A Prototype Embedding of Bluespec SystemVerilog in the SAL Model Checker

Dominic Richards and David Lester

Advanced Processor Technologies Group
The University of Manchester
Introduction

- Bluespec SystemVerilog (BSV) is a language for high level hardware design
- Developed from Term Rewriting Systems (TRS)
  - A language for designing and formally verifying hardware
- Elegant semantics => well suited for formal verification
- To date, a number of BSV designs have been verified with hand proof, but little work conducted on the application of automated reasoning.
- We have investigated automated reasoning for BSV, in the SAL model checker, and also the PVS theorem prover
Why Use Automated Reasoning?

- Hand proofs are convenient, but:
  - Can contain errors (analogy - doing arithmetic by hand v.s. on a calculator)
  - Proofs for large systems can be time consuming and tedious

- Automated reasoning has the potential to provide rigorous and efficient verification for some classes of systems...
  - … and these classes are ever expanding
Automated Reasoning for BSV

- Two approaches:
  - Verifying BSV designs with a model checker:
    - Presented today
  - Verifying BSV designs with a theorem prover:
    - Currently compile by hand
In This Presentation...

- Introduce BSV
- Introduce the SAL language
- Outline key challenges of embedding BSV in SAL
- Outline of our approach
- Experimental results: verifying a BSV implementation of Peterson's Protocol
Take Home Information

• Basic understanding of BSV SAL languages
• How to embed BSV in SAL
  – Surprisingly simple
• Understanding of advantages of verifying the embedding
  – Makes proof more rigorous
• Motivation to look at the paper for a strategy for verifying the translation
Bluespec SystemVerilog

- A Hardware Description Language based on the guarded action model of concurrency
- Hardware specified with modules, which associate elements of state with:
  - **Rules**: guarded actions that spontaneously change the state
  - **Methods**: functions that return values from the state and/or transform it
    - Methods from one module can be used to compose the rules and methods of other modules
Rules in BSV

```hs
rule my_rule (rl_guard);
  statement_1;
  statement_2;
  ...
endrule
```
The Semantics of a BSV Module

- Behaviour of a module can be understood with a simple semantics called Term Rewriting System (TRS) semantics
  - Also called one-rule-at-a-time semantics

- In a given state, a module chooses one rule for which the guard evaluates to `true' and applies the associated action

- If more than one guard is true, a non-deterministic choice is made
Bluespec SystemVerilog

- **Reg module:**
  - A register with 1 element of state and 2 methods: `_read` and `_write`

- Other modules can create instances of Reg, and use `_read` and `_write` in their rules and methods. Eg:

```verilog
define rule request_r1 (!request._read && !acknowledge._read));
    request._write(True);
endrule
```
The SAL Language

- Also a guarded action language, but simpler
- Guarded action systems defined in contexts that define:
  - Type of state
  - An initial state
  - A transition relation
The SAL Language

TRANSITION
[
  guarded_action_1 : guard_1 --> action_1
[
  guarded_action_2 : guard_2 --> action_2
[
  ...
]
]
The Challenges of Embedding BSV in the SAL Language

- BSV is a guarded action language
- Similar to specification languages of several proof tools:
  - Model checkers: SAL, SPIN etc.
  - Model checkable subset of the PVS theorem prover
- However, BSV is a more complex language in some respects...
The Challenges of Embedding BSV in a Automated Proof Tools

• Complex language constructs:
  – Modules and methods

• Widespread presence of data paths:
  – Can't always directly apply model checking to designs with data paths due to state space explosion
  – In SAL etc., we can build a specification that excludes data paths...
  – … but with BSV, the design is the specification
The Challenges of Embedding BSV in a Guarded Action Language

- Bridge the semantic gap
  - Express the constructs of BSV with the more limited constructs of the target language

- Bridge the abstraction gap
  - Abstract away from data path complexity to give abstract specifications that can be efficiently verified

- Our work concentrates on bridging the semantic gap
Bridging the Semantic Gap

- Translate BSV to SAL specifications that can be efficiently model checked, but bear little resemblance to the original BSV
  - Problematic, because difficult to rule out false positives and false negatives
- Verify the BSV-to-SAL translation with deductive proof
  - Currently performed in the PVS theorem prover
  - Simple proof, could possibly be done with an SMT solver
An Example Rule

rule p_critical (pcp._read == Critical && fifo.notFull);
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule
A Primitive Embedding in SAL

Reg \{T : type\} : CONTEXT = BEGIN
  State : type = [# data : T #];
END

FIFOF1 \{T : type\} : CONTEXT = BEGIN
  State : type = [# notFull : bool, notEmpty : bool, data : T #];
END
A Primitive Embedding in SAL

PC: TYPE = {Sleeping, Trying, Critical};

...  

pcp : Reg{PC}!State,

pcq : Reg{PC}!State,

turn : Reg{bool}!State,

fifo : FIFOF1{bool}!State
Rules in BSV

\[ p_{\text{critical}} : pcp.data = \text{Critical and fifo.notFull} \]

\[ \rightarrow \text{fifo'} = (\# \text{ data := true, notFull := false, notEmpty := true #}); \]

\[ \text{pcp'} = (\# \text{ data := Sleeping #}); \]

\[ \text{turn'} = (\# \text{ data := false #}) \]
rule p_critical (pcp._read == Critical && fifo.notFull);
    fifo.enq (True);
    pcp._write (Sleeping);
    turn._write (False);
endrule

p_critical : pcp.data = Critical and fifo.notFull
          --> fifo' = (# data := true,
                      notFull := false,
                      notEmpty := true #);
          pcp' = (# data := Sleeping #);
          turn' = (# data := false #)
BSV-to-SAL Translation

BSV Code → AST → Monadic PVS Embedding → Proof → Primitive PVS Embedding → Primitive SAL Embedding → Expanded AST → BSV Code

Currently in PVS, but might be possible in SMT Solver
A Module's State in PVS

Peterson : type = [# pcp : Reg [PC],
                 pcq : Reg [PC],
                 turn : Reg [bool],
                 fifo : Fifof1 [bool]
                 #]
Primitive Embedding in PVS

\[
p_{\text{critical\_primitive}} (\text{pre}, \text{post} : \text{Peterson}) : \text{bool}
\]
\[
= \text{pre} \text{'pcp'data = Critical} \land \text{pre}'\text{fifo'notFull}
\land \text{post} = \text{pre with [(fifo) := (# data := true, notFull := false, notEmpty := true #), (pcp) := (# data := Sleeping #), (turn) := (# data := false #)]}
\]
A Monadic Embedding in PVS

\[ p_{\text{critical}} = \text{rule} \ (pcp'\text{read} = \text{Critical} \land \ \text{fifo'notFull}) \]
\[ (\text{fifo'enq} \ (\text{true}) \gg \]
\[ \text{pcp'write} \ (\text{Sleeping}) \gg \]
\[ \text{turn'write} \ (\text{false})) \]
Rules in BSV

rule p_critical (pcp._read == Critical && fifo.notFull);  
fifo.enq (True);  
pcp._write (Sleeping);  
turn._write (False);  
endrule

p_critical = rule (pcp‘read = Critical \∧\ fifo‘notFull)  
(fifo‘enq (true) \⇒\  
pcp‘write (Sleeping) \⇒\  
turn‘write (false))
Experimental Results: Peterson's Protocol

- Verified a BSV implementation of 2 process Peterson's Protocol
- 50 lines of BSV code (extracts provided in paper)
- Hand embedded BSV code in SAL
- Verified the BSV translation in PVS
- All code will shortly be on sourceforge
  - Search on sourceforge for “Bluespec”
Example: Peterson's Protocol

mutex: THEOREM

\[ \text{System } \vdash \ G(\text{NOT}(\text{pcp.data = Critical AND pcq.data = Critical})) \]

“The two processes will never be in critical mode at the same time”

liveness: THEOREM

\[ \text{System } \vdash \ (G(F(\text{pcp.data = Trying})) \Rightarrow G(F(\text{pcp.data = Critical}))) \]
\[ \text{and } (G(F(\text{pcq.data = Trying})) \Rightarrow G(F(\text{pcq.data = Critical}))) \]

“A Trying process will always (eventually) gain access to the Critical mode”
Conclusion

• BSV is a semantically elegant HDL
  – Well suited for formal reasoning
  – But little work carried out on application of automated reasoning

• We have carried out investigations into the application of model checking and theorem proving for verifying BSV designs

• Today, I presented a strategy for embedding a subset of BSV in SAL model checker, where BSV-to-SAL translation is verified in PVS