# Formal Specification of BSV's Elaboration and Dynamic Semantics

Rishiyur S. Nikhil

Bluespec, Inc., July 28, 2015

**Abstract**

BSV is a High Level Hardware Design Language (HLHDL), intended for the full spectrum of digital hardware designs, from signal-processing accelerators to CPUs, SoCs and all kinds of Intellectual Property Blocks (IP Blocks). BSV has been in use commercially and in academia since the early 2000s.

Although BSV was inspired by formally-defined languages (specifically the Haskell functional programming language and Term Rewriting Systems), its formal elaboration and dynamic semantics have not been properly documented to date. So far, this has not been an issue, but there is now a growing interest in using BSV for formally verified hardware, because of its clean and high level semantics.

This document provides a formal specification of BSV elaboration and dynamic (execution) semantics. The specification is written as a Haskell program and is therefore executable. This document is accompanied by the full Haskell code as well as the (kernel) BSV source code for the fully-worked out examples which are described in this document.

## 1 Introduction and History

BSV [6] is a High Level Hardware Design Language (HLHDL), intended for the full spectrum of digital hardware designs, from signal-processing accelerators to CPUs, SoCs and all kinds of Intellectual Property Blocks (IP Blocks). BSV has been in use commercially and in academia since the early 2000s.

*Acronyms*

| | |
|---|---|
| HDL | Hardware Description Language |
| HLHDL | High Level Hardware Description Language |
| HLS | High Level Synthesis |
| HW | Hardware |
| PL | Programming Language |
| SoC | System on a Chip |
| SW | Software |
| TRS | Term Rewriting Systems |

### 1.1 Historical context

From the beginning, BSV has been inspired by formally-defined languages, specifically Term Rewriting Systems [19, 30, 4] (TRSs) and the Haskell functional programming language [28]. It started

with research by Shen and Arvind at MIT to use Term Rewriting Systems to formally specify and verify complex concurrent hardware, such as deep processor pipelines and distributed coherent caches for multi-core systems [2, 29]. Similar approaches (using rewrite rules to specify complex concurrency) may also be found in TLA+ [20] and its use to verify cache coherence protocols [1, 18], in UNITY [7] and in Event-B [23].

From the specification research of Shen and Arvind emerged the idea of directly *synthesizing* hardware from TRS specifications. This idea was developed significantly by Hoe and Arvind [12, 13, 11] at MIT.

In 2000, activity on productizing these ideas moved from MIT into Sandburst Corporation. There, Lennart Augustsson *et al.* designed and implemented a new language called Bluespec [3]. Whereas the previous prototype from MIT had only rudimentary circuit-description facilities, focusing more on solving the problem of synthesis from TRSs, the new language borrowed heavily from Haskell to also provide a powerful circuit description language (also called "static elaboration" in HDL jargon). It had Haskell-like syntax, type definitions, polymorphism, type classes, higher-order functions, total absence of side effects, laziness, and a monadic structure for static elaboration. Although it was an industrial-strength language, it was only intended for internal use inside Sandburst. Nowadays we refer to this second-generation language as "Bluespec Classic".

In 2003, activity moved again, to Bluespec, Inc., with an aim to make it a publicly available Hardware Design Language (HDL). In this third generation, first released in 2004, its syntax was changed from Haskell's very spare syntax to something more similar to SystemVerilog [17] which is more familiar to hardware design engineers, the primary audience. The name of the language was changed to *BSV* (Bluespec SystemVerilog). Support for multiple clock domains was added, with statically typed clocks and resets and clock domain discipline. But apart from this syntax change and addition of clocks, the underlying language and its basic semantics has remained the roughly same (even today you can successfully compile and run BSV programs from 2004!).

### 1.2 Why no earlier formalization?

Like many industrial-strength languages, BSV is not a tiny language (although the language-definition part of the Reference Guide [6] is barely 150 pages, compared to more than 1300 pages for SystemVerilog [17]), and so defining formal semantics in gory detail is not trivial (and may not be readable by anyone!). BSV's semantics is described informally in the Reference Guide, in a number of books and papers (e.g., [24, 27, 25, 26]), and in Bluespec's BSV training materials. By and large this has been perfectly adequate for practical use.

There is now a strong emerging interest in *mechanized* formal verification of computing systems that include components implemented in hardware. The interest spans CPUs, memory systems and SoCs as well as hardware accelerators, hardware implementations of cryptography, and so on. There are many groups—commercial, non-profit research and academic—who are keen on using BSV for this purpose because it is both industrial-strength and has a very clean and high level semantics. This activity requires formal specification of BSV, hence this document.

### 1.3 Document overview

This document is accompanied separately by Haskell code that specifies the semantics. The Haskell code is executable, and comes with several example "kernel BSV" programs on which it can be run.

That Haskell code, and not this document, should be regarded as the official formal specification. This document should be regarded as a reading guide to that Haskell code. The Haskell code fragments shown here are from the accompanying Haskell code. In case they are out of sync, the accompanying Haskell code should be taken as definitive.

*Note (July 27, 2015): This is the first public release of this document and the Haskell code. It is quite possible that both the semantics and this document need fixes and/or refinement. The author gladly welcomes comments and suggestion from readers.*

A detailed table of contents follows next. Then, Sec. 3 provides some general commentary about our approach. Sec. 4 describes the concrete and abstract syntax of our kernel language. Sec. 5 talks briefly about parsing from concrete to abstract syntax. Sec. 6 describes static elaboration of the source program into a collection of module instances.

Sec. 7, "Dynamic Semantics", is the heart of this document. Readers who are experienced at reading Haskell and/or formal semantics of languages may wish to skip preceding sections and go there directly.

Sec. 8 presents a set of four example kernel BSV programs and their semantic interpretation (included with the accompanying Haskell code). The first two are explained in detail and the latter two are left as exercises for the reader (although we provide Haskell execution transcripts for all of them). Sec. 9 concludes with some comparisons to other hardware description languages and with a discussion of the relationship of this formal semantics to the full BSV language and its compilation with the *bsc* compiler.

## 2 Table Of Contents

# 3 Preliminary comments

Certain aspects of BSV's formal semantics may seem novel to those already familiar with semantics of programming languages or TRSs. We discuss them in the following sections.

## 3.1 Static elaboration and static vs. dynamic semantics

All SW programming languages have a dynamic control structure.[1] Specifically, *frames* are pushed and popped on a stack as we dynamically call and return from functions, methods and exceptions.

In hardware, there is no such dynamic control structure. Hardware consists of physical electronics; they do not grow or shrink during execution![2]

Nevertheless, you will find functions and procedures in all modern HDLs. In BSV you will also see extensive use of methods, loops, and recursion. These are merely linguistic devices for *circuit description*, i.e., they are unfolded, or inlined by the HDL compiler/interpreter to describe a circuit. It is almost like statically pre-building the possible tree of stack frames for a SW program just by looking at its text.

In SW programming languages, *static semantics* usually refers to syntax, scoping and other well-formedness checks, type-checking, and perhaps some simple syntax desugaring. The "output" of static semantics is still a program text that may be recognizably similar to the original program text. Everything after that is described by *dynamic semantics*, including dynamic control behavior like conditionals, loops, function calls and returns, recursion, exceptions, and so on.

In HDLs, *static semantics* covers the same topics as in SW PLs, but goes further to encompass static elaboration, i.e., the unfolding of linguistic constructs like function application, method calls, elaboration loops and recursion, and so on. The result of static elaboration is a hardware circuit. This hardware circuit may have a logical hierarchical structure, i.e., a top-level module containing instantiated sub-modules which, in turn, may contain next-level instantiated sub-modules, and so on. At the leaves of this hierarchy will be primitive modules such as registers, FIFOs, RAMs, and I/O devices. All the modules are connected by *combinational circuits*. In HDLs, *dynamic semantics* concerns the behavior of this statically elaborated circuit.

In this document, we will not discuss the part of BSV static semantics that coincides directly with Haskell (specifically types, polymorphism, type classes, type-checking and monadic elaboration). We will describe BSV's static elaboration, because modules, module instances, and module *sharing* are fundamental concepts.

## 3.2 TRSs, Linear Orders, and Clocks

Classical TRSs are non-deterministic, and have no notion of clocks. The basic dynamic semantics of TRSs is almost trivial:

- Given the current state (a "term"), find a part of the state (a subterm) that matches the left-hand side of one of the rewrite rules. (There may be many such matches; choose one non-deterministically.)

[1] The only exceptions are ancient languages like early FORTRAN and perhaps a few languages for embedded programs.

[2] Of course it is always possible to *represent* a dynamic stack in memory in a hardware system, but that is a level of abstraction above our current level of discourse, namely physical hardware.

- Replace that subterm by the right-hand side of the rule (with suitable variable substitutions), producing the next state. We call this a *rule firing*.
- Repeat and rinse.

The description above is sequential (one rule at a time) and may seem at odds with our earlier claim that this is a favorite way amongst researchers to model complex concurrency. The apparent paradox is resolved by realizing (a) that it is non-sequential in that there is no prescribed "control flow" or "program counter" to sequence rules; it is purely reactive–which rule can fire depends only on the instantaneous state, and (b) *any* actual concurrent execution that preserves the logical sequentiality of the semantics is acceptable (*linearizability*). Another way of saying this is that each rewrite is an *atomic transaction* on the state [21].

BSV goes further and refines TRS semantics to include *clocks*, which are fundamental to most digital HW[3]:

- Every BSV rule fires "within" a clock.[4]
- Many BSV rules may fire within a clock, but there will always be a *logical linear order* that explains their behavior.
- There will be practical constraints (described later) that preclude certain rules from firing within the same clock in a certain order.

### 3.3 Haskell executability

Our specification is given as a Haskell program, and is executable. It can be characterized as a high-level implementation-independent operational semantics.

### 3.4 Kernel language only

As is common in the literature, we will describe BSV's semantics for a *kernel* or subset of the language. Extending it to the full language would be routine but tedious to describe and to understand, without providing any useful additional illumination. Nevertheless, our kernel language is more comprehensive than previous toy descriptions of BSV semantics, and captures the essence of the actual BSV code behavior (specifically: module instances and module sharing). Further, it is *the* mental model that one uses in practice when developing and debugging BSV code.

### 3.5 Rule firing constraints due to module sharing

As mentioned earlier, we will be introducing constraints that prevent certain rules from firing within a clock in a certain order. Many of these constraints have functional motivations. For example, whether a FIFO logically enqueues before it dequeues, or vice versa, can affect its functionality in certain corner cases (when it is full or empty).

[3] Digital circuits can also be *asynchronous* or clock-free. It is a fascinating question whether a BSV-like language can be compiled to asynchronous hardware, but we do not explore that further here.

[4] This is a design choice based on current tractability; a more general future approach may allow rules to use multiple clocks.

Other constraints arise due to *module sharing*. In digital hardware, a wire can be driven with only one, stable value ("0" or "1") during each clock. Consider two independent rules that each occasionally drive a particular input wire of a shared module. Those rules therefore cannot fire in the same clock. Our semantics will capture these constraints that arise due to module sharing.

### 3.6 The semantics does not rely on any built-in primitives

This semantics does not prescribe any particular predefined set of primitives (such as registers). For example, it is perfectly feasible with this semantics to define a version of BSV based entirely on FIFOs or RAMs, or even registers with different semantics. This semantics is, in effect, parameterized by a set of primitives. All it assumes is that each primitive is supplied with the primitive semantics of the methods when invoked (within an Action) and the parallel (intra-rule) and concurrent (inter-rule) ordering constraints on its methods.

Of course in practice our primitives include registers, Concurrent Registers, FIFOs, BRAMs and more. The accompanying Haskell code includes registers and Concurrent Registers (CRegs).

### 3.7 Schedule-construction is an orthogonal question and is not addressed here

We use the term *schedule* to mean a linear sequence of rule instances. In each clock, we will execute rule instances according to such a schedule (execution of a rule instance may be a no-op, because of conflicts or because the rule is not enabled).

It is important to distinguish two quite separate questions:

- How should one to construct a schedule from the rules in a BSV program? Typically, this is the job of a compiler which, may, for example, try to construct a schedule that maximizes concurrency.
- Given a schedule, what is a *correct execution* of the schedule?

In this document, we only address the latter question. A schedule is provided as an argument to the main semantic function, and may as well be provided by an oracle that selects the next rule instance to be considered at the very last moment (when we are ready to execute the next rule instance in the current clock). Thus, the semantics defined here can be seen as *universally quantified over all possible schedules* i.e., describing correct execution for any possible schedule.

In this semantics there is no such thing as a "wrong" schedule. Of course, from a practical point of view, some schedules are likely to be better than others because, for example, they admit more concurrency (more rules can fire in a clock). This semantics takes no position on how to construct such a schedule.

Our description here is therefore more general than the current *bsc* compiler for BSV, which:

- Only considers each rule once during each clock.
- Statically fixes a priority (linearization) of the rules.

In contrast, the semantic description here admits schedules that may vary from clock to clock, and may include duplicate rule execution within a clock.

# 4 Kernel Language Concrete and Abstract Syntax

The kernel language syntax is described below both as abstract syntax (a Haskell data type) and as concrete syntax. The parser reads concrete syntax from a text file and produces a Haskell data structure corresponding to the abstract syntax.

In the accompanying Haskell code, the file `AST.hs` defines the abstract syntax data types. The file `Parser.hs` implements the parser using Haskell's `Parsec` parser combinators.

Metasyntax: we use conventional BNF below for concrete syntax, with the following conventional extensions:

| | |
|---|---|
| { $x$ } | means zero or more repetitions of $x$ |
| $x$ ... $x$ | means zero or more repetitions of $x$ |
| [ $x$ ] | means zero or one repetitions of $x$ (i.e., $x$ is optional) |

`Typewriter` font is used for terminals, and *italics* for non-terminals.

## 4.1 Programs

A kernel BSV program is a list of module definitions followed by a schedule.

**Concrete syntax:**

*Program* ::= *ModuleDefinition* ... *ModuleDefinition Schedule*

**Abstract syntax:**

```
type Program  = ([Binding], Schedule)
```

In the abstract syntax, each module definition is just a binding of the module name to a lambda-expression whose arguments are the module parameters and whose body is a module expression.

There must be exactly one module definition called `main`, and it must have zero parameters. Static elaboration is performed by applying this niladic lambda expression (to an empty argument list).

Static elaboration will recursively instantiate modules; this is sometimes called the module hierarchy. Thus, each module instance has a unique hierarchical name, which is a sequence of identifiers that identify a path down this recursion, starting with `main`.

The hierarchical name of any module instance's component (such as a rule or method instance) is simply the hierarchical name of its enclosing module instance appended with component's name.

```
type Ide      = String      -- Identifiers
type RuleName = String      -- Rule names
type MethName = String      -- Method names
type HierName = [Ide]       -- Hierarchical name
```

The schedule is a list of hierarchical names, each referring to a unique post-elaboration instance of a rule.

```
type Schedule   = [HierName]  -- list of rule instances
```

Having a schedule at the end of the source file is just a convenience that allows us easily to exercise the semantics. As described in Sec. 3.7, this semantics takes no position on schedule construction, just on correct execution of any given schedule. One could write variants of this Haskell code where the schedule is constructed from analysis of the module definitions, or uses an run time oracle, etc. Some of our examples contain multiple schedules at the end of the source file, and we comment-out all but one of them for each execution.

## 4.2 Bindings

A binding associates an identifier with an expression/value.

**Concrete syntax:**

*Binding* ::= `let` *Ide* `=` *Expr*

**Abstract syntax:**

```
type Binding = (Expr, Expr)
```

The lhs uses `Expr` instead of `Ide` to allow for possible future extension where it could be an array selection, struct selection or pattern.

## 4.3 Module Definitions

The word "module" is used in quite different ways in different languages and often concerns mechanisms for separate compilation and namespace management, without any serious role in dynamic semantics. In BSV, as in many HDLs, a module is a fundamental dynamic semantic entity, representing a hardware unit with an interface. Source programs contain module expressions which evaluate to modules. Elaborated programs contain module instances. A single module expression may result in multiple module instances. A module instance may be *shared*, which can affect dynamic semantics.

A module expression is an expression which contains a sequence of bindings, rules, and interface method definitions.

**Concrete syntax:**

*ModuleDefinition* ::= `module` *Ide* [ `#` `(` *Ide* `,` ... `,` *Ide* `)` ] `;`
*Binding* `;` ... `;` *Binding*
`rules`
*Rule* ... *Rule*
`methods`
*MethodDef* ... *MethodDef*
`endmodule`

The optional "`#` `(` *Ide* `,` ... `,` *Ide* `)`" on the first line represents parameters to the module. If missing parameter, it is treated as `#()`, the empty argument list.

In the abstract syntax, such a module definition binding is converted into a `Binding` that binds the *Ide* (name of module definition) to a lambda-expression whose arguments are the parameters and body is a module-expression:

| *Binding:* `(`*Ide*`, (Lambda [`*Ide* `...` *Ide*`]` *module-expression*`))` |
|---|

**Abstract syntax** for module-expressions:

```
data Expr = ...
          | ModuleExpr [Binding] [Rule] [MethDef]    -- module-expression
```

(`Expr` is defined in more detail later.)

## 4.4 Rules

Rules are defined inside modules.

**Concrete syntax:**

*Rule* ::= `rule` *Ide* `(` [ Expr ] `) ;`
*Stmt*
...
*Stmt*
`endrule`

The *Expr* is the rule condition (a boolean expression). "*Stmt* ... *Stmt*" is a list of zero or more statements constituting the rule body. In the abstract syntax, the rule condition and body are combined into a "When" expression (described in more detail later in Sec. 4.6), and the statements are combined into a Block.

| `Rule` *Ide* `(When` *Expr* `(Block [`*Stmt* `...` *Stmt*`]))` |
|---|

**Abstract syntax:**

```
data Rule = Rule Ide Expr

data Expr = ...
          | Block [Stmt]            -- Block
          | When Expr Expr          -- When guard-cond value
```

## 4.5 Method Definitions in Modules

Methods are defined inside modules.

**Concrete syntax:**

*MethodDef* ::= `method` *MethodKind Ide* `(` *Ide* `,` ... `,` *Ide* `)` [ `if` `(` *Expr* `)` ] `;`
*Stmt* ... *Stmt*
`endmethod`

*MethodKind* ::= `V` | `A` | `AV`

The *MethodKind* specifies whether this is a value method (V), Action method (A) or ActionValue method (AV). Value methods are pure (no side-effects). Action methods just have a side effect, and do not return any value. ActionValue methods have a side effect and return a value.

This is followed by an *Ide*, the name of the method, and the arguments of the methods in parentheses. This is optionally followed by the keyword `if` and the method condition, a boolean *Expr*. A missing method condition is treated as `if (True)`. Then we have the method body, which is a sequence of statements.

In the abstract syntax, this is converted into a `MethDef` that contains the method kind, the method name, and a When-expression. whose condition is the method condition and whose body is a lambda-expression. The lambda-expression's arguments are the method arguments, and the body is a Block containing the method's statements:

```
MethDef  {methKind = "A/V/AV",
          methName = Ide,
          methBody = When Expr
                          Lambda [Ide ... Ide]
                              Block [Stmt ... Stmt]
         }
```

**Abstract syntax:**

```
data MethDef = MethDef { methKind :: String    -- "V", "A" or "AV"
                       , methName :: Ide
                       , methBody :: Expr      -- When-expr
                       }
```

## 4.6 Expressions

Finally, we can describe all the different kinds of expressions.

**Concrete syntax:**

| | | |
|---|---|---|
| *Expr* | ::= `(` *Expr* `)` | Parenthesized expression |
| | *Ide* | Identifier |
| | `( )` | Void literal |
| | *IntegerLiteral* | integer literal |

| | |
|---|---|
| *StringLiteral* | string literal |
| `True` \| `False` | boolean literal |
| `if (` *Expr* `)` *Expr* `else` *Expr* | conditional |
| `while (` *Expr* `)` *Expr* | loop |
| *Expr* `.` *Ide* | Method value |
| `begin` *Stmt* ... *Stmt* `end` | block |

**Abstract syntax:**

```
data Expr = EIde String             -- Identifier

          | Void                    -- Constant void
          | ConstI Int              -- Constant integer
          | ConstS String           -- Constant string
          | PrimFn Opcode           -- Constant primitive function value
          | Lambda [Expr] Expr      -- args have form 'EIde x'
          | App [Expr]              -- apply e0 to e1, e2, ...
          | If Expr Expr Expr       -- Conditional
          | While Expr Expr         -- Loop

          | MethodVal Expr MethName -- Method value: Expr is a module instance

          | Block [Stmt]            -- Block

          | When Expr Expr          -- When guard-cond value

          | ModuleExpr [Binding] [Rule] [MethDef]

          -- The following only appears during static elaboration
          | ModuleInstRef Int   -- index into global IntMap of module instances

          -- The following only appear during dynamic execution
          | Unavail             -- due to a method condition being False
          | Closure Env Expr    -- value of a lambda

data Stmt = StmtExpr Expr
          | StmtBinding Binding
```

The first several alternatives are quite conventional (identifiers, constants, lambda-expressions, applications, conditionals, loops). As usual, lambda-expressions will evaluate to closures at run time. A Block-expression contains a list (sequence) of statements, each of which can be either a let-binding or an expression (typically a method-call or function-call). The final value of the Block-expression is the value of the last item: Void if it is a binding, and the expression value if it is an expression.

A When-expression has a guard (a boolean expression) and a body (an expression). At run time, a When-expression evaluates to the special value `Unavail` if the guard expression evaluates to False or is itself `Unavail`. The body is only executed if the guard evaluates to True.

During static elaboration, a `ModuleExpr` creates a new module instance which has a unique integer identifier (more about this later), and is replaced by a `ModuleInstRef` $j$ value that refers to it.

A method value contains an expression and a method name. At run time, the expression evaluates to a particular module instance reference, and the method name selects a method in that module instance. In an application, the function value may be a primitive function value, a standard lambda-closure, or a method value.

# 5 Parsing from concrete syntax to abstract syntax

The Haskell file `Parser.hs` defines a parser from concrete syntax into abstract syntax, using the standard Haskell `Parsec` library. The main function is:

```
type Program    = ([Binding], Schedule)

parseProgramFromString :: String -> Program
```

Its input is a String representing a complete program (typically taken from a source file), and its output is a pair: the list of top-level bindings (module definition bindings), and the schedule (list of hierarchical names identifying rule instances).

This is a vanilla use of Haskell's Parsec library, and is not particulary interesting for a discussion about semantics of the abstract syntax, so we do not discuss it further here.

# 6 Static Elaboration

The Haskell code for static elaboration is in the file `StaticElab.hs`.

The source program is parsed into a list of bindings and a schedule. Each binding associates an identifier to a lambda-expression whose body is a module-expression, as described in Sec. 4.3. The schedule is not relevant for static elaboration; it will only play a role in dynamic semantics, discussed in Sec. 7.

One of the bindings in a source program must bind the distinguished name `main` to module with no parameters (more specifically, to a niladic lambda-expression whose body is a module-expression). The *static elaboration* step applies this lambda expression (to zero arguments) to produce a statically elaborated program, which is (abstractly) a description of a concrete hardware module hierarchy with behavior expressed as rules.

The process of static elaboration is recursive: applying `main` will result in a module instance; the bindings in this module, when evaluated, will create other module instances. The bindings inside those module instances may, in turn, bind further module instances, and so on, recursively. A particular source-program module definition may be instantiated multiple times. Ultimately we have a hierarchy (i.e., a *tree*) of module instances, with an instance of `main` at the root.

### 6.1 Module instances

A module instance is either an instance of a primitive module (built-in) or is an instance of a user-defined module in the source program (the value of a module-expression):

```
data ModuleInst = PMIReg      Int                -- reg data
                | PMICReg Int Int                -- # of ports, reg data
                ...
                 ... other primitives
                ...
                | UMI     Env [Rule] [MethDef]  -- User-defined module instance
```

(Mnemonics: PMI = Primitive Module Instance, UMI = User Module Instance).

Here we show "Registers" and "Concurrent Registers" as primitives, but these can be replaced or extended with an arbitrary set of primitives such as FIFOs, RAMs, and so on. Each primitive has its own representation of internal state. For example, the `Int` value for Registers represents the current register data contents. For Concurrent Registers, the first `Int` represents the number of its concurrent ports, and the second `Int` represents its current data contents.

Static elaboration is a classical lambda-calculus "Eval" of the source program. This is supplied with an initial environment that binds the names of constructors for primitive modules to primitive functions:

```
        primitivesEnv :: Env
        primitivesEnv = [ (EIde "mkReg",  PrimFn PrimMkReg)
                        , (EIde "mkCReg", PrimFn PrimMkCReg)
                        ]
```

When applied to arguments (during static elaboration), these primitive functions create (instantiate) `PMI` module instances.

When module-expressions are evaluated, they create `UMI` module instances. User-defined module instances are closures—the `Env` component represents the bindings in the module, which may be used by in its rules and method definitions.

For the dynamic semantics the module hierarchy is not important: the only important thing is that a rule in one module instance may invoke a method in another module instance, and thus must have a way to refer to another module instance. Thus, during static elaboration, we give each module instance a *unique integer name*, and we collect all module instances in a *mapping* (a Haskell `IntMap` object) that maps each unique integer name to a pair— the module instance's hierarchical name, and its instance value. We call this mapping a `System`:

```
type System = Intmap (HierName, ModuleInst)
```

Each time we evaluate a "`ModuleExpr [Binding] [Rule] [MethDef]`" expression, we create a new `ModuleInst` in the global `system`, and replace the expression with a `ModuleInstRef Int` value, which is a reference to the module instance.

```
data Expr = ...
          | ModuleInstRef Int    -- index into global IntMap of module instances
```

### 6.2 The Static Elaboration function

Static elaboration is carried out by this function:

```
staticElab :: [Binding] -> (System, Expr)
staticElab bindings = eval  []  ["main"]  DI.empty  (App [EIde "main"])
    where
        ...
```

The argument is the list of bindings from the source file produced by the parser. It merely calls the `eval` function, to be described next, giving it:

- an initially empty environment: `[]`,
- an initial hierarchical name: `["main"]`
- an initially empty map of module instances: `DI.empty`,
- and an initial application: `[main]` (`main` applied to no arguments).

Static elaboration is performed by a classical mutually recursive Eval-Apply pair performing evaluation and application (locally inside the `staticElab` function):

```
  eval :: Env -> HierName -> System -> Expr -> (System, Expr)
  app :: HierName -> System -> Expr -> [Expr] -> (System, Expr)
```

In both `eval` and `app`:

- The `HierName` argument is the current position in the module instance hierarchy, i.e, the list of module instance names above this recursive point.
- The `System` argument is the collection of module instances accumulated so far.

and the result is a pair: the updated `System`, and an `Expr` representing an evaluated value. In `eval`:

- The `Env` argument is the current environment binding identifiers to values.
- The `Expr` argument is the expression being evaluated.

Most of the clauses in `eval` are classical (identifiers, constants, primitive functions, lambdas, applications). The only unusual clause is for module definitions:

```
eval env hn system (ModuleExpr bs rs mds) =
                let
                    -- Convert the bindings into an environment,
                    -- sequentially evaluating the right-hand sides
                    f (system, env) (lhs @ (EIde s), rhs) = (system', env')
```

```
                    where
                        (system', v) = eval env (hn ++ [s]) system rhs
                        env' = env ++ [(lhs, v)]
                (system', env') = foldl f (system, []) bs

                -- Create a user-defined module instance
                new_umi = UMI (env ++ env') rs mds
            in
                extendSystem system' hn new_umi
```

A module-expression contains bindings `bs` of identifiers to expressions (typically module instantiations), rules `rs` and method definitions `mds`. Here, we leave `rs` and `mds` untouched. We convert `bs` into an environment `env'` by evaluating each right-hand side expression to a value. We do this sequentially, using `foldl`, so that values from earlier bindings are in scope for later bindings. We create a new user-defined module instance (`new_umi`). Finally, `extendSystem` adds this module instance to the system, allocating a new unique integer identifier $j$, returning the new extended system and `ModuleInstRef` $j$ as the value of the expression.

In `app`:

- The `Expr` argument is the value of a function to be applied (either a primitive, or the closure of a lambda expression if user-defined).
- The `[Expr]` argument is a list of argument values for the function.

Most of the clauses in `app` are classical (primitive functions and lambda-closures). The only unusual clauses are for primitive module constructors. In each case, we simply extend the incoming `system` with new primitive module instances:

```
app hn system (PrimFn PrimMkReg) [ConstI j]
    = extendSystem system hn (PMIReg j)

app hn system (PrimFn PrimMkCReg) [ConstI n, ConstI j]
    = extendSystem system hn (PMICReg n j)
```

# 7 Dynamic Semantics

Having parsed and elaborated a program, we are ready to turn to the central focus of this document– the dynamic semantics of rules.

## 7.1 The central intuitions behind clocks and schedules

Hardware execution is a sequence of clocks. In each clock we execute a sequence of rules[5]. We call a sequence of rules a *schedule*.

[5]We remind that the sequence of rules is only a logical concept to explain semantics and understand behavior. It may not be discernible at all in the clocked digital hardware that is the output of the *bsc* compiler.

As explained in Sec. 3.7, the semantics takes a schedule as a parameter, and takes no position on how that schedule is constructed, it just explains correct execution of the given schedule.

Thus, the central question answered by the semantics is:

> In the current clock,
> given that we have already performed rule executions $r_0$, $r_1$, ..., $r_{j-1}$ so far in this clock,
> and the schedule has determined that we must next attempt rule $r_j$,
>
> What is a correct execution of $r_j$?

This step is repeated (within this clock) until the schedule offers no more rules for this clock.

And all this is repeated forever for multiple clocks.

Correct execution of a rule is straightforward, and is summarized informally in Fig. 1. Internalizing this will make it easy to understand the formal description that follows. This is also (roughly) the mental model used every day by the practicing BSV programmer to understand BSV program execution and the schedules produced by the *bsc* compiler.

## 7.2 Formal description

The code for the dynamic semantics is in the file `Semantics.hs`. We skip over the first few definitions, `exec_n_clocks`, `findRule` and `initEnv` because they are about specifying schedules (here, we arbitrarily read a schedule at the end of each source file) and repeated execution of a schedule across clocks, etc., which, as explained earlier, are not the central focus of this document.

Instead, we pick it up at the following function:

```
exec_1_clock :: System -> [(HierName,Env,Rule)] -> IO (Bool, System)
exec_1_clock system ruleSeq = do (anyfired, system', _)
                                     <- foldM f  (False, system, [])  ruleSeq
                                 return (anyfired, system')
    where
        -- mcsPrev is the collection of methods-called in rules
        -- previously fired in this clock
        f (anyfired, system, mcsPrev) rule =
            do (fired, system', mcsPrev') <- execRule system mcsPrev rule
               return (anyfired || fired, system', mcsPrev')
```

The `system` argument is the `IntMap` from integers to module instances.

The `ruleSeq` argument is a list of rule instances representing the schedule. Each is a triple of a hierarchical name that uniquely identifies the rule, an environment representing the bindings in the module instance where this rule is found, and a Rule-expression.[6].

[6] There is no assumption that the schedule is constant across clocks. In fact, an alternative formulation may replace the `ruleSeq` argument by an oracle function on the current system state and list of rules fired so far in a clock, i.e., it can take a last-minute instantaneous decision on which rule should be considered next.

**Correct execution of a rule in a clock, given that we have previously executed certain rules in this clock.**
**(Informal Description)**

- We first "evaluate" the rule (which is a When-expression). This is mostly a conventional lambda-calculus Eval-Apply pair, with a few variations:
  - In any When-expression (typically rule and method bodies), if the guard condition evaluates to `False`, the When-expression evaluates to the special value `Unavail`. In fact, the guard condition may itself evaluate to `Unavail`, in which case also the When-expression's value is `Unavail`.
  - We collect a record of every method called, whether on a primitive module instance or on a user-defined module instance.
  - We collect *but do not perform* applications of primitive Action method values to their argument values. Note, therefore, that, so far, *rule evaluation has no side effects on the system state.*
- After rule evaluation, we decide whether the rule will fire or not. Any of the following conditions will stop a rule from firing.
  - There is at least one *intra-rule conflict*, i.e., the rule invokes two methods $m_1$ and $m_2$ that are not allowed to be invoked within the same Action (simultaneously/instantaneously/parallel). These prohibitions are axiomatic properties of methods on primitive modules, and are given in a fixed intra-rule conflict table.
  - There is at least one *inter-rule conflict*, i.e., a rule executed earlier in this clock invokes a method $m_1$ and this rule invokes a method $m_2$, and they are not allowed to be executed in that order. These prohibitions are axiomatic properties of methods on primitive modules, and are given in a fixed inter-rule conflict table.
  - The rule evaluates to `Unavail`. Either the rule condition expression, or the condition of some method (either in the rule condition expression or its body), evaluates to False.
- Finally, if we decide that the rule will fire, we execute the primitive Actions that we collected during rule evaluation. This transforms the system state. Conceptually, all these actions are performed simultaneously and instantaneously, i.e., in parallel.

Figure 1: The central idea in rule semantics: correct rule execution

The result is a pair. The `Bool` component indicates whether any rule fired in this clock. The `System` component is the updated system state (which will be the same as the `system` argument if the rule did not fire).

The `exec_1_clock` function is straightforward, just using `foldM` to apply the `execRule` function over the rules in the schedule, carrying along three values: a `Bool` indicating whether any rule fired, the current system state, and `mcsPrev`, the collection of methods called in rules that have fired so far in this clock.

A "method-called" is simply a method value, which is an expression:

```
type MC = Expr    -- MethodVal (ModuleInstRef j) MethName
```

### 7.2.1 Evaluating a Rule Expression

The central function of this semantics is this one, which is a direct implementation of the informal explanation of Fig. 1.

```
execRule :: System -> [MC] -> (HierName, Env, Rule) -> IO (Bool, System, [MC])
execRule  system  mcsPrev  (hname, env, Rule name e) =
  do -- Eval this rule
     (v,actions,mcsThis) <- evalExpr system env e

     let intra_cfs = intraRuleConflicts  system  mcsThis
     let inter_cfs = interRuleConflicts  system  mcsPrev  mcsThis
     let hw_cfs    = hwConflicts         system  mcsPrev  mcsThis

     -- has intra-rule method conflicts
     if not (intra_cfs == []) then return (False, system, mcsPrev)

     -- has inter-rule method conflicts
     else if not (inter_cfs == []) then return (False, system, mcsPrev)

     -- has hardware method conflicts
     else if not (hw_cfs == []) then return (False, system, mcsPrev)

     -- Rule is not enabled (rule cond, or some method cond, is false)
     else if (v == Unavail) then return (False, system, mcsPrev ++ mcsThis)

     -- Rule is enabled (rule-cond is True and all method-conds are True)
     -- Perform the rule's actions, updating the system state
     else do system' <- doActions system actions
             return (True, system', mcsPrev ++ mcsThis)
```

The `mcsPrev` argument is a collection of methods called in rules that have previously fired in this clock. The third argument is a rule instance (its hierarchical name, an environment representing

the bindings in its surrounding module, and the Rule-expression itself). The output is a boolean indicating whether the rule fired, the updated system state (same as input system state if the rule did not fire), and an updated collection of methods called (`mcsPrev` augmented by methods called in this rule, if any).

The `evalExpr` function evaluates an expression to a value, which is just an expression restricted to certain forms:

```
type Val    = Expr    -- Unavail
                      -- Void, ConstI j, ConstS s,
                      -- PrimFn op,
                      -- Closure env (Lambda ...),
                      -- MethodVal (MmoduleInstRef j) MethName
```

The `evalExpr` function is a classical lambda-calculus Eval-Apply pair, with a few variations.

```
evalExpr :: System -> Env -> Expr -> IO (Val, [Action], [MC])
evalExpr system env e = eval e
    where
        eval :: Expr -> IO (Val, [Action], [MC])

        ... clauses for classical eval of identifiers, constants, ...
        ...   primitive functions, lambdas ...
        ...   conditionals, loops, blocks ...
        ...
```

We omit here the classical clauses (identifiers, constants, conditionals, lambdas, ...). If the expression is `(MethodVal e methName)`, then `e` evaluates to a module-instance reference of the form `ModuleInstRef` $j$ where $j$ is an index into the `system` IntMap and selects a module instance.

```
        eval (MethodVal e methName) =
            do (miref, [], mcs) <- evalExpr system env e
               return (MethodVal miref methName, [], mcs)
```

Since this evaluation cannot have side-effects, the pattern-match asserts `[]` for the actions collected.

Evaluating a When-expression can return `True`, `False` or `Unavail`. The boolean values are encoded as `ConstI 1` and `ConstI 0`.

```
        eval (When econd ebody) =
            do (vcond, [], mcs) <- evalExpr system env econd
               if ((vcond == Unavail) || (vcond == (ConstI 0)))
               then return (Unavail, [], mcs)
               else do (vbody, as', mcs') <- evalExpr system env ebody
                       if (vbody == Unavail) then return (Unavail, [], mcs)
                       else return (vbody, as', mcs++mcs')
```

Evaluating an application is classical: evaluate the list of expressions representing the function and the arguments, and then apply the function value to the argument values except, if any of those values are `Unavail`, just return `Unavail`.

```
        eval (App es)   =
            do (vs, as, mcs) <- evalList es
               if any (== Unavail) vs then
                   return (Unavail, as, mcs)
               else do (v, as', mcs') <- apply system vs
                       return (v, as++as', mcs++mcs')
        ...
```

The `apply` function is classical for primitive functions and closures. For applying method values, we use the `applyMethod` function.

```
apply :: System -> [Expr] -> IO (Val, [Action], [MC])
apply system  (PrimFn opcode : vargs) =
    do (y, as) <- applyPrimFn (PrimFn opcode) vargs
       return (y, as, [])

apply system  (Closure env (Lambda xs e) : vargs) =
    evalExpr system (env ++ zip xs vargs) e

apply system  (mv @ (MethodVal miref methname): vargs) =
    applyMethod system mv vargs
```

The `applyPrimFn` function is just a direct implementation of each primitive function's semantics. We show the clause for `Plus` here:

```
applyPrimFn :: Val -> [Val] -> IO (Val, [Action])
...
applyPrimFn (PrimFn Plus)   [ConstI x, ConstI y] = return (ConstI (x+y), [])
...
```

The `PrimDisplay` function is an Action, so we just collect it for later execution after deciding that the rule will fire:

```
...
applyPrimFn (PrimFn PrimDisplay) [ConstI x] =
    return (Void, [App [PrimFn PrimDisplay, ConstI x]])
applyPrimFn (PrimFn PrimDisplay) [ConstS s] =
    return (Void, [App [PrimFn PrimDisplay, ConstS s]])
...
```

The `applyMethod` function records the method called. Then, for methods of primitive modules (like registers), it just implements the primitive's semantics for value methods, or collects the application for Action/ActionValue methods.

```
applyMethod :: System -> Val -> [Val] -> IO (Val, [Action], [MC])
applyMethod system  mc@ (MethodVal  miref @ (ModuleInstRef j)  methname)  vargs =
    do let (hn, mi) = system IntMap.! j

       -- Record this method-called, whether it's a PMI or UMI
       let mcs = [mc]
       -- thisActions will only be used if this is not a Value method
       let thisActions = [App (mc:vargs)]

       case (mi, methname, vargs) of
           -- Ordinary registers
           (PMIReg v, "_read", []) -> return (ConstI v, [], mcs)
           (PMIReg v, "_write", [v']) -> return (Void, thisActions, mcs)
           ...
```

Application of user-defined methods is similar to application of closures. A user-defined method is in fact a closure—it is a lambda-expression whose environment is the bindings in its enclosing module instance. We evaluate the lambda-expression to a closure value, and apply it to the arguments.

```
           ...
           -- User-defined modules
           (UMI env rules methDefs, _, _) ->
               do -- Select the method by name, getting its body and kind
                  let [(mbody, mkind)] =
                    = [(mbody', mkind')| md <- methDefs
                                       , let MethDef mkind' methname' mbody' = md
                                       , methname == methname' ]
                  -- mbody is a lambda expr, evaluating to closure v
                  (v, actions, mcs') <- evalExpr system env mbody
                  -- let actions' = thisActions ++ actions
                  let actions' = if (mkind == "V") then []
                                 else (thisActions ++ actions)
                  if (v == Unavail) then return (Unavail, actions', mcs++mcs')
                  else do (v', actions'', mcs'') <- apply system (v:vargs)
                          return (v', actions'++actions'', mcs++mcs'++mcs'')
```

### 7.2.2 Checking whether the rule has intra-rule conflicts

After evaluating the rule (which has no side-effects, since we only collect its Actions, we don't perform them yet), we check for conflicts. Checking whether the collection of methods-called within a rule has any intra-rule conflicts is done with this function:

```
intraRuleConflicts :: System -> [MC] -> [(MC,MC)]
intraRuleConflicts system mcs = ...
```

It checks each method-called in `mcs` with every other method-called in `mcs`, and returns a list of all pairs that conflict. This check is just a test for presence in the fixed `intraRuleConflictTable`. An example entry in this table is shown below.

```
intraRuleConflictTable :: [ (ModuleType, MethName, MethName) ]
intraRuleConflictTable =
    [ -- Reg ----------------
      -- _write cannot be called more than once in a rule
      (Reg, "_write", "_write")
      ...
    ]
```

### 7.2.3 Checking whether the rule has inter-rule conflicts

Checking whether the collection of methods-called within a rule has any inter-rule conflicts with any method called in rules previously fired in this clock is done with this function:

```
interRuleConflicts :: System -> [MC] -> [MC] -> [(MC,MC)]
interRuleConflicts  system  mcsPrev  mcsThis = ...
```

It checks each method-called in `mcsPrev` with each method-called in `mcsThis`, and returns a list of all pairs that conflict. This check is just a test for presence in the fixed `interRuleConflictTable`. An example entry in this table is shown below.

```
interRuleConflictTable :: [ (ModuleType, MethName, MethName) ]
interRuleConflictTable =
    [ -- Reg ----------------
      -- _write cannot precede _read
      (Reg, "_write", "_read")
      ...
    ]
```

### 7.2.4 Checking whether the rule has hardware conflicts

Checking whether there are any hardware conflicts is done with this function:

```
hwConflicts :: System -> [MC] -> [MC] -> [(MC,MC)]
hwConflicts  system  mcsPrev  mcsThis = ...
```

It checks each method-called in `mcsThis` with each other method-called in `mcsThis`, and it checks each method-called in `mcsThis` with each method-called in `mcsPrev`. Note that hardware-conflicts between methods-called in `mcsPrev` would have already been checked while executing earlier rules. It returns a list of all pairs that conflict.

A hardware conflict captures the fact that a hardware wire can only be driven by one value in each clock. Thus an Action or ActionValue method always has a hardware conflict with itself (the "enable" input wire can only be driven once). A Value method that has any arguments always has a hardware conflict with itself (the argument input wires can only be driven once). To put it another way, only a niladic Value method does not have a hardware conflict with itself. This is captured in the following function.

```
hwConflict :: System -> MC -> MC -> Bool
hwConflict system  (MethodVal (ModuleInstRef j)  methname)
                   (MethodVal (ModuleInstRef j2) methname2)
                       | j /= j2               = False    -- different modules
                       | methname /= methname2 = False    -- different methods
                       | otherwise             = b
  where
      (hname, mi) = (system IntMap.! j)
      b = case mi of
              (PMIReg _)    -> (methname == "_write") -- Action method

               ... other primitives ...

              (UMI _ _ _)   ->    -- Method on user-defined module
                  let
                      md               = select_UMI_method  methname  mi
                      Lambda args body = methBody md
                      niladic_Value_method = (   (methKind md == "V")
                                              && (length args == 0))
                  in
                      not niladic_Value_method
```

### 7.2.5 Firing the rule (performing its actions)

Finally, after evaluating a rule, checking that it has no inter-rule, intra-rule or hardware conflicts, and checking that its rule and method conditions are True (not `Unavail`), we can fire the rule, i.e., perform its actions to update the system state.

The function `doActions` just uses `foldM` to iterate over all the actions, repeatedly updating the system state:

```
doActions :: System -> [Expr] -> IO System
doActions  system  actions = foldM  f  system  actions
    where f ...
```

Each action is executed by `doAction`. Methods of primitive modules just implement the primitive's semantics, updating the module instance state to a new module instance state (like the register `_write` shown below). Methods of user-defined modules are ignored here: they were originally collected for checking hardware conflicts, but they were also applied to their arguments to produce (recursively), any underlying primitive-module actions. Ultimately, only primitive-module actions affect system state.

```
doAction :: System -> Int -> MethName -> [Val] -> System
doAction  system  j  methName  args = system'
    where
        -- Get the primitive module instance from the system
        (hname, mi) = system IntMap.! j

        -- Upate the primitive module instance according to the method
        mi' = case (mi, methName, args) of
                  (PMIReg v,    "_write",  [ConstI v']) -> PMIReg v'
                   ... other primitive-module actions ...
                   ... user-module actions are ignored ...

        -- Put the updated primitive module instance back in the system
        system' = IntMap.insert j (hname, mi') system
```

And we're done with this rule (`execRule`). Function `exec_1_clock` will call `execRule` again for the next rule in the schedule, until there are no more rules in the schedule. Then, `exec_n_clocks` will call `exec_1_clock` again for the next clock, and so on.

# 8 Examples

In this section we describe some examples that are in the `Progs/` directory. Each example is in a source file with extension `.kbsv` (for kernel BSV).

## 8.1 GCD

It seems obligatory with hardware-design languages to start with a GCD example (Greatest Common Divisor). The code is in the file `GCD.kbsv`. Here is the source code for the GCD module.

```
1  module mkGCD;
2      let  zero = 0;
3      let x = mkReg (zero);
4      let y = mkReg (zero);
5      let busy = mkReg (zero);
6
7    rules
8      rule swap (   x._read()     >  y._read()
```

```
9                && busy._read () == 1
10               && y._read ()    != 0);
11           x._write (y._read ());
12           y._write (x._read ())
13       endrule
14
15       rule subtract (   x._read ()    <= y._read ()
16                      && busy._read () == 1
17                      && y._read ()    != 0);
18           y._write (y._read () - x._read ())
19       endrule
20
21     methods
22       method A start (num1, num2) if (busy._read () == 0);
23           x._write (num1);
24           y._write (num2);
25           busy._write (1)
26       endmethod
27
28       method AV getResult () if ((busy._read () == 1) && (y._read () == 0));
29           busy._write (0);
30           x._read ()
31       endmethod
32   endmodule
```

L.2 (line 2) is just a trivial binding giving the name `zero` to the constant integer 0, just to demonstrate that you can have conventional let-bindings of names to expressions. The next three lines instantiate three registers with initial (reset) value 0, and bind `x`, `y` and `busy` to those instances.

L.8-L.13 define the `swap` rule. The rule condition tests if $x > y$, if we are busy and if $y \neq 0$. Note, we're using the `_read()` method on a register to retrieve its value (in BSV these method calls can be omitted and *bsc* automatically fills them in). The body of the rule, L.11-L.12 effectively swap the $x$ and $y$ values. Here we use the `_write()` method to store a value into a register (in BSV we typically use the more convenient infix `<=` syntax).

L.15-L.19 define the `subtract` rule. Here the rule condition tests if $x \leq y$ (and busy and $y \neq 0$). The rule body assigns $y - x$ to $y$.

L.22-L26 define the `start` method. The method condition ensures it can only be invoked when we are not busy. The method body stores the arguments into $x$ and $y$, and store True (coded as integer 1) into `busy`.

L.28-L.32 define the `getResult` method. The method condition ensures it can only be invoked when we are busy and $y = 0$. It returns the value of $x$ and writes False (0) into `busy`, so that the `start` method can now be invoked again.

Here is a "`main`" module to test our GCD module.

```
1    module main;
2        let state = mkReg (0);
```

```
    let gcd = mkGCD ();

  rules
    rule init (state._read () == 0);
        gcd.start (24, 16);
        state._write (1)
    endrule

    rule finish (state._read () == 1);
        let z = gcd.getResult ();
        $display ("The GCD is ");
        $display (z);
        state._write (2)
    endrule
  methods
endmodule
```

It instantiates a `state` register initialized to 0, and it instantiates the GCD module. Rule `init` can fire when `state` is 0; it invokes `gcd.start(24,16)` and writes 1 to `state`. Rule `finish` can fire when `state` is 1; it retrieves a result from `gcd`, displays it, and writes 2 to `state`. When `state` is 2 no rules will enabled, and this will terminate the simulation.

The final piece of source text is a schedule:

```
schedule
  [main, init]
  [main, finish]
  [main, gcd, swap]
  [main, gcd, subtract]
```

Each line is a hierarchical name. For example, the last line represents the `subtract` rule inside the `gcd` instance inside the `main` instance. In this example, the choice of schedule is not important, since the rule conditions ensure that only one of them can fire in a clock anyway.

### 8.1.1 Executing the semantics

We execute the semantics as follows:

```
$ runghc  Main  100  Progs/GCD.kbsv
```

(This command is also available in the `Makefile`.) "`runghc`" is the standard Glasgow Haskell Compiler command to execute a Haskell program. "`Main`" refers to the file `Main.hs` which is the top-level of the Haskell code provided, which invokes all the parsing, static elaboration and semantic functions discussed in this document. "`100`" limits the execution to a maximum of 100 (simulated) clock cycles. Finally, we provide the source file name.

The variable `debugLevel` in the file `Utils.hs` controls the verbosity of the output. When set to 0, it should only produce the output of `$display` functions. We set it to 1 and captured the output in the file `log_GCD`. It consists of three sections: a rendering of the abstract syntax tree (AST) produced by parsing, a rendering of the elaborated program produced by static elaboration and, finally, a clock-by-clock trace of the execution. We skip the AST here, since it is not very interesting.

The first part of the elaborated program looks like this:

```
---- Elaborated program
  0 ["main","state"]    PMIReg 0
  1 ["main","gcd","x"]    PMIReg 0
  2 ["main","gcd","y"]    PMIReg 0
  3 ["main","gcd","busy"]    PMIReg 0
  4 ["main","gcd"]
    UMI
        zero = 0
        x = ModuleInstRef 1
        y = ModuleInstRef 2
        busy = ModuleInstRef 3
      RULES
        rule swap when (And(And(Gt(x._read(),y._read()),Eq(busy._read(),1)),
                        Neq(y._read(),0)));
          begin
            x._write(y._read());
            y._write(x._read());
          end
        rule subtract when (And(And(Leq(x._read(),y._read()),Eq(busy._read(),1)),
                                Neq(y._read(),0)));
          begin
            y._write(Minus(y._read(),x._read()));
          end
      METHODS
        method A start when (Eq(busy._read(),0));
          (lambda (num1,num2)
            begin
              x._write(num1);
              y._write(num2);
              busy._write(1);
            end)
        endmethod
        method AV getResult when (And(Eq(busy._read(),1),Eq(y._read(),0)));
          (lambda ()
            begin
              busy._write(0);
              x._read();
            end)
        endmethod
```

```
    EUMI
```

This is the Haskell `IntMap` that maps integers (unique names) to module instances. For example, "3" refers to the register primitive module instance (PMI) with hierarchical name `main.gcd.busy`, i.e., the `busy` register inside the `gcd` module instance which is inside the `main` module instance. The initial (reset-value) data in that register is 0.

"4" refers to the GCD user-defined module instance (UMI). In the module, you can see that `busy` is bound to the value `ModuleInstRef 3`, i.e., the register instance discussed in the previous para. If we had two instances of the GCD module, each would have its own entry in the IntMap. Inside, the value bound to the name `busy` would different in the two GCD instances, each referring to its own register PMI, its own copy of `busy`. The rules and method definitions in the UMI are the same as in the original AST (this is true in all UMIs—only the bindings are evaluated, the rules and method definitions are left untouched).

The rest of the elaborated program is the UMI for `main`, and the finally we have the value of the expression "`main()`" which is the root of static elaboration.

```
  5 ["main"]
    UMI
        state = ModuleInstRef 0
        gcd = ModuleInstRef 4
      RULES
        rule init when (Eq(state._read(),0));
          begin
            gcd.start(24,16);
            state._write(1);
          end
        rule finish when (Eq(state._read(),1));
          begin
            let z = gcd.getResult()
            $display("The GCD is ");
            $display(z);
            state._write(2);
          end
      METHODS
    EUMI
value of main(): ModuleInstRef 5
```

After static elaboration, we see a clock-by-clock trace of execution. In each clock, it says what happened with each rule in the schedule. For example, clock 0:

```
.... Clock 0
  Rule ["main","init"] fired
  Rule ["main","finish"]
    inter-rule conflicts:
      ["main","state"] _write > _read
```

```
  Rule ["main","gcd","swap"]
    inter-rule conflicts:
      ["main","gcd","x"] _write > _read
      ["main","gcd","x"] _write > _read
      ["main","gcd","y"] _write > _read
      ["main","gcd","y"] _write > _read
      ["main","gcd","y"] _write > _read
      ["main","gcd","busy"] _write > _read
  Rule ["main","gcd","subtract"]
    inter-rule conflicts:
      ["main","gcd","x"] _write > _read
      ["main","gcd","y"] _write > _read
      ["main","gcd","y"] _write > _read
      ["main","gcd","busy"] _write > _read
```

This shows that the first rule in the schedule, `main.init`, fired. The second rule in the schedule, `main.finish` did not fire: it had an inter-rule conflict, where it was trying to execute the method `main.state._read`, whereas rule `main.init` has already fired and executed `main.state._write`, and `_read` cannot follow `_write` in a clock. And so on, for the remaining rules in the schedule, `main.gcd.swap` and `main.gcd.subtract`.

When `main.init` fires, it invokes the `main.gcd.start(24,16)` method which stores 24 and 16 into registers `main.gcd.x` and `main.gcd.y`, respectively. Continuing through clock 5, we see:

```
.... Clock 1
  ...
  Rule ["main","gcd","swap"] fired              So: x = 16, y = 24
.... Clock 2
  ...
  Rule ["main","gcd","subtract"] fired          So: x = 16, y = 8
.... Clock 3
  ...
  Rule ["main","gcd","swap"] fired              So: x = 8, y = 16
  ...
.... Clock 4
  ...
  Rule ["main","gcd","subtract"] fired          So: x = 8, y = 8
.... Clock 5
  ...
  Rule ["main","gcd","subtract"] fired          So: x = 8, y = 0
```

At this point, rule `main.finish` is enabled: its rule condition is true since `main.state`=1, and the condition of the method it calls, `main.gcd.getResult`, is true because `main.gcd.busy`=1 and `main.gcd.y`=0. Indeed, if we continue to clock 6 we see:

```
.... Clock 6
  ...
```

```
  Rule ["main","finish"] fired
$DISPLAY: The GCD is
$DISPLAY: 8
  ...
```

and we see the outputs from the rule's `$display` statements.

In clock 7, we see that no rules fire (they are all `Unavail`), so the simulation prints the final system state and stops. You can see that the `main.state` register has value 2. the `main.gcd.x` register has value 8, the `main.gcd.y` register has value 0, and the `main.gcd.busy` register has value 0.

```
.... Clock 7
  Rule ["main","init"] unavail
  Rule ["main","finish"] unavail
  Rule ["main","gcd","swap"] unavail
  Rule ["main","gcd","subtract"] unavail
No rule fired on cycle 7
---- final system:
  0 ["main","state"]    PMIReg 2
  1 ["main","gcd","x"]    PMIReg 8
  2 ["main","gcd","y"]    PMIReg 0
  3 ["main","gcd","busy"]    PMIReg 0
  4 ["main","gcd"]UMI ...
  5 ["main"]UMI ...
```

## 8.2 PipelineFIFO

The GCD example is nice, but it does not have any interesting concurrency since only one rule condition is enabled at any time (this is why the particular choice of its schedule does not matter either). We now look at a more interesting example, a 2-stage pipeline between which is a "PipelineFIFO". The code is in the file `Progs/PipelineFIFO.kbsv`.

The first module in the file is `mkPipelineFIFO`:

```
module mkPipelineFIFO;
    let full = mkCReg (2, 0);
    let data = mkCReg (2, 0);

  rules
    -- no rules

  methods
    method A enq (x) if (full._read1 () == 0);
      data._write1 (x);
      full._write1 (1)
    endmethod
```

```
13
14      method V notEmpty ();
15        full._read0 ()
16      endmethod
17
18      method V first () if (full._read0 () == 1);
19        data._read0 ()
20      endmethod
21
22      method A deq () if (full._read0 () == 1);
23        full._write0 (0)
24      endmethod
25  endmodule
```

This implements a one-element FIFO. The `full` register is 1 if the FIFO contains a datum, and 0 if it does not. The `data` register holds the datum itself. The module contains no rules, just method definitions. The `enq` method may be called when `full`=0, i.e., the FIFO is empty, in which case it stores the datum `x` in the `data` register and stores 1 in `full`. The `notEmpty` method just returns the `full` value. The `first` and `deq` methods may be invoked when `full`=1. The `first` method returns the datum in the FIFO, and the `deq` method writes 0 `full`, i.e., it marks the FIFO as empty.

If `full` were an ordinary register, we could never invoke `enq` and `deq` in the same cycle. Both methods execute `full._read()` and write `full._write()`. Whether we scheduled (a rule that invoked) `enq` before (a rule that invoked) `deq` or vice versa, we would run into an inter-rule conflict because `_read()` must precede `_write` in a clock.

Thus, we use *Concurrent Registers* (CRegs) for `full` and `data`. The methods on the output side of the FIFO, `notEmpty`, `first` and `deq`, use `_read0` and `_write0`. The method on the input side of the FIFO, `enq`, uses `_read1` and `_write1`. The `interRuleConflictTable` permits `_write0` and `_read0` to precede `_write1` and `_read1` in a clock. Thus, provided the output side methods are scheduled before the input side method, they can execute in the same clock. Conceptually, we can dequeue an element from the FIFO, thus emptying it, and enqueue a new element into the FIFO, thus re-filling it, *all within the same clock.*

The main test program is shown here:

```
1   module main;
2       let x = mkReg (0);
3       let f = mkPipelineFIFO ();
4
5     rules
6       rule feed;
7         f.enq (x._read());
8         x._write (x._read() + 1)
9       endrule
10
11      rule drain (f.notEmpty ());
```

```
      $display ("RESULT");
      $display (f.first ());
      f.deq ()
    endrule
  methods
endmodule
```

The `feed` rule repeatedly enqueues `x` into the FIFO and increments `x`. The `drain` rule repeatedly dequeues an item from the FIFO and displays it. At the bottom of the file, we show two candidate schedules (one of them is commented out):

```
-- Sched 1 (pipelining happens)
schedule
  [ main, drain ]
  [ main, feed ]

/*
-- Sched 2 (pipelining does not happen, rules alternate)
schedule
  [ main, feed ]
  [ main, drain ]
*/
```

### 8.2.1 Executing the semantics with schedule 1

We can execute the program; the transcript using Sched 1 is captured in `log_PipelineFIFO`. Looking at the first 3 clocks, we see:

```
.... Clock 0
  Rule ["main","drain"] unavail
  Rule ["main","feed"] fired
.... Clock 1
  Rule ["main","drain"] fired
$DISPLAY: RESULT
$DISPLAY: 0
  Rule ["main","feed"] fired
.... Clock 2
  Rule ["main","drain"] fired
$DISPLAY: RESULT
$DISPLAY: 1
  Rule ["main","feed"] fired
.... Clock 3
  Rule ["main","drain"] fired
$DISPLAY: RESULT
$DISPLAY: 2
  Rule ["main","feed"] fired
```

In clock 0, rule `drain` did not fire, since the FIFO is initially empty, and so the `notEmpty`, `first` and `deq` are not enabled. Rule `feed` fired, since the `enq` method is enabled when the FIFO is empty.

In each subsequent clock, now that the FIFO is no longer empty, we see that both rules fire, as expected. First `drain` fires and displays the result; this leaves the FIFO empty, enabling the `feed` rule, which also fires. The displayed outputs are 0, 1, 2, ... as expected.

The system stops after 100 clocks, as requested on the command line. The final output is 99, as expected, and the final state can also be seen:

```
.... Clock 100
  Rule ["main","drain"] fired
$DISPLAY: RESULT
$DISPLAY: 99
  Rule ["main","feed"] fired
Cycle limit reached: 100
---- final system:
  0 ["main","x"]    PMIReg 101
  1 ["main","f","full"]    PMICReg 2 1
  2 ["main","f","data"]    PMICReg 2 100
  3 ["main","f"]UMI ...
  4 ["main"]UMI ...
```

### 8.2.2 Executing the semantics with schedule 2

If we instead execute the program with the other schedule, Sched 2, we get a different execution. Its transcript is in file `log_PipelineFIFO_Sched2`. The execution trace begins at clock 0:

```
.... Clock 0
  Rule ["main","feed"] fired
  Rule ["main","drain"]             did not fire
    inter-rule conflicts:
      ["main","f","data"] _write1 > _read0
      ["main","f","full"] _write1 > _read0
      ["main","f","full"] _write1 > _read0
      ["main","f","full"] _write1 > _read0
      ["main","f","full"] _write1 > _write0
```

In clock 0, the FIFO is empty, so rule `feed` is enabled and fires, ultimately using methods `full._read1()`, `full._write1()` and `data._write1()`. Then, rule `drain` cannot fire, since it ultimately invokes `full._read0()`, `full._write0()` and `data._read0()`, and these cannot follow the already-invoked `full._read1()`, `full._write1()` and `data._write1()` (inter-rule conflict). Moving on to clock 1:

```
.... Clock 1
  Rule ["main","feed"] unavail      did not fire
  Rule ["main","drain"] fired
```

```
$DISPLAY: RESULT
$DISPLAY: 0
```

In clock 1, rule `feed` cannot fire because the FIFO is full and so the `enq` method is disabled. Rule `drain` can fire; it empties the FIFO, and displays the dequeued item.

```
.... Clock 2
  Rule ["main","feed"] fired
  Rule ["main","drain"]              did not fire
    ...
```

In clock 2, rule `feed` is once again enabled since the FIFO is empty, but this again prevents rule `drain` from firing, just like in clock 0. In subsequent clocks we see this pattern repeating, with either rule `feed` or rule `drain` firing.

```
.... Clock 3
  Rule ["main","feed"] unavail       did not fire
  Rule ["main","drain"] fired
$DISPLAY: RESULT
$DISPLAY: 1
.... Clock 4
  Rule ["main","feed"] fired
  Rule ["main","drain"]              did not fire
    ...
.... Clock 5
  Rule ["main","feed"] unavail
  Rule ["main","drain"] fired
$DISPLAY: RESULT
$DISPLAY: 2
.... Clock 6
  Rule ["main","feed"] fired
  Rule ["main","drain"]              did not fire
    ...
```

After we reach the requested cycle limit (100), simulation stops. The final output is 49 and the final state of register `x` is 51 (these were 99 and 101, respectively, with Sched 1).

```
.... Clock 99
  Rule ["main","feed"] unavail
  Rule ["main","drain"] fired
$DISPLAY: RESULT
$DISPLAY: 49
.... Clock 100
  Rule ["main","feed"] fired
```

```
  Rule ["main","drain"]
    inter-rule conflicts:
      ...
Cycle limit reached: 100
---- final system:
  0 ["main","x"]    PMIReg 51
  1 ["main","f","full"]    PMICReg 2 1
  2 ["main","f","data"]    PMICReg 2 50
  3 ["main","f"]UMI ...
  4 ["main"]UMI ...
```

In summary, observe that *both schedules provide functionally correct execution*: we enqueue and dequeue correct values. In this functional sense, neither schedule is "correct" or "incorrect".

However, Sched 1 admits more concurrency than Sched 2, because its sequence is "better aligned" with the inter-rule ordering constraints on the methods of the `data` and `full` CRegs. In fact, the *bsc* compiler will automatically pick Sched 1 using this criterion, i.e., it tries to maximize concurrency.

### 8.3 BypassFIFO

The next example is the "dual" of the PipelineFIFO and is called the BypassFIFO. The code is in the file `Progs/BypassFIFO.kbsv`. We leave it as an exercise for the reader, with just the following observations.

In the `PipelineFIFO`, the output-side methods (`notEmpty`, `first` and `deq`) of the FIFO used port 0 methods (`_read0` and `_write0`) of the `full` and `data` Concurrent Registers, and the input-side method (`enq`) used port 1 methods (`_read1` and `_write1`). This allowed concurrent input and output on the FIFO, provided the output side was scheduled before the input side.

In the `BypassFIFO`, we do exactly the opposite: the input-side method (`enq`) uses port 0 methods (`_read0` and `_write0`), and the output-side methods (`notEmpty`, `first` and `deq`) use port 1 methods (`_read1` and `_write1`) of the `full` and `data` Concurrent Registers. This allows concurrent input and output on the FIFO, provided the input side is scheduled before the output side.

The main test program and the two candidate schedules are identical to those in `PipelineFIFO.kbsv`. The transcript using Sched 1 is shown in file `log_BypassFIFO_Sched1`, and using Sched 2 is shown in file `log_BypassFIFO`. As with the PipelineFIFO, both show functionally correct execution, just different levels of concurrency. With Sched 2, both rules can fire on every clock; with Sched 1 they alternate.

### 8.4 PipeFwdBwd

Our final example combines the last two and is representative of structures we find, for example, in CPU pipelines. In a CPU pipeline, the main data flow is in the "forward" direction through the pipe, and the pipeline stages are separated by PiplineFIFOs. Thus, in each clock, rules corresponding to all stages can fire concurrently, allowing the entire pipeline to advance in each clock. However, within each clock, semantically the last stage executes first, then the previous one, then its previous one, and so on until the first stage.

Most CPUs also also have a "reverse" data flow, where later stages may provide updated register values or branch-redirection information back to early stages. For consistent rule ordering with the forward direction, therefore, we use BypassFIFOs in the reverse direction.

Our example is in the file `PipeFwdBwd.kbsv`. It also has two stages, using a PipelineFIFO in the forward direction from rule `feed` to rule `drain`, and a BypassFIFO in the reverse direction from rule `drain` to rule `feed`.

We leave it as an exercise to the reader to run `PipeFwdBwd.kbsv` with each of the two schedules, and to explain their respective behaviors. For convenience, `log_PipeFwdBwd` is a transcript of execution with Sched 1, and `log_PipeFwdBwd_Sched2` is a transcript of execution with Sched 2.

# 9 Conclusion and closing observations

We have described the dynamic execution semantics of BSV programs formally by providing a Haskell program directly implementing the semantics.

We also described a process of static elaboration, but that is not central here; any alternative mechanism is fine that results in a collection of module instances, where each module is either primitive or is user-defined, where user-defined modules have rules and methods that can invoke methods of primitives or other modules.

## 9.1 Relationship to RTL and Chisel: Atomic transactions

In the literature and training materials on BSV, we often refer to rules as *atomic transactions*. Atomicity is trivial in the semantics described in this document because we define *serial* execution of rules according to a schedule, and so there is no question of interleaving of rule actions.

A central issue encountered in concurrent programming is that atomic operations are non-modular, or non-compositional [10]. In other words, a system of modules, each of which implements (locally) atomic operations does not guarantee that system-level actions are atomic. The classic example is a pair of shared bank account modules with `deposit` and `withdraw` methods, each of which are guaranteed atomic. Two concurrent processes that attempt a logical `transfer` operation by withdrawing from one account and depositing into the other may nevertheless see inconsistent results due to bad interleavings of these methods.

This is the reason why user-level atomic transactions cannot be implemented correctly merely with libraries—the compiler and/or run-time system must have visibility over *all* actions of a user-defined atomic transaction, and this set may span many modules/functions/procedures (cf. "Transactional Memory" [22]).

This explains the fundamental power of BSV over traditional RTL languages like Verilog [14], SystemVerilog [17] and VHDL [15]. The semantics of those languages is based entirely on clocks and classical clocked digital logic. There is no concept of user-defined atomic transactions like rules in BSV.

The more recent language Chisel [5] is a much higher-level language than those Verilog, SystemVerilog or VHDL, but its semantics is also fundamentally clock based, just like RTL. Chisel has a syntactic "when" construct that at first glance look like rule and method conditions in BSV, but

they have nothing to do with atomic transactions; they are just a syntactic means of prioritizing updates to a state element, *locally to a syntatic scope* (in particular, they are not modular or compositional).

## 9.2 Relationship to HLS (High Level Synthesis)

The last 10 years has seen the emergence of a number of commercial tools for hardware design under the category of "High Level Synthesis", or HLS for short [8] (we'll refer to this as "classical HLS"). The input design language for these tools is an existing language, typically C or C++ (or SystemC, which is just a C++ library [16]). This is paradoxical, since C and C++ are, formally, completely sequential languages, from which classical HLS tries to generate hardware which, quintessentially, has massive, fine-grained parallelism. Classical HLS tools achieve this with "automatic parallelization", which builds on several decades of research into automatic parallelization of software for vector computers and SIMD computers. The central activity is to analyze for-loops over dense rectangular arrays and to recognize when iterations can be performed safely in parallel, from which they can be mapped into parallel hardware. As a consequence, classical HLS has the same strengths and limitations as classical automatic parallelization; a deeper discussion of these limitations may be found in [9].

It should be evident that BSV and BSV compilation has almost nothing in common with classical HLS. The BSV programmer (unlike the C/C++ programmer) thinks in parallel, and explicitly designs and implements parallel algorithms. The "high-level synthesis" in BSV and the *bsc* compiler refers to a quite different set of properties and activities:

- its extremely powerful Haskell-like type system,
- its extremely powerful Haskell-like static-elaboration,
- its automatic analysis of rules to produce highly concurrent rule schedules,
- and its generation of efficient hardware from this analysis to execute rules concurrently.

Unlike classical HLS, where hardware quality drops off sharply (and may even becomes infeasible) as one moves out of the "sweet spot" of simple loops on dense rectangular arrays, BSV has universal applicability to all kinds of clocked digital hardware and is, in that sense, a genuinely full replacement for RTL.

## 9.3 Relationship of this formal semantics to the full BSV language

Ideally, formal semantics should be defined directly on the source language used by the programmer. If this can be done, and the semantic definition is "simple", then the everyday practicing programmer can directly use it as a mental model while creating, debugging and analyzing actual programs. Everyday practicing programmers find it difficult to usefully exploit formal semantics that is defined on a kernel language that is too far removed from the source language, or that involve esoteric source-to-source transformations.

We have attempted to approach this ideal in this semantics. The language defined in Sec. 4 is admittedly much smaller and much simplified compared to full BSV. But the principal difference

is in types and type-checking, and in a few syntatic facilities like interface definitions and static-elaboration loops. These are quite orthogonal to dynamic semantics and do not affect the mental model held by the everyday practicing programmer. We have also omitted anything related to BSV packages, which again are just about namespace management and separate compilation, and quite orthogonal to dynamic semantics.

Thus, our kernel language captures the essence of BSV for purposes of explaining dynamic semantics. This semantics is in fact the mental model one uses while doing BSV programming.

### 9.4 Relationship of this formal semantics to the *bsc* compiler

This formal semantics is purely a dynamic semantics explained directly on the abstract syntax trees of source code—it takes no position on how to find schedules (rule sequences), nor on compilation to actual hardware. As a result, this semantics is more liberal than what can actually be achieved by any compiler (which, by definition, is working statically).

The *bsc* compiler only produces a *static* schedule—a single fixed schedule that is constant across clocks. Further, these schedules are always permutations of the rule-instance set, i.e., each rule-instance appears in the schedule exactly once. The dynamic semantics has no such restriction.

As described earlier in Sec. 3.7, this semantics takes no position on schedules. A schedule is a parameter to this semantics, and may therefore be produced by an oracle, even one that dynamically selects the next rule at the last moment. There is no such thing in this semantics as a "wrong" schedule; different schedules merely admit more or less concurrency (because they'll encounter less or more conflicts, respectively).

The *bsc* compiler chooses a schedule by static analysis of the rules in a program. In other words, it is statically predicting something about dynamic execution (rule and method conditions, rule conflicts and hardware conflicts). This semantics has exact information, but the compiler can only have estimated or approximate information.

Thus, there are rule firings that may be permitted in this dynamic semantics that will be prohibited in code compiled by *bsc*, because *bsc* can only use conservative approximations (e.g., *bsc* may see a potential conflict where the dynamic semantics knows there is none).

Thus, the semantics described here is more liberal than what is produced today by *bsc*, and covers possible future improvements in *bsc* schedule analysis and compilation.

---

## References


[1] A. Akhiani, D.Doligez, P.Harter, L.Lamport, J.Scheid, M.Tuttle, and Y. Yu. Cache Coherence Verification with TLA+. In *Proc. World Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1871–1872, September 20-24 1999.

[2] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1998.

[3] L. Augustsson, J. Schwartz, and R. S. Nikhil. Bluespec Language Definition. Technical report, Sandburst Corp., 2001. Earlier version of BSV, now called "Bluespec Classic".

[4] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. ISBN 0 521 45520 0.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proc. Design Automation Conferene (DAC)*, 2012. https://chisel.eecs.berkeley.edu/.

[6] Bluespec, Inc. Bluespec$^{TM}$ SystemVerilog Version Reference Guide, 2015. (BSV). Originally published 2003, with extensive subsequent revisions.

[7] K. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[8] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Design and Test of Computers*, pages 8–17, July/August 2009.

[9] S. A. Edwards. The Challenge of Hardware Synthesis from C-like Languages. In *Proc. Design Automation and Test in Europe (DATE), Munich, Germany*, March 2005.

[10] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *ACM Conf. on Principles and Practice of Parallel Programming (PPoPP'05)*, 2005.

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

[12] J. C. Hoe. *Operation-Centric Hardware Description and Synthesis.* PhD thesis, MIT, June 2000.

[13] J. C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *IEEE/ACM Intl. Conf. on Computer Aided Design (ICCAD)*, pages 511–518, 2000.

[14] IEEE. IEEE Standard Verilog (R) Hardware Description Language, March 2001. IEEE Std 1364-2001.

[15] IEEE. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, 2002.

[16] IEEE. IEEE Standard for Standard SystemC Language Reference Manual, January 9 2012. IEEE Std 1666-2011.

[17] IEEE. IEEE Standard for System Verilog—Unified Hardware Design, Specification and Verification Language, 21 February 2013. IEEE Std 1800-2012, 1800-2005.

[18] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking Cache-coherence Protocols with TLA+. *Formal Methods in System Design*, 22(2):125–131, March 2003.

[19] J. Klop. *Term Rewriting Systems*, volume 2, pages 1–116. Oxford University Press, 1992.

[20] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional (Pearson Education), 2002.

[21] B. W. Lampson. Atomic Transactions. In B. W. Lampson, M. Paul, and H. J. Siegert, editors, *Distributed Systems - Architecture and Implementation, An Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1981.

[22] J. Larus and C. Kozyrakis. Transactional Memory. *Communications of the ACM*, 51(7):80–88, July 2008.

[23] C. Metayer, J.-R. Abrial, and L. Voisin. The Event-B Language, May 31 2005. http://rodin.cs.ncl.ac.uk/deliverables.htm.

[24] R. S. Nikhil. *Bluespec: a general-purpose approach to High-Level Synthesis based on parallel atomic transactions*, pages 129–146. Springer, June 2008. Chapter 8. Each chapter is from a different tool vendor or research group in the field.

[25] R. S. Nikhil. Abstraction in Hardware System Design. *Communications of the ACM*, 54(10):36–44, October 2011.

[26] R. S. Nikhil. Types, Functional Programming and Atomic Transactions in Hardware Design. In *In search of elegance in the theory and practice of computation Essays dedicated to Peter Buneman (Festschrift), Lecture Notes in Computer Science, LNCS 8000*, pages 418–431. Springer-Verlag, Berlin, Heidelberg, 2013.

[27] R. S. Nikhil and K. R. Czeck. *BSV by Example*. CreateSpace, December 2010. ISBN-10: 1456418467; ISBN-13: 978-1456418465; avail: Amazon.com.

[28] S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. haskell.org.

[29] J. Stoy, X. Shen, and Arvind. Proofs of Correctness of Cache-Coherence Protocols. In *FME2001: Formal Methods for Increasing Software Productivity*, Berlin, Germany, March 2001. Springer Verlag LNCS 2021.

[30] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. Cambridge Tracts in Theoretical Computer Science 55.