Formal Verification of Transformations on Dependency Graphs in Optimizing Compilers*

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

Abstract

Dependency graphs are used as intermediate representations in optimizing compilers and software-engineering. In a transformational design approach, optimization and refinement transformations are used to transform dependency-graph-based specifications at higher abstraction levels to those at lower abstraction levels. An informal representation would lead to subtle errors, making it difficult to guarantee the correctness of the transformations. In this work, we investigated the formal specification and efficient mechanical verification of transformations on dependency graphs.

1 Introduction

Dependency graphs[1] are used to model model data and control flow in software and hardware design. A dependency graph 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. In this work, we have used a general notion of dependency graph based on an intermediate representation used in the SPRITE project at Philips Research Labs [KMN+92].

*To appear in the Proceedings of California Software Engineering Symposium, Irvine, CA, March 30, 1995

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

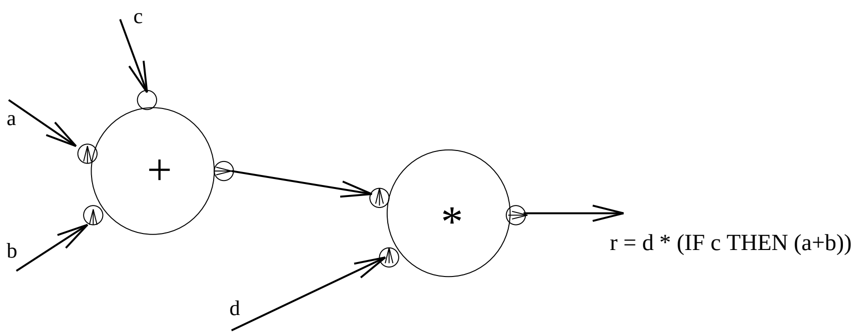

Figure 1: Example of a dependency graph with control specification.

We use Prototype Verification System (PVS) [SOR93] from SRI International for mechanical verification. In this work, we investigated the formal specification and efficient mechanical verification of transformations on dependency graphs. The specification and verification of the correctness of transformations we have presented here is independent of the behavior model underlying dependency graphs.

A typical transformation employed in optimizing compilers is *cross-jumping tail-merging* [EMH+93], shown in Figure 2. In this transformation, two identical nodes on dependency paths that are never active at the same time are merged into one node. However, as we found out using our approach explained in this paper, the transformation does not preserve behavior. Informally, the reason is as follows. The conditions True/False on open/filled bullets in Figure 2 on the bottom/top of the nodes control the execution of the nodes. Further, we stipulate[2] that the value of `p2` is determined by the edge that transmits a well-defined value to it - i.e. when `c` is *true*, `q20` determines the value of `p2`, while `q21` determines its value otherwise. In graph G1, when `c` is *false*, the value of `q10` is arbitrary, and so is the value of `p10`. If we choose the value of `r1` to be that of `p10`, the value of `r1` is also arbitrary. In graph G2, when `c` is *false*, the value of `p2` is that of `q21`. In this case, the value of `r2` is `(y1 * y2)`. Thus, the set of outputs for a given 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 5.

The main contributions of this work are the following:

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

[2]Section 3.2 explains this stipulation on how data conflicts at ports called *joins*, such as p2, are resolved.

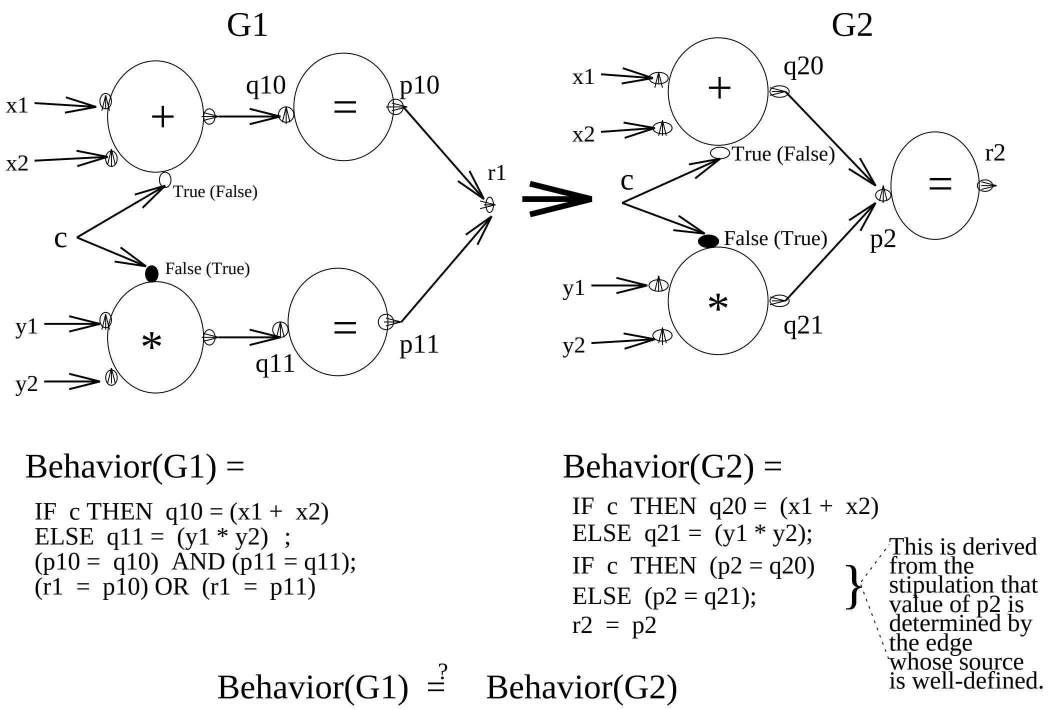

Figure 2: Cross jumping tail merging: incorrectly specified in informal document.

- A set of optimization and refinement transformations on dependency graphs have been mechanically verified. Generalization of transformations have also been proposed.
- Errors have been discovered in the transformations used in an industrial setting. Modifications for the erroneous transformations have been incorporated.

The rest of the paper is organized as follows. In Section 1.1, we discuss some of the past efforts in verification of transformations in dependency-graph-based formalisms. An overview of dependency graphs is given in Section 3. Section 2 provides a brief overview of PVS. We present the formal specification of dependency graphs in Section 4, followed by a short account of verification of transformations[3] in Section 5. Finally, we present the conclusions in Section 6.

1.1 Related work

There have been some efforts in analysis and verification of refinement transformations in the past. One of the earliest efforts on formalizing the correctness of

[3] A more detailed treatment of specification and verification transformations is given in a forthcoming thesis [Raj].

optimization of transformations is by Aho, Sethi and Ullman [ASU71]. They formalized a restricted class of transformations known as Finite Church-Rosser (FCR) transformations, and provided simple tests to check properties of such transformations. However, none of the other past work have dealt with transformations on dependency graphs in general. Most of the efforts have concentrated on specialized programming and specification languages languages [Ang94, McF93]. Many other efforts have provided a formal operational and denotational semantics for variations of dependency graph formalisms [dJ93,HK93,PBJ+91,CBL92]. But, the operational and denotational semantics, unlike axiomatic semantics, are tied to a specific behavior model. This makes them unsuitable to verify transformations on dependency graphs of arbitrary size. Further, it would not be possible to extend such concrete operational or denotational models to dependency graphs used in a multitude of formalisms such as in software engineering and real-time systems modeling.

2 Overview of PVS

The Prototype Verification System [4] (PVS) [SOR93, ORSvH95] 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.

PVS has been used for reasoning in many domains, such as in hardware verification [Cyr93,CRSS94,Raj94], protocol verification, and algorithm verification [LOR+93, ORSvH95]. We briefly give the features of the PVS specification language in Section 2.1, the PVS verification features in Section 2.2 and some notes on the syntax of the PVS specification language in Section 2.3.

2.1 PVS Specification Language

The specification language 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. The PVS specification language allows

[4] Information on PVS is accessible through World Wide Web: <http://www.csl.sri.com/sri-csl-fm.html>

embedding of special logics. In particular, property specifications can be expressed in temporal logics specialized for specific hardware/software/real-time applications.

2.2 PVS Verification Features

The PVS verifier 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) based state-space exploration may be invoked for model-checking and Boolean reasoning. It also features a variety of general induction schemes to tackle large-scale verification. Moreover, different verification schemes can be combined into general-purpose strategies for similar classes of problems, such as verification of microprocessors [Cyr93, CRSS94].

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.

2.3 Notes on Specification Notation

In PVS specifications an entity followed by a colon and a type indicates that the entity 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

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

```
0,1,2
x: integer | even(x) AND x /= 2
```

The symbol | 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 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:

```
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 Overview of Dependency Graphs

A dependency graph consists of nodes representing operations or processes, and directed edges representing data and control flow through the represented system. The behavior of a dependency graph is derived from the behaviors of structural building blocks of the graph. We describe the structural aspects in Section 3.1 and behavioral aspects in Section 3.2

3.1 Structural Aspects

The basic building blocks of a dependency 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 the following two kinds of edges:

- 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 dependency graph. A dependency graph itself can be viewed as one single node, and used to construct another dependency graph in a hierarchical manner. Figure 3 is an example of a general dependency graph.

3.2 Behavioral Aspects of Dependency Graphs

The behavior of a dependency 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.

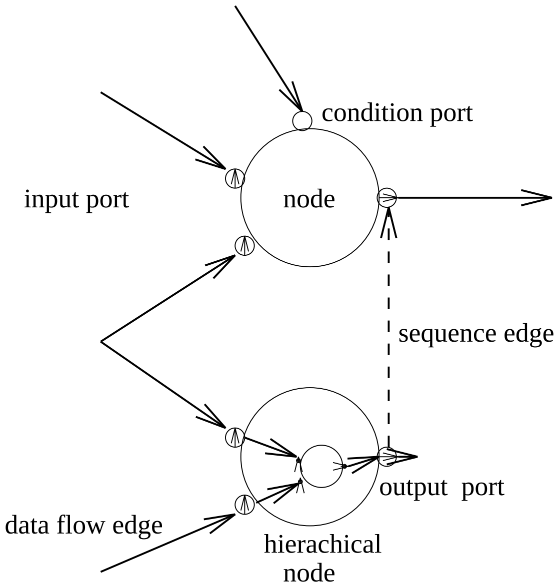

Figure 3: A general dependency graph

The ordering on data value 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*. 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 data value determines the behavior of the join. We stipulate that 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.

3.3 Transformations on Dependency Graphs

There are a variety of transformations described in literature [AC72, EMH$^+$93] for optimization and refinement of dependency graphs. Many of the optimization transformations are inspired by optimizing compiler techniques such as *common subexpression elimination*, *cross jumping tail merging* and algebraic transformations involving *commutativity*, *associativity*, and *distributivity*. Other optimization transformations include *loop transformations*. Refinement transformations include type transformations such as *real to integer*, *integer to Boolean*, and implementing data relations of the nodes by concrete operators [Mid94].

4 Formal Specification of Dependency Graphs

At the level of abstraction we have chosen to specify behavior, we bring about high-level properties of dependency graphs, refinement and equivalence that should hold independent of a detailed behavior model[5]. We can thus obviate the need to specify a concrete behavior model of dependency graphs. 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 asserted as axioms. While, other properties are derived as theorems by applications of inference rules on other theorems or axioms. 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 5.

We introduce an abstract refinement relation `silimp`, as a relation on ports[6] 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 dependency graphs *sileq* is defined by introducing the symmetry property in the refinement relation `silimp`:

Definition 4.1 (Equivalence)

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

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.

[5] In this paper we highlight only the key axioms and theorems due to space limitation.

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

A function `w` on ports would return a weight, that could be a number:

Definition 4.2 (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 dependency specifies that no two data flow edges communicate tokens simultaneously into a join, and no two weights on the edges forming a join can be equal. This suggests that we need a reflexive, transitive, and antisymmetric ordering relation on weights: such a relation is called *partial order*. We define a partial ordering relation $<=$ on weights, and assert the fact that the weights are ordered 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 4.1 (Partial order on weights: linear order on edge weights)

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

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:

Axiom 4.2 (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 4.1 (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 4.3 (Graph refinement)

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

4.1 Compositionality

The axiom of refinement 4.3 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 4.2 (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. The details are omitted due to space limitation.

5 Formal Verification of Transformations

In dependency-graph-based formalisms, there are a number of optimization and refinement transformations [EMH+93, PBJ+91], such as optimization transformations are inspired by optimizing compiler 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 [Mid94].

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

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

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

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

4. In the case of verifying refinement, we impose the constraint that the corresponding inputs of the dependency graphs before and after transformation are `silimp` – that is, the set of input values to the dependency graph after transformation is a subset of the set of input values to the dependency 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 dependency graph before transformation are `silimp` – that is, the outputs of dependency graph after transformation are refinements of corresponding outputs of the dependency 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 4. 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.

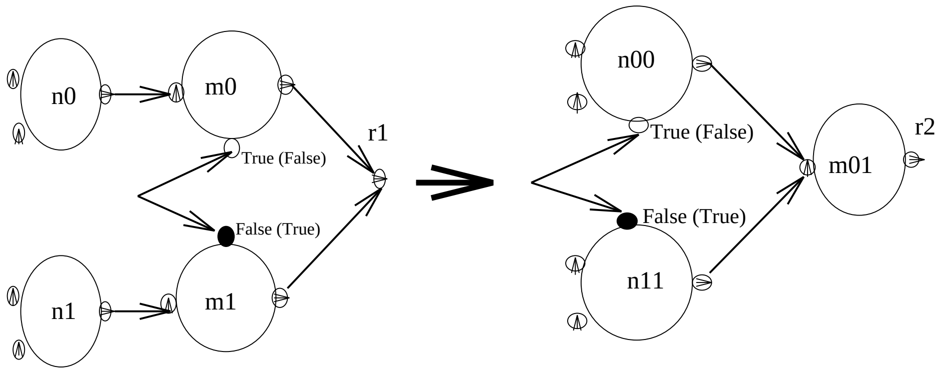

Figure 4: Cross jumping tail merging: corrected. Also, the concrete operations of Figure 2 such as "+" and "*" are replaced by general nodes.

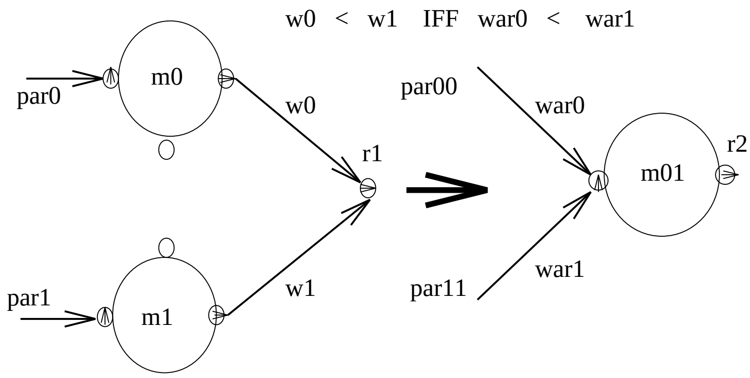

Figure 5: Cross jumping tail merging: generalized and verified. See Theorem 5.1

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 5. The cross jumping tail merging transformation is stated as a theorem as follows:

Theorem 5.1 (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
```

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

Table 1: Experimental results for proofs of various transformations on a Sparc-20 with 32 Mb memory

```
 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)
```

We show the experimental results for verifying the various transformations in Table 1. The proofs are semi-automatic. A few intermediate steps such as instantiation of existentially quantified variables need manual interaction, while other steps are automatic.

6 Conclusions and Future Work

One of the goals of a transformational approach is to achieve programs that are *correct by construction*. 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. We found errors in existing transformations, and suggested corrections. Furthermore, we have suggested generalization of transformations. The transformations we have verified are being used in industry in optimizing compilers and high-level synthesis of hardware. Applications of our work on dependency graphs in other fields such as real-time systems and software engineering have been described in a forthcoming thesis [Raj]. As part of future work, we would like to investigate the correctness of more complex transformations involving scheduling, and a tighter integration of formal methods in optimizing compilers and software engineering. For example, we have investigated the correctness of the transformations "off-line" (i.e. not the actual code that implements the transformations). We would like to develop an automatic verification method that works on real code implementing the optimizing transformations, based on our proof strategies.

Acknowledgments

The author thanks the anonymous reviewers for constructive comments. The author is grateful to N. Shankar for suggestions on an earlier draft of the paper.

References

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

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

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

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

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, pages 287–305, Bad Herrenalb (Blackforest), Germany, September 1994. Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.

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

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

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

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

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

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

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

[Mid94] P.F.A. Middelhoek. Transformational design of digital signal processing applications. In *Proceedings of the ProRISC/IEEE workshop on CSSP*, pages 175–180, Eindhoven The Netherlands, March 1994.

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

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

[Raj] Sreeranga P. Rajan. *Transformations on Dependency Graphs: Formal Specification and Efficient Mechanical Verification.* PhD thesis, University of British Columbia, Vancouver, Canada, forthcoming.

[Raj94] P. Sreeranga Rajan. Transformations in high level synthesis: Specification and verification. Technical Report NL-TN 118/94, Philips Research Laboratories, Eindhoven, The Netherlands, April 1994.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.