Correctness of Transformations in High Level Synthesis: Formal Verification*

Sreeranga P. 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

Abstract

This paper presents a formal approach to address the correctness of transformations in high-level synthesis. The novelty of the work is that a small set of properties that capture a general notion of refinement of control/data-flow graphs used in an industrial synthesis framework have been given, and the properties are independent of the underlying behavior model. We have mechanized the specification and verification of several optimization and refinement transformations used in industrial hardware design. This work has enabled to find and rectify errors in the transformations. Further, the work has led to generalization of transformations typically used in high-level synthesis.

1 Introduction

In high level synthesis, a sequence of transformations is used to optimize and refine specifications at the behavior level into implementations at the register transfer level. The transformations are correct if the set of behaviors allowed by the implementation is a subset 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 specification is represented by control/data-flow graphs composed of nodes and

*Part of this work will appear in the proceedings of the 1995 International Conference on Computer Hardware Description Languages, August 1995, Chiba, Japan.

edges. Typically, control/data-flow 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 guarantee the correctness of the transformations. The problem we undertake in this work is to provide guarantees for correctness of transformations on control/data-flow graphs, independent of the underlying behavior model[1].

A typical transformation used in high level synthesis is *cross jumping tail merging* [EMH 93], shown in Figure 1. In this transformation, two identical nodes on data flow paths that are never active at the same time are merged into one node. However, as we found out using our approach explained in this paper, the transformation does not preserve behavior. Informally, the reason is as follows. The conditions True/False on open/filled bullets in Figure 1 on the bottom/top of the nodes control the execution of the nodes. In graph G1, when `c` is false, the value of `q10` is arbitrary, and so is the value of `p10`. If we choose the value of `r1` to be that of `p10`, the value of `r1` is also arbitrary. In graph G2, when `c` is false, we could choose the value of `p2` to be that of `q21`. In this case, the value of `r2` is `(y1 * y2)`. Thus, the set of outputs for a given tt y1 and `y2` is the set of all values allowed by the type of the output port in graph G1. Whereas in graph G2, the set of outputs is a singleton set containing the element `(y1 * y2)`. Since, with identical inputs the corresponding sets of possible outputs are unequal, the behaviors of the graphs are not equivalent. More details about this transformation are presented in Section 3.2. The main contributions of this work are the following:

- A formal characterization of control/data-flow graphs has been achieved.
- A set of optimization and refinement transformations on control/data-flow graphs used in high level synthesis have been mechanically verified. Generalization of transformations have also been proposed.
- Errors have been discovered in the transformations used in industrial hardware design. Modifications for the erroneous transformations have been proposed and incorporated.

The context of our work is the synthesis framework as part of the SPRITE project at Philips Research Laboratories (PRL). The synthesis system is based on a general control/data-flow graph called SPRITE Input Language (SIL) [Kro 92, Klo 92]. 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 is used to represent both the specification at the behavior level

[1] We mean by behavior model, the model governing the type and history of data values that the graph may assume

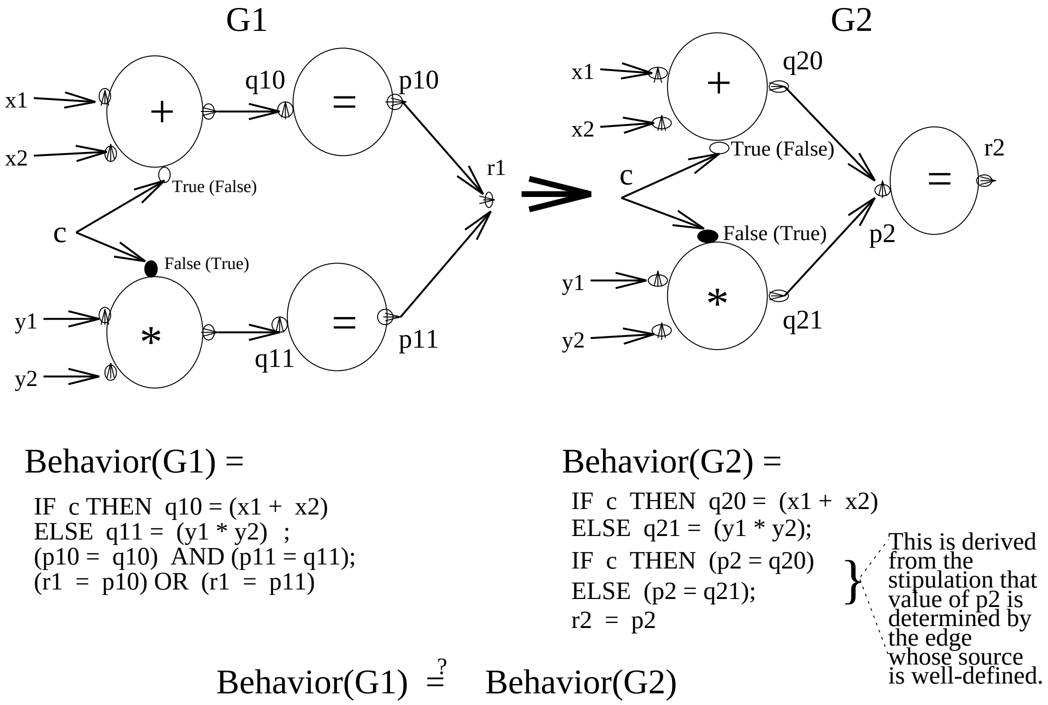


Figure 1: Cross jumping tail merging: incorrectly specified in informal document

and the implementation at the Register Transfer Level (RTL). Logic synthesis tools such as PHIDEO at PRL are used to obtain gate level designs from RTL descriptions. Important features of SIL include hierarchy and design freedom. It 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].

We use a property oriented approach to address the correctness problem. In this approach, a small set of basic properties corresponding to SIL graphs are asserted to be true. Such basic properties assumed to be true are called *axioms*. Other properties are checked to be true by applying a small number of inference rules on properties that have been already verified. Such derived properties are called *theorems*. We use the Prototype Verification System (PVS) from SRI International to mechanize the verification scheme. The PVS specification language allows us to specify the properties using a convenient level of abstraction. The PVS verifier features automatic procedures and interactive inference rules to check properties of specifications. The inference rules are based on higher order logic [SOR 93]. We show SIL and our work in the context of the synthesis system in Figure 2. The

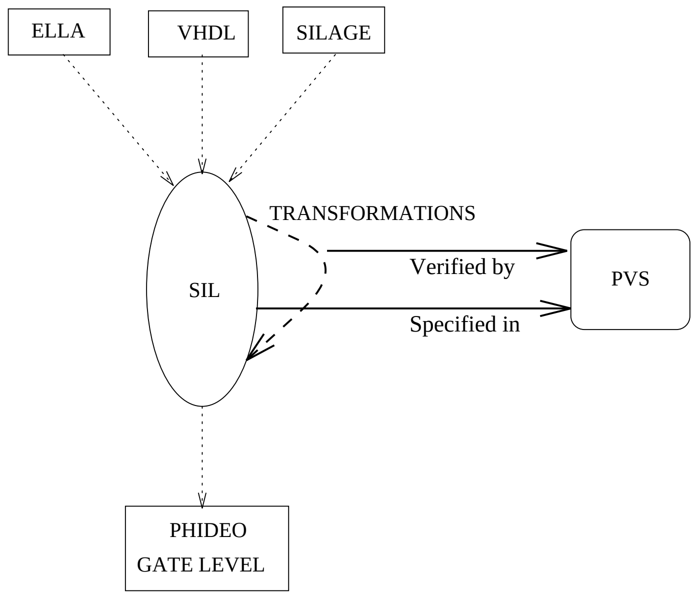


Figure 2: SIL transformations and verification in PVS in the context of high level synthesis

rest of the paper is organized as follows. We discuss related work in Section 1.1. Following an overview of SIL in Section 2, we provide a formal characterization for SIL, and a verification scheme for transformations in section 3. A brief account of the mechanization in PVS, and results from verifying various transformations are tabled in Section 4. Finally, Section 5 summarizes the conclusions.

1.1 Related Work

There have been some efforts in analysis and verification of refinement transformations in the past. However, none of the past work has dealt with correctness of transformations on control/data-flow graphs in general. Most of the efforts have concentrated on specialized hardware description languages tied to specific behavior models. In comparison, our work does not depend on any specific model of behavior. This makes it applicable to a variety of control/data-flow graph formalisms used in different high level synthesis frameworks.

A formal model was proposed for verifying correctness of high-level transformations by McFarland and Parker [McP 83]. A formal system using transformations for hardware synthesis has been discussed by Fourman [Fou 90]. A synthesis system for a language based on an algebraic formalism has been presented by Jones and Sheeran [She 90], and its formalization has been presented by Rossen [Ros 90]. Another algebraic approach to transformational design of hardware has been worked out by Johnson [Joh 94]. 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 logic synthesis at PRL. A formal analysis of transformations used in Systems Architect Workbench (SAW) high-level synthesis was studied by McFarland [McF 93]. 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 *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]. A formalization of SILAGE transformations in HOL was studied by Angelo [Ang 94].

2 Overview of SIL

The descriptions in SIL are presented as graphs. They are used to describe synchronous systems. The behavior of a SIL graph is derived from the behaviors of structural building blocks of the graph. We describe the structural aspects in Section 2.1 and behavioral aspects in Section 2.2

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 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. 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. Here, we address *sequence edge* and *data flow edge*:

- A *sequence edge* is used to specify the order in which data values get transmitted at ports.
- A *data flow edge* is used to specify the direction of communication of data values from a *source* port to a *sink* port.
- 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 3 is an example of a general SIL graph.

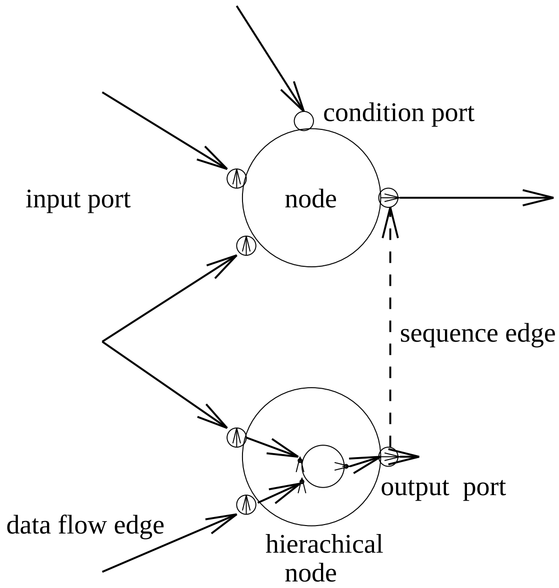


Figure 3: A general SIL graph

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 control/data flow. 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. 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 constrains the sequence in which the data values can appear at the ports of the nodes: this is particularly important in a hierarchically built node.

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 connecting it exactly once. The token is consumed at an input port (sink) to which the edge is connected. In terms of the token flow concept, sequence edges indicate the order of token arrival at ports: a sequence edge from port A to port B describes the token arriving at A before it arrives at B. A data flow edge has an implicit sequence edge from its source to its sink.

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

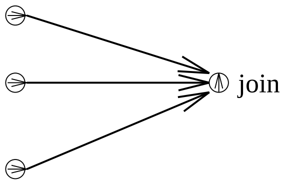


Figure 4: Join in SIL: informal description

Figure 4. 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. From the definition of SIL, there will be exactly one such data flow edge. 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.

A SIL graph models the behavior of a system during one clock cycle due to the single token flow concept. The repetition of a SIL graph, called *unfold*ing over multiple clock cycles gives the behavior of the system across clock cycles. We give an example of a cumulative adder specified using a *DELAY* node in Figure 5. Figure 6 shows an unfolding of the cumulative adder over multiple clock cycles. The SIL graph description of a multiplexor shown in Figure 7 illustrates the use of a sequence edge to get the desired behavior.

2.3 Transformations in SIL

There are a number of optimization and refinement transformations [EMH 93] in SIL. 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 94-1].

3 Formal Characterization of SIL

In this section, we first introduce a relation that describes refinement on ports. The refinement relation is specified without giving a concrete model of the behavior of ports. This is achieved by providing the basic properties that the relation has to satisfy under any model of behavior. We then specify the structural properties that the ports need to satisfy for behavioral refinement. These properties are then used to specify refinement of control/data-flow graphs. The fundamental properties are

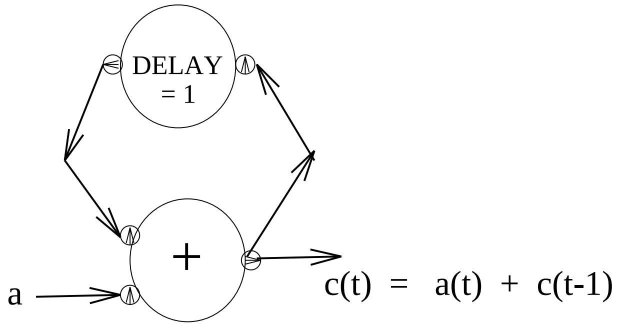


Figure 5: Cumulative adder using a DELAY node: single clock cycle

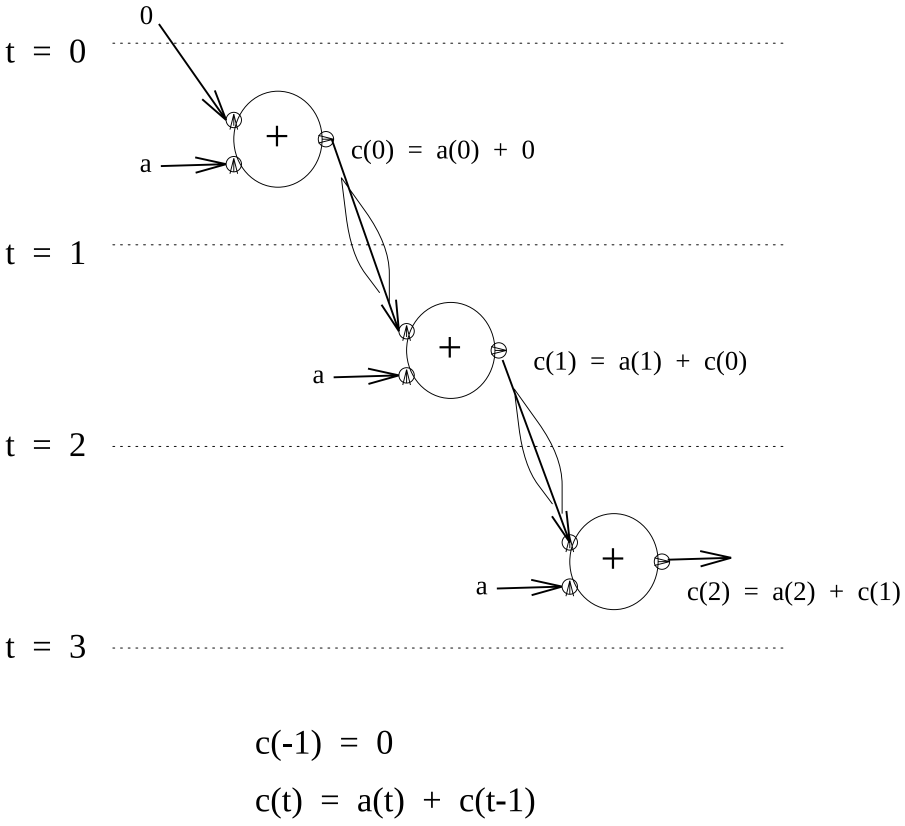
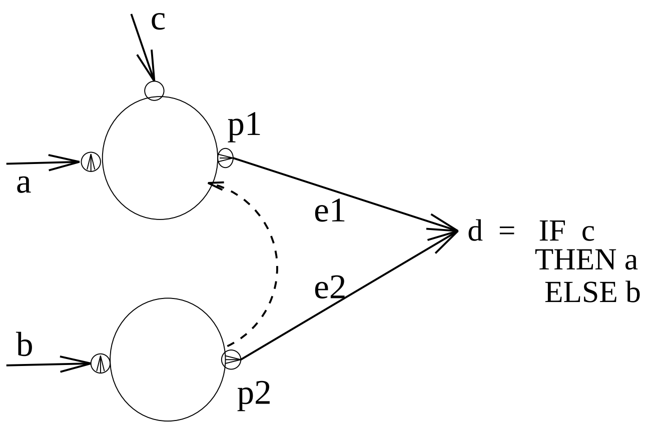


Figure 6: Cumulative adder: unfolded over multiple clock cycles



Sequence edge from p2 to p1 means token at p1
arrives at d later than that due to p2 when c is TRUE

Figure 7: SIL graph description of a multiplexor in a single clock cycle

asserted as axioms. While, other properties are derived as theorems by applications of inference rules on other theorems or axioms[2]. The verification scheme for correctness of transformations with an illustration of its application for the cross jumping tail merging transformation is discussed given in Section 3.2.

We introduce an abstract refinement relation `silimp`, as a relation on ports[3] of the nodes. Thus, if `p1` and `p2` are sets of ports, `silimp(p1,p2)` means that `p1` is a refinement of `p2`. This relation could be interpreted to mean that the set of values allowed by `p1` is a subset of the values allowed by `p2`. It should be noted however, we do not constrain the type or history of data values the ports could assume at any time. The refinement relation `silimp` is asserted to have the properties of reflexivity and transitivity. Such basic properties have to be satisfied under any behavior model of control/data-flow graphs. We do not impose antisymmetry to allow the definition of equivalence as a special case of refinement.

The equivalence of SIL graphs *sileq* is defined by introducing the symmetry property in the refinement relation `silimp`:

Definition 3.1 (Equivalence)

```
FORALL (p1:port),(p2:port):
sileq(p1,p2) = (silimp(p1,p2) AND silimp(p2,p1))
```

[2] We provide here the key axioms and theorems needed to prove properties of SIL graphs. A more detailed axiomatization appears in a forthcoming dissertation [RAJ 94].

[3] We could also allow the `silimp` relation to hold among sets of ports. The size of the sets of ports could be arbitrary

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. The data flow edges are ordered according to the order in which the tokens are communicated on the edges. We model this ordering by associating weights with the data flow edges. A function `w` on ports would return a weight, that could be a number:

Definition 3.2 (Weight)

```
w: [port,port -> 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[4] $<$ on weights, and assert the fact that the weights are ordered if and only if the associated data flow edges form a join. A data flow edge between port `p0` and `p1` is indicated by the relation `dfe(p0,p1)`. The axiom is specified as follows:

Axiom 3.1 (Partial order on weights)

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

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. We state the property that the join port is a refinement of the source whose associated data flow edge has the maximum weight:

[4] We do not use the usual notation <= to stress that no two weights on different edges forming a join can be equal.

Axiom 3.2 (Behavior of join)

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

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`. The behavior due to such an edge is derived as the following theorem:

Theorem 3.1 (Behavior in the absence of join)

```
FORALL (n1:graph),(p:port):
xdfe(outport(n1),p) IMPLIES silimp(outport(n1),p)
```

We can now associate the refinement and equivalence relation with a complete graph by using the properties expressed above. A graph `refines` or implements another graph, when the data relation of the implementing graph is contained in the data relation of the specification graph. This property is expressed by the final axiom as follows:

Axiom 3.3 (Graph refinement)

```
FORALL (n0:graph),(n1:graph):
 refines(n0,n1) AND silimp(inports(n0),inports(n1))
 IMPLIES
 silimp(outport(n0),outport(n1))
```

3.1 Compositionality

The axiom of refinement 3.3 allows us to provide a compositional proof of correctness of transformations: i.e., the refinement of component subgraphs of a graph `G` ensures the refinement of G. This allows us to assert that if a transformation is applied locally to a subgraph `g` of a graph `G`, leaving the rest of the graph unchanged, then the graph `G` undergoes a global refinement transformation.

Thus, we can state the following theorem:

Theorem 3.2 (Compositionality of Refinement)
Let g_1, g_2,...,g_N be the component subgraphs of `G`*, and g'_1, g'_2,...,g'_N the subgraphs of* `G´`*. If every g'_i is a refinement of g_i, then* `G´` *is a refinement of* `G`*. i.e.,*
($\forall$ i. refines(g'_i,g_i)) $\Longrightarrow$ refines(G',G)

Proof: The proof follows by induction on graph structure.

3.2 Verification of Transformations

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

1. Specify the structure of a SIL graph on which the transformation is to be applied.

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

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 `silimp` – 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 `sileq`: the sets of input values to both graphs are identical.

5. Verify the property that the outputs of the SIL graph before transformation are `silimp` – 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 `sileq`.

We illustrate this scheme in the cross jumping tail merging transformation mentioned in Section 1. A corrected version of the transformation is shown in Figure 8. In this 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.

In the course of our verification, we found the mistake in the informal specification of the transformation as shown in Figure 1. The error that occurred in the

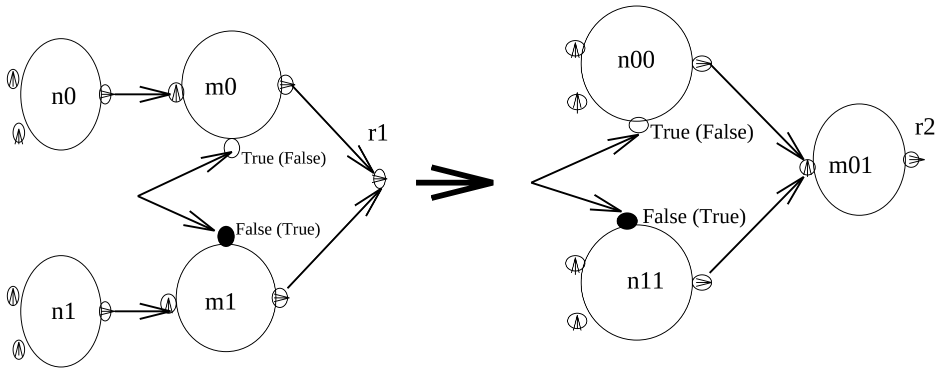


Figure 8: Cross jumping tail merging: corrected. Also, the concrete operations of Figure 1 such as "+" and "*" are replaced by general nodes.

original informal specification was the incorrect placing of the conditions on the nodes. With such a placing, as shown in Figure 1, the correctness of the transformation depends on the *ordering* of the output ports of the top "=" node and the bottom "=" node. When condition `c` is *true*, the values at `q11` and so `p11` are arbitrary, while the values at `q10` and `p10` are well-defined. Thus, if an ordering is imposed such that the port `r1` gets the value at `p11`, then that value would be arbitrary. However, in the transformed figure, the condition `c` being *true* results in an ordering such that `p2` gets the value of `q20` and the value of `q21` when `c` is *false*. Thus, the transformation would not be behavior preserving.

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 $m0$ and $m1$ in the original graph is the same as the ordering of the data flow edges coming into the node $m01$ in the optimized graph. We have suitably modified, generalized, and verified the transformation. The generalized transformation is shown in Figure 9.

The correctness result for the generalized cross jumping tail merging transformation of Figure 9 is stated as a theorem as follows:

Theorem 3.3 (Cross jumping tail merging)

```
FORALL par0,r1,r2,m0,m1,m01:
xdfe(par0,inports(m0)) AND xdfe(par1,inports(m1)) AND
(w(outport(m0),r1) < w(outport(m1),r1)
                  IFF
 w(par00,inports(m01)) < w(par11,inports(m01))) AND
dfe(outport(m0),r1) AND dfe(outport(m1),r1) AND
```

```
(FORALL pp: ((pp /= outport(m0)) OR (pp /= outport(m1)))
IMPLIES
             NOT dfe(pp,r1)) AND
dfe(par00,inports(m01)) AND dfe(par11,inports(m01)) AND
(FORALL par: (par /= par00 AND par /= par11)
IMPLIES
          NOT dfe(par,inports(m01))) AND
xdfe(outport(m01),r2) AND
sileq(par0,par00) AND sileq(par1,par11)
                IMPLIES
           sileq(r1,r2)
```

4 Mechanization in PVS

The Prototype Verification System (PVS) [OSR 93] 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 have specified and verified transformations such as copy propagation, constant propagation, common subexpression insertion, commutativity, associativity,

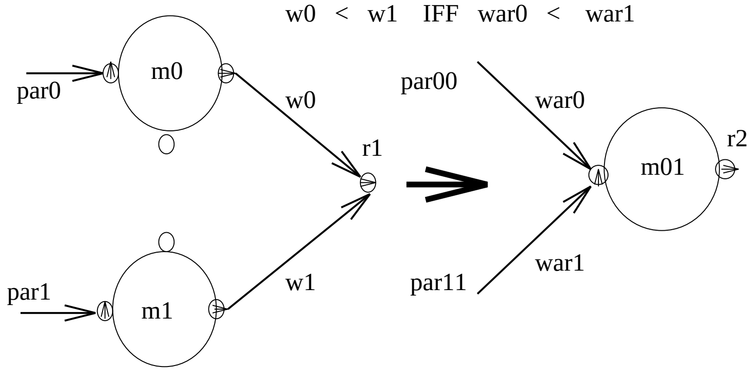


Figure 9: Cross jumping tail merging: generalized and verified. See Theorem 3.3

Transformation	Number of 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 1: Number of Inference rule applications for various transformations

distributivity and strength reduction described by Engelen and others [EMH 93] in PVS. 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 Boolean simplifier based on Binary Decision Diagrams (BDD) [BRB 90, Jan 93] integrated in PVS, is used to determine that the conjunction of Boolean expressions is indeed true. We show the number of major inference rule applications required for verifying the various transformations in Table 1.

5 Conclusions and Future Work

One of the goals of high level synthesis of hardware is to achieve designs *correct by construction*. In this work, we have attempted to help accomplish this goal by verifying the correctness of transformations used in high level synthesis, independent of the underlying behavior model. A formal characterization of a general control/data flow graph specification language has been achieved. We have given a small set of axioms that capture a general notion of refinement and equivalence of control/data flow graphs. Many transformations have been generalized by weakening the preconditions, and their correctness verified. We have specified and mechanically verified about a dozen of the optimization and refinement transformations. We found errors in this process, and suggested corrections. As part of future work, we would like to integrate our verification scheme with a high level synthesis CAD tool.

References

[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 et.al, *Event-Based Verification of Synchronous Globally Controlled, Logic Designs Against Signal Flow Graphs*, IEEE Trans. on CAD of ICs., Vol. 13, No. 1 Jan 1994.

[Ang 94] C. Angelo, *PhD Thesis*, IMEC, Leuven, Belgium, 1994.

[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.

[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.

[Cha 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.

[EMH 93] W.J.A Engelen, P.F.A. Middelhoek, C. Huijs, J. Hofstede, 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.

[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.

[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

[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, et al, *The SPRITE Input Language SIL-1, Language Report*, SPRITE, deliverable Ls.a.a / Philips / Y3 / M12 / 2, October 1992.

[Kro 92] Th. Krol et.al, *The SPRITE Input Language, An intermediate format for High Level Synthesis*, Proceedings of EDAC 92, Brussels, 16-19 March 1992, pp 186-192.

[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,

[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.

[Mid 94-1] 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.

[RAJ 94] Sreeranga P. Rajan, *Transformations on Dependency Graphs: Axiomatic Specification and Efficient Mechanical Verification*, PhD Thesis, University of British Columbia, July 1995.

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

[SOR 93] 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.

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

[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.