Transformations in High-Level Synthesis: Formal Specification and Efficient Mechanical Verification

P. Sreeranga Rajan
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

sree@csl.sri.com
Phone: +1 (415) 859-2873 Fax: +1 (415) 859-2844

October 1994

Abstract

Dependency graphs are used to model data and control flow in hardware and software design. In high-level synthesis of hardware, optimization and refinement transformations are used to transform dependency-graph-based specifications at the behavior level to dependency-graph-based implementations at the register-transfer level. Register-transfer-level implementations are mapped to gate-level hardware designs by low-level logic synthesis. In this work, we investigated the specification and mechanical verification of the correctness of transformations used in high-level synthesis of hardware.

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. The specification and verification schemes are general enough for applications in other synthesis frameworks and software design, where a transformational design approach is used.

In order to present our work in a concrete context, we focus on the high-level synthesis part of the SPRITE project at Philips Research Laboratories. The transformations in the high-level synthesis system are used for refinement and optimization of descriptions specified in a dependency graph language called the 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.

We used the Prototype Verification System (PVS) from SRI International to specify and mechanically verify the correctness of the transformations. The PVS specification language allows us to investigate the correctness problem using a convenient level of representation. The PVS verifier features automatic procedures and interactive verification rules to check properties of specifications.

Contents

List of Figures

List of Tables

Acknowledgments

A major part of the work presented in this report was done at Philips Research Laboratories, Eindhoven, The Netherlands, from September 1993 through April 1994. I thank Ton Kostelijk for the invitation to work on the project, and for providing illuminating suggestions, support and a homelike environment. I am grateful to Corrie Huijs, Wim Kloosterhuis, Thijs Krol, Jaap Hofstede, Peter Middelhoek, and Wim Smits for their cooperation, review, and corrections. Thanks to group leader Gerard Beenker for arranging a pleasant stay in Eindhoven and providing constant support for the project, and appreciation to the group members for a lively and stimulating atmosphere. Thanks to Iskender Agi, Mark Moriconi, Peter Neumann, Sam Owre, John Rushby, and N. Shankar for comments and suggestions, and M.K. Srivas for providing in-depth corrections and remarks. Thanks to Jens Ulrik Skakkebaek of TU Denmark, Jozef Hooman and Geert Janssen of TU Eindhoven, and Paul Miner of NASA for remarks and interesting discussions related to this work. I am grateful to Jeff Joyce of UBC for the encouragement, and for suggesting applications of the work in software engineering. Thanks to Paul Gilmore for detailed observations, and to Alan Mackworth and Mabo Ito of UBC for insightful remarks.

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. 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 designs by low-level logic synthesis. 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. An informal representation 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. 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 the model of behavior underlying dependency graphs.

A typical transformation employed in high-level synthesis 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 the formal approach explained in this paper, the transformation does not preserve behavior. Informally, the reason is as follows. In graph G1,

[1]In literature, they are also known as control-flow/data-flow graphs and signal-flow graphs.
[2]Usually known as input/output behavior.

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

when `c` is false, the value of `q0` is arbitrary, and so is the value of `p0`. If we choose the value of `pp0` to be that of `p0`, the value of `pp0` is also arbitrary. In graph G2, when `c` is false, we could choose the value of `p01` to be that of `q11`. In this case, the value of `pp00` is `(y1 * y2)`. Because the corresponding outputs could be unequal with identical inputs, the behaviors of the graphs are not equivalent. A corrected and generalized cross-jumping tail-merging transformation is presented in Chapter 6.

The main contributions of this work are the following:

- A formal specification of dependency graphs has been achieved.
- A set of optimization and refinement transformations on dependency graphs used in high level synthesis 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 were possible before.

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

Formal methods could be divided into two main categories: property-oriented methods and model-oriented methods [JMW 90]. 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 *axioms*, are asserted to be true, while other properties, called *theorems*, 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. As we discuss later in Chapter 7, 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.

Dependency graph[3] 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 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 [VHD 88], SILAGE [Hil 85], and ELLA [ELL 90]. 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

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

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 [WMM 94, Mid 94-2]. In one of the applications [Mid 94], 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 [Klo 92, Kro 92]. A detailed denotational semantics of SIL for showing the correctness of transformations has been worked out earlier [HHK 92, HuK 94]. The optimization and refinement transformations have been specified informally as part of the SPRITE project [EMH 93, Mid 93, Mid 94].

We use the Prototype Verification System (PVS) [OSR 93], an environment for formal specification and verification. The PVS specification language, based on typed higher-order logic, permits an axiomatic method to develop specifications. This method 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 dependency graphs. For example, a behavior model could be based on behavior expressions [McP 83], an imperative semantics [Cam 89], a denotational model [GGJ 93, HuK 94], or an operational model [GGJ 93]. In the axiomatic framework we discuss in this report, we can compare descriptions with respect to their behavior, and thus establish correctness of transformations, without specifying a behavioral model of a SIL description. 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 report is organized as follows: Chapter 2 gives an overview of SIL. In Chapter 3, we give a brief description of the PVS system. In Chapter 4, we describe the specification of structure of SIL graphs, while in Chapter 5 we describe the specification of behavior, refinement, and equivalence of SIL graphs. We present the specification and verification of transformations in Chapter 6. In that chapter, we also illustrate how our generalization and composition of transformations leads to new transformations for further optimization and refinement than would have been possible before. Finally, following a general discussion, conclusions are summarized in Chapter 7. A listing of the specification of SIL and its verified properties as it appears in PVS is given in Appendix A. Transcripts of the verification in PVS for two transformations discussed

Figure 1.3: SIL transformations and verification in PVS in the context of high level synthesis.

in detail in this paper are listed in Appendix B. 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. However, few have dealt with transformations on dependency graphs in general. Most of the efforts have concentrated on specialized hardware description languages and programming languages.

A formal model was proposed for verifying correctness of high-level transformations by McFarland and Parker [McP 83]. Transformations used in YIF (Yorktown Internal Form) [YIF 88] have been proved to be behavior preserving [Cam 89]. In this work, a strong notion of behavior equivalence based on an imperative semantics tied to a particular model of representation is used. A formal system using transformations for hardware synthesis has been discussed by Fourman [Fou 90]. 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 [Jon 90], and its formalization has been presented by Rossen [Ros 90]. This effort is explained briefly in Section 1.1.2. Another algebraic approach to transformational design of hardware has been worked out by Johnson [Joh 94]. A short discussion on this approach is presented in Section 1.1.3. In the work on tying formal verification to silicon compilation [JRS 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 [Vem 90].

An automatic method for functional verification of retiming, pipelining and buffering optimization has been presented by Kostelijk [KoW 93]. It has been implemented in a CAD tool called RetLab as part of PHIDEO at PRL. A formal analysis of transformations used in Systems Architect Workbench (SAW) high-level synthesis was studied by McFarland [McF 93]. 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 [AAD 93, Ael 94]. Denotational and operational models of generalized data-flow graphs have been developed, but they have not been used to study the correctness of transformations [GGJ 93]. A formalization of SILAGE transformations in HOL was studied by Angelo [Ang 94]. 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 [CBL 92] 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 [AaL 94], but it does not provide a mechanical verification for transformations in high-level synthesis. A brief discussion of the work on verification of transformations in synchronized transitions [Sta 90] is given in Section 1.1.6. In Section 1.1.7, we briefly discuss the work on formal specification and verification of refinement transformation in software design.

1.1.1 LAMBDA

LAMBDA [Fou 90] 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 [Jon 90], based on an algebraic formalism. The algebraic formalism consists of relations and operations on relations such as composition, inverse 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 to specify 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 [Ros 90] in a proof checking system called ISABELLE. ISABELLE, based on type theory, allows syntactic embedding other logics. A fragment of Ruby corresponding to combinational circuits, delay element, serial composition and parallel composition called Pure Ruby is specified as a type. Properties and proof rules such as induction on Ruby terms is 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 [Joh 94]. In this formalism, a functional specification is translated into a representation of a Deterministic Finite State Machine specification called behavior tables [RTJ 93]. 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 [McF 93] used in System Architect's Workbench (SAW) [Tho 98] is carried out. In this system, hardware described at the register-transfer level or higher using ISPB [Bar 81] 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 [Hil 85] is an applicative hardware description language. This language 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 [Ang 94], the syntax and semantics of SILAGE programs have been formalized as predicates in HOL [GoM 93]. 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 Synchronized Transitions in LP

Synchronization Transitions (ST) [Sta 90] is a formalism to specify states and transitions between states. It is based on UNITY [UNI 88] model of computation as a collection of atomic conditional assignments to state variables without explicit flow of control. The transitions are specified by guarded commands. State variables model storage and sharing of state variables model communication. This is unlike message passing in CSP [Hoa 85] formalism and token passing in SIL. There is no concept of clocks and sequencing. The temporal behavior is determined by guards. The formalism is geared towards direct realizations in synchronous and asynchronous circuits. The optimization and refinement transformations are not defined in the language. The conditions to be satisfied by an abstraction function, mapping a concrete state set to an abstract state set have been presented.

The specification that an ST program has to satisfy can be described as an invariant. An ST program could then be directly translated into Larch Prover (LP) [GaG 89], and invariants translated as proof obligations to be discharged. LP is a rewrite rule prover based on first order equational logic. Thus, an ST program can be both directly translated to LP and verified, and realized in hardware through synthesis.

1.1.7 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 [Bac 88] for specifications based on Dijkstra's guarded command language and weakest precondition semantics has been formalized in HOL [WrS 91]. 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 [Hoo 94]. 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 [Kid 90] 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 [HuK 94]. 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

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*:

Figure 2.1: Different kinds of SIL ports.

Figure 2.2: An example of a SIL graph description.

- 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

Figure 2.3: SIL node: informal description.

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 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 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) [Hil 85]. 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 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 can assume well-defined data values are selected. Then, among those 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 an 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 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 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

Figure 2.4: SIL edges: informal description.

Figure 2.5: SIL Join and Distribute: informal description.

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

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

Figure 2.8: Cumulative adder: unfolded SIL graph.

Figure 2.9: Partial specification of a multiplexor.

Figure 2.10: Implementation specification of a multiplexor.

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`.

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 compiler optimization techniques such as *Common Subexpression Elimination*, *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 [Mid 94]. We show a retiming transformation example in Figure 2.11

Figure 2.11: Example SIL transformation: retiming.

Chapter 3

Specification and Verification in PVS

The Prototype Verification System (PVS) [OSR 93, SOR 93-2] 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 [Cyr 93, CRS 94], protocol verification, and algorithm verification [LOR 93]. We briefly give the features of the PVS specification language in Section 3.1, the PVS verification features in Section 3.2 and some notes on 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 [OSR 93] 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 [SOR 93-2] 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) [BRB 90, Jan 93] 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 [Cyr 93, CRS 94].

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}
```

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

The symbol | has to be 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".

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 unspecified 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*

```
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:

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

```
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 steps that leads 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 [SOR 93-1]. 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 a 3 cent stamp and a 5 cent stamp 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!2 k!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

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 4.1, followed by the specification of edges in Section 4.2 and nodes and SIL graphs in Section 4.3. Finally, in Section 4.4 we establish the properties that need to hold for a SIL graph to be well-formed, and thus have a proper behavior.

4.1 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 *unspecified 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
```

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

Table 4.1: .
PVS types for data-flow edge and sequence edge; see Figure 4.1.

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]
               #]
```

4.2 Edges

An *edge* is a directed line connecting two *port*s. Mathematically, it is a relation on two *port*s. 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 ports. A *true* value of the predicate indicates the 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 4.1.

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

Figure 4.1: SIL data-flow and sequence edges; see Table 4.1.

```
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 4.1.

4.3 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 multplexers 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 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 port*s 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.

```
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 4.2: PVS specification of conditional node as a record type; see Figure 4.2.

In our specification, we first introduce a conditional node in PVS as a *record type* as shown in Table 4.2, 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 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 internally. 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 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.

Figure 4.2: SIL conditional node; see Table 4.2.

- *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 4.2.

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 4.3 by using this PVS subtyping feature. The Figure 4.3 illustrates this specification.

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:

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

Table 4.3: Node as a subtype of a conditional node; see Figure 4.3.

Figure 4.3: Node as a subtype of a conditional node; see Table 4.3.

```
graph: TYPE = cnode
```

4.4 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 5, 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 5

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 5.1, followed by a presentation of refinement and equivalence in Section 5.2.

5.1 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 [HuK 94]. 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 can 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 7.

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.

5.2 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 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 5.1. 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.

Figure 5.1: Example: refinement of ports due to non-deterministic choice.

Figure 5.2: Example: array refinement does not imply every individual port refinement.

```
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 5.2. 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 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*. In terms of the token flow concept, we recall from Chapter 2, that the ordering depends on the which of the tokens fired from the source ports determines the value of the 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[1] $<$ on weights, and assert the fact that the weights are ordered if and only if the associated data-flow edges form a join. We give the PVS specification of this property in Table 5.1 and illustrate it in Figure 5.3.

[1]We do not use the usual notation $\leq$ to stress that no two weights on different edges forming a join can be equal.

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

dfe_w_ax: AXIOM
p0 /= p1 IMPLIES
   dfe(p0,p2) AND dfe(p1,p2)
              IFF
  (w(p0,p2) < w(p1,p2) OR
   w(p1,p2) < w(p0,p2))
```

Table 5.1: Using weights for ordering data-flow edges: PVS specification; see Figure 5.3.

Figure 5.3: Using weights for ordering data-flow edges; see Table 5.1.

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

Table 5.2: Using weights to determine *join* behavior; see Figure 5.4.

Figure 5.4: Using weights to determine *join* behavior; see Table 5.2.

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. 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 5.2. 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 5.4.

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

```
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 5.3: Weight when the condition on a conditional node is false; see Figure 5.5.

Figure 5.5: Weight when the condition on a conditional node is false; see Table 5.3.

Figure 5.6: Absence of *join*: exclusive data-flow edge; see Table 5.4.

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 5.3, and illustrated in Figure 5.5.

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 5.4 and shown in Figure 5.6.

We can explicitly define an exclusive data-flow edge relation for arrays of ports as in Table 5.5. 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 an output port of

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

Table 5.4: Absence of join: exclusive data-flow edge; see Figure 5.6.

```
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 5.5: Array version of exclusive data-flow edge

```
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 5.6: A theorem on join of exactly two data-flow edges

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 would lead more optimization than if it were equivalence. This issue is discussed further in Chapter 6 as part of generalizations of transformations.

A useful theorem involving a join of exactly two data-flow edges, shown in Table 5.6, 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 Table 5.7 and show it in Figure 5.7. 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 5.8. The statement of the property is the theorem in Table 5.8.

```
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 5.7: Order preserved by refinement and optimization; see Figure 5.7.

Figure 5.7: Order preserved by refinement and optimization; see Table 5.7.

```
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 5.8: Order preserved by refinement and exclusive data-flow edge; see Figure 5.8.

Figure 5.8: Order preserved by refinement and exclusive data-flow edge; see Table 5.8.

```
sub_kind_implement_ax: AXIOM
FORALL (n0:node),(n1:node|same_size(n0,n1)):
 sub_kind(n0,n1) AND silimpar(inports(n0),inports(n1)) IMPLIES
 silimp(outport(n0),outport(n1))
```

Table 5.9: Graph refinement: property expressing relation between outputs and inputs of graphs independent of underlying behavior; see Figure 5.9.

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 *sub_kind* of the specification node. Instead of describing the graph refinement by 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. The PVS specification in Table 5.9 illustrates the property in Figure 5.9.

This allows us to compare output ports, given a relationship among the input 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 5.10 to express that two nodes, having the same number of input ports (i.e., they are of the same_size), are of the same kind if they have the same data relations.

Figure 5.9: Graph refinement: property expressing relation between outputs and inputs of graphs independent of underlying behavior; see Table 5.9.

```
same_kind(cn0,(cn1|same_size(cn1,cn0))) =
            datarel(cn0) = datarel(cn1)

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

Table 5.10: Predicates for expressing the sameness of nodes

Chapter 6

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 6.1. We explain in detail *common subexpression elimination* in Section 6.2 and *cross-jumping tail-merging* in section 6.3. We briefly mention specification and verification of other transformations and proofs in Section 6.4, and generalization and composition of transformations in Section 6.5. In Section 6.6, we illustrate with an example the usefulness of the axiomatic specification in investigating "what-if" scenarios. Finally, in Section 6.7, 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.

6.1 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.

Figure 6.1: Common subexpression elimination; see Table 6.1.

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`.

6.2 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 6.1.

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|same_kind(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 6.1.

6.3 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 6.2.

In the course of our specification in PVS, we found a mistake in the informal specification of the transformation. We show the erroneous transformation that was given in the original informal specification in Figure 6.3.

However, the same mistake was discovered later by inspection of the informal specification [Klo 94] 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

```
CSubE: THEOREM
FORALL dot0,(dot1|same_kind(dot1,dot0)),
    (dot01|same_kind(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 6.1: Correctness of common subexpression elimination; see Figure 6.1.

Figure 6.2: Cross-jumping tail-merging: corrected.

Figure 6.3: Cross-jumping tail-merging: incorrectly specified in informal document.

Figure 6.4: Cross-jumping tail-merging: generalized and verified; see Table 6.3.

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 `r2`, 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 6.3 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 $dot0$ and $dot1$ in the original graph is the same as the ordering of the data-flow edges coming into the node $dot01$ in the optimized graph. We have suitably modified, generalized, and verified the transformation. The generalized transformation is shown in Figure 6.4. The PVS specification of the preconditions is shown in Table 6.2, and the theorem statement is shown in Table 6.3.

6.4 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 number of high level inference rule applications required for verifying the various transformations in Table 6.4. The high level inference rules are the rules that the user would use to guide the PVS theorem prover to derive a proof of a theorem. Examples of high level inference rules [SOR 93-2] 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 could 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 B.

6.5 Generalization and Composition of Transformations

We have seen earlier, in Chapter 6.3, 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. 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.

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.

6.6 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

Figure 6.5: Cross-jumping tail-merging: inapplicable when two nodes are merged into one.

that resulted during the transformational design of a direction detector [Mid 94-2]. It involved a variation of the cross-jumping tail-merging transformation. In Figure 6.4, 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 6.5.

Since the nodes are merged, $w0 = w1$. While, due the ordering imposed by *join*, either $war0 < war1$ or $\mathrm{war1} < \mathrm{war0}$. 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.

6.7 Devising New Transformations

In Section 6.6, we argued that cross-jumping tail-merging could not be applied in cases as shown in Figure 6.5. However, we would like to have such a transformation for further optimization in cases as shown in Figure 6.6. We can view this as a transformation derived from the process of generalizing cross-jumping tail-merging 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 6.7. 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 6.8.

The problem can be solved by devising a new and simple transformation as follows. In the description of common subexpression elimination shown in Figure 6.1, the outputs of nodes `dot01` and `dot1` were required to be not connected to join ports. However,

Figure 6.6: Further optimization impossible using existing transformations.

Figure 6.7: Inapplicability of cross-jumping tail-merging after common subexpression elimination: due to precondition restrictions.

Figure 6.8: Inapplicability of common subexpression elimination after cross-jumping tail-merging: due to precondition restrictions.

Figure 6.9: A simple new transformation: obvious, post-facto.

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 6.9. 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.

```
sks(cn1:cnode,cn2:cnode) = same_kind(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 6.2: 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 same_kind(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 6.3: Correctness of cross-jumping tail-merging; see Figure 6.4.

Transformation	Number of high level inference rule applications
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 6.4: Number of high level inference rule applications for various transformations

Chapter 7

Discussion and Conclusions

One of the goals of high-level synthesis 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. 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 7.1. In Section 7.2, we briefly present our experience in developing a formal specification from an informal document. We highlight the advantages of an axiomatic approach in Section 7.3. Finally, Section 7.4 summarizes the conclusions.

7.1 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.

7.2 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 4.2, 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 4. 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 [KoW 93], 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.

7.3 Axiomatic Approach versus 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 [TDM 94, HMW 94] 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 5.9 and Figure 5.9, 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 behavioral 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. 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. This distinction is similar to the contrast between axiomatic semantics and denotational or operational semantics in the context of programming languages. Denotational and operational models worked out by de Jong and Huijs [GGJ 93, HuK 94] could be used as a concrete model that satisfies the axiomatic specification discussed in this report.

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.

7.4 Conclusions and Future Work

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. We have explored generating preconditions for transformations semiautomatically from the specifications. 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 also plan to use our framework to investigate the correctness of transformations involving scheduling and resource allocation.

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 study the correctness of transformations on generic dependency graphs. Finally, we have shown that our approach, and formal methods in general can creatively help discover new techniques in system design. As part of the future work, we are considering a seemless integration of our verification scheme with VLSI CAD tools for hardware design and CASE tools for software design.

References

[AaL 94] M. Aagaard and M. Leeser
PBS: Proven Boolean Algorithm, IEEE Trans. on CAD of ICs. Vol 13, No. 4, April 1994.

[AAD 93] F.V. Aelten, J. Allen, and S. Devadas
Verification of Relations between Synchronous Machines, IEEE Trans. on CAD of ICs. Vol. 12, No. 12, December 1993.

[Ael 94] F.V. Aelten, J. Allen, and S. Devadas
Even-Based Verification of Synchronous Globally Controlled, Logic Designs Against Signal Flow Graphs, IEEE Trans. on CAD of ICs., Vol. 13, No. 1 January 1994.

[Ang 94] C. Angelo
Formal Hardware Verification in a Silicon Compilation Environment by means of theorem proving, PhD Thesis, IMEC, Leuven, Belgium, February 1994.

[Bac 88] R.J.R. Back
A Calculus of Refinements for Program Derivations, Acta Informatica, Vol. 25, pp.593-624, 1988.

[Bar 81] M. R. Barbacci
Instruction Set Processor Specifications (ISPS): The notation and applications, IEEE Trans. on Computers, C-30(1): pp 24-40, 1981.

[BRB 90] K.S. Brace, R.L. Rudell, and R.E. Bryant
Efficient Implementation of a BDD Package, Proceedings of the 27th ACM/IEEE Design Automation Conference, Orlando, Florida, June 24-28, 1990, pp. 40-45.

[YIF 88] R.K. Brayton, R. Camposano, G. DeMicheli, R.H.J.M. Otten, and J.T.J. van Eijndhoven
The Yorktown Silicon Compiler System, Silicon Compilation, D. Gajski (Ed.), Addison-Wesley, 1988.

[BCM 90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang
Symbolic Model Checking: 10^{20} *states and beyond*, Proceedings of the Fifth Annual Symposium on Logic in Computer Science, June 1990.

[Cam 89] R. Camposano
Behavior Preserving Transformations in High Level Synthesis, Hardware Specification, Verification and Synthesis: Mathematical Aspects, Cornell MSI Workshop, Lecture Notes in Computer Science 408, pp 106-128, Springer-Verlag, July 1989.

[UNI 88] M. Chandy and J. Misra
Parallel Program Design: a foundation, Reading, Mass. : Addison-Wesley Pub. Co., c1988.

[CBL 92] R. Chapman, G. Brown, and M. Leeser
Verified High-Level Synthesis in BEDROC, Proceedings of the 1992 European Design Automation Conference, March 1992, IEEE Press.

[ELL 90] Computer General Electronic Design
The ELLA Language Reference Manual, The New Church,
Henry St. Bath BA1 1JR, U.K., issue 4.0, 1990.

[Cyr 93] D. Cyrluk
Microprocessor Verification in PVS: A methodology and simple example,
SRI-CSL-93-12, Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[CRS 94] D. Cyrluk, S. Rajan, N. Shankar, and M. Srivas
Effective Theorem Proving for Hardware Verification, Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design, Bad Heerenalb (Blackforest), Germany, 26-29 September, 1994.

[TDM 94] Tom DeMarco
Structured Analysis and System Specification, Yourdon Press, New Jersey, USA, 1979.

[EMH 93] W.J.A Engelen, P.F.A. Middelhoek, C. Huijs, J. Hofstede, and Th. Krol
Applying Software Transformations to SIL,
SPRITE deliverable Ls.a.5.2/UT/Y5/M6/1A, June 1993.

[Fou 90] M. P. Fourman
Formal System Design, Formal Methods for VLSI Design, J. Staunstrup (ed.), North-Holland, IFIP 1990.

[GaG 89] S. J. Garland and J. V. Guttag
An Overview of LP: the Larch Prover, Proceedings of the Third International Conference on Rewriting Techniques and Applications, Springer-Verlag, 1989.

[GoM 93] M. J. C. Gordon and T. F. Melham (Ed.)
Introduction to HOL: a theorem proving environment for higher order logic, Cambridge University Press, 1993.

[Hoa 85] C. A. R. Hoare
Communicating Sequential Processes, Prentice Hall, Hemel Hempstead, UK, 1985.

[Hil 85] P. N. Hilfinger
Silage: a High-level Language and Silicon Compiler for Digital Signal Processing, Proceedings of IEEE Custom Integrated Circuits Conference pp 213-216, Portland, OR, May 1985.

[Hoo 94] Jozef Hooman
Correctness of Real Time Systems by Construction, Proceedings, Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, LNCS, Springer-Verlag, September 20-24, 1994 (to appear).

[HHK 92] C. Huijs, J. Hofstede, and Th. Krol
Transformations and semantical checks for SIL-1,
SPRITE deliverable LS.a.5.1/UT/Y4/M6/1, November 1992.

[HuK 94] C. Huijs and Th. Krol
A Formal Semantic Model to fit SIL for Transformational Design, to appear in: Microprocessing and Microprogramming 39 (1994) Proceedings of Euromicro '94, September 5-8-1994 Liverpool.

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

[Jan 93] G. Janssen
ROBDD software, Department of Electrical Engineering, Technical University of Eindhoven, Eindhoven, Netherlands, October 1993.

[Joh 94] S. D. Johnson
Synthesis of Digital Designs from Recursion Equations, MIT Press, Cambridge, 1984.

[Jon 90] G. Jones and M. Sheeran
Circuit Design in Ruby, Formal Methods for VLSI Design, J. Staunstrup (ed.), North-Holland, IFIP 1990.

[GGJ 93] G.G de Jong
Generalized data flow graphs: theory and applications, PhD Thesis, Eindhoven University of Technology, October 1993.

[JRS 91] J. Joyce, E. Liu, J. Rushby, N. Shankar, R. Suaya, and F. von Henke
From Formal Verification to Silicon Compilation, Proceedings of the IEEE Compcon, San Francisco, CA, February 1991, pp. 450-455

[Klo 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, SPRITE, deliverable Ls.a.a / Philips / Y3 / M12 / 2, October 1992.

[Klo 94] W.E.H. Kloosterhuis
Personal Communication, January 1994.

[KoW 93] A. P. Kostelijk and A. van der Werf
Functional Verification for Retiming and Rebuffering Optimization, Proceedings of The European Conference on Design Automation with the European Event in ASIC Design, Paris, France, Feb 22-25, 1993, IEEE Computer Society Press.

[Kro 92] Th. Krol, J.v. Meerbergen, C. Niessen, W. Smits, and J. Huisken
The SPRITE Input Language, An intermediate format for High Level Synthesis, Proceedings of EDAC 92, Brussels, 16-19 March 1992, pp 186-192.

[LOR 93] Patrick Lincoln, Sam Owre, John Rushby, N. Shankar, F. von Henke
Eight Papers on Formal Verification, Technical Report SRI-CSL-93-4, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993.

[McF 93] M.C. McFarland
Formal Analysis of Correctness of Behavioral Transformations, Formal Methods in Systems Design Vol.2, No.3 pp. 231-257, Kluwer, June 1993.

[McP 83] M.C. McFarland and A.C. Parker
An Abstract Model of Behavior for Hardware Descriptions, IEEE Trans. on Computers C-32(7), pp.621-36, July 1983.

[KLM 92] Kenneth L. McMillan
Symbolic Model Checking, PhD Thesis, Technical Report, CMU-CS-92-131 pp 97-99, May 1992.

[Mid 93] P.F.A. Middelhoek
Transformational Design of Digital Circuits, Proceedings of the Seventh Workshop Computersystems, 26 November 1993, Eindhoven, The Netherlands, pp. 57-68.

[Mid 94] P.F.A. Middelhoek
Transformational Design of Digital Signal Processing Applications, Proceedings of the ProRISC/IEEE workshop on CSSP, 24 March 1994, pp. 175-180.

[Mid 94-2] P.F.A. Middelhoek
Transformational Design of a Direction Detector for the Progressive Scan Conversion Algorithm, Preliminary, Department of Computer Science, University of Twente, May 25, 1994.

[OSR 93] S. Owre, N. Shankar, and J.M. Rushby
User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release), Computer Science Laboratory, SRI International, Menlo Park, CA, USA, February, 1993.

[RTJ 93] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson
An Introduction to Behavior Tables, Technical Report No. 392, Computer Science Department, Indiana University, December 1993.

[Ros 90] Lars Rossen
Formal Ruby, Formal Methods for VLSI Design, J. Staunstrup (ed.), North-Holland, IFIP 1990.

[Sax 93] J. B. Saxe, J.J. Horning, J.V. Guttag, and S.J. Garland
Using Transformations and Verification in Circuit Design, Formal Methods in Systems Design Vol.3, No.3, pp. 181-209, Kluwer, December 1993.

[SOR 93-1] N. Shankar, S. Owre, and J.M. Rushby
A Tutorial on Specification and Verification using PVS (Beta Release), Computer Science Laboratory, SRI International, Menlo Park, CA, USA, March 31, 1993.

[SOR 93-2] 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, USA, March 31, 1993.

[Kid 90] Douglas R. Smith
KIDS: A Semi-Automatic Program Development System, Transactions on Software Engineering: Special Issue on Formal Methods, Vol. 16, No. 9, September, 1990.

[Sta 90] J. Staunstrup and M. Greenstreet
Synchronized Transitions, Formal Methods for VLSI Design, J. Staunstrup (ed.), North-Holland, IFIP 1990.

[Tho 98] D. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, and R.L. Blackburn
The System Architect's Workbench, Proceedings of the 25th Design Automation Conference, ACM/IEEE, pp 337-343, 1988.

[Vem 90] R. Vemuri
How to Prove the Completeness of a Set of Register Level Design Transformations, Proceedings of the 27th Design Automation Conference, pp. 207-212, ACM/IEEE, June 1990

[WMM 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, Proceedings of the SPRITE workshop on "VLSI Synthesis for DSP", Section 12, Philips Research Labs, Eindhoven, March 1994.

[JMW 90] Jeannette M. Wing
A Specifier's Introduction to Formal Methods, IEEE Computer, Vol. 23, Number 9, pp 8-22, IEEE Computer Society Press, September 1990

[HMW 94] M. Wong
Informal, Semi-formal, and Formal Approaches to the specification of software Requirements, Masters Thesis, Department of Computer Science, UBC, September 1994.

[WrS 91] J. von Wright and K. Sere
Program Transformations and Refinements in HOL, Higher Order Logic Theorem Proving and its Applications, International Workshop, Proceedings, M. Archer, J.J. Joyce, K.N. Levitt and P.J. Windley (Eds.), IEEE Computer Society Press, August 28-30, 1991.

Appendix A

Definitions, Axioms and Theorems

A.1 Definitions

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

cnode: TYPE =
         [#
           inports: parray,
           outport: port,       % strictly, this should also be
                                % parray (as in SPLIT) for
                                % hierarchical nodes.
           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]]
         #]

% derive a node as a subtype of cnode
node: TYPE = {n:cnode|cond(n)=LAMBDA (p:port):TRUE}
```

```
cn0,cn1: VAR cnode
par0,par1: VAR parray

% Useful functions in comparing nodes and parrays
same_size(cn0,cn1):boolean =
        size(inports(cn0)) = size(inports(cn1))
same_size(par0,par1):boolean =
        size(par0) = size(par1)

% same_size appears as a type constraint
same_kind(cn0,(cn1|same_size(cn1,cn0))) =
    datarel(cn0) = datarel(cn1)
sks(cn0,cn1) =
    same_size(cn0,cn1) AND same_kind(cn0,cn1)

% refinement/implementation relationship between nodes
sub_kind(cn0,(cn1|same_size(cn1,cn0))):boolean

sbks(cn0,cn1) = same_size(cn0,cn1) AND sub_kind(cn0,cn1)

% defines behavioral implication of sil graphs/ports
silimp: pred[[port,port]]

p0,p1,p2: VAR port
par,par1,par2,par3: VAR parray
i: VAR nat

% defines array version of silimp: note the weak axiom def
silimpar(par1,par2):boolean

% defines a behavioral equivalence of sil graphs
sileq(p1,p2):boolean =
        silimp(p1,p2) AND silimp(p2,p1)
```

```
% array version
sileqar(par1,par2):boolean =
        FORALL (i| i < size(par1)):
                sileq(port_array(par1)(i),port_array(par2)(i))

% Arbitrary functions corresponding to data_rel of nodes
silf: VAR [port->port]
silfar: VAR [parray->port]

<: pred[[weight,weight]]

% data flow edge is a relation on ports
dfe: [port,port -> boolean]
% an arbitrary fixed function corresponding to ports
w: [port,port->weight]

p,p0,p1,p2,p3,p4: VAR port
cn,cn0,cn1,cn2,cn3: VAR cnode
n,n0,n1,n2,n3: VAR node

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)

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

```
% array version of dfear and xdfear and ordering
war: [parray,parray->weight]    % or weightarray??

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

dfear(par0,(par1:{par|same_size(par,par0)})):boolean =
      (FORALL (i|i<size(par0)):
         dfe(port_array(par0)(i),port_array(par1)(i)))

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

% array version of the above corresponding theorems -
% illustrates clarity of specification
is_node_outport(p):boolean =
      EXISTS n: p = outport(n)

is_outportar(par):boolean =
   FORALL (i|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 =
   FORALL (i|i<size(par)): is_cnode_outport(port_array(par)(i))

% definition of an assignment node
asignment(cn:cnode):node =
   cn WITH [inports := inports(cn) WITH [size := 1]]
      WITH [dataf := LAMBDA (p:parray): outport(cn) =
                         port_array(inports(cn))(0)]

% definition of floor function for real-integer
% refinement transformation
floor(x): int =
   epsilon (LAMBDA y: y <= x AND y > (x-1))
```

A.2 Axioms

```
cn: VAR cnode
cnode_ax: AXIOM
FORALL cn: cond(cn)(condport(cn)) IMPLIES
           datarel(cn)(inports(cn),outport(cn))

silimpar_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)

% Reflexivity
silimp_refl_ax: AXIOM silimp(p1,p1)

% Transitivity
silimp_trans_ax: AXIOM
FORALL p0,p1,p2:
silimp(p0,p1) AND silimp(p1,p2) IMPLIES silimp(p0,p2)
silimpar_trans_ax: AXIOM
silimpar(par1,par2) AND silimpar(par2,par3)
                  IMPLIES
            silimpar(par1,par3)
```

```
% sub kind nodes implement each other
sub_kind_implement_ax: AXIOM
FORALL (n0:node),(n1:node|same_size(n0,n1)):
       sub_kind(n0,n1) AND silimpar(inports(n0),inports(n1))
                     IMPLIES
       silimp(outport(n0),outport(n1))

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

% partial order relation on weights
partial_order(<:pred[[weight,weight]])

dfe_port_ax1: AXIOM
dfe(p1,p2) IMPLIES is_outport(p1)

dfe_port_ax2: AXIOM
dfe(p1,p2) IMPLIES (is_inport(p2) OR is_condport(p2))

% We need these general axioms on dfes and partial order on w's
dfe_w_ax: AXIOM
(dfe(p0,p2) AND dfe(p1,p2))
            IFF
(w(p0,p2) < w(p1,p2) OR
 w(p1,p2) < w(p0,p2))

% if the conditional port val is false, then the w involving its
% output port is the least!: this is the property of the bottom
% we want!!
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)
% Generalized join axiom
join_ax: AXIOM
(dfe(p1,p2) AND
 (FORALL p: dfe(p,p2) IMPLIES w(p,p2) < w(p1,p2)))
             IMPLIES
           silimp(p1,p2)
```

```
% Partial order preservation: Advanced axiom (we can't prove it
% unless we introduce extra delay axioms for nodes/silimp)
p00,p11,p22,p33,p44: VAR port
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)

% Generalized join axiom for arrays
joinar_ax: AXIOM
FORALL par1,(par2|same_size(par2,par1)):
(dfear(par1,par2) AND
 (FORALL (par|same_size(par,par1)):
  dfear(par,par2) IMPLIES war(par,par2) < war(par1,par2)))
         IMPLIES
  silimpar(par1,par2)
```

A.3 Theorems

```
% property of an non-conditional node
node_data_rel_th: THEOREM
FORALL (n:node): datarel(n)(inports(n),outport(n))

silimpar_refl_th: THEOREM
silimpar(par,par)


% sileq_ar_refl
sileqar_refl_th: THEOREM
sileqar(par1,par1)

% sileq_ar_sym
sileqar_sym_th: THEOREM
FORALL par1,(par2:{par|same_size(par,par1)}):
sileqar(par1,par2)= sileqar(par2,par1)

% sileq_ar_trans
sileqar_trans_th: THEOREM
FORALL par1,(par2:{par|same_size(par,par1)}),
            (par3:{par|same_size(par,par1)}):
(sileqar(par1,par2) AND sileqar(par2,par3))
           IMPLIES
       sileqar(par1,par3)

% same kind non-conditional nodes propagate similar outputs
in_eqar_imp_outeq: THEOREM
FORALL (n0:node),(n1:node|sks(n0,n1)):
sileqar(inports(n0),inports(n1))
        IMPLIES
sileq(outport(n0),outport(n1))
```

```
% Make sure of no joins in this theorem:
% Holds only if the dfe connects an outport of a
% non-conditional node: partly taken care of by typing
% (n1 is an ordinary node type)
xdfe_sileq_th: THEOREM
xdfe(outport(n1),p2) IMPLIES sileq(outport(n1),p2)

% partial order preservation extension theorem
po_preserve_xdfe_th: THEOREM
(w(p0,p2) < w(p1,p2) AND
 sileq(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)

% Join of 2 dfes
dfe2_join_th: THEOREM
(dfe(p1,p3) AND dfe(p2,p3) AND
 (FORALL p0:
        ((p0 /= p1) OR (p0 /= p2))
        IMPLIES NOT dfe(p0,p3)))
       IMPLIES
IF w(p1,p3) < w(p2,p3) THEN
        sileq(p2,p3)
ELSE    sileq(p1,p3)
ENDIF
```

```
% array version theorems
inports_sileqar_th: THEOREM
FORALL (nd:node):
FORALL (n0:node|same_size(n0,nd)),(n1:node|same_size(n1,nd)):
(
 (FORALL (i|i<size(inports(nd))),n:
        xdfe(outport(n),inport(n0,i)) IFF
         (EXISTS (nn:node): sileq(outport(n),outport(nn)) AND
                           xdfe(outport(nn),inport(n1,i))
         )
  ) AND
 (FORALL (i|i<size(inports(nd))):
         EXISTS n: xdfe(outport(n),inport(n0,i)))
      IMPLIES
sileqar(inports(n0),inports(n1)))

inports_eqar_th: THEOREM
FORALL (n0:node),(n1:node|same_size(n0,n1)):
(
 (FORALL (i|i<size(inports(n0))),n:
         xdfe(outport(n),inport(n0,i)) IFF
         xdfe(outport(n), inport(n1,i))) AND
 (FORALL (i|i<size(inports(n0))):
    EXISTS n: xdfe(outport(n),inport(n0,i)))
          IMPLIES
    sileqar(inports(n0),inports(n1)))

xdfear_sileqar_th: THEOREM
FORALL (par0|is_outportar(par0)),
       (par1|same_size(par1,par0)):
xdfear(par0,par1) IMPLIES sileqar(par0,par1)
```

```
% Inports connected by exclusive data flow edge arrays
% to identical ports are sileqar
inportsar_eqar_th: THEOREM
FORALL (n0:node),(n1:node|same_size(n0,n1)):
((
  (FORALL (par|is_outportar(par) AND same_size(par,inports(n0))):
     xdfear(par, inports(n0)) IFF xdfear(par,inports(n1))) AND
  (EXISTS (par|is_outportar(par) AND same_size(par,inports(n0))):
                xdfear(par, inports(n0)))
 )
       IMPLIES
sileqar(inports(n0),inports(n1)))

% Inports connected by  exclusive data flow edge arrays
% to sileq port arrays sileqar
inportsar_sileqar_th: THEOREM
FORALL (n0:node),(n1:node|same_size(n0,n1)):
((
  (FORALL (par|is_outportar(par) AND same_size(par,inports(n0))):
         xdfear(par,inports(n0)) IFF
  (EXISTS (parr|is_outportar(par) AND
           same_size(parr,inports(n0))):
        (sileqar(par,parr) AND
         xdfear(parr,inports(n1))))) AND
   EXISTS (par|is_outportar(par) AND same_size(par,inports(n0))):
         xdfear(par,inports(n0))
)
IMPLIES sileqar(inports(n0),inports(n1)))
```

```
dfear2_join_th: THEOREM
FORALL par1,(par2|same_size(par2,par1)),
            (par3|same_size(par3,par1)):
(dfear(par1,par3) AND dfear(par2,par3)) AND
(FORALL (par|same_size(par,par1)):
         (par /= par1 OR par /= par2)
           IMPLIES
         NOT dfear(par,par3))
        IMPLIES
IF war(par1,par3) <= war(par2,par3)
THEN sileqar(par2,par3)
ELSE    sileqar(par1,par3)
ENDIF

% Common Subexpression Elimination Transformation
p0,p1,p2,p3: VAR port
par,parr: VAR parray
dot0,dot1,dot01: VAR node
% The preconditions can be weakend at par, such as -
% to exists par1: silimp(par1,par)
preconds(dot0,(dot1:{dot1|same_kind(dot0,dot1)})):boolean =
(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)))

CsubE: THEOREM
FORALL dot0,(dot1|same_kind(dot1,dot0)),
            (dot01|same_kind(dot01,dot0)):
((
 preconds(dot0,dot1) AND
 (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))):
        (sileqar(par,parr) AND xdfear(parr,inports(dot01)))))
 )
        IMPLIES
(FORALL p1,p2:
        ((xdfe(outport(dot0),p1) OR xdfe(outport(dot1),p1)) AND
          xdfe(outport(dot01),p2)) IMPLIES sileq(p1,p2))
)
```

```
% Cross Jumping Tail Merging Theorem
p0,p00,p1,p11,p2,p22,p3,p33,pp,pp0,pp00: VAR port
n,dot,dot0,dot1,dot01: VAR node
cn2,cn3,cn22,cn33: VAR cnode
par,par0,par1,par00,par11,par2,par3: VAR parray
% preconditions
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) =
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
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
dfear(par00,inports(dot01)) AND dfear(par11,inports(dot01))
                     AND
(FORALL (par|size(par)=size(par00)):
        (par /= par00 OR par /= par11)
                IMPLIES
         NOT dfear(par,inports(dot01)))
                     AND
xdfe(outport(dot01),pp00) AND
sileqar(par0,par00) AND
sileqar(par1,par11)
```

```
% Cross jumping tail merging transformations is correct when
% the preconditions are satisfied
CjtM: THEOREM
FORALL dot0:
 LET sk = LAMBDA n: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)):
preconds(dot0,dot1,dot01,par0,par1,par00,par11,pp0,pp00)
IMPLIES
sileq(pp0,pp00)
```

Appendix B

Proof Transcripts

B.1 Common Subexpression Elimination

Terse proof for `CSubE`.

`CSubE:`

$$
\begin{array}{|ll}
\hline
\{1\} & \forall\ \mathrm{dot0}, (\mathrm{dot1}\ \mid\ \mathrm{same\_kind}(\mathrm{dot1},\ \mathrm{dot0})),\\
& \quad (\mathrm{dot01}\ \mid\ \mathrm{same\_kind}(\mathrm{dot01},\ \mathrm{dot0})) :\\
& \quad ((\mathrm{preconds}(\mathrm{dot0},\ \mathrm{dot1})\\
& \qquad \wedge\\
& \qquad (\forall\ (\mathrm{par}\ \mid\ \mathrm{is\_outportar}(\mathrm{par})\ \wedge\ \mathrm{size}(\mathrm{par})\ =\ \mathrm{size}(\mathrm{inports}(\mathrm{dot0}))) :\\
& \qquad\quad \mathrm{xdfear}(\mathrm{par},\ \mathrm{inports}(\mathrm{dot0}))\\
& \qquad\qquad \Leftrightarrow\\
& \qquad\qquad (\exists\ (\mathrm{parr}\\
& \qquad\qquad\quad \mid\ \mathrm{is\_outportar}(\mathrm{parr})\ \wedge\ \mathrm{size}(\mathrm{parr})\ =\ \mathrm{size}(\mathrm{inports}(\mathrm{dot0}))) :\\
& \qquad\qquad\quad (\mathrm{sileqar}(\mathrm{par},\ \mathrm{parr})\ \wedge\ \mathrm{xdfear}(\mathrm{parr},\ \mathrm{inports}(\mathrm{dot01}))))))\\
& \qquad \supset\\
& \quad\ (\forall\ p_1,\\
& \qquad p_2 :\\
& \qquad ((\mathrm{xdfe}(\mathrm{outport}(\mathrm{dot0}), p_1)\ \vee\ \mathrm{xdfe}(\mathrm{outport}(\mathrm{dot1}), p_1))\\
& \qquad\quad \wedge\ \mathrm{xdfe}(\mathrm{outport}(\mathrm{dot01}), p_2))\\
& \qquad\quad \supset\ \mathrm{sileq}(p_1, p_2)))
\end{array}
$$

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.

B.2 Cross Jumping Tail Merging

Terse proof for `CjtM`.

`CjtM`:

$$
\begin{array}{l}
\{1\} \quad (\forall\, (\mathrm{pp0},\ \mathrm{pp00} : \mathrm{port}) : \\
\quad \forall\, (\mathrm{dot0} : \mathrm{node}) : \\
\quad\quad \textsc{let}\ \mathrm{sk} : [\mathrm{node} \to \mathrm{bool}] = \\
\quad\quad\quad\quad \lambda\, (n : \mathrm{node}) : \mathrm{same\_size}(n,\ \mathrm{dot0}) \wedge \mathrm{same\_kind}(n,\ \mathrm{dot0}), \\
\quad\quad\quad \mathrm{ios} : [\mathrm{parray} \to \mathrm{bool}] = \\
\quad\quad\quad\quad \lambda\, (\mathrm{par} : \mathrm{parray}) : \mathrm{is\_outportar}(\mathrm{par}) \\
\quad\quad\quad\quad\quad \wedge\ \mathrm{size}(\mathrm{par}) = \mathrm{size}(\mathrm{inports}(\mathrm{dot0})) \\
\quad\quad \textsc{in}\ \forall\, (\mathrm{dot1} : \mathrm{node} \mid \mathrm{sk}(\mathrm{dot1})), (\mathrm{dot01} : \mathrm{node} \mid \mathrm{sk}(\mathrm{dot01})), \\
\quad\quad\quad\quad (\mathrm{par0} : \mathrm{parray} \mid \mathrm{ios}(\mathrm{par0})), (\mathrm{par1} : \mathrm{parray} \mid \mathrm{ios}(\mathrm{par1})), \\
\quad\quad\quad\quad (\mathrm{par00} : \mathrm{parray} \mid \mathrm{ios}(\mathrm{par00})), \\
\quad\quad\quad\quad (\mathrm{par11} : \mathrm{parray} \mid \mathrm{ios}(\mathrm{par11})) : \\
\quad \mathrm{preconds}(\mathrm{dot0},\ \mathrm{dot1},\ \mathrm{dot01},\ \mathrm{par0},\ \mathrm{par1},\ \mathrm{par00},\ \mathrm{par11},\ \mathrm{pp0},\ \mathrm{pp00}) \\
\quad\quad \supset\ \mathrm{sileq}(\mathrm{pp0},\ \mathrm{pp00}))
\end{array}
$$

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.