Modular Compilation of Guarded Atomic Actions

Muralidaran Vijayaraghavan
Massachusetts Institute of Technology
vmurali@csail.mit.edu

Nirav Dave
SRI International
ndave@csl.sri.com

Arvind
Massachusetts Institute of Technology
arvind@csail.mit.edu

Abstract—**Over the last decade, Bluespec, a hardware description language of guarded atomic actions has been used to describe rapidly modifiable, modular, no-compromise hardware designs and generate circuits from them. While the language itself supports significant modularity, the compiler compiles a module with other modules as parameters by in-lining or flattening the module. This forces the user to either suffer large compile times or to change the modular structure of the design. In this paper we propose a new modular compilation scheme which supports compilation of modules with interface methods as parameters and preserves Bluespec's one-rule-at-a-time semantic model. This compilation process inherently requires the distributed scheduling of rules.**

I. Introduction

It is important for designers to be able to modify their designs rapidly to better search the micro-architectural space as performance and power pressures increase in the embedded systems. Such exploration has proven quite difficult in structural RTL because of its rigid timing requirements on modules. Bluespec, where one represents hardware using guarded atomic actions (GAAs) on state elements has been shown to enable rapid design exploration without compromising the quality of generated hardware.

A complex hardware design is hardly ever built as a monolithic block; instead it is designed as a collection of interacting modules. While Bluespec's Guarded Atomic Actions (GAAs) preserve the designers intent about cycle-level micro-architectural actions, the compilation strategy offers only limited support for non-tree-like modular hierarchies. This restriction disallows, for example, passing in communication channels to modules, and can sometimes destroy the natural modular structure of the code. For example, a module that interacts with an off-chip memory via a bus must include the bus and memory modules as submodules.

In this paper we present KBS2, a kernel for Bluespec where modules can have interface parameters. We specify the requirements for a modular interface to guarantee the language atomicity requirements. We then define an algorithm to calculate the "most-permissive" modular interface in the sense that if the module is instantiated with interface parameters which has an equal or more restrictive interface it is guaranteed to work correctly. Finally we define a translation of KBS2 to hardware, including invariants that a scheduler must follow. We argue that this scheme generates as efficient a hardware as the state-of-the-art compilation scheme which cannot handle parameters [13]. The main contribution of this paper is the modular compilation of modules which have *interface parameters*, *i.e.,* methods defined by other modules, as parameters.

Bluespec System Verilog (BSV, the implementation of Bluespec) has many features that are important for the ease of programming but not necessary for understanding its semantics. Most of these features are eliminated after the static elaboration phase of compilation. However, two features, guards and EHRs, remain after static elaboration and they affect the semantics of the language. These features are orthogonal to the issue of compiling modules with interface parameters and hence, to simplify the presentation, we discuss these features in passing and give the detailed compilation procedure only for the language without guards and EHRs. To further simplify the presentation, we present the language features in steps a) KBS0, a language with no modules [8], b) KBS1, a language with modules without interface parameters [13], and c) KBS2, a language with modules with interface parameters. All these languages except for KBS2 have been presented elsewhere [5], [8], [13], [12]. We revisit them here to make the compilation part self-contained and to use uniform notation.

Paper Organization: Section II discusses related work and the novelty of our work. Section III gives an example to motivate the need for modules with interface parameters. Section IV presents KBS0, its semantics and its compilation into hardware. Section V presents the same for KBS1 and discusses the issues with guards and EHRs. Section VI discusses the KBS2 language and its compilation procedure. Section VII gives the conclusion and offers some future directions for research.

II. Related work

Early rule-based systems often executed only one rule at a time and automatically satisfied the one-rule-at-a-time semantics [4]. The origins of the present work lie in work of Hoe and Arvind on hardware synthesis from rule-based based systems [1], [8]. Parallel scheduling of "non-conflicting rules" in a single clock cycle was essential for good quality hardware synthesis. The work was later extended by Rosenband and Arvind to include modules with proper interface properties to maintain one-rule-at-a-time semantics. However their techniques did not permit interface parameters and assumed that the methods of separate modules could not conflict with each other.

Further work on scheduling and modules went in several different directions. Dave *et al.* recharacterized the scheduling problem as the one of rule composition for better understandability [5]. Rosenband introduced the idea of *performance* *guarantees* to reduce the variability in the synthesis results and introduced EHRs (*Emphemeral History Registers*) to implement these guarantees [12]. Recently EHRs have been elevated to a new primitive which is directly available to the user to make it much easier to express concurrent rules in Bluespec. Karczmarek *et al.* showed how to extend hardware compilation to include multi-cycle combinational circuits [9].

The work by Dave *et al.* [6] illustrates several design patterns which can be exploited in Bluespec to allow modular refinement without breaking functionality. This paper focuses solely on modular compilation, and enhances the flexibility for modular refinement strategy proposed by them.

Another type of work on modular refinement has its origins in Carloni's work on Latency-Insensitive circuits [3]. He showed how a synchronous circuit can be decomposed into parts which can communicate through channels whose latency does not affect the functional correctness of the overall circuit. His technique required putting a shim around each part to keep track of the clock cycles in the specification. With proper extensions (see for example, [11], [14]) such Latency-Insensitive frameworks permit original circuits to be modified in an incremental modular manner. However the resulting circuit produces the same cycle-by-cycle behavior as in the original model. In contrast the work presented in this paper is about a high-level language where the designs are expressed to be modularly refineable; separate compilation is a requirement for such modular refinements.

The problem of synthesizing hardware from synchronous languages such as Esterel is generally not viewed as a scheduling problem [2], [7]. A program in Esterel, like an FSM, describes a unique behavior. In this sense, synthesizing Esterel is the problem of choosing between a set of equivalent FSMs. Rule scheduling in Bluespec also implements an FSM but it is from a much larger set of non-equivalent FSMs all of which satisfy the initial specification.

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. For example, soft scheduling is a framework proposed by Zhu and Gajski [15] 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. It also requires scheduler state to track decisions, whereas we only considered stateless schedulers.

III. A MOTIVATING EXAMPLE

We will use the example of a simple multi-stage processor pipeline to motivate the issues of modularity and modular compilation. The processor contains a Fetch stage, a Decode/register-read stage and an Execute/memory/writeback stage as shown in Figure 1. We represent each stage as a separate module so that it may be refined further for greater pipelining. Fetch passes instructions to Decode via the `f2d` FIFO while Decode passes the decoded instructions to Execute via the `d2e` FIFO. Execute communicates the correct target PC to Fetch via the `redirect` FIFO. These modules also communicate with each other through the register file and the scoreboard; Decode reads the register file through two read ports and Execute updates the register file though the write port. The scoreboard is used to keep track of instructions in the pipeline to avoid RAW and WAW hazards. Decode can search the scoreboard to examine if registers which an instruction reads are being updated by an instruction in the pipeline. If not, the instruction can be entered into the execute pipeline and enqueued in the scoreboard. Execute removes an instruction from the scoreboard after it has been committed. Fetch communicates with the instruction memory via the `iReq` and `iRes` ports while Execute communicates with the data memory using the `dReq` and `dRes` ports.

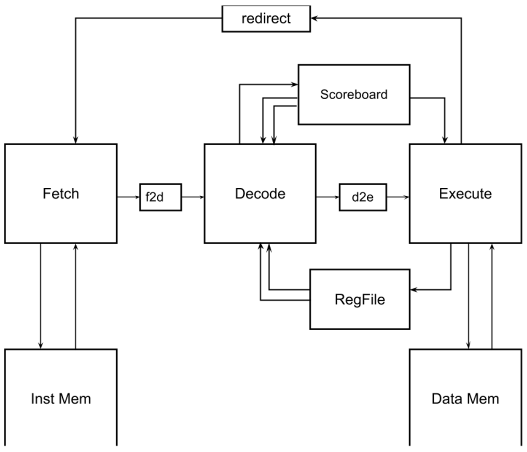

Fig. 1: A simple processor

Figure 2 shows the processor code organization, along with the methods called in each module. For our purposes the internal module details are not important and consequently rules for implementing the modules are not shown. `TypeX x <- X(ap1, ap2 ...)` instantiates a module `X` of `TypeX` with local instance name `x`, passing parameters `ap1`, `ap2` *etc.* We follow the naming convention where m_g refers to the method g of module m. Thus, `f2d_enq` represents the enqueue method of the FIFO module `f2d`. In this example, modules `mkProc`, `mkFetch`, `mkDecode` and `mkExecute` define no interface methods; they interact with other modules only through parameters. Modules `redirect`, `f2d`, `d2e`, `rf` and `sb` have defined methods to allow other modules to access their internal state.

We would like to compile each of the modules `mkProc`, `mkFetch`, `mkDecode`, `mkExecute`, `mkRegFile`, `mkScoreboard` and all the FIFOs separately. This is essential for abstraction and modular refinement. The designer should be able to change the implementation of say the `mkExecute` module without the internal knowledge of how the other modules are implemented. It would permit the testing of a module in various different contexts [5]. Modular compilation can also drastically reduce the compile time by avoiding recompilation.

```
Module mkProc(iReq, iRes, mReq, mRes)
  // Instantiation of modules w/o parameters
  Fifo redirect <- mkFifo()
  Fifo f2d <- mkFifo()
  Fifo d2e <- mkFifo()
  RegFile rf <- mkRegFile()
  Scoreboard sb <- mkScoreboard()

  // Instantiation of modules with parameters
  Fetch fetch <- mkFetch(iReq, iRes,
    redirect_deq, redirect_first, f2d_enq)
  Decode decode <- mkDecode(f2d_deq, f2d_first,
    rf_read0, rf_read1, d2e_enq,
    sb_enq, sb_search0, sb_search1)
  Execute execute <- mkExecute(d2e_first,
    d2e_deq, sb_deq, rf_write,
    redirect_enq, mReq, mRes)

Module mkFetch(iReq, iRes,
    redirectDeq, redirectFirst, f2dEnq)
  Reg pc <- mkReg(...)
  Reg epoch <- mkReg(...)
  ...
  rule fetch1, fetch2 ...
  // These rules calls the following methods:
   ... iReq, iRes, redirectFirst,
   redirectDeq,  f2dEnq ...

Module mkDecode(f2dDeq, f2dFirst,
   iReq, iRes, d2eEnq, rfRead0, rfRead1,
   sbEnq, sbSearch0, sbSearch1)
    ...
  rule decode1, decode2 ...
  // These rules calls the following methods:
   ... f2dFirst, f2dDeq, rfRead0, rfRead1,
   d2eEnq, sbSearch0, sbSearch1, sbEnq, ...

Module mkExecute(d2eFirst,
    sbDeq, rfWrite, d2eDeq,
    redirectEnq, mReq, mRes)
    ...
  rule execute1, execute2 ...
  // These rules calls the following methods:
   ... d2eFirst, d2eDeq, redirectEnq,
   mReq, mRes, sbDeq, rfWrite ...
```

Fig. 2: Processor organization - `m_g` is method `g` of `m`

Modular compilation is difficult because it needs to guarantee that the atomicity of the rules inside Fetch, Decode and Execute, is maintained even though the method calls are spread over many different modules. Since the rules are spread across multiple modules, modular compilation also needs to preserve the one-rule-at-a-time semantics of the whole system *i.e.,* any behavior of the system must be understood as if the rules of the whole system are executed one at a time in some order. Sometimes the methods of a module "conflict" with each other and may not be called simultaneously. For instance, the `f2d` FIFO may be such that its enqueue and dequeue methods update the same register; then no rule should be able to invoke these methods simultaneously, and two different rules calling these methods should not be scheduled simultaneously.

```
             // [...] represents a list in the meta language
         Prog ::=   [⟨instantiation⟩]
                    [⟨rule⟩]
instantiation ::=   x <- Reg(⟨c⟩)
                              // Register instantiation
       rule ::=  rule ⟨r⟩ ⟨a⟩ // r is the name identifier
    a ::=  ⟨x⟩_w(⟨e⟩)                // register write
       ||  if ⟨e⟩ then ⟨a⟩           // conditional action
       ||  ⟨a⟩|⟨a⟩                   // parallel action
       ||  let ⟨t⟩ = ⟨e⟩ in ⟨a⟩      // let bindings
    e ::=  ⟨x⟩_r()                   // register read
       ||  ⟨c⟩                       // a constant
       ||  ⟨t⟩                       // a variable
       ||  ⟨op⟩(⟨e⟩,⟨e⟩)             // primitive functions
       ||  let ⟨t⟩ = ⟨e⟩ in ⟨e⟩      // let bindings
   op ::=  && || plus || ...
```

Fig. 3: Grammar of the moduleless language KBS0

Note that current Bluespec does offer some degree of modular compilation for modules which do not have interface parameters. For instance, in Figure 2, `mkFIFO`, `mkScoreboard` and `mkRegFile` can be compiled separately, but `mkFetch`, `mkDecode` and `mkExecute` cannot. The next two sections give the necessary background on the current Bluespec language and compiler.

IV. KBS0: A LANGUAGE WITHOUT MODULES

The first kernel language we consider is KBS0, a language without user-level modules and methods (see Figure 3). In KBS0 registers hold the state and rules define how the state is to be transformed from one cycle to the next. A rule is an atomic action which is either a register update, a predicated action, or a parallel composition of multiple actions. An expression (which is either used to update a register, or in the predicate of a predicated action) is either the value of some register, *i.e.,* a register read, a constant, or a primitive operation on expressions. In addition to these, the language allows let-bindings of variables in both actions and expressions.

The operational semantics, given in Figure 4 specifies the effect of an action on the state in terms of a set of register updates. The sequent $\langle S,\ B\rangle\ \vdash\ e \Rightarrow\ v$ in Figure 4 represents the evaluation of an expression to a value in the context of the current state of the system (S) and the set of bindings created during the evaluation (B). Similarly, the sequent $\langle S,\ B\rangle\ \vdash\ a\ \Rightarrow\ U$ represents the set of updates to the registers of the system to be performed by action a. Figure 5 specifies how a program P transforms the state in one step (denoted by $\rightarrow$) by applying one rule to the state S. A state is *legal* only if it can be reached by applying one rule at a time to the initial state S_0, *i.e.,* if it is in $\rightarrow^*$, the transitive and reflexive closure of $\rightarrow$. We will call this the *one-rule-at-a-time* semantic model.

A. Scheduling multiple rules in KBS0

Before giving the details of compiling KBS0 into a hardware circuit (Section IV-B), we discuss concurrency issues which are essential to construct a high-performance implementation. While it is possible to provide a hardware implementation

$$\text{let-action}\ \frac{\langle S,B\rangle \vdash e \Rightarrow v \qquad \langle S, B\cup\{t\mapsto v\}\rangle \vdash a \Rightarrow U}{\langle S,B\rangle \vdash \texttt{let}\ t{=}e\ \texttt{in}\ a \Rightarrow U}$$

$$\text{reg-write}\ \frac{\langle S,B\rangle \vdash e \Rightarrow v}{\langle S,B\rangle \vdash x\_w(e) \Rightarrow \{x \mapsto v\}}$$

$$\text{if-false}\ \frac{\langle S,B\rangle \vdash e \Rightarrow \mathit{False}}{\langle S,B\rangle \vdash \texttt{if}\ e\ \texttt{then}\ a \Rightarrow \{\}}$$

$$\text{if-true}\ \frac{\langle S,B\rangle \vdash e \Rightarrow \mathit{True} \qquad \langle S,B\rangle \vdash a \Rightarrow U}{\langle S,B\rangle \vdash \texttt{if}\ e\ \texttt{then}\ a \Rightarrow U}$$

$$\text{parallel}\ \frac{\begin{array}{c}\langle S,B\rangle \vdash a_1 \Rightarrow U_1 \qquad \langle S,B\rangle \vdash a_2 \Rightarrow U_2 \\ U_1 \cap U_2{}^{a} = \{\}\end{array}}{\langle S,B\rangle \vdash a_1 | a_2 \Rightarrow U_1 \cup U_2}$$

$$\text{constant}\ \frac{}{\langle S,B\rangle \vdash c \Rightarrow c}$$

$$\text{variable}\ \frac{t \mapsto v \in B}{\langle S,B\rangle \vdash t \Rightarrow v}$$

$$\text{let-expr}\ \frac{\langle S,B\rangle \vdash e \Rightarrow v \qquad \langle S, B\cup\{t\mapsto v\}\rangle \vdash e \Rightarrow v'}{\langle S,B\rangle \vdash \texttt{let}\ t{=}e\ \texttt{in}\ e \Rightarrow v'}$$

$$\text{op}\ \frac{\begin{array}{c}\langle S,B\rangle \vdash e_1 \Rightarrow v_1 \qquad \langle S,B\rangle \vdash e_2 \Rightarrow v_2 \\ \mathit{op}(v_1, v_2) = v\end{array}}{\langle S,B\rangle \vdash \mathit{op}(e_1, e_2) \Rightarrow v}$$

$$\text{reg-read}\ \frac{}{\langle S,B\rangle \vdash x\_r() \Rightarrow S[x]}$$

[a] $U_1 \cap U_2$ denotes that the condition that two sets of updates are disjoint w.r.t the registers updated

Fig. 4: Operational Semantics of actions in KBS0

$$\text{rule-firing}\ \frac{\mathit{rule}\ r\ a \in P \qquad \langle S, \{\}\rangle \vdash a \Rightarrow U}{P \vdash S \rightarrow \mathit{update}(S, U)}$$

where $\mathit{update}(S,U)[x] =$ if $(x, v) \in U$ then v else $S[x]$

$$\text{legal-state}\ \frac{P \vdash S_0 \rightarrow^* S}{S \in \mathit{LegalState}(P, S_0)}$$

Fig. 5: Operational Semantics of rules in KBS0

where any sequence of rules can be executed in one cycle, such an implementation is likely to have a much longer critical path (slower clock) than the one implied by the slowest rule. To generate good hardware we restrict ourselves to "parallel" composition of the rules and disallow the duplication of rule bodies and the creation of implicit combinational paths between the circuits generated from different rules. Without loss of generality the parallel composition of two rules $\mathit{rule}\ r_1\ a_1$ and $\mathit{rule}\ r_2\ a_2$ can be expressed as $\mathit{rule}\ r_{12}\ (a_1|a_2)$. Of course the action $a_1|a_2$ may not be legal, *i.e.,* may have double write errors, and, even if legal, may not transform the state into a legal state as specified by the two rule system. According to the semantics in Figures 4 and 5 a legal state has to be reachable by applying either a_1 and then a_2 or vice versa.

Suppose rule r_1 does not write any register that rule r_2 reads. Then parallel execution of r_1 and r_2 would behave as if rule r_1 was scheduled before r_2. Similarly, if rule r_2 does not write any register that rule r_1 reads then parallel execution of r_1 and r_2 would behave as if rule r_2 was scheduled before r_1.

Definition 1. $r_1 < r_2$: Rule r_2 does not read or write any register that rule r_1 may write. □

Definition 2. $r_1 > r_2$: Rule r_1 does not read or write any register that rule r_2 may write. □

Notice that the $r_1 < r_2$ relation is statically determined *i.e.,* irrespective of the predicates of the conditional actions. Therefore even if there is one state in which r_1 writes a register that r_2 reads, then $r_1 < r_2$ is not true.

Theorem 1. Given $\mathit{rule}\ r_1\ a_1$ and $\mathit{rule}\ r_2\ a_2$, let rule r_{12} be $\mathit{rule}\ r_{12}\ (a_1|a_2)$.

$$r_1 < r_2 \ \Rightarrow\ \forall\ s.\ a_2(a_1(s)) \ = \ (a_1|a_2)(s)$$

where s represents a state and $a(s)$ is the state obtained after applying the updates of action a to s. □

Note that the $(r_1 < r_2)$ relation is not transitive, *i.e.,* $(r_1 < r_2) \wedge (r_2 < r_3)$ does not imply $(r_1 < r_3)$ as illustrated by the following program:

```
x <- Reg(0)
y <- Reg(0)
z <- Reg(0)
rule r_1 z_w(x_r())
rule r_2 x_w(y_r())
rule r_3 y_w(z_r())
```

$r_1 < r_2$ because of register x, $r_2 < r_3$ because of register y, but $r_1 < r_3$ is not true. Because of lack of transitivity, the schedulability of multiple rules has to be defined as follows [8]:

Theorem 2. Rules $r_1, \ldots, r_n$ can be scheduled concurrently conforming to the one-rule-at-a-time semantics, if there exists a permutation $(p_1, p_2, \ldots p_n)$ of $(1, \ldots, n)$ such that $\forall i, \forall (j > i).r_{p_i} < r_{p_j}$ □

The scheduling restriction may provide no valid total order for all the rules to be scheduled in a clock cycle. To resolve these concurrency restrictions we leverage the nondeterminism of the execution model to restrict one or more of the rules to not execute at all in a clock cycle, and instead choose a subset of rules such that there is a valid order for all the rules in the subset to be scheduled. This is represented via a scheduler circuit which provides an enable signal r_{en} for each rule r. The scheduler obeys the restriction implied by the above Theorem 2. The details of scheduler fall outside of the scope of this paper.

B. Compiling KBS0 into Hardware

An expression in KBS0 is essentially an acyclic network of primitive operators, *i.e.,* a combinational circuit. Some of the inputs for the expressions are the output of registers (for example, the output of register x is labeled as x_r_res) and some of the outputs are intended as input to registers (labeled as x_w_arg and x_w_en). We represent the network

```
x <- Reg(0)
y <- Reg(0)
rule r1
  let t = e in
    if t then
      x_w(y_r()) | y_w(x_r())
    if !t then
      x_w(y_r() + 1)

rule r2
  y_w(y_r() + 1)
```

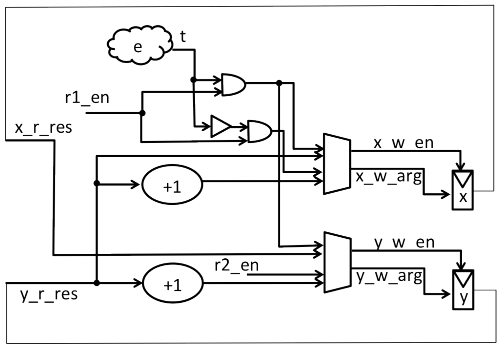

Fig. 6: Example to illustrate the translation of KBS0 into hardware circuits

of operators as a set of bindings; a straightforward procedure for compiling expressions into bindings is given in Figure 8.

For compiling actions, we need to collect all the possible updates that an action may make to the same register. Consider the example in Figure 6. Rule $r1$ writes to register x in two conditional actions. When t is true, it writes the value of register y into x, otherwise it writes $y{+}1$ into x. The hardware structure for writing multiple values has to be represented using muxes. Textually we will represent such muxes as $p_1.e_1 + p_2.e_2$. In a legal program, at most one predicate among p_1 and p_2 can be true. The syntax for describing bindings and predicated expressions is given in Figure 7.

To build such predicated expressions, the compiler needs to thread the bindings and pass in the predicate under which an action is to be performed. The procedure for compiling actions is given in Figure 9. For example, to compile $x\_w(e)$ under predicate P, we first compile expression e to e', and append $P.e'$ to the input argument for register x. We also have to set the enable for x whenever P is true. The predicate to be passed down changes whenever we encounter a conditional action. We do this by creating a new predicate whose expression is the conjunction of the incoming predicate and the predicate for the conditional action. To avoid duplicating the logic for the predicate, we assign it a name and pass down the new name instead of the "anded predicate expression".

Since it is also possible for two different rules to write into the same register, we need a mechanism to allow at most one of them to perform the update. At the hardware level, this capability is provided by an input enable signal (r_en) for each rule, and as discussed earlier, it is the responsibility of the scheduler to set r_ens to avoid double write errors. The final register update uses the same mux as discussed for multiple action updates – the bindings have to be threaded through all the rules. Figure 10 gives the syntax directed compilation of KBS0 rules into bindings.

Static error checking: Instead of generating a circuit to perform a dynamic check whenever multiple writes are attempted to the same register, the compiler accepts a program only if it can prove that multiple writes cannot happen. This can be checked by looking at the generated predicated expression corresponding to x_w_arg, for each register x. A rule becomes

$$\begin{array}{rl}
B ::= & [\langle b\rangle] \\
b ::= & \langle t\rangle = \langle ei\rangle \\
\| & \langle x\rangle\_w\_\text{arg} \; = \langle pe\rangle \\
\| & \langle x\rangle\_w\_\texttt{en} \; = \langle ei\rangle \\
ei ::= & \langle c\rangle \\
\| & \langle t\rangle \\
\| & \langle op\rangle\,(\langle ei\rangle, \langle ei\rangle) \\
\| & \langle x\rangle\_r\_\texttt{res} \\
pe ::= & \bot \\
\| & \langle be\rangle.\langle ei\rangle \text{ // } be \text{ is a boolean } ei \\
\| & \langle be\rangle.\langle pe\rangle \\
\| & \langle pe\rangle + \langle pe\rangle
\end{array}$$

Fig. 7: Syntax for bindings

$$\begin{array}{lll}
\text{CE} : \langle B\rangle \to \langle e\rangle \to (\langle ei\rangle, \langle B\rangle) & & \\
\text{CE } B & [\![x\_r()]\!] = & (x\_r\_\texttt{res},\ B) \\
\text{CE } B & [\![c]\!] = & (c,\ B) \\
\text{CE } B & [\![t]\!] = & (t,\ B) \\
\text{CE } B & [\![\texttt{let } t{=}e1 \texttt{ in } e2]\!] = & \\
\quad \text{let} & (e1', B1) & = (\text{CE } B\ [\![e1]\!]) \\
\quad \text{in} & (\text{CE } (B1[t] := e1')\ [\![e2]\!]) & \\
\text{CE } B & [\![op(e1, e2)]\!] = & \\
\quad \text{let} & (e1', B1) & = (\text{CE } B\ [\![e1]\!]) \\
 & (e2', B2) & = (\text{CE } B1\ [\![e2]\!]) \\
\quad \text{in} & (op(e1', e2'), B2) &
\end{array}$$

Fig. 8: KBS0 expression compilation procedure

$$\begin{array}{llll}
\text{CA} : \langle B\rangle \to \langle be\rangle \to \langle a\rangle \to \langle B\rangle & & & \\
\text{CA } B\ P & [\![x\_w(e)]\!] & = & \\
\quad \text{let} & (e', B1) & = (\text{CE } B\ [\![e]\!]) & \\
 & BT1 & = (B1[x\_w\_\text{arg}] := B1[x\_w\_\text{arg}]) + P.e') & \\
\quad \text{in} & (BT1[x\_w\_\text{en}] := BT1[x\_w\_\text{en}] \vee P) & & \\
\text{CA } B\ P & [\![\texttt{if } e \texttt{ then } a]\!] & = & \\
\quad \text{let} & (e', B1) & = (\text{CE } B\ [\![e]\!]) & \text{where t is fresh} \\
\quad \text{in} & (\text{CA } (B1[t] := P.e')\ t\ [\![a]\!]) & & \\
\text{CA } B\ P & [\![a1 \texttt{ | } a2]\!] & = (\text{CA } (\text{CA } B\ P\ [\![a1]\!])\ P\ [\![a2]\!]) & \\
\text{CA } B\ P & [\![\texttt{let } t{=}e \texttt{ in } a]\!] & = & \\
\quad \text{let} & (e', B1) & = (\text{CE } B\ [\![e]\!]) & \\
 & B2 & = B1[t\_use] := \bot & \\
\quad \text{in} & (\text{CA } (B2[t] := e')\ P\ [\![a]\!]) & &
\end{array}$$

Fig. 9: KBS0 action compilation procedure

$$\text{CR} : \langle B\rangle \rightarrow \langle rule\rangle \rightarrow \langle B\rangle$$
$$\text{CR}\ B\ [\![\texttt{rule}\ r\ a]\!] \quad = \text{CA}\ \ B\ \ \ \texttt{r\_en}\ \ [\![a]\!]$$

Fig. 10: KBS0 rule compilation procedure

```
       Prog ::=  Module ⟨M⟩
                 [⟨instantiation⟩]
                 [⟨rule⟩]
                 [⟨value0-method⟩]
                 [⟨value-method⟩]
                 [⟨action-method⟩]
 instantiation ::= m <- ⟨M⟩
 ...
 value0-method ::= v0-meth ⟨h⟩ = λ().⟨e⟩
  value-method ::= v-meth ⟨h⟩ = λx.⟨e⟩
 action-method ::= a-meth ⟨h⟩ = λx.⟨a⟩
                              // h is a method name identifier
 a ::=   ... // KBS0 actions
     ||  ⟨method-name⟩(⟨e⟩)    // method call
 method-name ::= ⟨m⟩_⟨h⟩
 e ::=   ... // KBS0 expressions
     ||  ⟨method-name⟩(⟨e⟩)    // method call
     ||  ⟨method-name⟩()       // parameterless method call
```

Fig. 11: Grammar of KBS1

illegal *i.e.,* if $x\_w\_arg = \ p_1.e_1 \ + \ p_2.e_2 \ + \ \dots \ p_n.e_n$ and $\exists i, j \neq \ i.\ p_i \ \wedge \ p_j$.

V. KBS1: Modules without Parameters

KBS0 adds a basic notion of module hierarchy to KBS0, starting with the register primitive module (`Reg`). KBS1 programs are constructed as a set of module definitions which encapsulate other submodules, rules and defined methods to manipulate the internal state of the module from outside. Submodules are instantiations of other modules. Only tree-like hierarchies in module instantiations are permitted. Methods come in two varieties: *action methods*, which encapsulate state updates, and *value methods*, which return values. In general methods can have arguments; we treat value methods without parameters as a distinct category for compilation purposes (Figure 11).

For succinctness we will use the word module for both module instance and definitions and rely on the context to disambiguate. Where appropriate we will use capital letters (M) to represent the definition and lowercase (m) for instances.

Figure 12 gives the example of an up-down counter module `MC`. `MC` instantiates a register `x` and defines two action methods `downcount` and `upcount`. (For simplicity, we ignore the issues of overflow, underflow, *etc.* .) `MC` is instantiated as `mc` by another module `M` which has two rules `r1` and `r2`, both of which update the counter under various conditions. Notice that rules `r1` and `r2` conflict because ultimately both rules try to update counter `x` simultaneously. Since `x` is encapsulated inside module `MC`, this conflict cannot be detected by `M` unless some information about the conflict between the methods of `MC` is available to `M`. Next, we discuss how this information is provided and used in the interface definitions.

```
 Module MC
   x <- Reg(0)
   a-meth downcount = λc.x_w(x_r() - c)
   a-meth upcount = λc.x_w(x_r() + c)

 Module M
   mc <- MC
   y <- Reg(0)
   z <- Reg(0)
   rule r1
     if (y > 0) then
       mc_downcount(1) | y_w(f(y_r()))
   rule r2
     if (z > 0) then
       mc_upcount(2) | z_w(g(z_r()))
```

Fig. 12: Example to illustrate conflicts in KBS1

A. Conflict Matrix CM

For each module, as part of its interface, we will define a conflict matrix which gives the pairwise conflict relations between its defined methods. For two methods h_1, h_2 defined in a module,

$$CM(h_1, h_2) \in \{\{\}, \{<\},\ \{>\},\ \{<,>\}\}$$

If $\{<\} \ \subseteq \ CM(h_1,\ h_2)$, then h_1 does not write any register (directly or indirectly) that h_2 reads or writes, and both methods can be invoked concurrently with the effect as if h_1 executed before h_2. ($\{>\}$ is defined as its dual.) If $\{<,\ >\} \ \subseteq \ CM(h_1,\ h_2)$, we say that the methods are *conflict-free*, *i.e.,* not only can they execute concurrently, but produce the same effect in either order of execution. On the other hand, if $CM(h_1, h_2) \ = \ \ \{\}$, then the two methods are said to *conflict* and cannot be invoked concurrently. A rule containing such conflicting method calls, which are not invoked in a mutually exclusive manner, is said to be illegal. If they are invoked in two different rules, then the two rules cannot be scheduled together.

Methods of a module are like *ports* when the module is synthesized into hardware (Section V-C). A method that has parameters, regardless of whether it's a value or an action method, cannot be invoked simultaneously by two different calls because of port conflict. Action methods even without parameters affect the same state and therefore always conflict with themselves. However, parameterless value methods, for example register reads, can be invoked in multiple places simultaneously and therefore we treat them specially. Multiple write errors discussed for KBS0 is a special case of conflicting port usage.

The CM relationship for the `Reg` module is given in Figure 13. Given the conflict matrix for registers, we can systematically derive the conflict relation between any pair of methods g, h defined by a module M as follows:

Procedure to calculate CM in KBS1

1) Compute $mcalls$: $mcalls(g)$ represents the methods called by the definitions of g in M (obtained statically). Such calls are

	r	w
r	$\{<,>\}$	$\{<\}$
w	$\{>\}$	$\{\}$

Fig. 13: CM `Reg` primitive

necessarily restricted to the methods in the submodules of M in KBS1.

2) Compute $Conflict$: For each $p \in mcalls(g)$ and each $q \in mcalls(h)$, compute the conflicts between p and q as follows:
$Conflict(p,q) = CM(p,\ q)$ where p,q are from the same module instance
$Conflict(p,q) = \{<,\ >\}$ where p,q are from different module instances
Since different module instances do not share state, their respective methods are always conflict-free with each other in KBS1.

3) Compute CM: For every pair of defined methods g, h, we compute CM as follows:

$$CM(g,h) = \bigcap_{\substack{p \in mcalls(g)\\ q \in mcalls(h)}} Conflict(p,q) \text{ where } g \neq h$$

$CM(g,g) = \{<,>\}$ where g is a value0-method

$CM(g,g) = \{\}$ where g is not a value0-method

The procedure given above is recursive, and essentially computes CM bottoms up in the module invocation hierarchy.

B. Concurrent scheduling of rules

The following property of *Sequential composability (SC)* is essential to understand the invariants of a correct scheduler:

Definition 3. Sequential Composibility (SC): Given a set of sets of method names $g_1, g_2, \ldots g_n$, $SC(g_1, g_2, \ldots g_n)$ is true if there exists a permutation, $p_1, p_2, \ldots p_n$ of $1, 2, \ldots n$ such that,

$$\forall i. \forall (j > i). \forall x \in g_{p_i}, \forall y \in g_{p_j}.(\{<\} \subseteq Conflict(x,y)) \quad \square$$

Consider a system where only the top module has rules. In such a system, rules $r_1, \ldots, r_n$ can be scheduled concurrently without violating the one-rule-at-a-time semantics if $SC(mcalls(r_1), \ldots, mcalls(r_n))$ is true.

When a submodule has rules, then the submodule's rule might conflict with its defined methods. If we assume that the scheduling is done outside in, the inner module can know which of its methods are being called by an external rule or method. We use a signal (h_en) to denote whether method h defined by the internal module is currently being called *i.e.,* enabled, externally. For the internal scheduler to work properly, it must consider the conflicts between an internal rule and its defined methods which are currently enabled.

Definition 4. (SC Invariant of KBS1) Suppose $r_1, \ldots, r_n$ are the rules of M being chosen to be scheduled in the current state, $h_1, \ldots, h_k$ are the methods of M being called externally, then M preserves SC Invariant for KBS1 iff

$$SC(mcalls(r_1), \ldots, calls(r_n), \\ mcalls(h_1), \ldots, mcalls(h_k))$$

is true $\square$

Expressions:

CE B P $[\![m\_h(e)]\!] =$
  let $(e', B1)$ = (CE B P $[\![e]\!]$)
    $BT1$ = $(B1[m\_h\_\texttt{arg}] := B1[m\_h\_\texttt{arg}] + P.e')$
    $BT2$ = $(BT1[m\_h\_\texttt{en}] := BT1[m\_h\_\texttt{arg}] \vee P)$
  in $(m\_h\_\texttt{res}, BT2)$

CE B P $[\![m\_h()]\!] =$
    $(m\_h\_\texttt{res}, (B[m\_h\_\texttt{en}] := B[m\_h\_\texttt{en}] \vee P))$

Actions:

CA B P $[\![m\_h(e)]\!] =$
  let $(e', B1)$ = (CE B P $[\![e]\!]$)
    $BT1$ = $B1[m\_h\_\text{arg}] := B1[m\_h\_\text{arg}] + P.e'$)
  in $(BT1[m\_h\_\texttt{en}] := BT1[m\_h\_\texttt{en}] \vee P)$

Fig. 14: Compilation for KBS1 expressions and actions

Theorem 3. If the scheduler of every module in a system preserves the SC invariant of KBS1 then system will obey the one-rule-at-a-time semantics. $\square$.

The above theorem shows how to construct modular schedulers assuming "method enabled" signals generated by the outer modules are available to the internal schedulers.

C. Compiling KBS1 to Hardware

Intuitively the compilation of KBS1 can be thought of as almost exactly taking the KBS0 program derived by inlining modules and adding the appropriate hardware-level module boundaries. This corresponds to each method `h` having an argument port `h_arg` and value methods having a result port `h_res`. Finally, as discussed before, we provide a signal `h_en` for each method `h` to indicate that it is being used.

For compiling KBS1, we simply add cases to handle method calls to the KBS0 compilation procedure for expressions and actions (Figure 14) We show only the incremental changes over KBS0. In the compilation procedure for expressions and actions, we simply add cases to handle method calls. The argument of a method call gets bound to a predicated expression containing the inputs to that method at various locations in the code. The method enable is bound to the logical OR of the predicates of each term in the predicated expression. Uunlike KBS0, in KBS1 the predicate must be passed to the expression compilation too. To compile the definition of a method, the argument of a method `h` is bound to the name `h_arg`, and the result of a method (in case of value methods) is bound to the expression corresponding to the method.

When modules are compiled separately, we must provide a linking phase to connect the bindings of the module of instance m (compiled separately using module M) to the bindings of M_0, the module instantiating M as m (see Figure 16). In our representation, this can be accomplished simply by prepending the name of the instantiated module to all state and binding names from the submodules' compilation. This is what the function *instantiate* does in Figure 16. It also initializes the predicated expression of the argument of every defined method

```
**Value methods:**
  CVM :     $\langle B\rangle \rightarrow \langle value\text{-}method\rangle \rightarrow \langle B\rangle$
  CVM $B$      $[\![$v-meth $h$=$\lambda x.e]\!]$ =
    let    $(e', B') =$ CE $(B[x] := h\_\mathtt{arg})\ h\_\mathtt{en}\ [\![e]\!]$
     in    $(B'[h\_\mathtt{res}] := e')$
**Value0 methods:**
  CV0M :   $\langle B\rangle \rightarrow \langle value0\text{-}method\rangle \rightarrow \langle B\rangle$
  CV0M $B$     $[\![$v0-meth $h$=$\lambda().e]\!]$ =
    let    $(e', B') =$ CE $B\ h\_\mathtt{en}\ [\![e]\!]$
     in    $B'[h\_\mathtt{res}] := e')$
**Action methods:**
  CAM :     $\langle B\rangle \rightarrow \langle action\text{-}method\rangle \rightarrow \langle B\rangle$
  CAM $B$      $[\![$a-meth $h$=$\lambda x.a]\!]$ =
                      CA $(B[x] := h\_\mathtt{arg})\ h\_\mathtt{en}\ [\![a]\!]$
```

Fig. 15: Compilation for KBS1 Methods

```
$CInst : CompiledModules \rightarrow \langle B\rangle \rightarrow \langle instance\rangle \rightarrow \langle B\rangle$
$CompiledModules = \langle M\rangle \rightarrow \langle B\rangle$
---
$CInst$   _            $B$   $[\![m1$ <- $Reg(c)]\!]$                  =
                                      $instantiate(Reg(c), m1)$
$CInst$   $compileds$   $B$   $[\![m1$ <- $M1(ap1, \ldots, apn)]\!]$   =
       let            $BM1$    $= compileds\ M1$
                      $BM2$    $= instantiate(BM1, m1)$
                      $BT0$    $= B + +BM2$
              $(h1, \ldots, hk)$   $= Defined\ methods\ in\ M1$
         in    $fold\ (CDefnSub\ m1)\ BT0\ [h1, \ldots hk]$
---
$CDefnSub : \langle m\rangle \rightarrow \langle B\rangle \rightarrow \langle h\rangle \rightarrow \langle B\rangle$
$CDefnSub\ m1\ B\ h = (B[m1\_h\_\mathtt{arg}] := \bot)$
```

Fig. 16: Compilation for KBS1 instances

in the instance as $\bot$.

Figure 17 shows the compilation of a module, which simply collects all the bindings defined internally (from instances, rules, and methods).

Static error checking: As with KBS0, not all KBS1 programs are valid. It is illegal if two conflicting methods $h_1, h_2, CM(h_1, h_2) = \emptyset$ are dynamically called within a rule. This can again be checked statically as follows: If $h_1\_arg = p_1.e_1 + p_2.e_2 + \ldots\ p_n.e_n$ and $h_2\_arg = \ p_{n+1}.e_{n+1} + \ldots + p_{n+m}.e_{n+m}$ such that $CM(h_1\_arg, h_2\_arg) = \ \{\}$, then it is illegal if $\exists i, j \neq \ i.\ p_i\ \wedge\ p_j$

```
$CM : \langle M\rangle \rightarrow$ CompiledModules $\rightarrow$ (CompiledModules, $\langle B\rangle$)
$CM[\![$Module $M$
        $m1$ <- $M1$
        rule $r1\ a1, \ldots$
        v0-meth $v1$=$\lambda().e1, \ldots$
        v-meth $f1$=$\lambda x.e2, \ldots$
        a-meth $g1$=$\lambda y.a2, \ldots]\!]$
        $compiledModules$ =
  let   $B = fold\ (CInst\ compiledModules)\ \phi$
            $[[\![m1$ <- $M1]\!], \ldots]$
     $B0 = fold\ CR\ B\ [[\![$rule $r1\ a1]\!], \ldots]$
     $B1 = fold$ CVM $B0[[\![$v-meth $f1 = \lambda x.e2]\!], \ldots]$
     $B2 = fold$ CV0M $B1[[\![$v0-meth $v1$=$\lambda().e1]\!], \ldots]$
     $B3 = fold$ CVM $B2[[\![$a-meth $g1 = \lambda x.a1]\!], \ldots]$
  in   $(\lambda x.if\ x = M\ then\ B3\ else\ compileModules(x), B3)$
```

Fig. 17: The compilation Procedure for KBS1 Modules

D. Further Language Enrichment

There are a few extensions to our kernel languages which have impact on the compilation scheme. While important for a practical implementation, these are orthogonal to the modularity question. We discuss them briefly for completeness.

1) Guards: Partially Valid Expressions and Actions: Often methods for an object may not be available every cycle. For instance, a FIFO may not have space to allow a new value to be enqueued or an FFT module may not have finished computing a result to report via the response operation. In these cases users of a module need to check explicitly that every method called is valid when used.

To increase modular reuse, this explicit check has been codified into a notion of a predicate guard (denoted via the *when* clause) which is implicitly checked, guaranteeing that a method's "readiness" property is never violated [5]. These guards propagate across parallel actions, *i.e.,* the following expressions are equivalent:

```
(a1 **when** e1)|(a2 **when** e2)≡(a1|a2) **when** (e1∧e2)
```

An guarded action or expression is valid only when it's predicate is true. An action or expression with a failing guard in a subexpression is itself not valid. Thus the following expressions are equivalent:

```
**if** p **then** (a **when** e)≡(**if** p **then** a) **when** (¬p∨e)
```

In general, guards can be naturally lifted to the top expressions and actions using similar transformations. These lifted guards can be safely transformed into *if* conditionals at the top of rule bodies or auxiliary methods for method bodies. In practice, this information is used by the scheduler to consider only those rules whose (top-level) guards are true.

For hardware understandability, it is more natural to reason about these guards as part of the method itself rather than an entirely separate method with a new set of ports. This translates into an additional boolean "ready" output port in the binding list signifying when the method is valid.

E. Ephemeral History Registers: Combinational passing within an atomic action

As stated previously, scheduling is constrained from introducing combinational paths from between rules. However, in practice, for high performance it is useful to allow controlled use of combinational paths. For example, a design which allows a new element to be enqueued into a full pipeline buffer if in the same cycle someone has taken an element out performs better than one which does not. To allow users to express this we introduce new primitives which internally allow information to be passed between methods calls combinationally. Of particular interest is the Ephemeral History Register (EHR) [12] with which one can model any other primitive.

In the presence of EHRs, a combinational loop can arise in the body of a single rule if the order imposed by the conflict relation between the methods of an EHR instance violates the corresponding dataflow order. This is resolved by statically

```
// [...] represents a list in the meta language
 Prog ::= Module ⟨M⟩ // M is a module name
            ([⟨fp⟩], [⟨cf⟩])
               // fp is a formal method parameter
               // cf is a conflict specification between 2 fp's
          ...
 instantiation ::= ⟨m⟩ <- ⟨M⟩ ([⟨method-name⟩])
                       // m is an "object" of "type" M
                 ||  ⟨m⟩ <- Reg(⟨c⟩)
                       // Register-primitive instantiation
 method-name ::= ⟨fp⟩ // Formal parameter call
                 ||  ⟨m⟩_⟨h⟩ // Instantiated module method call
 ...
```

Fig. 18: Grammar of the kernel language KBS2

checking to ensure that the conflict relation be consistent with the dataflow ordering.

VI. KBS2: Modules with parameters

Now we introduce the compilation procedure for KBS2, our full kernel language with interface parameters in modules (see Figure 18). Our language does not permit recursive method calls and consequently prohibits the following types of module instantiations.

```
Module M()
  m1 <- M1(m2_h2)
  m2 <- M1(m1_h1)
```

We also do not permit passing a method to multiple modules. For example, the following programs are illegal:

1)
```
Module M(fp)
  m1 <- M1(fp)
  rule r1 fp(1)
```
2)
```
Module M(fp)
  m1 <- M1(fp)
  m2 <- M2(fp)
```
3)
```
Module M()
  m1 <- M1()
  m2 <- M2(m1_h)
rule r1 m1_h(1)
```

This restriction does not fundamentally reduce the expressibility of the language, since one simply has to duplicate the method of an instance (with a new name) and pass the "new" method in case a duplicate is necessary.

A. Sharing state through interface parameters

Consider the example in Figure 19. Module M1 has two action methods h1 and h2, both of which write the register x. Rule r3 in module M2 calls the formal parameter p whose actual argument is m1_h2. In order to preserve one-rule-at-a-time semantics, none of the rules r1, r2 or r3 should execute simultaneously. Even though rules r1 and r2 call methods of different instances, formal parameters cause a conflict – they effectively write the same register x of instance m1. Similarly, rule r3 cannot be scheduled whenever method h3 is called (this can be detected using the scheme described for KBS1) or rule r1 is scheduled (again because they write the same state x). The conflict between rules r1 and r3 cannot be detected using the technique in KBS1.

B. Computing the conflict matrix

The interface of a module needs to convey more information than in KBS1. We give a solution to compute the conflict matrix assuming the interface includes the following information:

1) CM_{DM}: The conflict relation between each pair of defined methods
2) CM_{FP}: The conflict relation between each pair of formal parameters
3) $fpu_M(h)$: The set of formal parameters that a defined method h of module M uses (either directly or indirectly)
4) p_sch: A signal for each formal parameter p to indicate whether the external scheduling has prohibited the use of p in the internal rules or submodules. If rule r is selected to be executed by the scheduler, then all of the interface parameters which are called by r in the current state should be schedulable.

The calculation of CM_{DM} relation is the same as before, except that the $Conflict$ relation between methods defined in different submodules cannot be obtained because the two submodules can conflict because of the sharing of state between them via parameters. For instance, as discussed earlier, methods m1_h2 and m2_h3 conflict in Figure 19 even though methods h2 and h3 are defined in different instances.

We also need conflict information about the interface parameters. This, like the type information, has to be supplied by the designer. Our compiling procedure will make sure that the constraints imposed by CM_{FP} are not violated in the instantiations of the module with actual parameters.

The methods called by g, a defined method of some module M, can be either formal parameters of M or the methods defined by the modules instantiated inside M. We either know or can calculate the formal parameters used by each defined method of a module using the function $fpu_M(g)$. We explain fpu_M next. We define a function sub_M, which is a mapping for each instance $m \leftarrow M(ap_1, \ldots, ap_n)$ where $sub_m(fp_i) = ap_i$.

For registers:

$$fpu_{Reg}(Reg\_r) = fpu_{Reg}(Reg\_w) = \{\}$$

```
Module M ()
  m1 <- M1()
  m2 <- M2(m1_h2)
  rule r1 m1_h1(5)
  rule r2 m2_h3(6)
Module M1()
  x <- Reg(0)
  a-meth h1 = λa.x_w(a)
  a-meth h2 = λb.x_w(b+1)
Module M2(p)
  rule r3 p(6)
  a-meth h3 = λc.p(c+2)
```

Fig. 19: Illustrating the need for a distributed scheduler

In the following, mi is an instance of module Mi and M is the top level module instantiating Mi's.

If $mcalls(M\_g)$ contains fp, a formal parameter of M, then fp must also belong to $fpu_M(M\_g)$. The non-trivial case is when $mcalls(M\_g)$ contains $m1\_h1$. Consider the scenario where $h1$ defined in module $M1$ of which $m1$ is an instance of, uses a formal parameter of $M1$ which is supplied a formal parameter of M during its instantiation. One way to keep track of such indirect references is by using the function $fpu_{M1}(M1\_h1)$. $fpu_{M1}(M1\_h1)$ is by definition a subset of the formal parameters of $M1$. By substituting the actual parameters used in instantiating $M1$ we can determine all the methods called by $m1\_h1$ including the formal parameters of M. We will refer to this function as $apu(m1\_h1)$. Finally among $apu(m1\_h1)$, we may discover some formal parameters of M and thus we can calculate the indirect use of the formal parameters of M by $m1\_h1$. Among the methods in $apu(m1\_h1)$ which are not formal parameters of M, say, mj_hj, we recursively obtain $apu(mj\_hj)$, till we are left with only formal parameters of M. FPU_M below performs the recursive call. Thus, we obtain the formal parameters of M which are directly or indirectly used in g. fpu_M is calculated for all the methods defined by a module M when M is compiled.

Procedure to calculate fpu_M, the set of formal parameters called by a defined method g of a module M:

$$\begin{aligned} fpu_M(M\_g) &= FPU_M(mcalls(g)) \\ \\ FPU_M(\emptyset) &= \emptyset \\ FPU_M(\{fp\} \cup hs) &= \{fp\} \cup FPU_M(hs) \\ FPU_M(\{m\_g\} \cup hs) &= FPU_M(apu(m\_g) \cup hs) \\ \\ apu(m_i\_h) &= \{sub_{m_i}(fp) |\ fp \in fpu_{M_i}(m_i\_h)\} \end{aligned}$$

For the example in Figure 19, $fpu_M2(M2\_h3) = \{p\}$ according to the above procedure.

Using fpu_M (more specifically apu), we can obtain the $Conflict$ relation between every pair of methods (both interface parameters, and methods defined by instances) called inside M

$$\begin{aligned} Conflict(m_1\_h_1, m_1\_h_2) &= CM_{DM}(M_1\_h_1, M_1\_h_2) \\ Conflict(fp_1, fp_2) &= CM_{FP}(fp_1, fp_2) \\ Conflict(m_1\_h_1, m_2\_h_2) &= \bigcap_{\substack{p \in apu(m_1\_h_1) \\ q \in apu(m_2\_h_2)}} Conflict(p, q) \\ &\quad \text{where } m_1 \neq m_2 \\ Conflict(fp, m_2\_h_2) &= \bigcap_{q \in apu(m_2\_h_2)} Conflict(fp, q) \\ Conflict(m_1\_h_1, fp) &= \bigcap_{p \in apu(m_1\_h_1)} Conflict(p, fp) \end{aligned}$$

Calculating the actual CM_{DM} relation between every pair of methods defined by a module in KBS2 remains exactly the same as calculating CM in KBS1.

$$\begin{aligned} CM_{DM}(g, h) &= \bigcap_{\substack{p \in mcalls(g) \\ q \in mcalls(h)}} Conflict(p, q) \text{ where } g \neq h \\ CM_{DM}(g, g) &= \{<, >\} \text{ where } g \text{ is a value0-method} \\ CM_{DM}(g, g) &= \{\} \text{ where } g \text{ is not a value0-method} \end{aligned}$$

Calculating apu for a method has the worst case complexity of the number of callable methods in a module. Thus, calculating CM_{DM} for a pair of methods has a worst case complexity of $\mathcal{O}(n^2)$ where n is the number of callable methods in a module.

C. Scheduling multiple rules in KBS2

The module's scheduler must obey the following theorems:

Definition 5. (SC Invariant for KBS2) Suppose $r_1, \ldots, r_n$ are the rules of M being chosen to be scheduled in the current state, $h_1, \ldots, h_k$ are the methods of M being called externally, and $aps_i, \ldots, aps_j$ are the actual parameters to be passed into submodules of M and which have been chosen as schedulable, then M preserves SC Invariant for KBS2 iff

$$SC(mcalls(r_1), \ldots, calls(r_n), \\ mcalls(h_1), \ldots, mcalls(h_k), aps_1, \ldots, aps_j)$$

is true □

Theorem 4. If the scheduler of every moodule in a system preserves the SC Invariant for KBS2 then the system will obey the one-rule-at-a-time semantics. □

Scheduling strategies are omitted for brevity.

```
$CInst : CompiledModules \rightarrow \langle B \rangle \rightarrow \langle instance \rangle \rightarrow \langle B \rangle$
$CompiledModules = \langle M \rangle \rightarrow \langle B \rangle$
---------------------------------------------------------------
$CInst \quad compileds \quad B \quad [\![ m1$ <- $M1(ap1, \ldots, apn) ]\!] \quad =$
  let                $BM1$   $= compileds\ M1$
                     $BM2$   $= instantiate(BM1, m1)$
                     $BT0$   $= B ++ BM2$
         $(fp1, \ldots, fpn)$   $= Formal\ parameters\ of\ M1$
                     $BT1$   $= fold\ (CParamSub\ [[\![ap1]\!], \ldots]$
                                 $[[\![fp1]\!], \ldots]\ m1)\ BT0\ [1 \ldots n]$
           $(h1, \ldots, hk)$   $= Defined\ methods\ in\ M1$
  in   $fold\ (CDefnSub\ m1)\ BT1\ [h1, \ldots hk]$
---------------------------------------------------------------
$CParamSub : [\langle method\text{-}name \rangle] \rightarrow [\langle fp \rangle] \rightarrow \langle m \rangle \rightarrow$
                 $\langle B \rangle \rightarrow Integer \rightarrow \langle B \rangle$
$CParamSub\ aps\ fps\ m1\ B\ i =$
  let     $fp$    $= fps[i]$
          $ap$    $= aps[i]$
         $BT2$   $= (B[m1\_fp\_\mathtt{sch}] := \bot)$
         $BT3$   $= (BT2[m1\_fp\_\mathtt{res}] := ap\_\mathtt{res})$
         $BT4$   $= (BT3[ap\_\mathrm{arg}] :=$
                $BT3[ap\_\mathrm{arg}] + m1\_fp\_\mathtt{en}.m1\_fp\_\mathtt{arg})$
         $BT5$   $= (BT4[ap\_\mathtt{en}] := BT4[ap\_\mathtt{en}] \vee m1\_fp\_\mathtt{en})$
  in     $BT5$
```

Fig. 20: Compilation for KBS2 instances

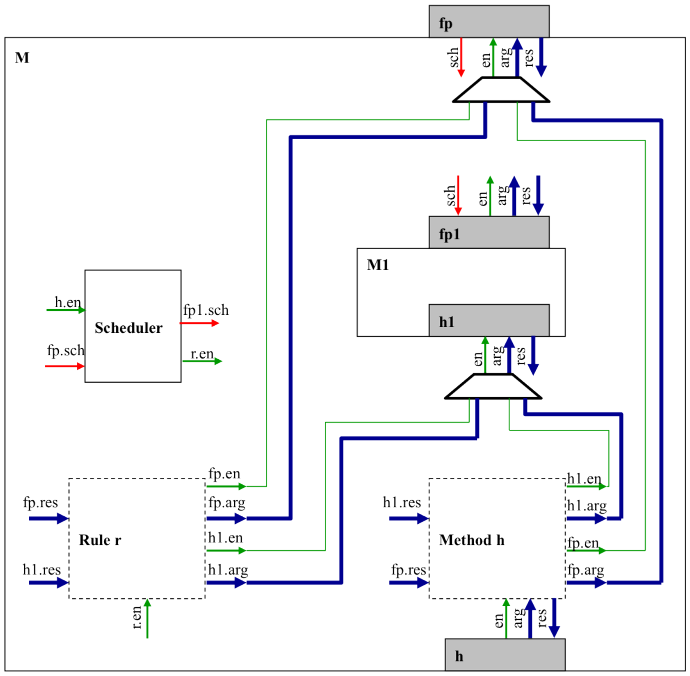

Fig. 21: Ports generated by compiling a module

D. Hardware compilation

The compilation of a KBS2 program into bindings is very similar to the compilation of a KBS1 program into bindings. In KBS2, a method call can either be of a method defined by an instantiated module or a call of the formal parameter. Both the called methods have the same set of ports as described previously. In addition, formal parameters, have an additional `fp_sch` port which will be set by the scheduler (of the external module) every clock cycle. Figure 21 gives the ports of various entities (like rules, methods, *etc.*) that the compilation process will generate.

The only major difference in the compilation of KBS2 with respect to KBS1 is the linking of the formal parameters of the module of an instance with the actual parameters supplied to the instance. Figure 20 shows the compilation procedure for instances. $CParamSub$ performs the required linking.

VII. Conclusion and Future Work

Modularity is a key requirement for effective hardware design. Our approach allows non-tree like method calling structures in a design which permits many otherwise natural decompositions, *e.g.,* the use of an off-chip memory.

Our approach codifies a resource-oriented view of representation and provides caller and callee interfaces and dictates their safe use. Though we did not have space to discuss it, our scheme applies naturally to modern scheduling algorithms and thus can be implemented with no additional hardware over the current state of the art. Furthermore, our extension does not affect the compilation of older programs.

In some sense this improvement solves all of the issues in current Bluespec decomposition of hardware. However, in the context of multiple clock domains and hardware/software computational domains [10], may require that we break our restriction prohibiting no cycles in the parameter passing structures. The next step is extending this work to allow such cyclic parameter passing.

Acknowledgement

This work was done in CSAIL in collaboration with SRI. The research at CSAIL was supported by Quanta Research agreement dated 4/1/05 and Angstrom Project under Grant No. HR0011-10-9-0009. The research at SRI was supported by Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 and National Science Foundation under Grant No. CCF-1217498. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Quanta Research, the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] Arvind and Xiaowei Shen. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro*, 19(3):36–46, May 1999.

[2] Gérard Berry, C. A. R. Hoare, and W. A. Hunt. Mechanized reasoning and hardware design. chapter Esterel on hardware, pages 87–104. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[3] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(9):1059–1076, 2001.

[4] K. Mani Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[5] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[6] Nirav Dave, Man Cheuk Ng, Michael Pellauer, and Arvind. A design flow based on modular refinement. In *MEMOCODE*, pages 11–20, 2010.

[7] Stephen A. Edwards. High-level synthesis from the synchronous language esterel. In *IWLS*, pages 401–406, 2002.

[8] 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.

[9] Michal Karczmarek and Arvind. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *ICCAD*, 2008.

[10] Myron King, Nirav Dave, and Arvind. Automatic generation of hardware/software interfaces. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 325–336, New York, NY, USA, 2012. ACM.

[11] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, pages 19–30, 2006.

[12] Daniel L. Rosenband. The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

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

[14] M. Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, pages 171–180, 2009.

[15] Jianwen Zhu and Daniel Gajski. Soft scheduling in high level synthesis. In *DAC*, pages 219–224, 1999.