Scheduling as Rule Composition

Nirav Dave, Arvind & Michael Pellauer
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: {ndave, arvind, mpellauer}@csail.mit.edu

Abstract

Bluespec is a high-level hardware description language used for architectural exploration, hardware modeling and synthesis of semiconductor chips. In Bluespec, one views hardware as a collection of stateful elements (e.g., registers, memories) and describes its behavior using rules*, or* Guarded Atomic Actions *which modify these elements. All legal behaviors of a Bluespec program can be explained in terms of rules being applied in some sequence.* Scheduling *is the process of selecting which rules to execute in parallel while maintaining this semantic invariant. The scheduling decision can have a large impact on critical design properties such as pipeline concurrency and clock frequency. What constitutes a good schedule often depends upon the application and requires the designer's input.*

In this paper we introduce BTRS, the kernel language for Bluespec and use it to explore the task of scheduling. We view scheduling as the process of restricting a Bluespec design's non-deterministic behavior to be deterministic. We define a small set of scheduling operators whose semantics are expressed in terms of rule composition*. We show how to represent the schedules generated by the Bluespec compiler using these compositions. More importantly, our scheduling primitives open a large class of new schedules which are needed for microarchitectural explorations.*

1 Introduction

When designing a circuit, concurrency expectations often play a critical role in system performance and verification. Consider a packet router which must maintain a certain line rate due to real-time constraints. There are many router microarchitectures to explore — pipelines, circular lookups, RAM-controllers and so on. Each will correctly perform the task of routing packets to their proper destination. However each will do so with vastly different pipeline throughput, clock-cycle frequency, circuit area, and power consumption. Some design points will turn out to compensate for pipeline bubbles with a high clock speed, whereas others will ultimately be unable to meet the real-time constraints. It is critical that the architect be able to perform such exploration without disrupting the fundamental functionality of routing packets to their correct destinations.

In this paper we present a set of scheduling primitives for rule-based hardware system which give the architect precise control over concurrency concerns. Moreover, they enable design-space exploration via allowing the user to separate concerns of *functional correctness* ("Is my router relaying packets to correct destinations?"), *performance correctness* ("Are there any harmful pipeline bubbles or dead cycles?") and *physical correctness* ("Does my design run at a fast enough clock speed and fit in a small enough area?").

In previous rule-based hardware design creating the rules was the purview of the designer, whereas the rules were scheduled using an automatically-synthesized scheduler circuit. However, in large-scale designs, it has been found that often the compiler does not have sufficient knowledge to get the required behavior or performance. Specifically, the generated scheduler avoided introducing combinational paths that may have worsened clock frequency, even when this decision may have eliminated some pipeline bubbles. Thus, physical concerns were always favored over performance concerns. As such, user knowledge had to be provided to the compiler via extra-language solutions such as pragmas in order to produce the desired implementation.

In this paper, we discuss a new way of approaching this problem. Instead of the compiler selecting the "correct" scheduler, the designer is responsible for generating the schedule and the tool is only responsible for implementing it as efficiently as possible. We present a formal system for reasoning about rule-based systems, and study how new rules can be derived from existing ones without changing the behavior of the system. Ultimately, we reformulate scheduling in terms of rule composition, and show how the entire scheduling space can be characterized using only three composition primitives. We show out technique using the longest-prefix example.

Paper organization: Section 2 begins by presenting BTRS, our kernel language of guarded atomic actions and modules, illustrated via a circular lookup example. In Section 3 we present the operational semantics of BTRS using SOS-style rules. In Section 4 we introduce a theory of derived rules and explain their generation via a small set of rule composition operators. Section 5 shows how scheduling can be im-

m ::=	`Module` $name$	
	[`Register` r v]	// Regs w/ initial values
	[`Rule` R a]	// Rules
	[`ActMeth` g $\lambda x.a$]	//Action method
	[`ValMeth` f $\lambda x.e$]	//Value method
a ::=	r `:=` e	// Register update
‖	`if` e `then` a	// Conditional action
‖	a `|` a	// Parallel composition
‖	a `;` a	// Sequential composition
‖	a `when` e	// Guarded action
‖	(t = e `in` a)	// Let action
‖	$m.g(e)$	// Action methcall g of m
e ::=	r	// Register Read
‖	c	// Constant Value
‖	t	// Variable Reference
‖	e op e	// Primitive Operation
‖	e `?` e `:` e	// Conditional Expression
‖	e `when` e	// Guarded Expression
‖	(t = e `in` e)	// Let Expression
‖	$m.f(e)$	// Value Methcall f of m
op ::=	`&&` | `||` | ...	// Primitive operations

Figure 1. BTRS Grammar for a Module

plemented with derived rules. In Section 6 we explore the implementation of our scheduling constructs and demonstrate how they enable microarchitectural exploration in our lookup example. Finally, we discuss related work in Section 7 and in Section 8 we conclude and discuss future work.

2 BTRS: An Introduction

In this section we will informally introduce our language of rules BTRS (pronounced *B-terse*) via our illustrative example, a circular lookup module.

2.1 The Language

BTRS roughly corresponds to the result of a Bluespec program after "static elaboration," (i.e., after type checking and module instantiations). We have also supplemented the language with a sequential connective which we will use for composition (Section 3.1). The grammar for BTRS is given in Figure 1. Most of the grammar is standard and needs little discussion. We will only highlight the unique aspects of the language. A BTRS module consists of 3 parts: a set of state elements (i.e. registers), a set of guarded atomic actions of *rules* which represent the internal state changes, and a set of *methods* which is used to interface with other modules.

Every action or rule in BTRS is deterministic. The nondeterminism in a description is introduced by the choice in the order of execution of these rules. The range of behaviors that a collection of modules can produce is succinctly described in the Figure 2.

Since this procedure involves a nondeterministic choice and the choice potentially affects the observed behaviors, our BTRS program is more like a specification as opposed to an implementation. To obtain an implementation we must selectively restrict the model to eliminate all nondeterminism and produce one behavior (scheduling).

Repeatedly:

1. Choose a rule in some module to execute
2. Compute U, the set of register updates, by evaluating the rule's action according to the rules given in Figure 4.
3. Update all the registers according to U.

Figure 2. BTRS Execution Procedure

2.2 Example: A Circular Lookup Module

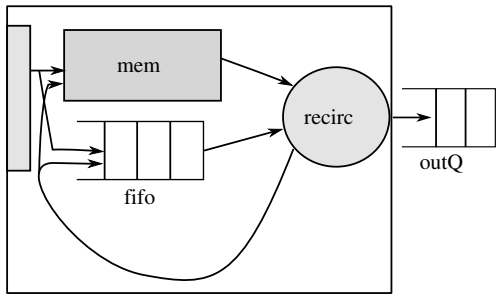
Consider the example given in Figure 3. This program represents the system of a circular pipeline to implement a complex table lookup. The table lookup may require fetching the data from the memory several times, akin to pointer chasing. The memory module (`mem`) has a request-response type of interface. It can take an arbitrary amount of time to generate a response for a request and can handle multiple outstanding requests. The memory response can be examined or dequeued. The system has an internal FIFO `fifo` to hold the outstanding requests while a memory reference is in progress. Once a lookup request is completed, the message leaves the system via another fifo `outQ`.

The top level module is called `lookup` and has one internal rule called `recirc` which takes responses from memory and the partial state of the corresponding request from `fifo` and depending on the result either passes the final result into `outQ` or recirculates the request, placing an updated partial state in `fifo` and a new memory request. The `lookup` module has only one method called `enter` which simply enqueues a new request into the `fifo` queue and sends an initial request to the memory.

The `recirc` rule contains a complex action which is a parallel composition of an action to accept the memory response and a conditional action involving the predicate $p(x)$. The then-side action of this conditional action is a parallel composition of an action to dequeue from `fifo` and another one to enqueue into `outQ`. The else-side action is a parallel composition of a request to the memory and another compound action, which is itself a sequential composition of the dequeue action on `fifo` followed by a enqueue action on the same queue. All the functions like `p`, `f1`, `f2` and `addr` represent combinational circuits, whose details are not relevant for the current discussion. "`x=mem.res()`" and "`y=fifo.first()`" represent pure bindings of values being returned by the methods `mem.res()` and `fifo.first()`.

Notice the use of the sequential connective in `recirc` rule is necessary for correct behavior. Had we chosen to use parallel composition, when we need to recirculate a request we would need to be able to enqueue (there's space in `fifo`) and dequeue (there's an element in `fifo`) at the beginning of the cycle. Thus if the queue were full, we'd be unable to execute `recirc` and the system would deadlock. By sequencing the enqueue after the dequeue, the enqueue action observes the free space left by the dequeue.

Figure 3 includes an implementation of a one-element FIFO as well. Its interface has two action methods, `enq` and




```
Module lookup
  Rule "recirc"
       x = mem.res() in
       y = fifo.first() in
       mem.resAccept()|
       if p(x) then
          (fifo.deq() | outQ.enq(f1(x,y)))
       else
           mem.req(addr(x))|(fifo.deq();
                             fifo.enq(f2(x,y)))
  ActMeth enter(x) =
       fifo.enq(x)| mem.req(addr(x));

Module fifo
  Register vf0 false
  Register  f0
  ActMeth enq(x) =
      (vf0 := true | f0 := x) when !vf0
  ActMeth deq() = (vf0 := false) when vf0
  ValMeth first() = f0 when vf0
```

Figure 3. Example: A Table-Lookup Module and one-element FIFO

deq, and one value method first. It has a register f0 to hold a data value and a one-bit register vf0 to hold the valid bit for f0. The encoding of all methods is self-explanatory but it is worth pointing out that all methods have guards represented by the when clauses. The guards for first and deq signify that the FIFO is not empty while the guard for enq signifies that the FIFO is not full.

Although it is easy to understand the meaning of enq and deq in isolation, the interaction between the two methods is not obvious. Can the user simultaneously enqueue and dequeue? The recirc rule uses the FIFO in such a way that deq must somehow come before enq. Furthermore this transaction must happen atomically with no other users being able to observe the intermediate state.

We have not shown an implementation of the memory module but the system allows for it to be pipelined and to hold many requests simultaneously. The guard of mem.req will indicate when it can accept a new request. The guards of value method mem.res and action method mem.resAccept would indicate when mem has a result available. The guard condition of the enter method is simply the conjunction of the guards (implicit conditions) of fifo.enq and mem.req methods.

The reader may wonder what happens if in some state both the rule recirc and the method enter can fire. Such a situation could occur if the fifo and mem could hold multiple entries. The BTRS semantics allow either rule to be executed in this situation. In fact, the reference semantics to be presented shortly includes this type of non-determinism. In Section 5 we present a new way to reason about the scheduling necessary to remove this nondeterminism for hardware synthesis.

3 Semantics of Rule Execution in BTRS

We present the operational semantics of a rule execution in BTRS using SOS-style evaluation rules (Figure 4), where $\twoheadrightarrow$ means expression evaluation. The meaning of each composite atomic action will be explained in terms of its constituent atomic actions.

Let S represent the values of all the registers before the rule executes. The effect of executing an atomic action will be represented by U, the set of register updates implied by the execution. Conflicting updates to the same register produce a *dynamic error*. Our system can easily handle dynamic errors, but doing so would clutter the presentation and obscure the primary contribution. Therefore, for the purposes of this paper, we will assume that the sufficient static analysis has been applied to all system to prevent dynamic errors from occurring.

The semantic machine is incomplete in the sense that there are cases where the execution gets *stuck* because none of the rules in Figure 4 apply. In such cases we will say that the action produced no updates. This allows us to present a much more succinct set of rules which are not cluttered by dealing with $\perp$ propagation.

Now we discuss the less standard aspects of BTRS.

3.1 Action Composition

The language provides two ways to compose actions together: *parallel composition* and *sequential composition*.

Two actions $A_1|A_2$ composed in parallel both observe the same initial state and do not observe each other's actions. Thus the action $r_1 := r_2 \mid r_2 := r_1$ swaps the values in registers r_1 and r_2. Since all rules are determinate, there is never any ambiguity due to the order in which subactions complete. Thus, parallelly composed actions are forbidden from updating the same state simultaneously. Again, it is preferable if such an error is disallowed by static checking in an earlier compilation step.

Sequential composition is more in line with other languages with atomic actions. The $A_1; A_2$ is the execution of A_1 followed by A_2. A_2 observes the full effect of A_1. No other action observes A_1's updates without also observing A_2's updates.

3.2 Conditional versus Guarded Actions

BTRS has both conditional actions (ifs) as well as guarded actions (whens). These are similar as they both restrict the evaluation of an action based on some condition. The difference is their scope: conditional actions have only a local effect whereas guarded actions have a global effect. If an if's predicate evaluates to false, then that action

Action Rules:

reg-update
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow v,\ v \neq \text{NR}}{\langle S,U,B\rangle \vdash r := e \twoheadrightarrow \{\}[v/r]}$$

if-true
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow true,\ \langle S,U,B\rangle \vdash a \twoheadrightarrow U'}{\langle S,U,B\rangle \vdash \texttt{if}\ e\ \texttt{then}\ a \twoheadrightarrow U'}$$

if-false
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow false}{\langle S,U,B\rangle \vdash \texttt{if}\ e\ \texttt{then}\ a \twoheadrightarrow U}$$

a-when-true
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow true,\ \langle S,U,B\rangle \vdash a \twoheadrightarrow U'}{\langle S,U,B\rangle \vdash a\ \texttt{when}\ e \twoheadrightarrow U'}$$

par
$$\frac{\langle S,U,B\rangle \vdash a_1 \twoheadrightarrow U_1,\ \langle S,U,B\rangle \vdash a_2 \twoheadrightarrow U_2}{\langle S,U,B\rangle \vdash a_1 \mid a_2 \twoheadrightarrow (U_1 \uplus U_2)}$$

seq
$$\frac{\langle S,U,B\rangle \vdash a_1 \twoheadrightarrow U_1; \langle S,U_1{+}{+}U,B\rangle \vdash a_2 \twoheadrightarrow U_2}{\langle S,U,B\rangle \vdash a_1\ ;\ a_2 \twoheadrightarrow U_2{+}{+}U_1}$$

a-let-sub
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow v, \langle S,U,B[v/t]\rangle \vdash a \twoheadrightarrow U'}{\langle S,U,B\rangle \vdash t = e\ \texttt{in}\ a \twoheadrightarrow U'}$$

a-meth-call
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow v,\ ,v \neq \text{NR},\ m.g = \langle \lambda t.a\rangle,\ \langle S,U,B[v/t]\rangle \vdash a \twoheadrightarrow U'}{\langle S,U,B\rangle \vdash m.g(e) \twoheadrightarrow U'}$$

Expression Rules:

reg-read
$$\langle S,U,B\rangle \vdash r \twoheadrightarrow (U{+}{+}S)(r)$$

const
$$\langle S,U,B\rangle \vdash c \twoheadrightarrow \underline{c}$$

variable
$$\langle S,U,B\rangle \vdash t \twoheadrightarrow B(t)$$

op
$$\frac{\langle S,U,B\rangle \vdash e_1 \twoheadrightarrow v_1,\ v_1 \neq \text{NR},\ \langle S,U,B\rangle \vdash e_2 \twoheadrightarrow v_2,\ v_2 \neq \text{NR}}{\langle S,U,B\rangle \vdash e_1\ op\ e_2 \twoheadrightarrow v_1\ \underline{op}\ v_2}$$

tri-true
$$\frac{\langle S,U,B\rangle \vdash e_1 \twoheadrightarrow true,\ \langle S,U,B\rangle \vdash e_2 \twoheadrightarrow v}{\langle S,U,B\rangle \vdash e_1\ ?\ e_2\ :\ e_3 \twoheadrightarrow v}$$

tri-false
$$\frac{\langle S,U,B\rangle \vdash e_1 \twoheadrightarrow false,\ \langle S,U,B\rangle \vdash e_3 \twoheadrightarrow v}{\langle S,U,B\rangle \vdash e_1\ ?\ e_2\ :\ e_3 \twoheadrightarrow v}$$

e-when-true
$$\frac{\langle S,U,B\rangle \vdash e_2 \twoheadrightarrow true,\ \langle S,U,B\rangle \vdash e_1 \twoheadrightarrow v}{\langle S,U,B\rangle \vdash e_1\ \texttt{when}\ e_2 \twoheadrightarrow v}$$

e-when-false
$$\frac{\langle S,U,B\rangle \vdash e_2 \twoheadrightarrow false}{\langle S,U,B\rangle \vdash e_1\ \texttt{when}\ e_2 \twoheadrightarrow \text{NR}}$$

e-let-sub
$$\frac{\langle S,U,B\rangle \vdash e_1 \twoheadrightarrow v_1, \langle S,U,B[v/t]\rangle \vdash e_2 \twoheadrightarrow v_2}{\langle S,U,B\rangle \vdash t = e_1\ \texttt{in}\ e_2 \twoheadrightarrow v_2}$$

e-meth-call
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow v,\ v \neq \text{NR},\ m.f = \langle \lambda t.e_b\rangle,\ \langle S,U,B[v/t]\rangle \vdash e_b \twoheadrightarrow v'}{\langle S,U,B\rangle \vdash m.f(e) \twoheadrightarrow v'}$$

Merge Functions:

$$U_1 \uplus U_2 \quad = \text{error if } \exists r.\{r \mapsto v_1\} \in U_1 \wedge \{r \mapsto v_2\} \in U_2 \text{ otherwise } U_1 \cup U_2$$

$$\{\}(x) \quad = \bot$$
$$S[v/t](x) \quad = v \text{ if } t = x \text{ otherwise } S(x)$$

Each action rule gives a list of register updates given an environment $\langle S, U, B\rangle$ where S represents the register state, U is the observable updates, and B represents the local bindings. NR represents the "not-ready" value and can be stored in a binding, but *not* assigned to a register. The strictness of method calls is enforced by checking that parameter values are not NR. Initially U and B are empty and S contains the value of all registers. One can think of ++ as list concatenation. If the system gets stuck because no rule is applicable, it is assumed than an empty U is returned.

Figure 4. Operational semantics of a BTRS Rule

A.1	$(a_1\ \texttt{when}\ p) \mid a_2$	$\equiv$	$(a_1 \mid a_2)\ \texttt{when}\ p$
A.2	$a_1 \mid (a_2\ \texttt{when}\ p)$	$\equiv$	$(a_1 \mid a_2)\ \texttt{when}\ p$
A.3	$(a_1\ \texttt{when}\ p)\ ;\ a_2$	$\equiv$	$(a_1\ ;\ a_2)\ \texttt{when}\ p$
A.4	$a_1\ ;\ (a_2\ \texttt{when}\ p)$	$\equiv$	$(a_1\ ;\ a_2)\ \texttt{when}\ p'$ (p' is p after a_1)
A.5	$\texttt{if}\ (e\ \texttt{when}\ p)\ \texttt{then}\ a$	$\equiv$	$(\texttt{if}\ e\ \texttt{then}\ a)$ $\texttt{when}\ p$
A.6	$\texttt{if}\ e\ \texttt{then}\ (a\ \texttt{when}\ p)$	$\equiv$	$(\texttt{if}\ e\ \texttt{then}\ a)$ $\texttt{when}\ (p \vee \neg e)$
A.7	$(a\ \texttt{when}\ p)\ \texttt{when}\ q$	$\equiv$	$a\ \texttt{when}\ (p \wedge q)$
A.8	$r := (e\ \texttt{when}\ p)$	$\equiv$	$(r := e)\ \texttt{when}\ p$
A.9	$m.h(e\ \texttt{when}\ p)$	$\equiv$	$m.h(e)\ \texttt{when}\ p$
A.10	$\texttt{Rule}\ n\ \texttt{if}\ p\ \texttt{then}\ a$	$\equiv$	$\texttt{Rule}\ n\ (a\ \texttt{when}\ p)$

Figure 5. When-Related Axioms on Actions

doesn't happen (produces no updates). If a `when`'s predicate is false, the subaction (and as a result the whole atomic action) is invalid. One of the best ways to understand the differences between `whens` and `ifs` is to examine the axioms in Figure 5.

Axioms A.1 and A.2 collectively say that a guard on one action in a parallel composition affects all the other actions. Axioms A.3 and A.4 say similar things about guards in a sequential composition. We take some liberty with notation in writing "p' is p after a" to mean an expression which is the same as p except that it is evaluated after the effects a have been taken into account. Axioms A.5 and A.6 state that guards in conditional actions are reflected only when the condition is true, but guards in the predicate of a condition are always evaluated. Axiom A.7 deals with merging `when` clauses. A.8 and A.9 translates expression `when`-clauses to action `when`-clauses. Axiom A.10 states that top-level `whens` in a rule can be treated as an `if` and vice versa.

Guards are restrictions used to reflect resources. Thus it would be appropriate to guard a method which enqueues into a bounded FIFO with the condition that the FIFO has space to accept the new value.

3.3 Behaviors and Equivalence of a BTRS system

We define a *state* of a BTRS system design to be the value of all of its registers. A rule represents a relation from states to states, where a state is taken to the state that results from executing the rule. The *behavior* of a system given

```
Rule "A" (r1 := r2 + 1) when (r2 < 5)
Rule "B" (r2 := r1 + 1) when (r1 < 4)
Rule "D" (if (r2 < 5) then
                  (r1 := r2 + 1 |
                   (if (r2 < 3) then
                       r2 := r2 + 2))
            |  (if (r2 >= 5) && (r1 < 4) then
                   r2 := r1 + 1)
```

Figure 6. Rule D acts as A, B or A then B

a starting state is the transitive closure of the union of rule relations.

A system is said to be *determinate* if we can move from state S_0 to either S_1 or S_2 in the system, there exists some S_3 such that we can move from S_1 to S_3 and from S_2 to S_3. We now use these definitions to build a framework of derived rules.

4 Derived Rules

If rule A takes state S to state S' and rule B takes state S' to state S'', we can derive a new rule AB which directly takes S to S''. Such *derived rules* do not change the behavior of a system. In a practical sense, a derived rule like AB may represent a faster way of executing the system. If rule AB replaces A or rule B then it may reduce the possible traces of the system and in some sense reduce the amount of choice we have when selecting rules to execute.

From this basic notion, we can generate a powerful framework for reasoning about restricting behaviors and increasing parallelism in BTRS. In this section we will formalize the notion of a derived rule, and the behavior and equivalence of BTRS systems *derived BTRS systems* (systems containing derived rules of a *principal BTRS system*).

4.1 What is a Derived Rule?

We define a derived rule D of a system S with set of rules $\mathcal{R}$ if adding D to $\mathcal{R}$ does not change the behavior of S. Since adding a rule to a system trivially maintains all its old behaviors, this is equivalent to saying that any possible executions of D in S must be expressible as some sequence of execution of the rules in $\mathcal{R}$.

Consider the example in Figure 6. The rule D behaves as either A, B or A followed by B depending on the state.

4.2 Derived Rules via Composition

For any system a large number of derived rules exist. Rather than generating those rules in an ad-hoc fashion, we propose to generate derived rules via rule composition: functions which take a set of rules and return a new rule derived from the input rules.

Many such composition functions are possible. The only constraint required for a rule composition operator is that all behaviors of the new rule can be expressed as a sequence of executions of the parameter rules.

In Figure 7 we introduce three basic rule compositions ($compose$, par and $restrict$). We shall demonstrate that these three operators are sufficient to describe a large space

$$
\begin{array}{rl}
DR ::= & R \\
\| & compose(DR, DR) \\
\| & par(DR, DR) \\
\| & restrict(DR, DR)
\end{array}
$$

Figure 7. BTRS Scheduling Language

of rule compositions which generate interesting behavior. In section 3.1 we introduced a sequential connective `;` to BTRS which has composition semantics that do not currently exist in Bluespec. Now we make use of this connective to naturally explain these derived rules.

For clarity, we will assume that all parameter rules to a composition are in the form a `when` p: they have had all guards lifted to the top following the axioms presented in Figure 5. This is not necessary for these compositions to be correct, but may allows us to make claims about mutual exclusion which are useful in discussion.

Sequencing Rules: $compose$

An important rule one could want is one which executed as if $R1$ and then $R2$ happened in a single step. For correctness all of $R1$'s updates must be observed by $R2$. This is exactly what the sequential connective `;` does on actions. Thus our new rule is:

```
compose(Rule R1 a1 when p1, Rule R2 a2 when p2) =
       Rule R1R2 (a1 when p1);(a2 when p2)
```

It's important to note that this new rule will only be enabled when *both* $R1$ and $R2$ can fire in sequence.

Merging Mutually Exclusive Rules: par

Often, two rules $R1$ and $R2$ are never enabled at the same time. In this case, it makes sense to treat the two rules as two halves of a single rule. We are guaranteed such a rule must be expressible as a sequence of $R1$ and $R2$ as it will always behave as either $R1$ or $R2$. Similarly to $compose$ we could express this notion with:

```
Rule R1orR2 (if p1 then a1)|(if p2 then a2)
             when (p1 ∨ p2)
```

If the rules are not mutually exclusive, the new rule may exhibit new behaviors. For instance if $R1$ was $r1 := r2$ and $R2$ was $r2 := r1$, then the above action would swap the values, a behavior not expressible via sequential executions of $R1$ and $R2$.

We prevent this situation from occurring by forcing the new rule to only be enabled when exactly one of the rules is ready. The new operator is:

```
par(Rule R1 a1 when p1, Rule R2 a2 when p2) =
  Rule R1R2 (if p1 then a1)|(if p2 then a2)
               when (p1 ⊕ p2)
```

Choosing from Rules: $restrict$

Sometimes we care to generate a derived rule which operates as $R1$ if possible (i.e. $R1$ is ready) and otherwise it acts as $R2$. Instead of generating an entirely new composition, we will simply limit one rule ($R2$) to be enabled only when the other rule ($R1$) is not. Then this new rule ($R2'$) would be mutually exclusive with the second rule ($R1$) and

could they could thus be composed via par. This naturally extends to generating priorities for three and more rules.

```
restrict(Rule R1 a1 when p1, Rule R2 a2 when p2)=
         Rule R1R2 a2 when (¬p1 ∧ p2)
```

One extension of this composition would be to allow restrictions against general Boolean values instead of just a rule guard, however we do not make use of this in this work.

Correctness of these Compositions

All rules generated from these compositions do not add new behaviors to a system. This is straightforward to show. The execution of $compose(R1, R2)$ produces the same result as the execution of $R1$ followed by the execution of $R2$. The execution of $par(R1, R2)$ will either behave exactly as $R1$ or $R2$. All executions of $restrict(R1, R2)$ match an execution of $R2$.

Further, since adding a rule generated via these compositions do not add behavior to the system, any composition expressible via these compositions also produce derived rules of the parameter rules.

4.3 Expressing Other Rule Compositions

With these three compositions, we can generate a slew of new rule compositions by successively applying these three operators to principal rules, adding the new derived rule to the set of available rules and repeating the process until the desired derived rule is generated.

For example, it is common to use $restrict$ in conjunction with par. For instance we may want to generate a derived rule of A and B which operates as B unless A could be done. This occurs frequently enough that we will introduce the following operator:

```
pri(R1, R2) = par(R1, restrict(R1, R2))
```

As another example, consider rule composition seq. This composition take rules $R1$ and $R2$ and tries to execute $R1$ and then (whether it was executed or not) attempts to execute $R2$. This is most naturally expressed as:

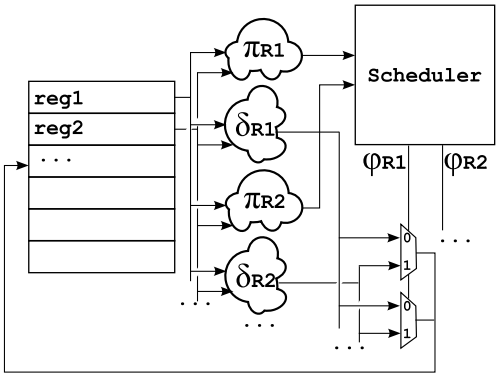
```
seq(Rule R1 a1 when p1, Rule R2 a2 when p2) =
    Rule R1R2 (if p1 then a1);(if p2 then a2)
                    when (p1 ∨ p2)
```

We can express the seq operator via our original compositions as:

```
seq(A, B) = S where AB = compose(A,B)
                    A´ = restrict(AB,A)
                    B´ = restrict(A,B)
                    S  = par(AB,A´, B´)
```

5 Scheduling and Schedulers

BTRS's execution model consists of non-deterministically choosing a rule from the set of rules in the system and executing it and repeating the process. However, when we implement this in hardware we need to generate a deterministic system where one or more rules happen each clock cycle. Thus, we need to restrict the behavior of our BTRS system. Notice that the goal of scheduling is not to generate a just determinate BTRS system, but also determine the concurrency of rules in the implementation (which rules fire together in a clock cycle). The task of making these restrictions is referred to as *scheduling*.



A scheduler attempts to execute rules in parallel while maintaining consistency with serial behavior. By convention we call the input signals to the scheduler πs, which can be thought of as the set of rules which are "ready." The output of the scheduler is a set of ϕs, those rules which are actually chosen to execute. The δs of those rules are the effect which is applied to the current state to calculate the next state.

Figure 8. Scheduler Example

In implementation, scheduling is traditionally accomplished with the addition of a hardware circuit (a *scheduler*) which selects which rules to execute each clock cycle. A traditional scheduler is depicted in Figure 8. Each choice of scheduler results in a different FSM implementation.

In this section we show that rule composition and derived rules can be used to accomplish this task simply and powerfully, giving the user a new way to reason about their system. Then we use this rule composition to examine existing approaches to rule scheduling.

5.1 Expressing Scheduling Using Rule Composition

Previously, it was stated that scheduling removes the non-determinism of rule choice introduced in the BTRS execution process described in Figure 2. But consider a system with only one rule. This choice has been removed since there is only one rule which can be chosen!

This leads us to our key insight: a system S with many rules can be transformed into a single-rule system S' via rule composition. S corresponds to the principle BTRS specification, and S' to the scheduled implementation. The schedule then is exactly the rule composition used on the principle rules to generate a final derived rule. Note that a scheduler can refer to the same rule multiple times, and can in fact execute multiple times a cycle.

This gives additional benefits. Scheduling is no longer an extra-language operation and can be reasoned about naturally in the BTRS language itself. Since a BTRS system with a scheduler is a unique FSM, one can ask questions about FSM equivalence of two systems, something which is not possible in a system without an explicit schedule. Additionally, since schedulers are now precise, easily expressed,

and are generated compositionally, we end with a natural and precise notion of a partial scheduler.

Expressing scheduling via rule composition makes intuitive sense. What does adding a scheduler do, but join all the rules into a single deterministic action? In fact all previous scheduling algorithms can be naturally expressed in terms of rule compositions. We will show how popular scheduling algorithms, such as Esposito Scheduling [4] and Performance Guarantees [8] can be described as such.

In this work we will only consider "stateless" schedulers. That is, the firing of rules each cycle based solely on the explicit state of the system. This is not a crippling restriction, since any system can be programmatically transformed by adding state to the system itself and modifying the rules to read and modify it appropriately.

5.2 Simple Schedulers

The simplest scheduler one could imagine would fire exactly one rule each cycle. Consider for example, a fair scheduler that takes a fixed static order on rules and applies them one by one in that order, if possible, moving on to the next rule, if not.

We can express this idea programmatically by introducing a counter `cnt` and modifying each rule to test the value of the counter and increment it if it was that rule's turn to execute. Thus, assuming rule `R` has body `a when p` where there are no other `whens` in a and the rule is assigned the i^{th} slot and there are n rules, this rule would become:

```
Rule R if (cnt == i) then
((if  p then a) | cnt := mod(cnt + 1, n))
```

Since each of the rules are now mutually exclusive, we can describe the scheduler as the simple parallel composition of these rules:

```
par(R1, par(R2, par(R3,...)))
```

Naturally, this scheduler tends to result in poor performance. If two rules operate on disjoint data, allowing both rules to fire in the same cycle would exploit natural parallelism in the system without much effect on the hardware quality. Similarly, mutually exclusive rules could be schedule to fire in the same cycle with no penalty as only one will ever have an effect. Thus, the Bluespec compiler's goal for scheduling is to select a maximal number of rules to fire concurrently each cycle while keeping the appearance of firing one rule at a time. Generating an optimal scheduler involves an exponential search. In order to avoid this, the Bluespec compiler uses a well-tested heuristic known as Esposito Scheduling.

5.3 The Esposito Scheduler Algorithm

The Esposito scheduler algorithm[4] is the standard scheduler generation algorithm in the Bluespec Compiler. The algorithm is an efficient heuristic designed to generate minimal scheduling hardware while still achieving reasonable rule-level parallelism. The algorithm works on a few basic principles:

1. To prevent lengthening combinational paths, the scheduler should introduce no combinational paths between different rules to keep atomicity (e.g. a logically later rule should not read a value to which a earlier rule could write)
2. To keep muxing logic simple, the scheduler will consider only one sequence of rules in logical execution order. Each cycle the subset of rules that occur will execute in this order.
3. When two rules are forbidden from firing, a single global priority ordering (*urgency*) will determine which rule is chosen.

To generate the scheduler, the algorithm first builds a directed graph where each node corresponds to an individual rule. An edge exists in the graph from rule R_1 to rule R_2 if and only if it is possible that rule R_1 followed by rule R_2 is valid to fire and the effects of the most recent execution of R_1 affects the execution of R_2.

In practice, the above condition is too difficult to calculate and a combination of mutual exclusion analysis and some conservative interaction analysis is used [7]. A cycle in this graph represents a situation where all the rules in the cycle cannot fire correctly together without introducing a combinational path. The algorithm systematically removes these cycles by restricting the guard of one of the rules, so that it is mutually exclusive with an adjacent rule in the cycle thereby removing the edge and cutting the cycle.

Once all cycles are removed, the now acyclic graph represent a partial ordering on rules where sources of the directed edges are "later" than the sinks. This lets us generate a total logical ordering for the execution of the rules each cycle.

Consider the following system of rules A, B, and C where A is more urgent than B, which is more urgent than C:

```
Rule "A" (r1 := r2 + 1) when (r2 < 10)
Rule "B" (r2 := r3 + 1) when (r3 < 10)
Rule "C" (r3 := r1 + 1) when (r1 < 10)
```

The resulting graph would consist of three edges: from A to C, C to B, and B to A. To cut this cycle we choose to remove the edge from A to C, restricting C not to fire when A is valid to fire. Now the graph is acyclic and we can generate a total ordering on the rules (A then B then C). Rules A and B will fire when their guards are ready, and C will fire when $(r1 < 10) \wedge \neg(r2 < 10)$.

The Esposito Algorithm via Rule Compositions

It's very natural to express the Esposito algorithm using rule compositions. An edge is added between R_1 and R_2 where executing the R_1 may affect R_2's execution. This is exactly saying *not* to make an edge when there is no combinational path (i.e. $seq(R_1, R_2)$ behaves the same as composition $par^*(R_1, R_2)$) or the rules are mutually exclusive (i.e. $par(R_1, R_2)$ behaves the same as $par^*(R_1, R_2)$).

```
par*(Rule R1 a1 when p1, Rule R2 when p2) =
    Rule R1R2 (if p1 then a1)|(if p2 then a2)
                        when (p1 ∨ p2)
```

One of these equivalences is needed because while $par*$ does not add combinational paths between the two rules, it may allow for incorrect behavior if both rules fire. Verifying that it matches with a valid composition guarantees that using $par*$ will be correct.

Restricting the firing conditions of rules to cut cycles, can be expressed by replacing the restricted rule R with $restrict(R', R)$ where R' is the rule whose guard is augmented. Once all cycles have been removed, composing all rules using the $par*$ on the chosen total logical order for rules $R_0 < R_1 < ... < R_n$ we can use the $par*$ composition to join the rules together.

$$S = par^*(R_0, par^*(R_1, ...par^*(R_{n-1}, R_n))))$$

Since we composed the rules so all no sequential data passing was needed, the resulting rule will be guaranteed to not add any new behaviors.

5.4 Performance Guarantees

The Esposito scheduler has proven effective at generating efficient schedulers. However, it does not allow for data generated in one rule to be passed to another rule executed in the same cycle. This intra-cycle passing is sometimes desired for performance reasons. One solution to this is to add special state which passes data between two methods (an RWire). However, this makes the meaning of a rule dependent on the implementation of the modules which it uses, heralding the addition of the sequential action connective.

In light of this, Performance Guarantees[8] were proposed to handle these shortcomings. In this system the user was responsible for partitioning the set of rules into a sequence of sets of rules. A scheduler algorithm similar to the Esposito algorithm was used to generate schedulers. However, while no combinational paths are allowed between rules in a single set, all rules in later sets observe the state modifications of previous rules.

In addition to allowing combinational paths to be selectively added through the scheduler, performance guarantees allowed rules to be executed multiple times in a cycle.

Performance Guarantees via Rule Compositions

Performance Guarantees also can be expressed via rule composition. Each of the sets of rules is scheduled (via the Esposito scheduler) into single rules. This sequence of rules, is then composed using the seq connective adding the observations. Repeating rules is also well defined.

6 User-Defined Schedules

Using the rule composition primitives introduced above, the user has complete control over scheduling. This frees the designer to explore schedules which balance high-level performance properties (pipeline throughput) vs. low-level physical properties (clock frequency, area) without changing the rules themselves.

In this section we present such an exploration using the circular lookup example introduced in Section 2.2.

```
Module lookup
  Rule "recirc"
       x = mem.res() in
       y = fifo.first() in
       ((mem.resAccept();mem.req(addr(x))|
             (fifo.deq();fifo.enq(f2(x,y))))
        when !p(x))
  Rule "exit"
       x = mem.res() in
       y = fifo.first() in
       ((mem.resAccept()|fifo.deq()|
             outQ.enq(f1(x,y)))
         when p(x))
  Rule "enter"
       x = inQ.first() in inQ.deq()|
       fifo.enq(x)| mem.req(addr(x))
```

Figure 9. New LPM Design Rules

6.1 Scheduling the Circular Lookup

An important issue in the circular pipeline discussed earlier is if a packet can enter the system in the same clock cycle when another one is leaving the system, i.e., can the `recirc` rule and the `enter` method execute concurrently. If these actions do not take place in the same cycle then the system is supposed to contain a *dead cycle*. Is this a serious issue? Suppose our concrete lookup algorithm takes at most 3 lookups for each request. The dead cycle in this case would increase the total number of cycles needed to serve the incoming requests by at least 33%! The user can avoid such a dead cycle by giving an appropriate schedule. However, as we will show later, exploiting this level of concurrency may increase the critical combinational path delay. A designer may want to consider the total performance when selecting a schedule.

In general `recirc` and `enter` cannot execute together without an additional memory port. However, as designers we can see that in the case of an exit, the `recirc` rule and the `enter` method do not conflict, i.e., do not use the same methods, and thus it is possible to execute them concurrently. It is easier to understand this if we split the original `recirc` rule into two rules: `recirc`, which only recirculates requests back into the pipeline and `exit`, which handles requests entering `outQ`. Additionally, we transform the `enter` method into the `enter` rule which explicitly picks up incoming messages from an `inQ`. These rules are given in Figure 9. Note that since all three rules interact with `fifo`, scheduling this system also requires determining the inter-method properties of the FIFO module. We also need both `fifo` and `mem` to hold more than one request to deal with realistic memory latencies. Therefore for our explorations we replaced the one-element FIFO with a 6-element one.

6.2 The Three Schedules

We are going to consider three schedules: Schedule 1 where `enter` and `exit` do not execute concurrently, Schedules 2 and 3 where they do. Here we do not explore schedules which will require dual-ported memories; such schedules will make it possible to process more than one message per cycle.

	Schedule 1	Schedule 2	Schedule 3
Composition	`pri(recirc, pri(exit, enter))`	`pri(seq(exit, enter), recirc)`	`pri(recirc, seq(exit, enter))`
Clock Period (ns)	2.0	2.0	2.0
Area (μm^2)	36,320	42,940	43,206
Max Latency (CCs)	15	18	15
Benchmark Perf. (CCs)	28,843	18,927	18,927

Worst-Case Latency refers to the maximum number of clock cycles that an operation can take to complete the pipeline. Although schedules 2 and 3 had the same performance on our synthetic benchmark, the worst-case latencies of schedule 2 is worse

Figure 10. Implementation Results

Schedule 1: `pri(recirc, pri(exit,enter))`
Schedule 2: `pri(recirc,seq(exit, enter))`
Schedule 3: `pri(seq(exit,enter), recirc)`

Schedule 1 executes only one rule per clock cycle, with `recirc` being the highest priority, followed by `exit`, then `enter`. Schedule 2 can execute `exit` and `enter` in the same cycle. It will choose to execute `recirc` over either or both of these. Schedule 3 also allows `exit` and `enter` to execute in the same cycle. However, in this schedule both of these rules take priority over `recirc`.

6.3 Performance Results

Since the Bluespec compiler does not currently support the sequential connective or derived-rule-based scheduling, each design was manually compiled to a single rule sans the sequential connective. It is possible to do this in general but requires generating new interfaces for the `fifo` and connecting these interfaces correctly to method calls in the rules. All designs were compiled using Bluespec Compiler version 2006.11 and synthesized using Synopsys Design Compiler version Y-2006.06 with TSMC 180 nm libraries. The performance and synthesis results are shown in Figure 10. To keep both area and timing comparable, we show results within 100 ps of the minimal clock period.

We can see that all schedules are able to meet a 2 ns timing requirement, but schedules 2 and 3 result in significantly larger area than schedule 1.

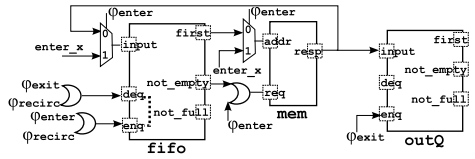
Schedule 1 takes 28,803 cycles to complete our synthetic benchmark. In contrast, both schedules 2 and 3 only take 18,927 cycles, an improvement of nearly 35%. This matches our intuition of the cycle-level effect of allowing `exit` and `enter` to execute concurrently.

The same analysis shows that in the worst case an operation under schedule 2 can take 3 more cycles to complete than an operation in schedule 3. This is because when `recirc` has priority it prevents a sixth instruction from entering the memory until something has left, whereas in schedule 3, we will enter new requests until `fifo` is full (i.e. we will have 6 in-process requests). Thus, though our benchmark did not exercise this feature, the design generated from schedule 2 has better performance in this regard.

A designer considering these three schedules would thus choose either schedule 1 or 2, depending what mixture of area and performance they valued more.

6.4 Understanding the Generated Designs

We've argued that the choice of schedules 2 and 3 required longer combinational paths, but have not explained how such paths come to exist. In this subsection we will examine the particulars of the differences in these design.




```
πenter = fifo.not_full
πrecirc = fifo.not_empty
πexit = fifo.not_empty
         && outQ.not_full

φenter = πenter && !φexit && !φrecirc
φrecirc = πrecirc
φexit = πexit && !φrecirc
```

Schedule 1 (`pri(recirc, pri(exit,enter))`).

```
πenter = fifo.not_full || φexit
πrecirc = fifo.not_empty
πexit = fifo.not_empty
         && outQ.not_full

φenter = πenter && !φrecirc
φrecirc = πrecirc
φexit = πexit && !φrecirc
```

Schedule 2 (`pri(seq(exit, enter), recirc)`).

```
πenter = fifo.not_full || φexit
πrecirc = fifo.not_empty
πexit = fifo.not_empty
         && outQ.not_full

φenter = πenter
φrecirc = πrecirc && !(φexit || φenter)
φexit = πexit
```

Schedule 3 (`pri(recirc,seq(exit, enter))`).

The circuits only differ in their π- and ϕ-signals, and in the `fifo` module. A combinational path between the `enq` and `deq` methods is required for Schedules 2 & 3.

Figure 11. Three LPM Implementations

At a high level, each of these designs looks exactly like the diagram in Figure 11. Each design varies in some of the definitions of the internal logic values as well the internals of the FIFO. We explain change of the logic with the changes to the FIFO externalized in the LPM module (specifically, the effect of `deq` on `notFull` the implicit condition of `enq`). The described logic does not change.

Since schedule 1 allows only one rule to execute in each cycle, its π-signals only depend on the state of the FIFOs. The ϕ-signals form a priority scheduler for the three rules. Since `enter` can happen logically after `exit` in a cycle

in both schedules 2 and 3, we see that the ϕ-signals are set true whenever `exit` is to happen in the same cycle (i.e. ϕ`exit` is true). It is this addition that causes the longer combinational path, and results in larger area.

7 Related Work

Previous research has been directed at frameworks for reasoning about scheduling. However, this work differs from classic task scheduling approaches in that rather than scheduling a fixed set of tasks onto a known topology of resources, we are providing a way for the designer to generate resources as needed for the behavior we desire. In this way, our work is more similar to a restricted form of the high-level synthesis problem [5] than classical task scheduling.

The problem of synthesizing synchronous languages such as Esterel explored by Berry, Edwards et al. [1, 3] is similar, but is generally not viewed as a scheduling problem because an Esterel program already describes a unique behavior. Synthesizing Esterel is the problem of choosing the optimal FSM from a set of equivalent FSMs to implement this behavior, whereas rule scheduling describes a set of FSMs which are not equivalent, but all adhere to the same initial system specification.

Early rule-based systems were often satisfied with performing one rule at a time[2]. Early work on using rule-based synthesis to describe hardware showed that systems could achieve good performance simply by executing in parallel rules that did not conflict. Hoe showed that this restriction could be loosened to allow rules with write-after-write and write-after-read hazards to go in parallel[6]. This work provides a formal framework for reasoning about scheduling and providing precise concurrency control.

Other formal frameworks for reasoning about scheduling have been explored. Soft scheduling is a framework proposed by Zhu and Gajski [10] which allows a scheduler to reason about its own performance and update its choices dynamically at runtime. This differs from our system in that each task can take different amounts of time — perhaps even unpredictable amounts of time. Also it requires scheduler state to track decisions, whereas we only considered stateless schedulers in this paper.

Saviou, Shukla, and Gupta proposed a methodology for improving SystemC scheduling via merging tasks [9]. This merging of tasks is similar to the rule composition techniques proposed here. The approaches differ in that since their goal is to improve simulator performance, their system must thus maintain strict behavioral equivalence. Our approach allows the user to specify a range of behaviors adhering to a behavioral specification.

8 Discussion

In this paper we presented a novel approach to the scheduling problem. We present a formal semantics for the Guarded Atomic Action model and use this model to define notions of system behavior and equivalence. Starting from the insight that a system with only one rule in it is trivially scheduled, we built up a theory of scheduling via derived rules. We introduced three primitive rule composition operators and demonstrated that they are sufficient to describe a large set of interesting schedules. We demonstrated how these operators can be used to enable the system architect to perform a new type of architectural exploration simply by changing the schedule.

Although this work was originally motivated by insufficiencies in automatically-generated schedulers, we do not see user-created schedules as a complete replacement. Rather, it would be interesting to explore using automatically generated schedules as a default provided to the user, and the tools presented in here applied selectively to reason about critical areas. Other future ideas which could be explored include generating stateful schedulers, particularly those with fairness properties. Additionally we hope to increase the modularity of this algorithm and perform case studies on larger systems such as a microprocessor pipeline.

Acknowledgments

This work has been supported by the National Science Foundation (#CCF-0541164) and Nokia Inc. We would also like to thank Anna Ingolfsdottir for her help with notion and Tarmo Uustalu for his helpful discussion about the formal definition of behaviors.

References

[1] G. Berry. Esterel on hardware. In *Mechanized Reasoning and Hardware Design*, pages 87–104. Prentice Hall, Hertfordshire, UK, 1992.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[3] S. Edwards. High-Level Synthesis from the Synchronous Language Esterel. In *Proceedings of IWLS'02*, New Orleans, LA, 2002.

[4] T. Esposito, M. Lis, R. Nanavati, J. Stoy, and J. Schwartz. System and method for scheduling TRS rules. United States Patent US 133051-0001, February 2005.

[5] D. D. Gajski, N. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, MA, 1992.

[6] J. C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of ICCAD'00*, pages 511–518, San Jose, CA, 2000.

[7] D. L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.

[8] D. L. Rosenband and Arvind. Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. In *Proceedings of ICCAD'05*, San Jose, CA, 2005.

[9] N. Savoiu, S. Shukla, and R. Gupta. Improving SystemC Simulation Through Petri Net Reduction. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2005.

[10] J. Zhu and D. D. Gajski. Soft Scheduling in High Level Synthesis. In *DAC '99: Proceedings of the 36th Conference on Design Automation*, pages 219–224, New York, NY, USA, 1999. ACM Press.