Verified High-Level Synthesis in BEDROC

Richard Chapman
Computer Science Department
Cornell University
Ithaca, NY 14853

Geoffrey Brown, Miriam Leeser
School of Electrical Engineering
Cornell University
Ithaca, NY 14853

Abstract

We show how formal semantics that we have given for representation languages used in the BEDROC high-level synthesis system are used to prove that the designs produced by the system conform to the input specification. We discuss proving correctness of the translation from behavioral specification to intermediate form, proving correctness of optimizations, and plans for proving correctness of scheduling.

1 Introduction and related work

We have given formal semantics to the specification language and intermediate form used in the BEDROC hardware synthesis system, and we are using these semantics in the proofs that the designs produced by the system conform to their specifications. Such proofs increase the confidence of a circuit designer in the correctness of the system's output, without the circuit designer having to learn verification techniques and apply them to each design.

There have been two efforts to give formal semantics to working high-level synthesis systems. Michael McFarland gave a predicate transformer-based semantics using behavior expressions to the hardware description language ISPS [10], based on previous work with Parker [9]. He used that as a basis for giving a similar semantics to the intermediate representation for ISPS, the Value Trace(VT) in the System Architect's Workbench. Raul Camposano also verified transformations used to improve scheduling and datapath allocation in the Yorktown Silicon Compiler [2].

Brown and Leeser [6] give operational and denotational semantics to an earlier version of HardwarePal to verify that translation to a three-address code intermediate form is correct. Unfortunately, the intermediate form chosen is not well suited for efficient optimization or scheduling, as it contains no dependence information.

First, we present the HardwarePal hardware description language and formal operational and denotational semantics for it, briefly discussing our proof of the two semantics' equivalence. We then discuss our intermediate representation, dependence flow graphs (dfg's) [11], and the operational semantics of dfg's. We describe the translation from HardwarePal to dependence flow graphs and outline our proof that this translation preserves the meaning of the initial HardwarePal program. We conclude by discussing our plans for proofs that register-transfer level design produced by BEDROC implements the dependence flow graph.

2 HardwarePal

A HardwarePal program is a behavioral description of hardware given as an algorithm in an imperative, Pascal-like language. The parser outputs an abstract syntax tree, which the dependence analyzer uses to construct a dependence flow graph containing the necessary data and control flow information to do scheduling and datapath allocation. The dfg can also be input to a simulator that produces C code to simulate the circuit at a behavioral level, if desired. The optimizer performs dfg-to-dfg transformations.

Recursion is not allowed in HardwarePal, and all procedure and function calls are expanded in the front-end. HardwarePal is strongly typed. The types and bit-widths of all variables and I/O ports are declared in the program. Types supported in HardwarePal include integers, booleans, input, and output ports. Identifiers serve only to transport data from its definition point to its uses; decisions about register and wire allocation are made at lower levels of the system. Control structures are if, while, repeat, for, and case. No notion of communicating processes is supported in HardwarePal in the present system, but we
hope to add that capability in the future. An example HardwarePal program is shown in Figure 1.

```
program test;

var inA, inB: input[4];
var x, y: integer[4];
var outC, outD: output[4];

begin
  repeat
    begin
      (read(x, inA) || read(y, inB));
      if y=1 then
        write(x, outC)
      else
        write(x, outD)
    end
  until y=0
end.
```

Figure 1: A HardwarePal program

## 3 HardwarePal semantics

For purposes of our correctness proofs, we give an operational semantics for HardwarePal. We found that it was also easy to give an equivalent denotational semantics to the language. We believe that these two characterizations of HardwarePal give a good basis for such proofs. In particular, we find them superior to using a Hoare Logic or predicate-transformer based semantics for that purpose. While a Hoare logic can be used to prove that transformations within a single language preserve certain properties, it seems cumbersome to us as a tool for proving that translations between languages preserve properties.

The operational rules describe state transitions. The state \(< C, \rho, i, \omega >\) consists of a compound command \(C\), an environment \(\rho\), a mapping of input port names to streams of values \(i\) and a similar mapping of output port names to streams of values \(\omega\). The environment also stores the width in bits of each identifier (for brevity, we ignore that here). The streams of values for the input ports must be supplied as part of the initial environment. The initial state, for some program \(C_1\) whose input port names are \(\{i_1, i_2, ..., i_n\}\), whose output port names are \(\{o_1, o_2, ..., o_m\}\) and whose variables are \(\{x_1, x_2, ..., x_k\}\) is:

\[
< C_1, \emptyset, \{ (i_j \rightarrow < v_{j1}, v_{j2}, ... > | j = 1..n),
            \{(o_j \rightarrow <>)| j = 1..m} >
\]

A transition rule is of the form:

\[
\text{condition} \quad \Rightarrow \quad < C, \rho, i, \omega > \rightarrow < C', \rho', i', \omega' >
\]

where \(\text{condition}\) is a predicate involving \(C, \rho, i, \omega\). Examples of transition rules are given in Figure 2.

We find HardwarePal's denotational semantics (example rules given in Figure 3) to be straightforward, giving us confidence that our operational semantics is also reasonable. The only arguments needed by the semantic function are the program, a map from identifiers to values, and the values of the I/O ports.

We have proven the operational and denotational semantics to be equivalent[3]: the configuration reached at the end of abstract execution starting in some initial configuration is the same as the value returned by applying the denotational function to the program and the initial condition. We have two theorems to demonstrate this, one for each direction of the bi-implication:

For all \(C \in \text{Comm}, \forall \rho \in \text{Env}, \forall i, \omega \in \text{Port}:

\[
(\langle C, \rho, i, \omega \rangle \Rightarrow \langle C', \rho', i', \omega' \rangle) \iff
(\langle C \rangle(\rho, i, \omega) = \langle C' \rangle(\rho', i', \omega'))
\]

Proof of the \(\Rightarrow\) direction is by induction on the length of the sequence of states derived via the operational rules and is straightforward. Proof of the \(\Leftarrow\) claim is by structural induction on \(C\). The loop case is proven by fixpoint induction.

## 4 Dependence flow graphs

BEDROC incorporates an intermediate representation developed for optimizing, parallelizing compilers: the dependence flow graph. In contrast, most current high-level synthesis systems use a control flow graph and a collection of data flow graphs as an intermediate form for scheduling and data path allocation. Like such forms, dfgs support efficient implementation of optimizations, such as constant propagation, dead code elimination, and loop unrolling. Dependence flow graphs (dfgs) have an important advantage: they possess a compositional formal semantics, allowing easy verification of the building the dfg, of optimizing transformations, and of scheduling.

We extend the dependence flow graph model developed for compilers to handle dependencies between I/O operations in addition to the value, output, anti, and flow dependencies present in programs. Figure 4 contains the dependence flow graph for the program in Figure 1. The lighter arcs represent the dependencies that exist between I/O port accesses, and the solid
5 Translation to dfg

The algorithm proceeds in two phases; in the first phase we generate a dependence flow graph closely following a sequential execution of the HardwarePal program, by doing a single walk of the abstract syntax tree of the HardwarePal program. We proceed control region by control region, routing arcs for each identifier into and out of every control region. Within a control region we generate the directed acyclic graph for the computation and I/O that occurs there. In the second phase we increase parallelism in the dfg by doing dfg-to-dfg transformations.

There are a number of clear problems with the initial dfg: it routes too many dependence arcs into control regions, and it delays operations by routing their dependences through control regions in which are not used, delaying scheduling of those operations. Our reasons for adopting this algorithm are that it is fast and easy to verify, and that the transformations required to optimize and parallelize it are also fast and easy to verify.

6 Proving the translation correct

Given operational semantics for both HardwarePal and dependence flow graphs, we want to show:

\[
\forall C \in \text{Comm} \Rightarrow \forall (\rho, t, \omega) \in \text{Env} \times \text{Port} \times \text{Port}.
\]

\[
(< C, \rho, t, \omega > \Rightarrow (\rho', t', \omega')) \iff (3p'' \in \text{Env.abs.ex}(\text{dfg}(C), (\rho, t, \omega)) = (\rho'', t', \omega'))
\]

where by \(\text{abs.ex}(\text{dfg}(C), (\rho, t, \omega))\) we mean the final configuration arrived at by abstract execution, according to the operational semantics, of the dfg computed from the HardwarePal command \(C\).

That is, for any initial configuration \((\rho, t, \omega)\) abstract execution of a HardwarePal program \(P\) gives rise to a final configuration \((\rho', t', \omega')\) if and only if abstract execution of the dfg for \(P\) arrived at by the algorithm of the previous section gives rise to a final configuration \((\rho'', t', \omega')\) in which the I/O ports have...
There are two implications to be proven in this claim. To show the \(\Rightarrow\) direction, we use induction on the length of the HardwarePal program's abstract execution sequence. In fact, we show something a little stronger: that if the starting configurations correspond in the necessary way, we can apply one rule to the HardwarePal program, then apply some rules to the dfg, and the resulting configurations will remain in correspondence. To make the proof work, as we run the dfg-generating algorithm we save information telling, for each node and arc in the dfg, which statement or variable in the HardwarePal program it came from. We can then use this information to order the application of rules in the abstract execution of the dfg. With our operational semantics for HardwarePal, the abstract execution sequence is deterministic. However, several rules may apply to a single configuration of a dfg. We know that the order of application of applicable rules will not affect the final configuration ([11], p.20, Theorem 2). The intuition behind the ordering of dfg rule application we choose is that we want the abstract executions of HardwarePal program and dfg to proceed in a lock-step fashion: for each rule application to the HardwarePal program, execute the dfg rules for the nodes corresponding to the HardwarePal statement executed, then show that the configurations still have the desired correspondence.

To prove the \(\Leftarrow\) direction, starting with an arbitrary abstract execution of the dfg, we need to show that the HardwarePal program's execution gives the same final configuration. To do this we use Pingali's theorem ([11], p 18) to re-order abstract execution of the dfg.

In the second phase we parallelize the dfg with these optimizations:

- If a dependence or value arc is not used in either branch of a conditional, replace the switch/merge nodes with a single arc going around the conditional.
- If a dependence or value arc is not used in a loop body, route the arc around rather than through the loop.
- If an arc is not used from a given point until the end of the program, strip it out of all control regions back to its last use.

To achieve maximum parallelism in the resulting dfg, the first two optimizations should be carried out in a single pass over the dfg, working from innermost loops to outermost loops.

Other optimizations implemented on the dependence flow graph include constant propagation, common sub-expression elimination, dead code elimination, unreachable code elimination, loop invariant removal, strength reduction, loop interchange, loop unrolling.
Because the dfg has a local execution semantics, one can show that an optimization is correct by showing that the sub-graph of the dfg that is modified has the same behavior before and after the transformation. That is,

\[(dfg_2 = \text{optimize}(dfg_1)) \iff (\exists \omega', \rho', \rho'' \forall \rho_1, \omega, (\text{abs.ex}(dfg_1, (\rho, \omega)) = (\rho', \omega')) \wedge (\text{abs.ex}(dfg_2, (\rho, \omega)) = (\rho'', \omega')))\]

7 Translation to RTL

The BEDROC scheduler uses an $RTA^*$ algorithm; the datapath and register allocate use clique partitioning. Together, these generate a register-transfer level design. To show equivalence of a dfg to a RTL design, some semantics for the RTL-level description language must be given. Several semantics have been provided for RTL languages [5][4], but not much has been done to show translations to and from the RTL language preserve any desirable properties.

The abstract execution of the RTL design, for given input sequences, is deterministic. For the dfg, several rewrite rules may be applicable at any time. The scheduling of dfg operations is an extension of the partial order given in the dfg. To show semantic equivalence, one must show the existence of an ordering of the application of rewrite rules to the dfg such that the configurations reached match up at some important points, such as the end of each machine cycle, with configurations reached via abstract execution of the RTL design.

8 Plans and acknowledgements

We aim to verify the BEDROC system at all levels. Proofs for the scheduling and data path allocation programs are the next focus of our effort. Finally, we would like to acknowledge Richard Johnson and Micah Beck for writing the dfg-processing software for programs that we started with, and David Cooper for writing the HardwarePal parser and lexical analyzer.

References


