# A Prototype Embedding of Bluespec SystemVerilog in the PVS Theorem Prover

Dominic Richards and David Lester
The University of Manchester
Manchester, U.K.
richardd@cs.man.ac.uk, dlester@cs.man.ac.uk

**Abstract**

Bluespec SystemVerilog (BSV) is a Hardware Description Language based on the guarded action model of concurrency. It has an elegant semantics, which makes it well suited for formal reasoning. To date, a number of BSV designs have been verified with hand proofs, but little work has been conducted on the application of automated reasoning. We present a prototype shallow embedding of BSV in the PVS theorem prover. Our embedding is compatible with the PVS model checker, which can automatically prove an important class of theorems, and can also be used in conjunction with the powerful proof strategies of PVS to verify a broader class of properties than can be achieved with model checking alone.

## 1 Introduction

Bluespec SystemVerilog (BSV) [Nik04] is a language for high-level hardware design, and produces hardware that's competitive with hand-written Register Transfer Level designs in terms of time and area for many applications [WNRD04, Nik04]. It developed from research using Term Rewriting Systems (TRS) to produce hardware specifications that could be synthesised and formally verified [AS99]; in common with TRS, BSV is semantically elegant, which makes it well suited for formal reasoning [ADK08]. However, only one previous investigation has been made into the mechanised verification of BSV designs [SS08], which concentrated on direct model checking of BSV designs in the SPIN model checker [Hol03], without the capacity for interactive reasoning or abstraction.

In this paper we present a prototype shallow embedding of BSV in the PVS theorem prover [ORS92]. Our embedding is compatible with the PVS model checker [ORR$^+$96], which can automatically prove an important class of theorems. Furthermore, the PVS model checker can be used in combination with the powerful proof strategies of PVS to verify a far broader class of properties than can be achieved with model checking alone.

We use a novel application of monads to capture the complex state-encapsulation mechanisms of BSV in a concise and readable way. Our embedding supports several advanced language features of BSV, including module instantiation, side-effecting methods and rule composition from side-effecting methods. In this work we present an embedding strategy: we have yet to build a BSV-to-PVS compiler, which will be developed in conjunction with our related research into automated abstraction of BSV designs.

The material in this paper is presented at length in the principal author's PhD thesis [Ric10a], and the associated code is available online [Ric10b]. Throughout the paper we present code fragments in BSV and PVS; to make the difference clear, all BSV code is presented in Courier font and surrounded by frames labeled 'BSV'. All PVS code is presented in Times New Roman, without surrounding frames.

## 2 Bluespec SystemVerilog

BSV is a language for high-level hardware design. For an in-depth introduction, see the online lecture series at [Blu]. In BSV, hardware is specified in terms of 'modules' that associate elements of state with

'rules' (guarded actions) that transform the state and 'methods' that can be called by other modules to return values from the state, and possibly transform it in the process.

A simple example of a module is `Reg`, which specifies a register with one element of state, no internal rules and two methods, `_read` and `_write`. Other modules can create instances of `Reg`, and use the methods `_read` and `_write` in their own rules and methods. For example, the following rule uses two registers, `request` and `acknowledge`, both of which hold elements of type `Bool`[1]:

```
rule request_rl (!(request._read() || acknowledge._read()));
  request._write(True);
endrule
```
BSV

The rule has a guard, which is a predicate on the state of the registers, and an 'action', which transforms the state of the `request` register. In general, a rule has the form:

```
rule my_rule (rl_guard);
  statement_1;
  statement_2;
  ...
endrule
```
BSV

The statements in the rule body are individual actions that transform the state in some way. The set of statements are applied *in parallel* to the state; each statement is applied to the state as it was immediately before the rule was activated, so that the changes made by `statement_1` aren't seen by `statement_2`, or any other statement. The BSV compiler ensures that the statements in a rule don't conflict with each other by simultaneously attempting to write to the same elements of state.

# 3 The Challenges of Embedding BSV in PVS

BSV uses guarded actions to express concurrency, which makes it similar to the guarded action languages that were developed for the formal study of concurrency; these include UNITY [CM88], Promela [Hol03], the SAL language [dMOR+04] and the model checkable subset of the PVS language [ORR+96]. There is a rich body of literature on the use of model checkers and, to a lesser extent, theorem provers for verifying systems expressed in these languages. However, BVS is a more complex language in some respects, being intended for hardware design rather than abstract specification:

1. **Complex encapsulation of state.** As seen in §2, BSV has a 'module' construct that allows elements of state to be associated with 'rules' and 'methods'. Rules can be composed from the methods provided by other modules; in this way, the execution of a rule in one module can alter the state in another.

2. **Widespread presence of data paths.** Model checking is a useful tool for efficiently verifying finite state concurrent systems, but can be confounded by the presence of data paths (circuits that hold elements of data). Data paths can have very large state spaces, which can cause state space explosions in model checkers; for this reason, model checking has been more widely used to verify control-based systems. When abstract specification languages such as UNITY are used for hardware verification, a model can be constructed that only specifies the control-based components

[1] BSV users will notice that we've de-sugared the method calls for Regs; we do this throughout the paper to simplify the semantics, and also to emphasize the use of methods inside rules.

of a design (when the data path is irrelevant to the properties being verified), or specifies some abstract interpretation of the data path. With an embedding of BSV, however, the hardware design *is* the specification; we can't chose to omit data paths from our formal model. Because of this, we must find a tractable way to abstract away from data path complexity within our proof environment.

So, in order to produce a usable general purpose embedding of BSV in a formal guarded action language, we must first bridge the *semantic gap* by expressing the constructs of BSV with the more limited constructs of our target specification language, preferably in such a way that equivalence between the two can easily be established [CDH$^+$00]. When we have achieved this, we must also bridge the *abstraction gap*, to obtain abstract specifications that can be efficiently verified [SS99, CDH$^+$00].

In this paper we concentrate on the first of these two steps. We offer an embedding strategy that bridges the semantic gap between BSV and the model checkable subset of PVS with a novel use of monads [Bir98]. In further work, we hope to employ automatic abstraction [SS99] to our PVS specifications in order to bridge the abstraction gap.

We actually introduce two embedding strategies with different strengths and weaknesses, and combine them in a hybrid technique that preserves the strengths of both. In §5 we introduce an embedding strategy that we call 'primitive embedding', where we specify BSV modules with 'primitive transition relations'. Primitive transition relations can be efficiently model checked, but bear little resemblance to the BSV modules they represent, and we discuss the problems that this creates. In §6 we introduce an embedding strategy that we call 'monadic embedding', where we specify BSV modules with 'monadic transition relations'. Monadic transition relations are syntactically similar to the BSV modules they represent, which cleanly bridges the semantic gap. However, verification is demanding in terms of CPU time. We go on to introduce a hybrid technique that allows us to perform proofs over monadic transition relations with the efficiency normally associated with proofs over primitive transition relations. We demonstrate the practical applicability of our approach in §7 with a hand embedding of a BSV fair arbiter, for which we verify deadlock freedom, mutual exclusion and fairness properties.

# 4 The Semantics of a BSV Module

The behavior of a BSV module can be understood in terms of a simple semantics called Term Rewriting System (TRS) semantics, also called 'one-rule-at-a-time' semantics. According to TRS semantics, a module evolves from a given state by choosing *one* rule for which the guard is true and applying the associated action to transform the state. If more than one guard is true, a nondeterministic choice is made. When actual hardware is generated from BSV designs, a number of optimisations are applied (for example, a clock is introduced and multiple rules are executed per clock cycle) but the behavior is guaranteed to comply with the TRS semantics.

The TRS semantics gives the designer a simple, high-level way of understanding his design, and we use it in our embedding. In PVS, we describe a rule as a predicate over pairs of states:

my_rule (pre, post: Module_State): bool = rl_guard (pre) $\wedge$ post = rl_action (pre)

Here, rl_guard and rl_action are the PVS representations of the rule's guard and action, and Module_State is the PVS representation of the module's state. We can then express the TRS semantics of a module by composing its rules together:

TRS_transition_relation (pre, post: Module_State): bool = $\text{rule}_1$ (pre, post) $\vee$ $\text{rule}_2$ (pre, post) $\vee$ ...

TRS_transition_relation is a binary relation on 'pre' and 'post'. It relates 'pre' to 'post' if any of its constituent rules relates them when applied in isolation: we have a nondeterministic, one-rule-at-a-time semantics. We consider the possible forms of Module_State and $\text{rule}_1$ etc. in the following sections.

# 5 A Primitive Embedding of BSV in PVS

PVS has a set of proof strategies for using model checking to verify temporal logic assertions about guarded action systems [ORR+96, SS99]. The state of a system can be defined inductively from boolean and scalar types, using tuples, records or arrays over subranges, and the transition relation is defined as a binary relation over pairs of states (as with TRS_transition_relation in §4).

Consider the following rule, which comes from the arbiter example of §7:

```
rule ack1_with_tok (token1._read() && req1._read()
                    && !(ack1._read() || ack2._read() || ack3._read()));
  ack1._write (True);
  move_token;
endrule
```
BSV

The guard of this rule is a predicate over the state of five registers and the action changes the state of register `ack1`, and also calls the 'local action' `move_token`:

```
Action move_token = (action token1._write(token3._read());
                            token2._write(token1._read());
                            token3._write(token2._read()); endaction);
```
BSV

We can specify the state of the `Reg` module as a PVS record with one field, and use this to specify the state of our arbiter system (which has nine boolean registers):

Reg: TYPE = [# val: $T$ #]

Arbiter: TYPE = [# req1, req2, req3, ack1, ack2, ack3, tok1, tok2, tok3 : Reg[bool]#]

The state of the `Reg` module could be expressed with a variable of type T, rather than a single-element record. In general, however, modules have more complex states (for example, even a simple two element 'First In, First Out' buffer would need at least 3 variables) so we factor the complexity of nested module states into records. This also helps to simplify our monadic embedding in §6.

We can now express the rule `ack1_with_tok` as a predicate over pairs of Arbiters:

```
ack1_with_tok_concrete (pre, post: Arbiter): bool =
  pre`tok1`val ∧ pre`req1`val ∧ ¬ (pre`ack1`val ∨ pre`ack2`val ∨ pre`ack3`val)
  ∧ post = pre WITH [(ack1) := (# val := TRUE #),
                     (tok1) := (# val := pre`tok3`val #),
                     (tok2) := (# val := pre`tok1`val #),
                     (tok3) := (# val := pre`tok2`val #)]
```

This is a straightforward way to express rules, and it's compatible with the PVS model checker. The state of the module is expressed as a record; rule guards are then expressed as predicates over the fields of this record, and rule actions are expressed as record updates. When a statement calls a local action, the action can be expanded in-place to a series of record updates (as we did here with the local action `move_token`). Methods, which generally return some value and perform an action to transform the state, can be expanded in-place by expressing the value they return as a pure function on the state, and the action as a series of record updates. When methods appear in the guard, they will be side-effect free (this is guaranteed by the BSV compiler) and can be expanded to pure functions on the state. This

is our 'primitive' embedding strategy. We can represent a BSV module by specifying all of its rules in this way, and combining them using the TRS_transition_relation function of §4; we then refer to this as a 'primitive transition relation'.

This approach seems quite simple, but it has a drawback. If we express a rule by fully expanding all of the actions and method calls, we expose its full complexity; BSV provides the module and method constructs to avoid just this. If we specify a more complex module in this way (for example, one where rules and methods call methods that themselves call methods, all returning values and producing side-effects) we end up with a long-winded specification that bears little resemblance to the BSV module it represents. If we assume that the process of translating from BSV to PVS is not formally verified, it becomes difficult to provide assurance that the translation is accurate; we have a large and unverified semantic gap. This makes it difficult to rule out false positives when a property is proven, or conversely false negatives when a property is disproved. We would like to have a PVS representation that's compatible with the PVS model checker, but also relates back to the BSV code in a simple and transparent way. Our solution is to use a variation of the state monad from functional programming [Bir98].

# 6 A Monadic Embedding

Monads are a technique for representing state in pure functional languages. A monad is simply a function that takes an element of state as input and returns a value and a new state. Monads can by composed sequentially in the same way as statements in a procedural language; this is achieved with the infix operator $\gg$ (pronounced 'seq'), which is equivalent to the semi-colon in procedural languages, and also a related function $\gg=$ ('bind').

In this section we show that monads allow us to embed BSV rules at a higher level of abstraction than our primitive embedding of §5. If this is your first exposure to monads, the first half of the section should be fairly accessible (up to the definitions of $\gg=$ and $\gg$). The rest of the section may require some background reading; for example, [Bir98] has an excellent chapter on monads. Also, the principal author's PhD thesis [Ric10a] gives an introduction to monads in the context of the work presented here.

We develop a new embedding strategy for rules, where actions and method calls are expressed as monads, rather than being expanded in-place. Before getting into the details, let's take a look at the result. This is our monadic embedding of the rule used in §5:

```
ack1_with_tok = rule (tok1`read ∧ req1`read ∧ ¬ (ack1`read ∨ ack2`read ∨ ack3`read))
                     (ack1`write (TRUE) ≫
                      move_token)
```

At the level of syntax, this is very similar the original rule. In contrast to the primitive embedding strategy of §5, the complexity of methods and actions is factored out into monads. This yields rule specifications that are syntactically similar to the BSV rules that they represent, so that errors in the BSV-to-PVS translation process will be discernible by inspection; with a far smaller semantic gap, false positives and false negatives can be more easily ruled out. Because of the syntactic similarity, the BSV-to-PVS translator will be simpler; translation will essentially be a task of converting between two very similar concrete syntaxes. This is important because we expect that the BSV-to-PVS translator won't be formally verified, so we must keep it as simple as possible. Furthermore, we hope that the structured, compositional nature of our monadic specifications will simplify deductive reasoning when it's necessary.

ack1_with_tok is a function. It has the type $\big[[\text{Arbiter, Arbiter}] \to \text{bool}\big]$ and is, in fact, extensionally equivalent to ack1_with_tok_primitive from §5; it's just a more concise way of writing the same function. If we fully expand all of the functions in the definition of ack1_with_tok, we end up with the definition of ack1_with_tok_primitive; a simple function involving record updates and functions over

record fields (we can do this in the PVS proof environment with the `(expand*)` proof strategy). We can specify a BSV module by forming monadic specifications of its rules, and combining them using the TRS_transition_relation function of §4; we then refer to this as a 'monadic transition relation'.

The PVS model checker fails when it's called directly on a monadic transition relation; this is possibly because of the extensive use of higher order functions. One solution is to expand the monadic transition relation to the equivalent primitive transition relation with the `(expand*)` proof strategy, then call the `(model-check)` strategy, and finally discharge a small number of subgoals with the `(grind)` strategy. In our experience, this approach is fine for small examples but the expansion step is quite demanding in terms of CPU time, which might become problematic for larger examples.

We can avoid expanding the monadic transition relation *during every proof* if we compile a PVS theory containing the primitive transition relation. We can state in a lemma that the primitive transition relation is extensionally equivalent to the monadic transition relation, and prove it with a single application of the proof strategy `(grind-with-ext)`. The PVS prover then treats the lemma as a rewrite rule, so that we can call the `(model-check)` proof strategy directly on the monadic transition relation, and the prover automatically rewrites it to the equivalent primitive transition relation, which can be efficiently model checked. This approach makes proofs faster in CPU time because we don't need to expand the monadic transition relation during every proof. There are additional overheads incurred because we must prove the equivalence lemma and compile the PVS theory containing the primitive transition relation, but these are only done once and can be re-used for all proofs. We evaluate the two verification approaches in §7. However, we can't yet evaluate the overhead incurred by compilation of the primitive transition relation because we have yet to build a BSV-to-PVS translator; we currently compile by hand.

## 6.1 A State Monad in PVS

So, how can we use monads to express actions and methods without expanding their full complexity in-place? Consider the body of action `move_token`:

```
token1._write(token3._read());
token2._write(token1._read());
token3._write(token2._read());
```
BSV

We have three statements, each composed of two method calls. The meaning we want to capture for the whole statement block is that an initial state is transformed independently by the three statements, and the changes made by each are combined to give a new state. We can actually achieve the same effect by applying the statements sequentially; we can apply the first statement to get a partially updated state, then apply the second statement to update this new state, and apply the third statement to update the result. This is possible because the statements are conflict-free; no two statements will update the same element of state, so we don't need to worry about later statements over-writing the updates made by earlier statements. However, each statement needs access to the initial state, as earlier statements might update elements of state that later statements need to read. This suggests that we specify statements as instances of the type:

$$\text{Monad: TYPE} = \big[\big[S,\ S\big] \rightarrow \big[A,\ S\big]\big]$$

'S' is the type of the module's state (in the case of `move_token`, it's Arbiter from §5). 'A' is the type of some return value; for the statements under consideration, it's Null. Instances of Monad take two copies of the module state (representing the initial state, and a partially updated state) and return a value and a new instance of the state, with any additional updates added to those of the partially updated state. We can compose statements to form rule bodies with the standard monad connectors (see [Bir98] for a good introduction) with $\gg=$ adapted to accept a pair of input states rather than a single input state:

```
>>= (m: Monad[S, A], k: [A -> Monad [S, B]]): Monad [S, B] =
  λ (init, updates: S): LET (val, new_updates) = m (init, updates) IN k (val) (init, new_updates)

>> (m: Monad[S, A], n: Monad[S, B]): Monad[S, B] = m >>= λ (val : A) : n
```

Shortly, we will introduce monads that specify register methods `token1._read`, `token1._write` etc. and call them tok1‘read, tok1‘write etc. We can use these to compose a specification of `move_token`:

```
move_token: Monad [Arbiter, Null] = tok1‘read >>= tok2‘write >>
                                    tok2‘read >>= tok3‘write >>
                                    tok3‘read >>= tok1‘write
```

## 6.2 Monad Transformers

What form will the monads tok1‘read, tok1‘write etc. have? Given that move_tok has type Monad[Arbiter, Null], they are constrained to have the type Monad[Arbiter, T] for some T; that is to say, they must operate on instances of the Arbiter state, despite the fact that they only influence one register within that state. We can achieve this by specifying the two generic register methods (read and write) as monads that act on type Reg[T], and lifting them to monads that act on type Arbiter with monad transformers:

```
read : Monad[Reg[T], T] = λ (init, updates: Reg[T]): (init‘val, updates)

write (d: T) : Monad [Reg[T], Null] = λ (init, updates: Reg[T]): (null, (# val := d #))

Transformer : TYPE = [Monad [R, A] -> Monad [S, A]]

transform (get_R: [S -> R], update_R: [[S, R] -> S]): Transformer =
  λ (m: Monad [R, A]) (init, updates: S) : LET (val, new_updates) = m (get_R (init), get_R (updates))
                                           IN (val, update_R (updates, new_updates))
```

A function of type Transformer takes a monad over state R and lifts it to become a monad over state S. We can use the ‘transform’ function to produce a Transformer that lifts the generic register functions (read and write) to operate on our Arbiter state. For example, we can do this for the tok1 register:

```
tok1T: Transformer[Reg[bool], Arbiter, T] = transform(get_tok1, update_tok1)
```

where get_tok1 and update_tok1 access and update the tok1 field of an Arbiter record. We can then define our lifted monads tok1‘read and tok1‘write:

```
tok1: [# read : Monad[Arbiter, bool], write : [bool -> Monad [Arbiter, Null]] #]
    = (# read := tok1T [bool] (read),     write := λ (x : bool) : tok1T [Null] (write (x)) #)
```

We can use these monads in the guard if we overload the standard Boolean and equality operators with functions over monads. For example:

```
∧ (m, n: Monad[S, bool]) : Monad [S, bool]
    = λ (init, updates : S) : LET b1 = (m (init, updates))‘1, b2 = (n (init, updates))‘1
                              IN (b1 ∧ b2, updates)
```

This allows us to construct guard predicates in a readable way, having a concrete syntax similar to guards in BSV. An example of this was seen earlier in the section, in the guard of rule ack1_with_tok.

Finally, when we have monadic specifications of a rule’s guard and body, we can form a ‘rule’ that is a predicate over pairs of states, using the function:

rule (guard: Monad [S, bool]) (action: Monad [S, Null]) (pre, post: S): bool =
  (guard (pre, pre))`1 ∧ post = (action (pre, pre))`2

## 7 Experimental Results: Embedding a Fair Arbiter

To evaluate our embedding strategy, we produced a BSV design for the control circuitry of a 3-input fair arbiter, which we embedded in PVS and verified with the PVS model checker. Our design is a variation of the synchronous bus arbiter presented in [dM04], and amounts to just over 100 lines of BSV.

We represent the three inputs by Boolean 'request' registers, which are used to request access to the output. The arbiter also has a Boolean 'acknowledge' register for each input, which is used to inform the input client that its request has been acknowledged, and that it has exclusive access to the output until the request is removed. In order to guarantee fair access to the output, the arbiter also has a separate Boolean 'token' register for each input; of the three 'token' registers, only one is set to true at any time. When an input 'has the token' the arbiter gives it priority over all other incoming requests. Each time a request is granted, the token is cycled round to another input, so that all inputs are guaranteed to receive the token (and therefore access to the output) infinitely often.

We use the Computation Tree Logic of PVS [ORR+96] to specify deadlock freedom, mutual exclusion and fairness properties for our arbiter, using an initial state defined by mkArbiter. The deadlock freedom theorem declares that every state that can be reached from the initial state by repeated application of the transition relation can reach another, different state with one more application. The mutual exclusion theorem declares that no two inputs will ever be acknowledged (and hence have access to the output) at the same time. The fairness theorem declares that a request by any input will be acknowledged:

deadlock_freedom: THEOREM AG (transitions, λ $a$ : EX (transitions, λ $a_1$ : $a \neq a_1$) ($a$)) (mkArbiter)

mutex: THEOREM AG (transitions, mutex_pred) (mkArbiter)
  WHERE mutex_pred ($a$) =
    ¬ ($a$`ack1`val ∧ $a$`ack2`val ∨ $a$`ack2`val ∧ $a$`ack3`val ∨ $a$`ack3`val ∧ $a$`ack1`val)

fairness: THEOREM AG (transitions, fairness_pred) (mkArbiter)
  WHERE fairness_pred ($a$) = ($a$`req1`val ∧ ¬ $a$`ack1`val ⇒ AF (transitions, λ $a_1$ : $a_1$`ack1`val) ($a$))
    ∧ ($a$`req2`val ∧ ¬ $a$`ack2`val ⇒ AF (transitions, λ $a_1$ : $a_1$`ack2`val) ($a$))
    ∧ ($a$`req3`val ∧ ¬ $a$`ack3`val ⇒ AF (transitions, λ $a_1$ : $a_1$`ack3`val) ($a$))

We proved all three theorems with the 'expansion' and 'rewrite' approaches introduced in §6. With both approaches, the proofs are relatively simple to carry out. For the 'expansion' approach, we call `(expand*)` followed by `(model-check)` and `(grind)`. The 'rewrite' approach requires a proof of equivalence between the monadic and primitive transition relations, which could be done with the strategy `(grind-with-ext)`, and thereafter the theorems of interest are verified with `(model-check)` and `(grind)`. In terms of CPU time, we found that the 'rewrite' approach was significantly faster:

| Theorem | Proof with Expansion | Proof with Rewrite | Speedup |
|---|---|---|---|
| deadlock_freedom | 39 secs | 0.54 secs | × 72 |
| mutex | 15 secs | 0.39 secs | × 38 |
| fairness | 79 secs | 2.0 secs | × 40 |
| equivalence of transition relations | – | 3.3 secs | – |

The figures were obtained on a MacBook Pro with an Intel Core 2 Duo 2.53 GHz processor and 2 GB

1067 MHz DDR3 memory. We could not quantify the extra CPU time required to compile the primitive transition relation in the 'rewrite' approach, as we currently compile by hand.

# 8 Related Work

Prior to our research, there was one investigation into the mechanised verification of BSV designs. In [SS08] Singh and Shulka present a translation schema from a subset of BSV into Promela, the specification language of the SPIN model checker. They use SPIN to verify that rule scheduling is a valid refinement, and also to verify LTL assertions. The subset of BSV that they consider is similar to ours, but doesn't include: instantiation of non-trivial nested modules; methods with arbitrary side-effects and return values; rule composition from methods with arbitrary side-effects and return values. They translate directly to Promela, in contrast to our approach, which uses monads to bridge the semantic gap.

In addition to the PVS embedding in this paper, we have also embedded the same subset of BSV in the SAL model-checker [RL10]. There are a number of advantages in compiling to a specialized model checker such as SAL. The PVS model checker is symbolic, but it's often useful to have access to explicit or bounded model checking; SAL provides both of these. Also, the PVS model checker doesn't provide counter-example traces when it fails to prove a property, whereas SAL does. However, we could not recreate our monadic embedding in SAL, because it requires interactive proof strategies. Instead, we produce a primitive embedding in SAL and use PVS to prove that instances of this embedding are equivalent to instances of a monadic embedding expressed in PVS. This is a good example of the benefits of interactive theorem proving, even for systems which at first inspection seem to fit well into fully automated proof tools.

In the wider literature, our work sits somewhere between theorem prover embeddings of formal guarded action languages and model checking of main-stream hardware and software languages. There are a number of theorem-prover embeddings of guarded action languages. In [Pau00], Paulson embeds UNITY in the Isabelle [Pau94] theorem prover. Rather than using model checking to discharge repetitive proofs, he uses a set-based formalism in his embedding that allows efficient use of Isabelle's proof tactics. In [CDLM08], Chaudhuri *et. al.* present a proof environment for TLA$^+$ [Lam02]; as with Paulson, they use automated deduction rather than model checking to lessen the proof burden.

Because languages like UNITY and TLA$^+$ were developed for specification rather than design, there is less emphasis on the use of abstraction for state-space reduction, which will be necessary in a general-purpose verification tool for BSV. There have been a number of applications of abstraction and model checking to verify programs expressed in main-stream hardware and software languages. Some of the larger projects include the Java model checkers Bandera [CDH$^+$00] and Java PathFinder [VHBP00], and the C model checkers FeaVer [HS00] and SLAM [BR02]. All of these tools employ some combination of static analysis, slicing, abstract interpretation and specialization to reduce the state space.

Monads have been used several times before to express state in theorem prover specification languages; notable examples include [Fil03, KM02, BKH$^+$08].

# 9 Conclusions and Further Work

We have presented a shallow embedding for a subset of Bluespec SystemVerilog in the PVS theorem prover. Our embedding uses monads to bridge the semantic gap, whilst allowing efficient use of the PVS model checker to discharge several important classes of proofs.

In further work, we plan to extend our approach with automated abstraction. PVS has a number of proof strategies for automatic predicate abstraction [SS99] that are closely integrated with the PVS model checker; this will be our first line of investigation. We hope to combine our work in PVS and SAL

by producing a translator that compiles BSV designs to both languages, giving users the benefits of both tools. We intend to road-test our combined system by verifying a BSV model of the communications network of the SpiNNaker super-computer, which we have previously verified using Haskell [RL09].

## References


[ADK08] Arvind, N. Dave, and M. Katelman. Getting formal verification into design flow. In *Proc. FM*, 2008.

[AS99] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3), 1999.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edition, 1998.

[BKH+08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *Proc. TPHOLs '08*, 2008.

[Blu] Bluespec Inc. Bluespec SystemVerilog online training. `http://www.demosondemand.com/`.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project. In *Proc. POPL '02*, 2002.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE '00*, 2000.

[CDLM08] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA+ Proof System. In *Proc. KEAPPA*, 2008.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison Wesley, 1988.

[dM04] Leonardo de Moura. SAL: Tutorial. Technical report, SRI International, November 2004.

[dMOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proc. CAV'04*, 2004.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *JFP*, 13(4), 2003.

[Hol03] Gerard Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, 2003.

[HS00] G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2), 2000.

[KM02] S. Krstic and J. Matthews. Verifying BDD algorithms through monadic interpretation. In *Proc. VMCAI '02*, 2002.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[Nik04] R. Nikhil. Bluespec SystemVerilog. In *Proc. MEMOCODE '04*, 2004.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. CAV '96*, 1996.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS. In *Proc. CADE-11*, 1992.

[Pau94] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. LNCS. 1994.

[Pau00] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 2000.

[Ric10a] Dominic Richards. *Automated Reasoning for Hardware Description Languages*. PhD in Preparation, The University of Manchester, UK. 2010.

[Ric10b] Dominic Richards. Source code. `https://sourceforge.net/projects/ar4bluespec`, 2010.

[RL09] Dominic Richards and David Lester. Concurrent functions: A system for the verification of networks-on-chip. *Proc. HFL'09*, 2009.

[RL10] Dominic Richards and David Lester. A prototype embedding of Bluespec SystemVerilog in the SAL model checker. *Proc. DCC'10*, 2010.

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In *CAV '99*, 1999.

[SS08] Gaurav Singh and Sandeep K. Shukla. Verifying compiler based refinement of Bluespec specifications using the SPIN model checker. In *Proc. SPIN '08*, Berlin, Heidelberg, 2008.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. ASE '00*, 2000.

[WNRD04] W-F Wong, R.S. Nikhil, D.L. Rosenband, and N. Dave. High-level synthesis. In *Proc. CAD04*, 2004.