Verification of Microarchitectural Refinements in Rule-based Systems

Nirav Dave*, Michael Katelman†, Myron King*, Arvind*, José Meseguer†
* Massachusetts Institute of Technology - Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, U.S.A.
{ndave, mdk, arvind}@csail.mit.edu
† University of Illinois at Urbana-Champaign - Department of Computer Science
Urbana, IL 61801, U.S.A.
{katelman, meseguer}@uiuc.edu

Abstract—**Microarchitectural refinements are often required to meet performance, area, or timing constraints when designing complex digital systems. While refinements are often straightforward to implement, it is difficult to formally specify the conditions of correctness for those which change cycle-level timing. As a result, in the later stages of design only those changes are considered that do not affect timing and whose verification can be automated using tools for checking FSM equivalence. This excludes an essential class of microarchitectural changes, such as the insertion of a register in a long combinational path to meet timing. A design methodology based on guarded atomic actions, or rules, offers an opportunity to raise the notion of correctness to a more abstract level. In rule-based systems, many useful refinements can be expressed simply by breaking a single rule into smaller rules which execute the original operation in multiple steps. Since the smaller rule executions can be interleaved with other rules, the verification task is to determine that no new *behaviors* have been introduced. We formalize this notion of correctness and present a tool based on SMT solvers that can automatically prove that a refinement is correct, or provide concrete information as to why it is not correct. With this tool, a larger class of refinements at all stages of the design process can be easily verified. We demonstrate the use of our tool in proving the correctness of the refinement of a processor pipeline from four stages to five.**

I. Introduction

Modular refinement is an important technique in designing complex digital systems because it eases architectural exploration for better performance, area, and power. For modular refinement to be viable it should be relatively easy to determine if a local change preserves the overall correctness of the design. Generally, it is extremely difficult for a designer to give a full formal correctness specification for a system. Specifying correctness requires a level of knowledge of the overall system and familiarity with formal verification methods that few designers possess. As a consequence, common practice is to settle for partial verification via testing. Testing works, but as test suites tend to be built in conjunction with the design itself, designers rarely gain sufficient confidence in their refinements' correctness until the final stages of the design cycle.

An alternative is to restrict the types of refinements to ones whose local correctness guarantees that the overall behavior will remain unaffected, and designs usually rely on the notion of equivalence supported by the design language semantics for proving or testing local equivalence. As most hardware description languages describe synthesizable systems at the level of gates and wires, this amounts to FSM (finite-state machine) equivalence. Tools usually require the designer to specify the mapping of state elements (*e.g.,* flip-flops), and thus reduce the problem of FSM equivalence to combinational equivalence, which can be performed efficiently. FSM-equivalence-preserving refinements have proven to be quite useful because tools are available to prove the local correctness automatically and there is no negative impact on the overall verification strategy. However, FSM refinement is too restrictive, disallowing many desirable changes such as adding a buffer to cut a critical path in a pipeline. Thus these tools are limited to verification in the later stages of design when the timing has been decided.

Recently, languages like Bluespec [2], which describe designs not as gates and wires but as a set of guarded atomic actions (or *rules*) on state elements, have been proposed. Over the last six years, it has been established that Bluespec programs not only can produce no-compromise hardware [1], but that keeping programs at the rule level allows more flexibility in design and refinements [4]. For instance, the addition of a pipeline stage can be implemented in a natural way by splitting the rule corresponding to the appropriate stage into multiple rules, and introducing state to hold the intermediate results.

A Bluespec program can be reasoned about at two levels. At the first level we deal with rules in an unscheduled manner. The semantics state that we compute by selecting any valid rule (*i.e.,* one whose guard evaluates to true) for execution, update the state by executing the body of the rule, and then repeat the process. This means that the program is naturally non-deterministic, and programs at this level are meant to be correct for all possible traces of execution. At the second level the compiler adds a scheduler which is responsible for resolving the non-determinism so that we may synthesize the program into a high-quality FSM implementation. The choice of scheduler is a purely performance-based concern and should not affect the correctness. We exploit this separation of concerns and focus on the equivalence of systems before scheduling.

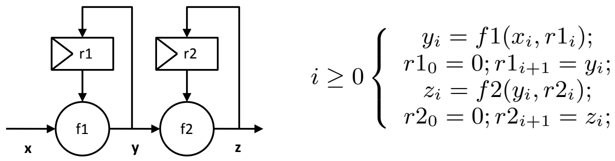


Fig. 1. Initial FSM

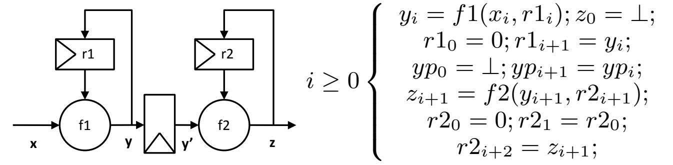


Fig. 2. Refined FSM

Despite the guarded atomic action formalism's deep relation to term-rewriting systems and formal proofs, little work has been done to verify rule-based programs at anything beyond the implementation level. *The main contribution of this paper is to define a notion of simulation between rule-based programs and describe an SMT-based algorithm that automatically verifies this notion of simulation*. A tool based on this algorithm is able to prove the correctness of interesting refinements in a matter of tens of seconds, well within range to be useful as a debugging aid for the designer. We use the tool to show the correctness of several examples including the refinement of a four-stage processor pipeline into a five-stage pipeline.

Paper Organization: In Section II, we discuss the kinds of refinements we want to make and why their correctness cannot be formulated at the FSM level. We also discuss the challenge of verification at the level of rules and discuss how nondeterministic specifications affect the verification task. In Section III, we formalize a notion of equivalence in the context of rule refinements. In Section IV, we discuss the algorithm used by our tool to mechanically verify equivalence using an SMT solver. In Section V, we discuss the verification of a processor program. In the last section we discuss related work and present our conclusion.

II. Motivating Refinement Example

To understand the challenges of refinement in rule-based systems we must first understand how such refinements differ from refinements of FSMs, motivating our notion of behavior and explaining where the new method and tool are needed.

A. Refining an FSM

Consider the hardware represented by the FSM system shown in Figure 1. The system consists of two registers `r1` and `r2`, both initially zero, and some combinational logic implementing functions `f1` and `f2`. The critical path in this system goes from `r1` to `r2` via `f1` and `f2`. In order to improve performance, a designer may want to break this path by adding a buffer (say, a one element FIFO) on the critical path as shown in Figure 2. Though we have not shown the circuitry to do so, we will assume that `r2` does not change and the output `z` is not defined when the FIFO is empty.

In this refined system, the operation that was done in one cycle is now done in two; `f1` is evaluated in the first cycle, and `f2` in the second. The computation is fully pipelined so that each stage is always productive (except the first cycle of the second stage, when the FIFO buffer is empty) and we have the same cycle-level computation rate. However the clock period in the refined system can be much shorter, thereby increasing system throughput. Though the cycle-by-cycle state of the two FSMs do not match directly, a little bit of analysis will show that the sequence of values assumed by `r2` and `z` are the same in both systems. In other words, the refined system produces the same answer as the original system but one cycle later. Therefore, in many situations such a refinement may be considered correct even though the FSMs of the two systems are not equivalent.

The problem here is that if we don't rely on FSM equivalence then how should we define equivalence? A solution could be to introduce the notion of a message or valid input and output and then define equivalence in terms of input-output sequences of messages as opposed to cycle-by-cycle behavior of input and output. Such a technique is not likely to work for sub-components of a synchronous circuit. In the following section we discuss a rule-based description of this example and show how refinements are expressed in such systems.

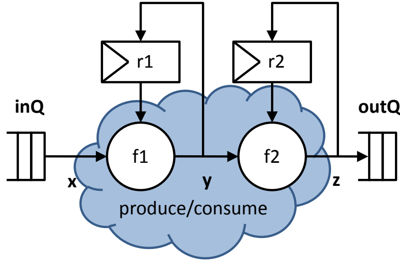
B. Refinements at the Rule Level

When designers specify systems using rules, they often have in their mind a particular datapath and FSM, although the exact datapath and FSM is generated by the compiler. For example, a designer may express the FSM design in Figure 1 using a single rule as shown in Figure 3. While it is reasonable to deal with streams of inputs in FSMs, it makes more sense in rule-based programs to think of input and output in terms of queues which we have added. For simplicity we can assume that `inQ` is never empty and `outQ` is never full. If we assume that a rule executes in one clock cycle then the rule in Figure 3 specifies that every cycle `r1` and `r2` should be updated, one value should be dequeued from `inQ`, and one value should be enqueued in the `outQ`. (The approximate logic generated by each rule is shown as a cloud in all the figures; we have omitted the control logic to avoid clutter.)

An *execution* in a rule-based program can be thought of in terms of the sequence of values assumed by various state elements. Assuming `inQ` has values x_0, x_1, x_2, ..., we will observe the sequence of states shown in Figure 4.

The sequences of values for `r1` and `r2` match exactly with those in Figure 1. Similarly, the values on the wires `x` and `z` which serve as input and output in the original program match the sequence of values in `inQ` and `outQ`.

The refined FSM in Figure 2 may be described by splitting our single rule into two rules: `produce` and `consume`, which communicate via the FIFO `q` as shown in Figure 5.




```
register r1 = 0, r2 = 0
fifo inQ, outQ;
rule produce_consume when (!inQ.empty() && !outQ.full()):
  let x = inQ.first(); inQ.deq();
  let y = f1(x,r1); let z = f2(y,r2);
  r1 := y;  r2 := z; outQ.enq(z);
```

Fig. 3. A Rule-based Specification of the Initial Program

$$
\begin{array}{ll}
([x_0, x_1, x_2, ...], r1_0, r2_0, []) \longrightarrow & \text{where: } r1_0 = 0; r2_0 = 0; \\
([x_1, x_2, ...], r1_1, r2_1, [z_1]) \longrightarrow & \quad r1_{i+1} = f1(x_i, r1_i); \\
([x_2, ...], r1_2, r2_2, [z_1, z_2]) \longrightarrow & \quad r2_{i+1} = f2(r1_{i+1}, r2_i); \\
... & \quad z_i = r2_i
\end{array}
$$

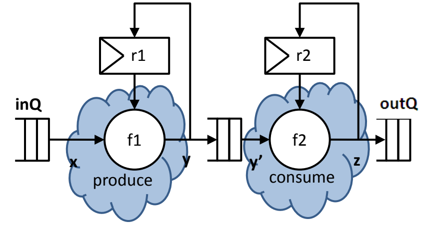
Fig. 4. The behavior of the program in Figure 3. State is represented by quadruples where the first and final member are the contents of `inQ` and `outQ`, and the second and third members are the values of `r1` and `r2`

The first thing to understand about this two-rule program is that it represents a nondeterministic specification which can be implemented by many different FSMs. For multiple rule programs, the semantics only states that any *enabled* rule (*i.e.,* a rule in a state where its guard is true) can be executed; it does not determine which rule to choose if more than one is enabled. The following are possible schedules for this program:

Schedule 1	Schedule 2	Schedule 3
produce	produce	produce
consume	produce	produce
produce	consume	consume
consume	consume	produce
...	...	consume
...	...	...

In the first schedule the program repeatedly enters a token into the FIFO and then immediately takes it out. This emulates the execution of the rule in the unrefined program (Figure 3) and leaves the FIFO `q` empty after each `consume` rule execution. This schedule also does the same set of updates to registers `r1` and `r2` as the original program. The second schedule repeatedly queues up two tokens before removing them. Note that, this schedule will be valid only if `q` has space for two tokens. In the third schedule, except when the program starts, there will always be at least one token in `q`.

In case of multiple-rule programs, the *behavior* of the program must be thought of in terms of the set of *permitted executions* or more precisely, the set of the sequences of values assumed by various state elements. A *scheduler* picks a specific execution from this set. A schedule is chosen by the compiler (with voluntary inputs from the designer) based on some goodness criteria. The current Bluespec compiler [6]




```
register r1 = 0, r2 = 0
fifo q, inQ, outQ
rule produce when (!q.full() && !inQ.empty()):
  let x = inQ.first(); inQ.deq();
  let y = f1(r1,x);
  q.enq(y); r1 := y
rule consume when (!q.empty() && !outQ.full()):
  let y = q.first;  q.deq();
  let z = f2(y,r2);
  outQ.enq(z); r2 := z;
```

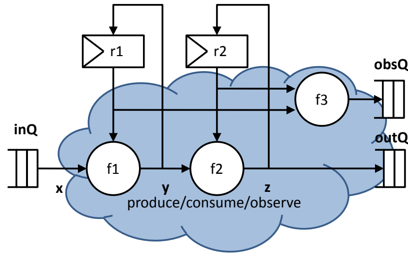
Fig. 5. A Refinement of the Program in Figure 3

schedules as many enabled rules as possible every cycle as long as the rules do not conflict with each other. The behavior produced by a parallel scheduler must be consistent with some one-rule-at-time schedule. For the example at hand the Bluespec compiler will schedule only the producer in the first cycle and then repeatedly schedule consumer followed by producer in the each subsequent cycle. If the compiler is implemented correctly, parallel scheduling cannot introduce any new behaviors by definition. Thus, for verification purposes we only have to consider executions caused by one-rule-at-time schedulers.

How should we compare the behaviors of our original program (Figure 3) and the refined program (Figure 5) since they have different sets of state elements? Both programs have `r1` and `r2` but in the two programs they get updated at different times and may not match. Intuitively we know that if `q` contains an element, then we are part way through at least one round of an equivalent `produce_consume` atomic computation and `r1` and `r2` will appear out of sync in the two programs. Conversely, whenever `q` is empty we should be able to draw a correspondence between the executions of the two programs.

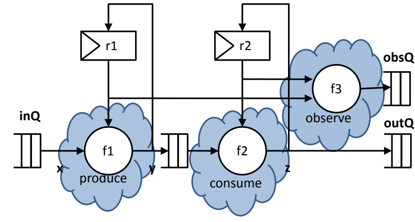
We can guarantee that the refined program does not add new behaviors if we can show that for any execution in the refined program, whenever it reaches a state where `q` is empty we can find a corresponding execution in the original program which has the same values for the matching state elements, *i.e.,* `r1` and `r2`. To guarantee that we haven't lost behaviors we must also show the converse: namely that any computation in the original program can be mimicked by the refined program. This is quite easy to show because `produce` followed by `consume` behaves exactly as `produce_consume`.

We first offer an intuitive reason why the original program can mimic the refined one). Consider those prefixes of a schedule in the refined program which have equal number of `produce` and `consume` rule executions. At the end of such a prefix, `q` must be empty. Further, since `produce` always




```
register r1 = 0, r2 = 0
fifo inQ, outQ, obsQ;
rule produce_consume_observe when (!inQ.empty()
                                   && !outQ.full() && !obsQ.full()):
  let x = inQ.first(); inQ.deq();
  let y = f1(x,r1); let z = f2(y,r2);
  let a = f3(r1,r2);
  r1 := y; r2 := z;
  outQ.enq(z); obsQ.enq(a);
```

Fig. 6. Program of Figure 3 with an Observer




```
register r1 = 0, r2 = 0
fifo q, q1, inQ, outQ, obsQ
rule produce when (!q.full()):
  let x = inQ.first(); inQ.deq();
  let y = f1(r1,x);
  q.enq(y); r1 := y
rule consume when (!q.empty()):
  let y = q.first(); q.deq();
  let z = f2(y,r2);
  outQ.enq(z); r2 := z;
rule observe when (!obsQ.full()):
  let a = f3(r1, r2); obsQ.enq(a);
```

Fig. 7. An incorrect refinement of the program in Figure 6

adds a token and `consume` always removes one, we must have a non-empty `q` when we have an unequal number of `produces` and `consumes`. However, in any state where `q` is non-empty, we can always execute an appropriate number of `consumes` to make the `q` is empty. Given this, all we must show is that 1) for all prefixes with an equal number of `produce` and `consume` rule executions, we can match `r1` and `r2` with a computation in the original program, and 2) for all other sequences we can take them to a state where the `q` is empty.

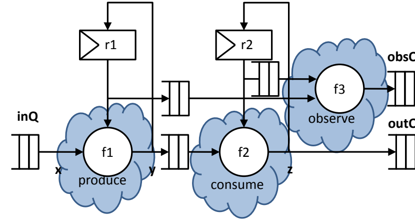
C. An Example to Illustrate Incorrect Refinements

While refinements are often easy to implement, it is not uncommon for a designer to make subtle mistakes. Consider the original one-rule produce-consume example augmented with observation logic as shown in Figure 6. In addition to doing the original computation, this program computes a function of the state of `r1` and `r2`, and at each iteration inserts the result into a new FIFO queue(`obsQ`). A designer may want to do the same rule splitting exercise he had done with the first program, leading to the program in Figure 7.

This refinement is clearly wrong; we can observe `r1` and `r2` out-of-sync via the new observer circuit. Thus, the sequence `produce observe consume` has no correspondence in the original program. For our tool to be useful to a designer, it must be able to correctly determine that this refinement is incorrect (or rather that it failed to find a matching behavior in the original program). A correct refinement is shown in Figure 8, where extra queues have been introduced to keep relevant values in sync. The correct solution would be obvious to an experienced hardware designer because all paths in a pipeline have the same number of stages.

D. Refinements in Nondeterministic Programs

The examples that we have considered so far have started with a single rule program. Such programs by definition

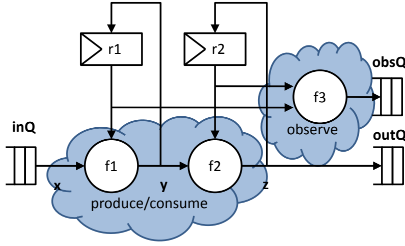



```
register r1 = 0, r2 = 0
fifo inQ, outQ, obsQ, r1Q, r2Q, q;
rule produce when (!inQ.empty() && !r1Q.full()
                                && !q.full()):
  let x = inQ.first(); inQ.deq();
  let y = f1(r1,x);
  r1Q.enq(r1); q.enq(y); r1 := y;
rule consume when (!q.empty() && !r2Q.full()
                              && !outQ.full()):
  let y = q.first(); q.deq();
  let z = f2(y,r2);
  r2Q.enq(r2);
  outQ.enq(z); r2 := z;
rule observe when (!obsQ.full() && !r1Q.empty()
                                   && !r2Q.empty()):
  let x = f3(r1Q.first(),r2Q.first());
  r1Q.deq(); r2Q.deq(); obsQ.enq(x);
```

Fig. 8. A correct refinement of the program in Figure 6

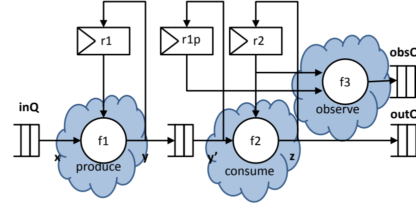
produce deterministic behaviors. Much of the value of rule-based programs comes from the ability to specify programs which can have multiple distinct behaviors. An example of a useful nondeterministic specification is that of a speculative processor whose correctness does not depend upon the number of instructions which are executed on the incorrect path. What does it mean to do a refinement in such a program?

Consider the example in Figure 9, which is a variation of our producer-consumer example with an observer (Figure 6). Unlike the lockstep version which did one observation for each iteration, in this program we are allowed to not only miss some updates of `r1` and `r2`, but are permitted to repeatedly make the same observations. An implementation, *i.e.,* a particular




```
register r1 = 0, r2 = 0
fifo inQ, outQ, obsQ;
rule produce_consume when (!inQ.empty() && !outQ.full()):
  let x = inQ.first(); inQ.deq();
  let y = f1(x,r1); let z = f2(y,r2);
  r1 := y;  r2 := z; outQ.enq(z);
rule observe when (!obsQ.full()):
  let x = f3(r1,r2);  obsQ.enq(x);
```

Fig. 9. A program with a nondeterministic observer




```
register r1 = 0, r2 = 0, r1p = 0
fifo inQ, outQ, q, obsQ
rule produce when (!q.full()):
  let x = inQ.first(); inQ.deq();
  let y = f1(x,r1);
  r1 := y; q.enq(y);
rule consume when (!q.empty()):
  let x = q.first(); q.deq();
  let z = f2(x,r2);
  r1p := x; r2 := z; outQ.enq(z);
rule observe when (!obsQ.full())
  let x = f3(r1p, r2); obsQ.enq(x);
```

Fig. 10. Correct refinement of Figure 9

schedule, of this rule-based specification would pick some deterministic sequence of observations from the allowed set. By giving such a specification, the designer is saying, in effect, that any schedule of observations is acceptable. In that sense, the observations made in the program in Figure 6 are an acceptable implementation of this nondeterministic program. By the same reasoning we could argue that the refinement shown in Figure 8 is a correct refinement of Figure 9.

But suppose we did not want to rule out any behaviors prematurely in our refinements, then a correct refinement will have to preserve *all* possible behaviors. We show a correct refinement of the nondeterministic program in Figure 10, where we introduce an extra register, `r1p`, to keep a relevant copy of `r1` in sync with `r2` with which to make legal observations.

It is nontrivial to show that all behaviors in the new program can be modeled by the original nondeterministic specification and vice versa. As we demonstrate later, our tool can automatically verify this condition, though we do require the programmer to specify a projection function, by which state in the two different programs can be related. The partial function relationship is both natural for designers to come up with and easy to specify. Having manually defined this function, the designer passes it to the tool which tells him either that the refinement is correct, or if it is not, returns an execution from one program which can not be simulated by the other.

III. Formalizing Behaviors and Correctness of Refinements

We model the behavior of a program using a state transition system. A program P has a collection of state elements and a set of rules R_P. The states in the transition system are the values assumed by these state elements. Thus, the state transition system of a program with 2 32-bit registers would have 2^{64} states.

A. Equivalence of Programs

Definition 1 (**State Transition System** of Program $P_{\mathcal{S}}$)**.** A program $P_{\mathcal{S}}$ is modeled by a *state transition system* $\mathcal{S}$, represented as a tuple:

$$\mathcal{S} = (S, S_0, \longrightarrow_{\mathcal{S}}, \twoheadrightarrow_{\mathcal{S}})$$

where S is the set of *states*; $S_0 \subseteq S$ is the set of states corresponding to initial configurations of P; $\longrightarrow_{\mathcal{S}} \subseteq S \times S$ is the transition relation where $(s, s') \in \longrightarrow_{\mathcal{S}}$ if and only if there exists some rule R in P whose execution takes the state s to s'; and $\twoheadrightarrow_{\mathcal{S}}$ is the *reflexive, transitive closure* of relation $\longrightarrow_{\mathcal{S}}$. ■

It is sometimes useful to know which rule R caused the transition from s to s'; we will denote this by writing:

$$s \xrightarrow{R} s'$$

Similarly we write the sequence of rule executions $\sigma = R_1, R_2, ...R_n$ where:

$$s_0 \xrightarrow{R_1} s_1 \xrightarrow{R_2} s_2 ... \xrightarrow{R_n} s_n$$

as:

$$s_0 \overset{\sigma}{\twoheadrightarrow} s_n$$

Intuitively two programs P_1 and P_2 with the same set of states are equivalent if every transition in one system can be modeled by the other. That is, every finite execution $s \twoheadrightarrow s'$ in P_1 has a corresponding execution $s \twoheadrightarrow s'$ in P_2 and vice versa.

Definition 2 (**Equivalence** of Programs)**.** Two programs $P_{\mathcal{S}}$ and $P_{\mathcal{S}'}$ modeled by state transition systems $\mathcal{S} = (S, S_0, \longrightarrow_{\mathcal{S}}, \twoheadrightarrow_{\mathcal{S}})$ and $\mathcal{S}' = (S', S'_0, \longrightarrow_{\mathcal{S}'}, \twoheadrightarrow_{\mathcal{S}'})$ are equivalent if:

$$(S = S') \wedge (S_0 = S'_0) \wedge (\twoheadrightarrow_{\mathcal{S}} = \twoheadrightarrow_{\mathcal{S}'}) \qquad \blacksquare$$

This definition captures the fact that two program may have a different set of rules but may still be equivalent in terms of their transitive closure. This ability allows us to *add* "derived" rules whose execution is always expressible in terms of the other rules in the program without affecting the meaning of the program.

Modeling the behavior of a program in terms of the transitive closure of executions also allows us to define several other notions precisely:

Definition 3 (**Deterministic Programs).** A program $P_{\mathcal{S}}$ is deterministic if all pair-wise executions are confluent, that is: $((s_0 \twoheadrightarrow_{\mathcal{S}} s_1) \wedge (s_0 \twoheadrightarrow_{\mathcal{S}} s_2)) \implies \exists s_3.((s_1 \twoheadrightarrow_{\mathcal{S}} s_3) \wedge (s_2 \twoheadrightarrow_{\mathcal{S}} s_3))$.
A program is called *non-deterministic* if it is *not* a deterministic program. $\blacksquare$

Examples of nondeterministic programs are shown in Figures 7, 9, and 10. All other examples given in Section II are deterministic by this definition.

As we have shown, implementing a rule-based program requires choosing a schedule. A scheduler by definition implements a specific execution sequence. Thus, in the case of non-deterministic programs the scheduler eliminates all non-determinism and produces one specific behavior among the allowed set of behaviors. Even for deterministic programs the scheduler is sometimes not able to produce a "complete" behavior because of unfair scheduling of rules. In such cases we say that an implementation is partially correct:

Definition 4 (**Partially Correct Implementation).** Program $P_{\mathcal{S}'}$ modelled by $\mathcal{S}' = (S', S'_0, \longrightarrow_{\mathcal{S}'}, \twoheadrightarrow_{\mathcal{S}'})$ is a *partially correct* implementation of $P_{\mathcal{S}}$, modelled by $\mathcal{S} = (S, S_0, \longrightarrow_{\mathcal{S}}, \twoheadrightarrow_{\mathcal{S}})$ when:

$$(S = S') \wedge (S_0 = S'_0) \wedge (\twoheadrightarrow_{\mathcal{S}'} \subseteq \twoheadrightarrow_{\mathcal{S}})$$

B. Correctness of Refinements

The correctness is slightly more complicated to define for the refinements we described in the previous section. To begin with, the state elements of the *specification program* are a proper subset of the state elements in the *implementation program* (we refer to the "refined" program as an implementation). In addition, the set of rules in the implementation is formed by removing a rule from the specification rule set, replacing it by two rules (which together simulate the removed rule) that operate on the new implementation state, and lifting the remaining rules to operate in the implementation state. This is clearly more complicated than the simple addition of a derived rule described in the previous section.

To reason about the correctness of refinement, we need a *projection function* to relate implementation state to specification state. This projection function is necessarily partial. Alternatively, we could employ a *lifting function* to map specification state to implementation state. This lifting function is necessarily total. The partial projection function must be a *retraction* (left inverse) of the lifting function. The states which are in the domain of the projection function are called *relatable*.

Consider the transitive reflexive closure of the transition relation $\twoheadrightarrow_{\mathcal{T}}$ of the implementation, from which all nodes (and associated edges) corresponding to states for which the projection function is undefined have been removed. The resulting graph should be identical to $\twoheadrightarrow_{\mathcal{S}}$ if the refinement is correct. (In cases where the projection function associates multiple states in the implementation with a single state in the specification some merging of the states must happen for the relations to be identical.)

However, there is one small complication with the intuition presented above: The fact that the lifted specification graph and reduced implementation graph are identical implies only the existence of a path in the implementation graph which corresponds to every path in the specification graph. This condition covers all finite executions in the implementation program which start and end in relatable states, but not those which start in a relatable state but do not end in one. To address those executions, we must verify that *every* finite execution in the implementation beginning in a relatable state which does not end in a relatable state, can eventually reach one. In concrete terms, this corresponds to the option to empty the FIFO in the refinement presented in Figure 5.

Definition 5 (A **Correct** Refinement**).** The refined program $P_{\mathcal{T}}$ modeled by $\mathcal{T} = (T, T_0, \longrightarrow_{\mathcal{T}}, \twoheadrightarrow_{\mathcal{T}})$ is a correct refinement of the program $P_{\mathcal{S}}$ modeled by $\mathcal{S} = (S, S_0, \longrightarrow_{\mathcal{S}}, \twoheadrightarrow_{\mathcal{S}})$ given lifting function $L : S \longrightarrow T$ relating the states when the following conditions hold:

1) **Soundness:**
$\forall t_1, t_2 \in T, s_1, s_2 \in S.$
$((L(s_1) = t_1) \wedge (L(s_2) = t_2) \wedge (t_1 \twoheadrightarrow_{\mathcal{T}} t_2))$
$\implies (s_1 \twoheadrightarrow_{\mathcal{S}} s_2)$
2) **Limited Divergence:**
$\forall t_0 \in T_0, t_1 \in T.(t_0 \twoheadrightarrow_{\mathcal{T}} t_1)$
$\implies \exists s \in S.(t_1 \twoheadrightarrow_{\mathcal{T}} L(s))$ $\blacksquare$

The first clause states that every possible execution in the implementation whose starting and ending states have corresponding states in the specification must have a corresponding execution in the specification. The second clause states that from any reachable state in the implementation we can always get back to a state which corresponds to a state in the specification. Note that these cases alone do not guarantee that the specification has been fully implemented. To guarantee that a specification has been fully implemented, we need the notion of *total* correctness.

Definition 6 (A **Totally Correct** Refinement**).** A totally correct refinement is a correct refinement which meets the additional conditions:

1) $T_0 = \{L(s) | s \in S_0\}$
2) $\forall s_1, s_2 \in S.(s_1 \twoheadrightarrow_{\mathcal{S}} s_2) \implies (L(s_1) \twoheadrightarrow_{\mathcal{T}} L(s_2))$ $\blacksquare$

The first clause merely says that our initial states directly correspond with each other. The second clause states that all executions in the specification program is preserved in the implementation. The conditions for total correctness are quite easy to verify.

IV. Checking Simulation Using SMT Solvers

We can understand the execution of rule R as the application of a pure function f_R of type $S \longrightarrow S$ to the current state. When the guard of R fails, it causes no state change (*i.e.,* $f_R(s) = s$). We can compose these functions to generate a function f_σ corresponding to a sequence of rules σ. To prove the correctness of refinements, we pose queries about f_σ to an SMT solver.

SMT solvers are conceptually Boolean Satisfiability (SAT) solvers extended to allow predicates relating to non-boolean domains (characterized by the particular theories it implements). SMT solvers do not directly reason about computation, but rather permit assertions about the input and output relation of functions. They provide concrete counter-examples when the assertion is false. For example, suppose we wish to verify that some concrete function f behaves as the identity function. We can formulate a universal quantification representing the property: $\forall x, y.(x = f(y)) \wedge (x = y)$. An SMT solver can be used to solve this query, provided the domains of x and y are finite, and f is expressed in terms of boolean variables. If the SMT solver can find a counter-example, then the property is false. If not, then we are assured that f must be the identity. The speed of SMT solvers on large domains is due to their ability to exploit symmetries in the search space [3].

When we reason about rule execution it is often useful to discard all executions where a rule produces no state update (a *degenerate* execution); it is clearly equivalent to the same execution with that rule removed. As such, when posing questions to the solver it is useful to add clauses which state that sequential states of an execution are different. To represent this assertion for the rule R, we define the predicate function $\hat{f}_R(s_2, s_1)$ which asserts that the guard of rule R evaluates to true in s_1 and that s_2 is the updated state:

$$\hat{f}_R(s_2, s_1) = (s_2 = f_R(s_1)) \wedge (s_2 \neq s_1)$$

As with the functions, we can construct larger predicate $\hat{f}_\sigma(s_2, s_1)$ which is true when a non-degenerate execution of σ takes us from s_1 to s_2.

Now we will explain how the propositions in Definition 5 can be turned into a small set of easily answerable SMT queries.

A. Checking Correctness

For this discussion let us assume we have a specification program $P_\mathcal{S}$ and a refinement $P_\mathcal{T}$. Their respective transition systems $\mathcal{S} = (S, S_0, \longrightarrow_\mathcal{S}, \twoheadrightarrow_\mathcal{S})$ and $\mathcal{T} = (T, T_0, \longrightarrow_\mathcal{T}, \twoheadrightarrow_\mathcal{T})$ are related by the lifting function $L : S \longrightarrow T$.

Now let's consider the soundness proposition from Definition 5:

$$\begin{aligned} &\forall t_1, t_2 \in T, s_1, s_2 \in S. \\ &\quad ((L(s_1) = t_1) \wedge (L(s_2) = t_2) \wedge (t_1 \twoheadrightarrow_\mathcal{T} t_2)) \\ &\qquad \implies (s_1 \twoheadrightarrow_S s_2) \end{aligned}$$

A naïve approach to verifying this property entails explicitly enumerating all pairs (t_1, t_2) in the relation $\twoheadrightarrow_\mathcal{T}$ and checking for a corresponding pair (s_1, s_2) in the relation $\twoheadrightarrow_\mathcal{S}$. As the set of states in both system are finite, both of these relations are similarly finite (bounded by $|T|^2$ and $|S|^2$, respectively) and thus we can mechanically check the implication.

We can substantially reduce this work by noticing two facts. First, because of transitivity, if we have already checked the correctness of $t_1 \overset{\sigma_1}{\twoheadrightarrow}_\mathcal{T} t_2$ and $t_2 \overset{\sigma_2}{\twoheadrightarrow}_\mathcal{T} t_3$, then there is no need to check the correctness of execution $\sigma = \sigma_2\sigma_1$. Second, if have already found an execution σ such that $t \overset{\sigma}{\twoheadrightarrow}_\mathcal{T} t'$ then we can ignore all other executions $\sigma' \neq \sigma$ which have the same starting and ending states as they must also be correct. This essentially reduces the task from checking the entire transitive closure to checking only a *covering* of it. Unfortunately, the size of this covering is still very large.

The insight on which our algorithm is built is that proving this property for a small set of finite rule sequences is tantamount to proving the property for any execution. We explain this idea using the program in Figure 5.

- Let's begin by considering all rule sequences of length one: `produce` and `consume`.
- The sequence `consume` is never valid for execution starting in a relatable state so we need not consider it further.
- The sequence `produce` is valid to execute but does not take us to a relatable state, so we construct more sequences by extending it with each rule in the implementation. These new sequences are `produce produce` and `produce consume`.
- The sequence `produce consume` always takes a relatable state to another relatable state. We check that all concrete executions of `produce consume` have a corresponding execution in the specification. We do this check over a finite set of sequences in $\mathcal{S}$ (in this case: `produce_consume`), the selection of which we will explain later. Since all executions of `produce consume` take us to a relatable state, we need not extend it.
- `produce produce` never takes us from relatable state to relatable state, so again extend the sequence to get new sequences `produce produce produce` and `produce produce consume`.
- `produce produce produce` is degenerate if `q` is of length 2 (`q` has to have some known finite length).
- Suppose we could prove that the sequence `produce produce consume` always behaves like `produce consume produce`. Then any execution prefixed by `produce produce consume` is equal to an execution prefixed by `produce consume produce`. Notice, that we need not consider any sequences prefixed by `produce consume produce` because itself has the prefixed `produce consume`. Therefore we need

not consider further sequences prefixed by `produce produce consume`.

- Because we have no new extension to consider, we have proved the correctness of this refinement.

Each of these steps involved an invocation of the SMT solver on queries which are much simpler than the general query presented previously, though the solver still must conceptually traverse the entire state space. The queries themselves are simple because they are always presented using rule sequences of concrete length, which are much smaller than the sequences in $\twoheadrightarrow_{\mathcal{T}}$. The only problem with this procedure is that in the worst case this algorithm will run for the maximum number of states in $\mathcal{S}$. If we give up before the correctly terminating condition, this only means we have failed to establish the correctness of the refinement. We think it is unlikely that the type of refinements we consider in this paper will enter this case. In fact most refinements can be shown to be correct with very small number of considered sequences.

B. The Algorithm

The algorithm constructs three sets, each of whose elements corresponds to a set of finite executions of T. For each iteration, R_σ represents the set of finite sequences for which we have explicitly found a corresponding member, and U represents the set of finite executions we have yet to verify (each element of U conceptually represents all finite sequences starting with some concrete sequence of rule executions σ). NU is the new value of U being constructed for the next iteration of the execution.

The Verification Algorithm:
Initially:

- $R_\sigma := \emptyset$
- $U := \{R_i | R_i \in R_{P_\mathcal{T}}\}$
- $NU := \emptyset$

1) if $U = \emptyset$, we have verified all finite executions. Exit with Success.
2) Check if we have reached our iteration limit. If so, give up, citing the current U set as the cause of the uncertainty.
3) For each $\sigma \in U$:
   a) Check if the execution of σ from a relatable state is ever non-degenerate:
   $(\exists s_1 \in S, t_2 \in T.(L(s_1) \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} t_2)$
   If no execution exists we can stop considering σ immediately.
   b) Check if σ should be added to R_σ. That is, that some execution of σ should have a correspondence in $\mathcal{S}$:
   $\exists s, s' \in S.(L(s) \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} L(s'))$
   If so $R_\sigma := R_\sigma \cup \{\sigma\}$.
   c) Check if all finite executions of σ that should have a correspondence in $\mathcal{S}$ have such a correspondence:
   $\forall s, s' \in S.(L(s) \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} L(s')) \implies \exists \sigma'.(s \overset{\sigma'}{\twoheadrightarrow}_{\mathcal{S}} s')$
   If this fails due to some concrete execution of σ, exit with Failure providing the counter example as justification.
   d) For every execution where σ does not put us in a relatable state, we must show that extensions of the form $\sigma\sigma'$ have an equivalent execution $\sigma_1\sigma_2\sigma'$, where σ_1 is a member of R_σ and $|\sigma_1\sigma_2| \le |\sigma|$. Thus, the correctness of $\sigma\sigma'$ is reduced to the correctness of the shorter sequence $\sigma_2\sigma'$.
   $\forall s \in S, t \in T.(L(s) \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} t)$
   $\implies \exists \sigma_1 \in R_\sigma, \sigma_2, s' \in S.$
   $(|\sigma_1\sigma_2| \le |\sigma|) \wedge (L(s') = \sigma_1(L(s)))$
   $\wedge (\sigma_2(L(s')) = t)$.
   If this succeeds, we need not consider executions for which σ is a prefix. If not, partition all the extensions into the $|R_P|$ sets of rules by extending σ by one rule execution. $NU := NU \cup \{\sigma.R_i | R_i \in R_{\mathcal{P}_\mathcal{T}}\}$.
4) $U := NU$. $NU := \emptyset$. Go to Step 1.

C. Formulating the SMT Queries

The four conditions in the inner-most loop of the algorithm can be formulated as the following SMT queries using the $\hat{f}_\sigma$ predicate and the lifting function L:

1) *Existence of valid execution of σ starting from a relatable state:*
   $\exists s_1 \in S, t_2 \in T.\hat{f}_\sigma(t_2, L(s_1))$
2) *Verifying that each execution of σ in the implementation starting and ending in a relatable state has a corresponding execution in the specification:*
   $\forall s_1, s_2 \in S, t_1, t_2 \in T.$
   $(L(s_1) = t_1) \wedge (L(s_2) = t_2) \wedge \hat{f}_\sigma(t_2, t_1) \implies$
   $\bigvee_{\sigma' \in EC(\sigma)}(\hat{f}_{\sigma'}(s_2, s_1))$
   where EC is the "expected correspondences" function which takes a sequences of rules σ in $\mathcal{T}$ and returns a finite set of sequences in $\mathcal{S}$ to which σ is likely to correspond. This function can be easily generated by the tool or the user, since given the refinements are rule splitting, it is easy to predict the candidates in the specification that could possible mimic σ. For instance, consider the refinement of the program in Figure 3 to the one in Figure 5. Each occurrence of `produce` in the implementation should correspond to an occurrence of `produce_consume` in the specification. Thus, the sequence `produce produce consume produce`, if it has a correspondence at all, could only correspond to the sequence `produce_consume produce_consume produce_consume`.
3) *Checking that every valid execution of σ in the implementation has an equivalent sequence which is correct by concatenation of smaller sequences*:
   $\forall t_1, t_2, t_m \in T, s_1, s_m \in S.$
   $(L(s_1) = t_1) \wedge \hat{f}_\sigma(t_2, t_1) \implies (L(s_m) = t_m) \wedge$
   $\bigvee_{\sigma_1 \in R_\sigma}(\bigvee_{\sigma_2 \in EA(\sigma,\sigma_1)}(\hat{f}_{\sigma_1}(t_m, t_1) \wedge \hat{f}_{\sigma_2}(t_2, t_m)))$
   Our algorithm requires us to find, given σ and σ_1 in $\mathcal{T}$, a σ_2 such that the execution of σ is the same as the execution of $\sigma_1\sigma_2$, and $|\sigma_1| + |\sigma_2| \le |\sigma|$. We will assume the existence of a "expected alternatives" function EA which enumerates all possible values of σ_2 given σ and σ_1.

D. Step-By-Step Demonstration

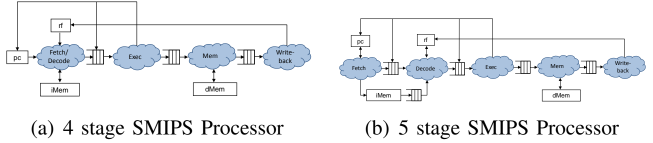
Figure 11 gives the traces of reasoning through which our algorithm progresses in order to verify three distinct refinements, all of which were presented in Section II. Each node represents an element in the set U, and the path from the root to any node in the graph corresponds to the concrete value σ for that node. At each node, we verify the correctness of all corresponding finite executions of σ: nodes displayed as $\perp$ are vacuously true by Step 3a, while other leaf nodes are either true by Step 3d or incorrect by Step 3c. In this section, we attempt to give the reader further intuition about the algorithm through a discussion of one of these traces.

The leftmost trace corresponds to the steps required to check the refinement of the design in Figure 3 to the design in Figure 5. Since this refinement has been discussed extensively, we provide no further explanation. The graph in the center corresponds to the refinement of the design in Figure 6 to the one in Figure 7 and provides a valuable illustration of how the algorithm rejects an *incorrect* refinement:

- We begin by considering all rule sequences of length one executed in a relatable state: `produce`, `consume`, and `observe`. The rule `observe` always ends in a relatable state, and corresponds directly to the `observe` rule in the specification program. `consume` is never valid to execute, so the only sequence which we extend is `produce` since it never ends in a relatable state.
- We now extend `produce`, giving us three new sequences to consider: `produce produce`, `produce consume`, and `produce observe`. `produce consume` always ends in a relatable state and corresponds to the execution of `produce_consume` in the specification. Neither `produce produce`, nor `produce observe` ever end in a relatable state, and since we are unable to prove their equivalence to an execution we have already verified, we extend both.
- In the third iteration, we consider the sequence `produce observe consume`, which always ends in a relatable state. This exposes an error in the refinement since there is no possible sequence of rule in the specification which produces this final state (in this case, the implementation enqueues a value to `obsQ` which the specification is unable to replicate.

V. The Debugging Tool and Evaluation

We use the symbolic algorithm as the basis for a single-threaded Haskell implementation. Our tool accepts both the reference and refined implementations as post-elaborated Bluespec SystemVerilog. It then generates the queries specified in Section IV as boolean propositions over bitvectors, and passes them to the STP solver [5]. To improve efficiency, we exploit simple path rewriting analysis to avoid invoking the SMT solver for trivial cases. For instance, with the program in Figure 10, we establish that `produce observe` is always `observe produce`, thus handling the P_3 and P_6 sequences of the rightmost graph in Figure 11 without additional SMT queries.



(a) 4 stage SMIPS Processor (b) 5 stage SMIPS Processor

Fig. 12. SMIPS processor refinement

To demonstrate our tool, we consider the refinement of a Simplified MIPS (SMIPS) processor, whose ISA contains a representative subset of 35 instructions from MIPS. While the ISA semantics are specified one instruction at a time, our program is pipelined with five stages in the style of the DLX processor [9], and resembles soft-cores used in many FPGA designs. The execution of the final implementation is split into the following five separate stages (see Figure 12(b)):

1) *Fetch* requests the next instruction from the instruction memory (`imem`) based on the `pc` register which it then updates speculatively to the next consecutive `pc`.
2) *Decode* takes the data from the instruction memory and fetch stage, decodes the instruction, and passes it along to the execute stage. It also reads the appropriate locations in the register file `rf`, stalling to avoid data hazards (stall logic is not shown).
3) *Execute* removes decoded instructions from the execute queue, executing the ALU instructions and translating addresses for memory instructions. When resolving branch instructions, mispredicted instructions are killed and `pc` is set.
4) *Memory* performs reads and writes to the data memory, passing the data to the writeback state. (A further refinement might introduce a more realistic split-phase memory, which would move some of this functionality into the writeback stage).
5) *Writeback* gets instructions in the form of register destination and value pairs, performing the update on the register file.

We implement this program using one rule per stage, and stages communicate via FIFO connections. If we choose to execute the rules for each stage in reverse order (starting from writeback and finishing with fetch), then we get a fully pipelined system. If we implement each FIFO with a single, this is indistinguishable from the standard processor complete with pipeline stalls. If instead we execute the rules in pipeline order, we end up with a system where the instructions fly through the processor one-at-a-time. For code simplicity, our final implementation actually decomposes the execute stage into three mutually exclusive cases, implementing each with a separate rule(`exec, exec_branch, and exec_branch_mispredict`). Since the rule guards are mutually exclusive, this does not modify the pipeline structure, nor does it change analysis.

Our implementation is relatively complicated and we would like to know if it matches the ISA. One way to do achieve this is to start with a single-rule description of the behavior (transliterated directly from the documentation, which we

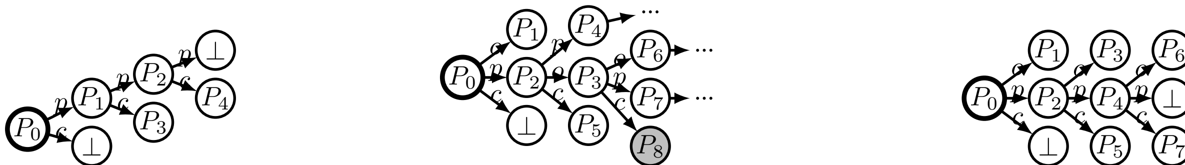


Fig. 11. Tree visualizations of the algorithmic steps to check the refinements from Figures 3 to 5, 6 to 7, and 9 to 10 (from left to right)

consider to be correct), and incrementally refine the program towards the final five-stage implementation. After each refinement, our tool can be used to verify correctness with regards to the previous iteration. For the sake of brevity, we examine only the final refinement, which takes a four-stage processor (Figure 12(a)) and splits the fetch-decode stage. Though the transformation is straightforward, the tool must be able to correctly resolve the effect of speculative execution from branch prediction.

The tool is able to establish strict stuttering simulation of this 7 rule program by the original 6 rule program in under 7 minutes. To do so it needed to check 21 executions in the refined program of maximum length 3, finding correspondences in the four-stage program for the 5 corresponding rules, `fetch_decode` for `fetch decode`, and `exec_branch_mispredict` for the mispeculating sequences `fetch exec_branch_mispredict` and `fetch fetch exec_branch_mispredict`.

VI. Discussion

This paper discusses the correctness of refinements for rule-based programs general enough to allow a number of microarchitectural techniques, like pipelining and speculation. These notions are closely related to the the well-known notions of simulation and bisimulation (see, *e.g.,* [8]) and stuttering simulation (see, *e.g.,* [7]).

This exact notion of correct refinement intended by a designer that the user wants is naturally and succinctly represented in our tool by a function. The tool is then able to establish the correctness of the refinement, returning a possible non-corresponding match in a short time. This makes it a viable candidate for designers to use to explore their refinements.

Though the execution time for our realistic example is still larger than one would want in the design cycle, there is substantial room for improvement to the current implementation. Most obviously the algorithm is highly parallelizable and would benefit from multicores. Furthermore, there are substantial inefficiencies from the file-level interaction with the SMT solver. More than half of compute time comes from marshaling and unmarshaling the representation. Not only would direct software hooks cut this time substantially, but they would also enable incremental construction of SMT queries, which would further reduce redundancy. Finally, improvements in SMT efficiency could be garnered by exploiting more theories beyond those of bitvectors, especially those of uninterpreted functions, FIFOs, and arrays. With these additions, we are confident that this tool can help shape how designers approach their work and encourage further formal reasoning in design.

Acknowledgments

This work has been supported by the National Science Foundation (#CCF-0541164).

References

[1] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Jose, CA, 2004.

[2] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[4] Kermin Fleming, Chun-Chieh Lin, Nirav Dave, Gopal Raghavan, Jamey Hicks, and Arvind. H.264 decoder: A case study in multiple design points. In *In Proceedings of Formal Methods and Models for Codesign (MEMOCODE 2008)*, Anaheim, CA, USA, June 2008.

[5] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *19th International Conference on Computer Aided Verification (CAV-07)*, volume 4590, pages 519–531, 2007.

[6] James C. Hoe and Arvind. Operation-Centric Hardware Description and Synthesis. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 23(9), September 2004.

[7] P. Manolios. A compositional theory of refinement for branching time. In *CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2003.

[8] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *FSTTCS'97*, volume 1346 of *Lecture Notes in Computer Science*, pages 284–296. Springer, 1997.

[9] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface, Second Edition.* Morgan Kaufmann, 1997.