D. Johnson, editor. *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, Chiba, Japan, August 1995. Proceedings published in a single volume jointly with ASP-DAC '95, CHDL '95, and VLSI '95, IEEE Catalog no. 95TH8102.

[JS90] G. Jones and M. Sheeran. Circuit design in ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design.* IFIP, North-Holland, 1990.

[JS93] Jeffrey J. Joyce and Carl-Johan H. Seger. Linking Bdd-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference.* Association for Computing Machinery, 1993.

[KeH+92] W.E.H. Kloosterhuis, M.R.R. eyckmans, J. Hofstede, C. Huijs, Th. Krol, O.P. McArdle, W.J.M. Smits, and L.G.L. Svensson. The SPRITE input language SIL-1, language report. Technical Report SPRITE, deliverable Ls.a.a / Philips / Y3 / M12 / 2, Philips Research Laboratories, Eindhoven The Netherlands, October 1992.

[KK94] Ramayya Kumar and Thomas Kropf, editors. *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit*

Design, Bad Herrenalb (Blackforest), Germany, September 1994. Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.

[Klo94] W.E.H. Kloosterhuis. Personal communication. January 1994.

[KMN+92] Th. Krol, J.v. Meerbergen, C. Niessen, W. Smits, and J. Huisken. The SPRITE input language, an intermediate format for high level synthesis. In *Proceedings of European Design Automation Conference*, pages 186–192, Brussels, March 1992.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, pages 333–354, December 1983.

[KvdW93] A. P. Kostelijk and A. van der Werf. Functional verification for retiming and rebuffering optimization. In *Proceedings of The European Conference on Design Automation with the European Event in ASIC Design*, Paris, February 1993. IEEE Computer Society.

[LOR+93] Patrick Lincoln, Sam Owre, John Rushby, N. Shankar, and Friedrich von Henke. Eight papers on formal verification. Technical Report SRI-CSL-93-4, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993.

[LPT93] P.G. Larsen, N. Plat, and H. Toetenel. A formal semantics of data flow diagrams. *Formal Aspects of Computing*, 3(1), 1993.

[McF93] M.C. McFarland. Formal analysis of correctness of behavioral transformations. *Formal Methods in System Design*, 2(3):231–257, June 1993.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.

[Mid93] P.F.A. Middelhoek. Transformational design of digital circuits. In *Proceedings of the Seventh Workshop Computersystems*, Eindhoven

The Netherlands, November 1993.

[Mid94a] P.F.A. Middelhoek. Transformational design of a direction detector for the progressive scan conversion algorithm. Technical report, Computer Science, University of Twente, Enschede, The Netherlands, May 1994. Preliminary.

[Mid94b] P.F.A. Middelhoek. Transformational design of digital signal processing applications. In *Proceedings of the ProRISC/IEEE workshop on CSSP*, pages 175–180, Eindhoven The Netherlands, March 1994.

[MP83] M.C. McFarland and A.C. Parker. An abstract model of behavior for hardware descriptions. *IEEE Transactions on Computers*, C-32(7):621–636, July 1983.

[NRP95] Vijay Nagasamy, Sreeranga P. Rajan, and Preeti R. Panda. Fiber channel protocol: Formal specification, verification, and design trade-off/analysis. In *Proceedings of the 1995 Silicon Valley Networking Conference and Exposition.* Systech Research, April 1995.

[oEE88] The Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual 1076-88.* IEEE Press, New York, 1988.

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

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[OSR93b] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release).* Computer

Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[Par89] D. Park. Finiteness is mu-effable. Technical Report 3, The University of Warwick, March 1989. Theory of Computation Report.

[PBJ+91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *18th ACM Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, 1981.

[Raj92] Sreeranga P. Rajan. Executing hol specifications: Towards an evaluation semantics for classical higher order logic. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications (5th International Workshop, HUG '92)*, Leuven, Belgium, September 1992. North-Holland.

[Raj94a] Sreeranga P. Rajan. Transformations in high-level synthesis: Formal specification and efficient mechanical verification. Technical Report SRI-CSL-94-10, SRI International, Menlo Park, California, October 1994.

[Raj94b] Sreeranga P. Rajan. Transformations in high level synthesis: Specification and verification. Technical Report NL-TN 118/94, Philips Research Laboratories, Eindhoven, The Netherlands, April 1994.

[Raj95a] Sreeranga P. Rajan. Correctness of transformations in high level synthesis. In Johnson [Joh95], pages 597–603.

[Raj95b] Sreeranga P. Rajan. Formal verification of transformations on dependency graphs in optimizing compilers. In *Proceedings of the California Software Engineering Symposium*. IRUS, University of California, Irvine, March 1995.

[RJS93] Sreeranga P. Rajan, J.J. Joyce, and C-J. Seger. From abstract data types to shift registers. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, Vancouver, Canada, August 1993. Number 780 in Lecture Notes in Computer Science, Springer-Verlag.

[Ros90] Lars Rossen. Formal ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. IFIP, North-Holland, 1990.

[RRV95] Sreeranga P. Rajan, P. Venkat Rangan, and Harrick M. Vin. A formal basis for structured multimedia collaborations. In *Submitted to 2nd IEEE International Conference on Multimedia Computing and Systems*, Washington, D.C., 1995. IEEE Press.

[RSS95] Sreeranga P. Rajan, N. Shankar, and M. Srivas. An integration of model-checking with automated proof checking. In *7th Conference on Computer-Aided Verification*, July 1995.

[RTJ93] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. An introduction to behavior tables. Technical report, Computer Science Department, Indiana University, Bloomington, IN, December 1993. No. 392.

[Rus93] John Rushby. Formal methods and certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also available as NASA Contractor Report 4551, December 1993.

[Sha94] N. Shankar. Personal communication. October 1994.

[SM95] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to a commercial microprocessor. In Johnson [Joh95], pages 493–502.

[Smi90] Douglas R. Smith. Kids: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, SE-16(9), September 1990.

[SOR93a] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[SOR93b] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[TDW+88] D. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, and R.L. Blackburn. The system architect's workbench. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 337–343. Association for Computing Machinery, 1988.

[Tea91] Teamwork. *Teamwork CASE Tool Manuals*, 1991.

[vdWvMM+94] A. van der Werf, J.L. van Meerbergen, O. McArdle, P.E.R. Lippens, W.F.J. Verhaegh, and D. Grant. Processing unit design. In *Proceedings of the SPRITE workshop on "VLSI Synthesis for DSP"*, Eindhoven, March 1994. Philips Research Labs.

[vEJ94] C. A. J. van Eijk and G. L. J. M. Janssen. Exploiting structural similarities in a BDD-based verification method. In Kumar and Kropf [KK94], pages 151–166.

[Vem90] R. Vemuri. How to prove the completeness of a set of register level design transformations. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 207–212. Association for Computing Machinery, 1990.

[vWS91] J. von Wright and K. Sere. Program transformations and refinements in hol. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Higher Order Logic Theorem Proving and its Applications (4th International Workshop, HUG '91)*, Davis, CA, August 1991. IEEE Computer Society.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–22, September 1990.

[Won94] M. Wong. Informal, semi-formal, and formal approaches to the specification of software requirements. Technical report, Department of Computer Science, UBC, Vancouver, Canada, September 1994.

[WP94] P.G. Whiting and R.S.V. Pascoe. A history of data-flow languages. Technical Report TR-SA-94-02, Division of Information Technology, Commonwealth Scientific and Industrial Research Organization, Carlton, Australia, March 1994.

[Zha94] Y. Zhang. A foundation for the design and analysis of robotic systems and behaviors. Technical report, Department of Computer Science, UBC, Vancouver, Canada, September 1994.

[ZM92] Y. Zhang and A.K. Mackworth. Constraint nets: A semantic model of real-time embedded systems. Technical Report 92-10, Department of Computer Science, UBC, Vancouver, Canada, 1992.

[ZM93] Y. Zhang and A. K. Mackworth. Constraint programming in constraint nets. In *First Workshop on Principles and Practice of Constraint Programming*, pages 303–312, 1993.

[ZM94] Y. Zhang and A. K. Mackworth. Will the robot do the right thing? In *Proceedings of Artificial Intelligence*, pages 255–262, Banff, Alberta, May 1994.

[ZM95] Y. Zhang and A. K. Mackworth. Constraint nets: a semantic model for hybrid dynamic systems. *Theoretical Computer Science*, 138(1), February 1995.

Appendix A

Peterson's Mutual Exclusion Algorithm: Automatic verification

```
safe :

  |-------
{1}   (FORALL
       (state: staterec):
         (init(state) AND initsem(state))
           IMPLIES
           AG(N,
              LAMBDA
```

```
            state:
              NOT
                (critical?(pc1(state))
                   AND critical?(pc2(state))))(state))

Running step: (Model-Check)

init  rewrites init(state!1)
   to idle?(pc1(state!1)) AND idle?(pc2(state!1))
initsem  rewrites initsem(state!1)
   to NOT sem1(state!1) AND NOT sem2(state!1)
EU  rewrites
  EU(N, (LAMBDA u: TRUE),
     NOT LAMBDA state: NOT (critical?(pc1(state)) AND
                            critical?(pc2(state))))
   to mu(LAMBDA
         (Q: pred[staterec]):
           (NOT
              LAMBDA
              state: NOT (critical?(pc1(state)) AND
                          critical?(pc2(state)))
              OR ((LAMBDA u: TRUE) AND EX(N, Q))))
EF  rewrites
  EF(N, NOT LAMBDA state: NOT (critical?(pc1(state)) AND
                               critical?(pc2(state))))
   to mu(LAMBDA
         (Q: pred[staterec]):
           (NOT
              LAMBDA
              state: NOT (critical?(pc1(state)) AND
                          critical?(pc2(state)))
              OR ((LAMBDA u: TRUE) AND EX(N, Q))))
AG  rewrites
  AG(N, LAMBDA state: NOT (critical?(pc1(state)) AND
                           critical?(pc2(state))))
   to NOT
        mu(LAMBDA
           (Q: pred[staterec]):
             (NOT
                LAMBDA
                state: NOT (critical?(pc1(state)) AND
```

```
                        critical?(pc2(state)))
                OR ((LAMBDA u: TRUE) AND EX(N, Q))))
NOT rewrites
  (NOT
     mu(LAMBDA
        (Q: pred[staterec]):
          (NOT
             LAMBDA
             state: NOT (critical?(pc1(state)) AND
                         critical?(pc2(state)))
             OR ((LAMBDA u: TRUE) AND EX(N, Q)))))(state!1)
  to NOT
       mu(LAMBDA
          (Q: pred[staterec]):
            (NOT
               LAMBDA
               state: NOT (critical?(pc1(state)) AND
                           critical?(pc2(state)))
               OR ((LAMBDA u: TRUE) AND EX(N, Q))))(state!1)
nextp1 rewrites nextp1(u, v)
  to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
     ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
     ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
         OR (exiting?(pc1(v)) AND NOT try(v))
     ELSE idle?(pc1(v))
     ENDIF
nextp2 rewrites nextp2(u, v)
  to IF idle?(pc2(u)) THEN idle?(pc2(v)) OR
                                (entering?(pc2(v)) AND NOT try(v))
     ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
     ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
         OR (exiting?(pc2(v)) AND try(v))
     ELSE idle?(pc2(v))
     ENDIF
nextsem rewrites nextsem(u, v)
  to IF
        (entering?(pc1(u))
           OR entering?(pc2(u)) OR exiting?(pc1(u)) OR
                                   exiting?(pc2(u)))
```

```
        THEN (entering?(pc1(u)) IMPLIES sem1(v))
          AND (entering?(pc2(u)) IMPLIES sem2(v))
                AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                      AND (exiting?(pc2(u)) IMPLIES NOT sem2(v))
      ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
      ENDIF
N  rewrites N(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                                  critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
        AND
        IF idle?(pc2(u)) THEN idle?(pc2(v))
            OR (entering?(pc2(v)) AND NOT try(v))
        ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
        ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
            OR (exiting?(pc2(v)) AND try(v))
        ELSE idle?(pc2(v))
        ENDIF
          AND
          IF
              (entering?(pc1(u))
                 OR entering?(pc2(u))
                      OR exiting?(pc1(u)) OR exiting?(pc2(u)))
            THEN (entering?(pc1(u)) IMPLIES sem1(v))
              AND (entering?(pc2(u)) IMPLIES sem2(v))
                    AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                          AND
                          (exiting?(pc2(u))
                             IMPLIES NOT sem2(v))
          ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
          ENDIF
```

By rewriting and mu-simplifying,

```
Q.E.D.


Run time  = 31.66 secs.
```

```
Real time = 40.55 secs.
NIL
>
```

The proof transcript of the liveness property proceeds in the same manner as above:

```
live :

  |-------
{1}   (FORALL
       (s: staterec):
         AG(N,
            (LAMBDA s: entering?(pc1(s)))
              IMPLIES EF(N, LAMBDA s: critical?(pc1(s))))(s))

Running step: (Model-Check)
EU  rewrites EU(N, (LAMBDA u: TRUE), LAMBDA s: critical?(pc1(s)))
   to mu(LAMBDA
          (Q: pred[staterec]):
            (LAMBDA s: critical?(pc1(s)) OR ((LAMBDA u: TRUE) AND EX(N, Q))))
EF  rewrites EF(N, LAMBDA s: critical?(pc1(s)))
   to mu(LAMBDA
          (Q: pred[staterec]):
            (LAMBDA s: critical?(pc1(s)) OR ((LAMBDA u: TRUE) AND EX(N, Q))))
EU  rewrites
  EU(N, (LAMBDA u: TRUE),
     NOT
       ((LAMBDA s: entering?(pc1(s)))
          IMPLIES
          mu(LAMBDA
             (Q: pred[staterec]):
               (LAMBDA s: critical?(pc1(s))
```

```
                  OR ((LAMBDA u: TRUE) AND EX(N, Q))))))
  to mu(LAMBDA
         (Q: pred[staterec]):
           (NOT
              ((LAMBDA s: entering?(pc1(s)))
                 IMPLIES
                 mu(LAMBDA
                    (Q: pred[staterec]):
                      (LAMBDA s: critical?(pc1(s))
                         OR ((LAMBDA u: TRUE) AND EX(N, Q)))))
              OR ((LAMBDA u: TRUE) AND EX(N, Q))))
EF rewrites
  EF(N,
     NOT
       ((LAMBDA s: entering?(pc1(s)))
          IMPLIES
          mu(LAMBDA
             (Q: pred[staterec]):
               (LAMBDA s: critical?(pc1(s))
                  OR ((LAMBDA u: TRUE) AND EX(N, Q))))))
  to mu(LAMBDA
         (Q: pred[staterec]):
           (NOT
              ((LAMBDA s: entering?(pc1(s)))
                 IMPLIES
                 mu(LAMBDA
                    (Q: pred[staterec]):
                      (LAMBDA s: critical?(pc1(s))
                         OR ((LAMBDA u: TRUE) AND EX(N, Q)))))
              OR ((LAMBDA u: TRUE) AND EX(N, Q))))
AG rewrites
  AG(N,
     (LAMBDA s: entering?(pc1(s)))
       IMPLIES
       mu(LAMBDA
          (Q: pred[staterec]):
            (LAMBDA s: critical?(pc1(s))
               OR ((LAMBDA u: TRUE) AND EX(N, Q)))))
  to NOT
       mu(LAMBDA
          (Q: pred[staterec]):
```

```
            (NOT
               ((LAMBDA s: entering?(pc1(s)))
                  IMPLIES
                  mu(LAMBDA
                     (Q: pred[staterec]):
                       (LAMBDA s: critical?(pc1(s))
                          OR ((LAMBDA u: TRUE) AND EX(N, Q)))))
                  OR ((LAMBDA u: TRUE) AND EX(N, Q))))
NOT rewrites
  (NOT
     mu(LAMBDA
        (Q: pred[staterec]):
          (NOT
             ((LAMBDA s: entering?(pc1(s)))
                IMPLIES
                mu(LAMBDA
                   (Q: pred[staterec]):
                     (LAMBDA s: critical?(pc1(s))
                        OR ((LAMBDA u: TRUE) AND EX(N, Q)))))
             OR ((LAMBDA u: TRUE) AND EX(N, Q)))))(s!1)
  to NOT
       mu(LAMBDA
          (Q: pred[staterec]):
            (NOT
               ((LAMBDA s: entering?(pc1(s)))
                  IMPLIES
                  mu(LAMBDA
                     (Q: pred[staterec]):
                       (LAMBDA s: critical?(pc1(s))
                          OR ((LAMBDA u: TRUE) AND EX(N, Q)))))
               OR ((LAMBDA u: TRUE) AND EX(N, Q))))(s!1)
nextp1 rewrites nextp1(u, v)
  to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
     ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
     ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
         OR (exiting?(pc1(v)) AND NOT try(v))
     ELSE idle?(pc1(v))
     ENDIF
nextp2 rewrites nextp2(u, v)
```

```
   to IF idle?(pc2(u)) THEN idle?(pc2(v)) OR
                                (entering?(pc2(v)) AND NOT try(v))
      ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
      ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
          OR (exiting?(pc2(v)) AND try(v))
      ELSE idle?(pc2(v))
      ENDIF
nextsem  rewrites nextsem(u, v)
   to IF
         (entering?(pc1(u))
            OR entering?(pc2(u)) OR exiting?(pc1(u)) OR
                                    exiting?(pc2(u)))
        THEN (entering?(pc1(u)) IMPLIES sem1(v))
          AND (entering?(pc2(u)) IMPLIES sem2(v))
                AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                      AND (exiting?(pc2(u)) IMPLIES NOT sem2(v))
      ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
      ENDIF
N  rewrites N(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
        AND
        IF idle?(pc2(u)) THEN idle?(pc2(v))
            OR (entering?(pc2(v)) AND NOT try(v))
        ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
        ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
            OR (exiting?(pc2(v)) AND try(v))
        ELSE idle?(pc2(v))
        ENDIF
          AND
          IF
              (entering?(pc1(u))
                 OR entering?(pc2(u))
                      OR exiting?(pc1(u)) OR exiting?(pc2(u)))
            THEN (entering?(pc1(u)) IMPLIES sem1(v))
```

```
              AND (entering?(pc2(u)) IMPLIES sem2(v))
                    AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                          AND
                          (exiting?(pc2(u))
                             IMPLIES NOT sem2(v))
          ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
          ENDIF
nextp1  rewrites nextp1(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
nextp2  rewrites nextp2(u, v)
   to IF idle?(pc2(u)) THEN idle?(pc2(v)) OR
                                (entering?(pc2(v)) AND NOT try(v))
      ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
      ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
          OR (exiting?(pc2(v)) AND try(v))
      ELSE idle?(pc2(v))
      ENDIF
nextsem  rewrites nextsem(u, v)
   to IF
          (entering?(pc1(u))
             OR entering?(pc2(u)) OR exiting?(pc1(u)) OR
                                     exiting?(pc2(u)))
        THEN (entering?(pc1(u)) IMPLIES sem1(v))
          AND (entering?(pc2(u)) IMPLIES sem2(v))
                AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                      AND (exiting?(pc2(u)) IMPLIES NOT sem2(v))
      ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
      ENDIF
N  rewrites N(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
```

```
      OR (exiting?(pc1(v)) AND NOT try(v))
    ELSE idle?(pc1(v))
    ENDIF
      AND
      IF idle?(pc2(u)) THEN idle?(pc2(v))
          OR (entering?(pc2(v)) AND NOT try(v))
      ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
      ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
          OR (exiting?(pc2(v)) AND try(v))
      ELSE idle?(pc2(v))
      ENDIF
        AND
        IF
            (entering?(pc1(u))
               OR entering?(pc2(u))
                    OR exiting?(pc1(u)) OR exiting?(pc2(u)))
          THEN (entering?(pc1(u)) IMPLIES sem1(v))
            AND (entering?(pc2(u)) IMPLIES sem2(v))
                  AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                        AND
                        (exiting?(pc2(u))
                           IMPLIES NOT sem2(v))
        ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
        ENDIF
```

By rewriting and mu-simplifying,

```
Q.E.D.


Run time  = 47.86 secs.
Real time = 66.98 secs.
NIL
>
```

The following proof that every path is free of deadlock fails as expected:

```
not_live :

  |-------
{1}   (FORALL
       (s: staterec):
         AG(N,
            (LAMBDA s: entering?(pc1(s)))
              IMPLIES AF(N, LAMBDA s: critical?(pc1(s))))(s))

Rule? (Model-Check)
EG  rewrites EG(N, NOT LAMBDA s: critical?(pc1(s)))
   to nu(LAMBDA
          (Q: pred[staterec]): (NOT LAMBDA s: critical?(pc1(s)) AND
                                              EX(N, Q)))
AF  rewrites AF(N, LAMBDA s: critical?(pc1(s)))
   to NOT
        nu(LAMBDA
           (Q: pred[staterec]):
             (NOT LAMBDA s: critical?(pc1(s)) AND EX(N, Q)))
EU  rewrites
  EU(N, (LAMBDA u: TRUE),
     NOT
       ((LAMBDA s: entering?(pc1(s)))
          IMPLIES NOT
                    nu(LAMBDA
                       (Q: pred[staterec]):
                         (NOT LAMBDA s: critical?(pc1(s)) AND
                                        EX(N, Q)))))
   to mu(LAMBDA
          (Q: pred[staterec]):
            (NOT
               ((LAMBDA s: entering?(pc1(s)))
                  IMPLIES NOT
                            nu(LAMBDA
                               (Q: pred[staterec]):
                                 (NOT LAMBDA s: critical?(pc1(s))
                                    AND EX(N, Q))))
               OR ((LAMBDA u: TRUE) AND EX(N, Q))))
EF  rewrites
  EF(N,
     NOT
```

```
      ((LAMBDA s: entering?(pc1(s)))
         IMPLIES NOT
                   nu(LAMBDA
                      (Q: pred[staterec]):
                        (NOT LAMBDA s: critical?(pc1(s)) AND
                                       EX(N, Q)))))
  to mu(LAMBDA
         (Q: pred[staterec]):
           (NOT
              ((LAMBDA s: entering?(pc1(s)))
                 IMPLIES NOT
                           nu(LAMBDA
                              (Q: pred[staterec]):
                                (NOT LAMBDA s: critical?(pc1(s))
                                   AND EX(N, Q))))
              OR ((LAMBDA u: TRUE) AND EX(N, Q))))
AG rewrites
  AG(N,
     (LAMBDA s: entering?(pc1(s)))
       IMPLIES NOT
                 nu(LAMBDA
                    (Q: pred[staterec]):
                      (NOT LAMBDA s: critical?(pc1(s)) AND EX(N, Q))))
  to NOT
       mu(LAMBDA
          (Q: pred[staterec]):
            (NOT
               ((LAMBDA s: entering?(pc1(s)))
                  IMPLIES NOT
                            nu(LAMBDA
                               (Q: pred[staterec]):
                                 (NOT LAMBDA s: critical?(pc1(s))
                                    AND EX(N, Q))))
               OR ((LAMBDA u: TRUE) AND EX(N, Q))))
NOT rewrites
  (NOT
     mu(LAMBDA
        (Q: pred[staterec]):
          (NOT
             ((LAMBDA s: entering?(pc1(s)))
                IMPLIES NOT
```

```
                          nu(LAMBDA
                             (Q: pred[staterec]):
                               (NOT LAMBDA s: critical?(pc1(s))
                                  AND EX(N, Q))))
              OR ((LAMBDA u: TRUE) AND EX(N, Q)))))(s!1)
   to NOT
        mu(LAMBDA
           (Q: pred[staterec]):
             (NOT
                ((LAMBDA s: entering?(pc1(s)))
                   IMPLIES NOT
                             nu(LAMBDA
                                (Q: pred[staterec]):
                                  (NOT LAMBDA s: critical?(pc1(s))
                                     AND EX(N, Q))))
                  OR ((LAMBDA u: TRUE) AND EX(N, Q))))(s!1)
nextp1  rewrites nextp1(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                              critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
nextp2  rewrites nextp2(u, v)
   to IF idle?(pc2(u)) THEN idle?(pc2(v)) OR
                                (entering?(pc2(v)) AND NOT try(v))
      ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
      ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
          OR (exiting?(pc2(v)) AND try(v))
      ELSE idle?(pc2(v))
      ENDIF
nextsem  rewrites nextsem(u, v)
   to IF
         (entering?(pc1(u))
            OR entering?(pc2(u)) OR exiting?(pc1(u)) OR
                                    exiting?(pc2(u)))
       THEN (entering?(pc1(u)) IMPLIES sem1(v))
         AND (entering?(pc2(u)) IMPLIES sem2(v))
               AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
```

```
                    AND (exiting?(pc2(u)) IMPLIES NOT sem2(v))
      ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
      ENDIF
N  rewrites N(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
        AND
        IF idle?(pc2(u)) THEN idle?(pc2(v))
            OR (entering?(pc2(v)) AND NOT try(v))
        ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
        ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
            OR (exiting?(pc2(v)) AND try(v))
        ELSE idle?(pc2(v))
        ENDIF
          AND
          IF
              (entering?(pc1(u))
                 OR entering?(pc2(u))
                      OR exiting?(pc1(u)) OR exiting?(pc2(u)))
            THEN (entering?(pc1(u)) IMPLIES sem1(v))
              AND (entering?(pc2(u)) IMPLIES sem2(v))
                    AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                          AND
                          (exiting?(pc2(u))
                             IMPLIES NOT sem2(v))
          ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
          ENDIF
nextp1  rewrites nextp1(u, v)
  to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                         critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
```

```
      ENDIF
nextp2  rewrites nextp2(u, v)
   to IF idle?(pc2(u)) THEN idle?(pc2(v)) OR
                                (entering?(pc2(v)) AND NOT try(v))
      ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
      ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
          OR (exiting?(pc2(v)) AND try(v))
      ELSE idle?(pc2(v))
      ENDIF
nextsem  rewrites nextsem(u, v)
   to IF
         (entering?(pc1(u))
            OR entering?(pc2(u)) OR exiting?(pc1(u)) OR
                                    exiting?(pc2(u)))
        THEN (entering?(pc1(u)) IMPLIES sem1(v))
          AND (entering?(pc2(u)) IMPLIES sem2(v))
                AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                      AND (exiting?(pc2(u)) IMPLIES NOT sem2(v))
      ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
      ENDIF
N  rewrites N(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
        AND
        IF idle?(pc2(u)) THEN idle?(pc2(v))
            OR (entering?(pc2(v)) AND NOT try(v))
        ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
        ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
            OR (exiting?(pc2(v)) AND try(v))
        ELSE idle?(pc2(v))
        ENDIF
          AND
          IF
              (entering?(pc1(u))
                 OR entering?(pc2(u))
```

```
                    OR exiting?(pc1(u)) OR exiting?(pc2(u)))
          THEN (entering?(pc1(u)) IMPLIES sem1(v))
            AND (entering?(pc2(u)) IMPLIES sem2(v))
                  AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                        AND
                        (exiting?(pc2(u))
                           IMPLIES NOT sem2(v))
        ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
        ENDIF
```

By rewriting and mu-simplifying,
this simplifies to:

```
not_live :

{-1}   TRUE
  |-------

Rule? q
```

Do you really want to quit? (Y or N): y

```
Run time  = 31.82 secs.
Real time = 59.33 secs.
>
```

The following proof illustrates the use of fairCTL operators to verify the property that there is at least one departing fair path from the initial state.

```
fair :

  |-------
{1}   (FORALL
       (s: staterec):
         (init(s) AND initsem(s))
           IMPLIES Fair?(N, LAMBDA s: critical?(pc1(s)))(s))
```

```
Running step: (Model-Check)
init  rewrites init(s!1)
   to idle?(pc1(s!1)) AND idle?(pc2(s!1))
initsem  rewrites initsem(s!1)
   to NOT sem1(s!1) AND NOT sem2(s!1)
fairEG  rewrites fairEG(N, (LAMBDA u: TRUE))(LAMBDA s:
                                                    critical?(pc1(s)))
   to nu(LAMBDA
          (Q: pred[staterec]):
            (LAMBDA u: TRUE)
              AND
              mu(LAMBDA
                 (P: pred[staterec]):
                   EX(N,
                      (LAMBDA u: TRUE)
                        AND
                        ((LAMBDA s: critical?(pc1(s)) AND Q)
                           OR P))))
Fair?  rewrites Fair?(N, LAMBDA s: critical?(pc1(s)))
   to nu(LAMBDA
          (Q: pred[staterec]):
            (LAMBDA u: TRUE)
              AND
              mu(LAMBDA
                 (P: pred[staterec]):
                   EX(N,
                      (LAMBDA u: TRUE)
                        AND
                        ((LAMBDA s: critical?(pc1(s)) AND Q)
                           OR P))))
nextp1  rewrites nextp1(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                          critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
nextp2  rewrites nextp2(u, v)
```

```
   to IF idle?(pc2(u)) THEN idle?(pc2(v)) OR
                                (entering?(pc2(v)) AND NOT try(v))
      ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
      ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
          OR (exiting?(pc2(v)) AND try(v))
      ELSE idle?(pc2(v))
      ENDIF
nextsem  rewrites nextsem(u, v)
   to IF
         (entering?(pc1(u))
            OR entering?(pc2(u)) OR exiting?(pc1(u)) OR
                                    exiting?(pc2(u)))
        THEN (entering?(pc1(u)) IMPLIES sem1(v))
          AND (entering?(pc2(u)) IMPLIES sem2(v))
                AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                      AND (exiting?(pc2(u)) IMPLIES NOT sem2(v))
      ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
      ENDIF
N  rewrites N(u, v)
   to IF idle?(pc1(u)) THEN idle?(pc1(v)) OR
                                (entering?(pc1(v)) AND try(v))
      ELSIF entering?(pc1(u)) AND try(u) AND NOT sem2(u) THEN
                                           critical?(pc1(v))
      ELSIF critical?(pc1(u)) THEN (critical?(pc1(v)) AND try(v))
          OR (exiting?(pc1(v)) AND NOT try(v))
      ELSE idle?(pc1(v))
      ENDIF
        AND
        IF idle?(pc2(u)) THEN idle?(pc2(v))
            OR (entering?(pc2(v)) AND NOT try(v))
        ELSIF entering?(pc2(u)) AND NOT sem1(u) THEN critical?(pc2(v))
        ELSIF critical?(pc2(u)) THEN (critical?(pc2(v)) AND NOT try(v))
            OR (exiting?(pc2(v)) AND try(v))
        ELSE idle?(pc2(v))
        ENDIF
          AND
          IF
              (entering?(pc1(u))
                 OR entering?(pc2(u))
                      OR exiting?(pc1(u)) OR exiting?(pc2(u)))
            THEN (entering?(pc1(u)) IMPLIES sem1(v))
```

```
            AND (entering?(pc2(u)) IMPLIES sem2(v))
                  AND (exiting?(pc1(u)) IMPLIES NOT sem1(v))
                        AND
                        (exiting?(pc2(u))
                           IMPLIES NOT sem2(v))
          ELSE (sem1(v) = sem1(u)) AND (sem2(v) = sem2(u))
          ENDIF
```

By rewriting and mu-simplifying,

```
Q.E.D.


Run time  = 12.84 secs.
Real time = 25.13 secs.
NIL
>
```

Appendix B

Definitions, Axioms and Theorems

B.1 Definitions

```
definitions : THEORY
  BEGIN

  port : TYPE
```

```
parray :
  TYPE  = [# size : nat, port_array : ARRAY[{i : nat | i < size} → port] #]

cnode :
  TYPE  =
    [# inports : parray,
       outport : port,
       intports : parray,
       condport : port,
       condit : pred[port],
       datarel : pred[[{p : parray | size(p) = size(inports)}, port]],
       orderrel : pred[[{p : parray | size(p) = size(inports)}, port]],
       intrel : pred[[parray, parray]] #]

node : TYPE  = {n : cnode | condit(n) = λ (p : port) : TRUE}

cn0, cn1 : VAR cnode

par, parr, par0, par1, par00, par11, par2, par3 : VAR parray

same_size(cn0, cn1) : boolean  = size(inports(cn0)) = size(inports(cn1))

same_size(par0, par1) : boolean  = size(par0) = size(par1)

refines(cn0, cn1) : bool

equates(cn0, cn1) : bool  = refines(cn0, cn1) ∧ refines(cn1, cn0)

sks(cn0, cn1) : boolean  = same_size(cn0, cn1) ∧ equates(cn0, cn1)

silimp : pred[[port, port]]

p0, p1, p2, p3, p4 : VAR port

i : VAR nat

silimpar(par1, (par2 : {par | same_size(par, par1)})) : boolean

sileqar(par1, (par2 : {par | same_size(par, par1)})) :
  boolean  = silimpar(par1, par2) ∧ silimpar(par2, par1)

sileq(p1, p2) : boolean  = silimp(p1, p2) ∧ silimp(p2, p1)

weight : TYPE

IMPORTING orders[weight]
```

$\leq$: pred[[weight, weight]]

dfe : [port, port $\rightarrow$ boolean]

w : [port, port $\rightarrow$ weight]

cn, cn2, cn3 : VAR cnode

n, n_0, n_1, n_2, n_3 : VAR node

p : VAR port

inport(cn, (i : {j : nat | j < size(inports(cn))})) :
  port = (port_array(inports(cn)))(i)

intport(cn, (i : {j : nat | j < size(intports(cn))})) :
  port = (port_array(intports(cn)))(i)

is_outport(p) : boolean = ($\exists$ cn : p = outport(cn))

is_inport(p) : boolean =
  ($\exists$ cn, (i : {j : nat | j < size(inports(cn))}) : p = inport(cn, i))

is_condport(p) : boolean = ($\exists$ cn : p = condport(cn))

xdfe(p_1, p_2) : boolean =
  dfe(p_1, p_2) $\wedge$ ($\forall$ p : ($p \neq p_1$) $\supset$ $\neg$ dfe(p, p_2))

war : [parray, parray $\rightarrow$ weight]

dfear(par0, (par1 : {par | same_size(par, par0)})) : boolean =
  ($\forall$ (i : nat | i < size(par0)) :
    dfe(port_array(par0)(i), port_array(par1)(i)))

xdfear(par0, (par1 : {par | same_size(par, par0)})) : boolean =
  ($\forall$ (i : nat | i < size(par0)) :
    xdfe(port_array(par0)(i), port_array(par1)(i)))

is_node_outport(p) : boolean = $\exists$ n : p = outport(n)

is_outportar(par) : boolean =
  $\forall$ (i : nat | i < size(par)) : is_node_outport(port_array(par)(i))

is_cnode_outport(p) : boolean = $\exists$ (cn : cnode) : p = outport(cn)

```
is_cnoutportar(par) : boolean  =
  ∀ (i : nat | i < size(par)) : is_cnode_outport(port_array(par)(i))

END definitions
```

B.2 Axioms

```
axioms : THEORY
  BEGIN

  IMPORTING definitions

  cn, cn0, cn1, cn2, cn3 : VAR cnode

  cnode_ax : AXIOM
    ∀ cn :
      condit(cn)(condport(cn))
        ⊃ datarel(cn)(inports(cn), outport(cn))

  par, par0, par1, par2, par3 : VAR parray

  i : VAR nat

  p, p0, p1, p2, p3, p4 : VAR port

  silimpar_ax : AXIOM
    ∀ par1, (par2 : parray | same_size(par1, par2)) :
      ∀ (i : nat | i < size(par1)) :
        silimp(port_array(par1)(i), port_array(par2)(i))
          ⊃ silimpar(par1, par2)

  sileq_refl_ax : AXIOM sileq(p1, p1)

  sileq_sym_ax : AXIOM sileq(p1, p2) = sileq(p2, p1)

  sileq_trans_ax : AXIOM
    ∀ p0, p1, p2 :
      (sileq(p0, p1) ∧ sileq(p1, p2) ⊃ sileq(p0, p2))

  refinement_ax : AXIOM
    ∀ (n0 : node), (n1 : node | same_size(n0, n1)) :
      refines(n0, n1) ∧ silimpar(inports(n0), inports(n1))
        ⊃ silimp(outport(n0), outport(n1))

  edge_ax : AXIOM partial_order?(≤ : pred[[weight, weight]])
```

n, n_0, n_1, n_2, n_3 : VAR node

dfe_port_ax1 : AXIOM $\text{dfe}(p_1, p_2) \supset \text{is\_outport}(p_1)$

dfe_port_ax2 : AXIOM $\text{dfe}(p_1, p_2) \supset (\text{is\_inport}(p_2) \vee \text{is\_condport}(p_2))$

self_edge_not_ax : AXIOM $\neg\ \text{dfe}(p, p)$

dfe_w_ax : AXIOM
$\forall\ p_0, p_1, p_2 :$
$p_0 \neq p_1$
$\supset$
$((\text{dfe}(p_0, p_2) \wedge \text{dfe}(p_1, p_2))$
$\supset$
$(w(p_0, p_2) \leq w(p_1, p_2)$
$\vee\ w(p_1, p_2) \leq w(p_0, p_2)))$

cond_bottom_ax : AXIOM
$\neg\ (\text{condit(cn)(condport(cn)))}$
$\supset$
$(\forall\ p :$
$\text{dfe(outport(cn)}, p)$
$\supset$
$(\forall\ p_0 :$
$\text{dfe}(p_1, p)$
$\supset\ w(\text{outport(cn)}, p_0) \leq w(p_1, p_0)))$

join_ax : AXIOM
$(\text{dfe}(p_1, p_2) \wedge (\forall\ p :\ \text{dfe}(p, p_2) \supset w(p, p_2) \leq w(p_1, p_2)))$
$\supset\ \text{sileq}(p_1, p_2)$

distrb_ax : AXIOM $\text{xdfe}(p, p_1) \wedge \text{xdfe}(p, p_2) \supset \text{sileq}(p_1, p_2)$

$p_{00}, p_{11}, p_{22}, p_{33}, p_{44}$: VAR port

po_preserve_ax : AXIOM
$(w(p_0, p_2) \leq w(p_1, p_2)$
$\wedge\ \text{sileq}(p_0, p_{00})$
$\wedge\ \text{sileq}(p_1, p_{11})$
$\wedge\ \text{sileq}(p_2, p_{22})$
$\wedge\ \text{dfe}(p_{00}, p_{22}) \wedge \text{dfe}(p_{11}, p_{22}))$
$\supset\ w(p_{00}, p_{22}) \leq w(p_{11}, p_{22})$

joinar_ax : AXIOM
$\forall\ \text{par1}, (\text{par2} :\ \text{parray}\ \mid\ \text{same\_size(par2, par1)}) :$
$(\text{dfear(par1, par2)}$

```
        ∧
       (∀ (par : parray | same_size(par, par1)) :
         dfear(par, par2)
            ⊃ war(par, par2) ≤ war(par1, par2)))
       ⊃ sileqar(par1, par2)

dfe_war_ax : AXIOM
  ∀ par0,
     (par1 : parray | same_size(par1, par0)),
     (par2 : parray | same_size(par2, par0)) :
    (dfear(par0, par2) ∧ dfear(par1, par2))
      ⇔
     (war(par0, par2) ≤ war(par1, par2)
        ∨ war(par1, par2) ≤ war(par0, par2))

END axioms
```

B.3 Theorems

```
theorems : THEORY
  BEGIN

  IMPORTING axioms

  p0, p1, p2, p3, p4 : VAR port

  par, parr, par0, par1, par00, par11, par2, par3 : VAR parray

  n, n0, n1, n2, n3 : VAR node

  node_data_f_th : THEOREM ∀ n : datarel(n)(inports(n), outport(n))

  sileq_trans_inv_th : THEOREM
    ∀ p0, p1, p2 :
     (sileq(p0, p1) ∧ sileq(p0, p2) ⊃ sileq(p1, p2))

  i : VAR nat

  sileqar(par1, (par2 : {par | same_size(par, par1)})) : boolean =
   ∀ (i | i < size(par1)) :
    sileq(port_array(par1)(i), port_array(par2)(i))

  sileqar_refl_th : THEOREM sileqar(par1, par1)

  sileqar_sym_th : THEOREM
    ∀ par1, (par2 : {par | same_size(par, par1)}) :
     sileqar(par1, par2) = sileqar(par2, par1)

  sileqar_trans_th : THEOREM
    ∀ par1,
      (par2 : {par | same_size(par, par1)}),
      (par3 : {par | same_size(par, par1)}) :
     (sileqar(par1, par2) ∧ sileqar(par2, par3))
       ⊃ sileqar(par1, par3)

  sileqar_trans_inv_th : THEOREM
    ∀ par1,
      (par2 : {par | same_size(par, par1)}),
```

```
      (par3 : {par | same_size(par, par1)}) :
    (sileqar(par1, par2) ∧ sileqar(par1, par3))
      ⊃ sileqar(par2, par3)

in_eqar_imp_outeq : THEOREM
  ∀ (n0 : node), (n1 : node | sks(n0, n1)) :
    sileqar(inports(n0), inports(n1))
      ⊃ sileq(outport(n0), outport(n1))

cn, cn0, cn1, cn2, cn3 : VAR cnode

xdfe_sileq_th : THEOREM xdfe(outport(n1), p2) ⊃ sileq(outport(n1), p2)

xdfe2_sileq_th : THEOREM
  (xdfe(outport(n0), p1)
     ∧ xdfe(outport(n2), p3) ∧ sileq(outport(n0), outport(n2)))
    ⊃ sileq(p1, p3)

p00, p11, p22, p33, p44 : VAR port

po_preserve_xdfe_th : THEOREM
  (w(p0, p2) ≤ w(p1, p2)
     ∧ sileq(p2, outport(n3))
         ∧ xdfe(outport(n3), p4)
             ∧ dfe(p00, p44)
                 ∧ dfe(p11, p44)
                     ∧ sileq(p0, p00)
                         ∧ sileq(p1, p11)
                             ∧ sileq(p4, p44))
    ⊃ w(p00, p44) ≤ w(p11, p44)

dfe2_join_th : THEOREM
  (dfe(p1, p3)
     ∧ dfe(p2, p3)
       ∧
       (∀ p0 :
          ((p0 ≠ p1) ∨ (p0 ≠ p2)) ⊃ ¬ dfe(p0, p3)))
    ⊃ IF w(p1, p3) ≤ w(p2, p3) THEN sileq(p2, p3)
      ELSE sileq(p1, p3)
      ENDIF

inports_sileqar_th : THEOREM
  ∀ (nd : node) :
    ∀ (n0 : node | same_size(n0, nd)),
    (n1 : node | same_size(n1, nd)) :
      ((∀ (i | i < size(inports(nd))), n :
```

```
        xdfe(outport(n), inport(n0, i))
            ⇔
          (∃ (nn : node) :
            sileq(outport(n), outport(nn))
              ∧ xdfe(outport(nn), inport(n1, i))))
        ∧
        (∀ (i | i < size(inports(nd))) :
          ∃ n : xdfe(outport(n), inport(n0, i)))
         ⊃ sileqar(inports(n0), inports(n1)))

inports_eqar_th : THEOREM
  ∀ (n0 : node), (n1 : node | same_size(n0, n1)) :
   ((∀ (i | i < size(inports(n0))), n :
      xdfe(outport(n), inport(n0, i))
        ⇔ xdfe(outport(n), inport(n1, i)))
      ∧
      (∀ (i | i < size(inports(n0))) :
        ∃ n : xdfe(outport(n), inport(n0, i)))
       ⊃ sileqar(inports(n0), inports(n1)))

xdfear_sileqar_th : THEOREM
  ∀ (par0 | is_outportar(par0)), (par1 | same_size(par1, par0)) :
   xdfear(par0, par1) ⊃ sileqar(par0, par1)

inportsar_eqar_th : THEOREM
  ∀ (n0 : node), (n1 : node | same_size(n0, n1)) :
   (((∀ (par | is_outportar(par) ∧ same_size(par, inports(n0))) :
       xdfear(par, inports(n0)) ⇔ xdfear(par, inports(n1)))
      ∧
      (∃ (par
            |
          is_outportar(par)
            ∧ same_size(par, inports(n0))) :
        xdfear(par, inports(n0))))
     ⊃ sileqar(inports(n0), inports(n1)))

inportsar_sileqar_th : THEOREM
  ∀ (n0 : node), (n1 : node | same_size(n0, n1)) :
   (((∀ (par | is_outportar(par) ∧ same_size(par, inports(n0))) :
       xdfear(par, inports(n0))
          ⇔
        (∃ (parr
              |
            is_outportar(par)
              ∧ same_size(parr, inports(n0))) :
          (sileqar(par, parr)
```

```
                ∧ xdfear(parr, inports(n1)))))
        ∧
        ∃ (par
            |
           is_outportar(par) ∧ same_size(par, inports(n0))) :
          xdfear(par, inports(n0)))
        ⊃ sileqar(inports(n0), inports(n1)))

dfear2_join_th : THEOREM
   ∀ par1,
      (par2 | same_size(par2, par1)), (par3 | same_size(par3, par1)) :
     (dfear(par1, par3) ∧ dfear(par2, par3))
       ∧
      (∀ (par | same_size(par, par1)) :
        (par ≠ par1 ∨ par ≠ par2) ⊃ ¬ dfear(par, par3))
       ⊃ IF war(par1, par3)
                ≤ war(par2, par3) THEN sileqar(par2, par3)
           ELSE sileqar(par1, par3)
           ENDIF

dot, dot0, dot1, dot00, dot01 : VAR node

preconds(dot0, (dot1 : {dot1 | equates(dot0, dot1)})) : boolean =
  (∀ (par | is_outportar(par) ∧ same_size(par, inports(dot0))) :
    xdfear(par, inports(dot0)) ⇔ xdfear(par, inports(dot1)))
    ∧
   (∃ (par | is_outportar(par) ∧ same_size(par, inports(dot0))) :
     xdfear(par, inports(dot0)))

CSubE : THEOREM
   ∀ dot0,
      (dot1 | equates(dot1, dot0)), (dot01 | equates(dot01, dot0)) :
     ((preconds(dot0, dot1)
         ∧
        (∀ (par
             |
            is_outportar(par)
               ∧ same_size(par, inports(dot0))) :
           xdfear(par, inports(dot0))
              ⇔
            (∃ (parr
                 |
                is_outportar(parr)
                   ∧ same_size(parr, inports(dot0))) :
              (sileqar(par, parr)
                 ∧ xdfear(parr, inports(dot01))))))
```

```
          ⊃
        (∀ p1, p2 :
          ((xdfe(outport(dot0), p1) ∨ xdfe(outport(dot1), p1))
             ∧ xdfe(outport(dot01), p2))
            ⊃ sileq(p1, p2)))

  pp, pp0, pp00 : VAR port

  cn22, cn33 : VAR cnode

  preconds(dot0, (dot1 : {dot | sks(dot, dot0)}),
           (dot00 : {dot | sks(dot, dot0)}),
           (par0 : {par | is_outportar(par) ∧ same_size(par, inports(dot0))}),
           (par1 : {par | is_outportar(par) ∧ same_size(par, inports(dot0))}),
           (par00 : {par | is_outportar(par) ∧ same_size(par, inports(dot0))}),
           (par11 : {par | is_outportar(par) ∧ same_size(par, inports(dot0))}),
           pp0, pp00) :
    boolean =
    xdfear(par0, inports(dot0))
      ∧ xdfear(par1, inports(dot1))
          ∧
         (war(par0, inports(dot0)) ≤ war(par1, inports(dot1))
            = w(outport(dot0), pp0) ≤ w(outport(dot1), pp0))
           ∧
          (war(par0, inports(dot0)) ≤ war(par1, inports(dot1))
             = war(par00, inports(dot00))
                 ≤ war(par11, inports(dot00)))
            ∧ dfe(outport(dot0), pp0)
                ∧ dfe(outport(dot1), pp0)
                    ∧
                   (∀ pp :
                     ((pp ≠ outport(dot0))
                        ∨ (pp ≠ outport(dot1)))
                       ⊃ ¬ dfe(pp, pp0))
                    ∧
                    dfear(par00,
                          inports(dot00))
 ∧
 dfear(par11,
       inports(dot00))
∧
(∀ (par
    |
   size(par)
     = size(par00)) :
  (par ≠ par00
```

```
    ∨ par  ≠ par11)
   ⊃ ¬
       dfear(par,
             inports(dot00)))
∧
xdfe
 (outport(dot00),
  pp00)
 ∧
 sileqar(par0,
         par00)
∧
sileqar(par1,
        par11)

 CjtM : THEOREM
    ∀ dot0 :
     LET sk  = λ n : sks(n, dot0),
         ios  = λ par :
                  is_outportar(par) ∧ same_size(par, inports(dot0))
      IN
       ∀ (dot1 | sk(dot1)),
       (dot00 | sk(dot00)),
       (par0 | ios(par0)),
       (par1 | ios(par1)), (par00 | ios(par00)), (par11 | ios(par11)) :
        preconds(dot0, dot1, dot00, par0,
                 par1, par00, par11, pp0, pp00)
           ⊃ sileq(pp0, pp00)

END theorems
```

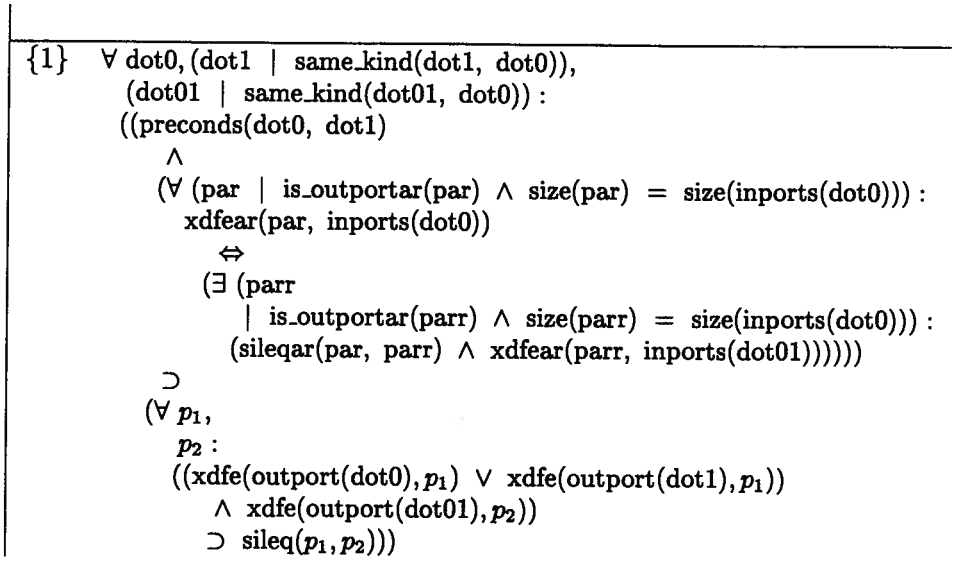
Appendix C

Proof Transcripts

C.1 Common Subexpression Elimination

Terse proof for `CSubE`.

CSubE:



Expanding the definition of preconds,

For the top quantifier in 1, we introduce Skolem constants: (dot0!1 dot1!1 dot01!1),

Applying disjunctive simplification to flatten sequent,

Applying inportsar_eqar_th where n_0 gets dot0!1, n_1 gets dot1!1,

Replacing using formula -2,

Replacing using formula -3,

Invoking decision procedures,

Applying inportsar_sileqar_th where n_0 gets dot0!1, n_1 gets dot01!1,

Replacing using formula -5,

Replacing using formula -4,

Invoking decision procedures,

Deleting some formulas,

For the top quantifier in 1, we introduce Skolem constants: $(p_1' \ p_2')$,

Applying sileqar_trans_inv_th where par1 gets inports(dot0!1), par2 gets inports(dot1!1), par3 gets inports(dot01!1),

Invoking decision procedures,

Applying in_eqar_imp_outeq where n_0 gets dot0!1, n_1 gets dot01!1,

Applying in_eqar_imp_outeq where n_0 gets dot1!1, n_1 gets dot01!1,

Applying xdfe_sileq_th where

Instantiating quantified variables,

Instantiating quantified variables,

Instantiating quantified variables,

Invoking decision procedures,

Deleting some formulas,

Applying sileq_trans_inv_th where

Instantiating the top quantifier in -1 with the terms: outport(dot0!1), p_1', outport(dot01!1),

Instantiating the top quantifier in -1 with the terms: outport(dot1!1), p_1', outport(dot01!1),

Applying sileq_trans_ax where

Instantiating the top quantifier in -1 with the terms: p_1', outport(dot01!1), p_2',

Applying bddsimp, which is trivially true. This completes the proof of **CSubE**.

Q.E.D.

C.2 Cross Jumping Tail Merging

Terse proof for `CjtM`.

`CjtM:`

$$
\begin{array}{|l}
\hline
\{1\}\quad (\forall\ (\mathrm{pp0},\ \mathrm{pp00} :\ \mathrm{port}) : \\
\qquad \forall\ (\mathrm{dot0} :\ \mathrm{node}) : \\
\qquad\quad \textsc{let}\ \mathrm{sk} :\ [\mathrm{node} \rightarrow \mathrm{bool}]\ = \\
\qquad\qquad\quad \lambda\ (n :\ \mathrm{node}) :\ \mathrm{same\_size}(n,\ \mathrm{dot0})\ \wedge\ \mathrm{same\_kind}(n,\ \mathrm{dot0}), \\
\qquad\qquad\ \mathrm{ios} :\ [\mathrm{parray} \rightarrow \mathrm{bool}]\ = \\
\qquad\qquad\quad \lambda\ (\mathrm{par} :\ \mathrm{parray}) :\ \mathrm{is\_outportar}(\mathrm{par}) \\
\qquad\qquad\qquad \wedge\ \mathrm{size}(\mathrm{par})\ =\ \mathrm{size}(\mathrm{inports}(\mathrm{dot0})) \\
\qquad\quad \textsc{in}\ \forall\ (\mathrm{dot1} :\ \mathrm{node}\ \mid\ \mathrm{sk}(\mathrm{dot1})), (\mathrm{dot01} :\ \mathrm{node}\ \mid\ \mathrm{sk}(\mathrm{dot01})), \\
\qquad\qquad\quad (\mathrm{par0} :\ \mathrm{parray}\ \mid\ \mathrm{ios}(\mathrm{par0})), (\mathrm{par1} :\ \mathrm{parray}\ \mid\ \mathrm{ios}(\mathrm{par1})), \\
\qquad\qquad\quad (\mathrm{par00} :\ \mathrm{parray}\ \mid\ \mathrm{ios}(\mathrm{par00})), \\
\qquad\qquad\quad (\mathrm{par11} :\ \mathrm{parray}\ \mid\ \mathrm{ios}(\mathrm{par11})) : \\
\qquad\quad \mathrm{preconds}(\mathrm{dot0},\ \mathrm{dot1},\ \mathrm{dot01},\ \mathrm{par0},\ \mathrm{par1},\ \mathrm{par00},\ \mathrm{par11},\ \mathrm{pp0},\ \mathrm{pp00}) \\
\qquad\ \supset\ \mathrm{sileq}(\mathrm{pp0},\ \mathrm{pp00}))
\end{array}
$$

Expanding the definition of preconds,

For the top quantifier in 1, we introduce Skolem constants: (pp0!1 pp00!1),

For the top quantifier in 1, we introduce Skolem constants: (dot0!1),

For the top quantifier in 1, we introduce Skolem constants: (dot1!1 dot01!1 par0!1 par1!1 par00!1 par11!1),

Applying disjunctive simplification to flatten sequent,

Applying xdfear_sileqar_th where

Instantiating quantified variables,

Instantiating quantified variables,

Applying dfe2_join_th where

Instantiating the top quantifier in -1 with the terms: outport(dot0!1), outport(dot1!1), pp0!1,

Replacing using formula -8,

Replacing using formula -9,

Replacing using formula -10,

Applying dfear2_join_th where

Instantiating the top quantifier in -1 with the terms: par00!1, par11!1, inports(dot01!1),

Replacing using formula -12,

Replacing using formula -13,

Replacing using formula -14,

Letting war01 name war(par0!1, inports(dot0!1)) $\leq$ war(par1!1, inports(dot1!1)),

Letting war001 name war(par00!1, inports(dot01!1)) $\leq$ war(par11!1, inports(dot01!1)),

Letting w_{01} name w(outport(dot0!1), pp0!1) $\leq$ w(outport(dot1!1), pp0!1),

Replacing using formula -1,

Hiding formulas: -1,

Replacing using formula -1,

Hiding formulas: -1,

Replacing using formula -1,

Hiding formulas: -1,

Invoking decision procedures,

Deleting some formulas,

Deleting some formulas,

Replacing using formula -6,

Replacing using formula -5,

Hiding formulas: -5, -6,

Applying sileqar_trans_inv_th where

Instantiating the top quantifier in -1 with the terms: par0!1, inports(dot0!1), par00!1,

Instantiating the top quantifier in -1 with the terms: par1!1, inports(dot1!1), par11!1,

Applying in_eqar_imp_outeq where

Instantiating the top quantifier in -1 with the terms: dot0!1, dot01!1,

Instantiating the top quantifier in -1 with the terms: dot1!1, dot01!1,

Applying xdfe_sileq_th where

Instantiating quantified variables,

Invoking decision procedures,

Deleting some formulas,

Applying sileqar_trans_th where

Instantiating the top quantifier in -1 with the terms: inports(dot1!1), par11!1, inports(dot01!1),

Instantiating the top quantifier in -1 with the terms: inports(dot0!1), par00!1, inports(dot01!1),

Invoking decision procedures,

Deleting some formulas,

Applying sileq_trans_inv_th where

Instantiating the top quantifier in -1 with the terms: outport(dot0!1), pp0!1, outport(dot01!1),

Instantiating the top quantifier in -1 with the terms: outport(dot1!1), pp0!1, outport(dot01!1),

Applying sileq_trans_ax where

Instantiating the top quantifier in -1 with the terms: pp0!1, outport(dot01!1), pp00!1,

Applying bddsimp,

which is trivially true.

This completes the proof of `CjtM`.

Q.E.D.