A Prototype Embedding of Bluespec SystemVerilog in the PVS Theorem Prover

Dominic Richards and David Lester

Advanced Processor Technologies Group
The University of Manchester
Introduction

• Bluespec SystemVerilog (‘Bluespec’) is a formally-inspired Hardware Description Language
• Elegant semantics => well suited for formal verification
• To date, a number of Bluespec designs have been verified with hand proof, but little work has been done on the application of automated reasoning
Introduction

• We're using PVS to experiment with automatic proof for Bluespec
  – We have embedded a subset of Bluespec in PVS
  – Embedding is compatible with the PVS model checker
  – This allows us to experiment with verification strategies that use a combination of model checking and interactive proof
  – We currently translate from Bluespec to PVS by hand
• All code is on sourceforge
Why Investigate Automated Reasoning for Bluespec?

- Strong demand in the Integrated Circuit industry for automatic proof support, to combat increasing design complexity
- International Technology Roadmap for Semiconductors 2009:
  - “[The] cost of design is the greatest threat to the continuation of the semiconductor roadmap”
  - Describes verification as “a bottleneck that has now reached crisis proportions”
  - Includes an extensive manifesto for the increased role of formal methods
Why Investigate Automated Reasoning for Hardware Description Languages?

• In 2009, 9.4% of design errors exposed using formal verification.

• ITRS '09 recommends **by 2024, 45% of all design errors** exposed using formal verification. Achieved by:
  – Increasing use of formalized languages at early stage in design cycle.
  – Complete mechanical proof of equivalence between all system specifications.
Why Investigate Automated Reasoning for Hardware Description Languages?

- Technology to maintain ITRS schedule:
  - Up to 2012 with tools currently in use
  - Up to 2016 with tools currently in development
  - **No known solutions** to maintain schedule past 2016

Broader scope — Greater capacity — More reliable — **New research required**
The Potential for Formal Methods in a Bluespec Design Flow
Bluespec SystemVerilog

Intel, IBM, Texas Instruments, Analog Devices, STMicroelectronics, Nokia, Qualcomm, Denali Software, Mercury Computer Systems
The Rest Of This Presentation...

- Bluespec SystemVerilog
- Strategies for embedding Bluespec in PVS
  - First, a simple, intuitive strategy which can be efficiently model checked, but has several drawbacks (`primitive' embedding)
  - A monadic embedding: a more sophisticated strategy, which allows efficient model checking, but avoids the problems associated with `primitive' embedding
- Experimental results: verifying a Bluespec arbiter
Bluespec SystemVerilog

- Creates hardware that's competitive with hand-written RTL in terms of time and area for many applications
- A formally inspired Hardware Description Language:
  - Based on the guarded action model of concurrency
    - Similar to model checking languages such as SAL, Promela, model checkable subset of the PVS language
Bluespec SystemVerilog

• Hardware specified with modules, which associate elements of state with:
  – **Methods**: functions that return values from the state and/or transform it
  – **Rules**: guarded actions that spontaneously change the state
Rules in Bluespec

rule my_rule (rl_guard);
    statement_1;
    statement_2;
    ...
endrule
The Semantics of a Bluespec Module

• Behaviour of a module can be understood with a simple semantics 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
The Semantics of a Bluespec Module

• 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:

```plaintext
rule request_rl (!request._read && !acknowledge._read));
  request._write(True);
endrule
```
The Model Checkable Subset of the PVS Language

• A guarded action language
  – Similar to Bluespec, but simpler
  – We define a state machine with:
    • A state, defined inductively from boolean and scalar types, using tuples, records and arrays
    • A transition relation, defined as a binary relation over pairs of states
  – No equivalent to the `module construct'
The State of Module `Arbiter'

Reg : TYPE = [# val: T #]

Arbiter: TYPE = [# req1, req2, req3, ack1, ack2, ack3, tok1, tok2, tok3 : Reg [bool] #]
A Rule from Module `Arbiter'

```plaintext
rule ack1_with_tok (tok1._read && req1._read
    && !(ack1._read || ack2._read || ack3._read));

    ack1._write (True);
move_token;
endrule
```
A Method from Module `Arbiter'

Action move_token =
(action tok1._write(tok3._read);
  tok2._write(tok1._read);
  tok3._write(tok2._read);
endaction);
A Rule in PVS

ack1_with_tok_primitive (pre, post: Arbiter): bool =
pre\textquoteleft tok1\textquoteleft val \land pre\textquoteleft req1\textquoteleft val \land \neg (pre\textquoteleft ack1\textquoteleft val \lor pre\textquoteleft ack2\textquoteleft val \lor pre\textquoteleft ack3\textquoteleft val)
\land post = pre WITH [ (ack1) : = (# val : = TRUE #),
(tok1) : = (# val : = pre\textquoteleft tok3\textquoteleft val #),
(tok2) : = (# val : = pre\textquoteleft tok1\textquoteleft val #),
(tok3) : = (# val : = pre\textquoteleft tok2\textquoteleft val #) ]

move_token
rule ack1_with_tok (token1._read && req1._read 
    && !(ack1._read || ack2._read || ack3._read));
    ack1._write (True);
    move_token;
endrule
A Monadic Embedding in PVS

\[
\text{ack1\_with\_tok = rule (tok1\'read} \land \text{ req1\'read} \\
\land \neg (\text{ack1\'read} \lor \text{ack2\'read} \lor \text{ack3\'read}) \\
(\text{ack1\'write (TRUE) >> move\_token})
\]
A Monadic Embedding in PVS

```
rule ack1_with_tok (tok1._read && req1._read
    && !(ack1._read || ack2._read || ack3._read));
    ack1._write (True);
move_token;
endrule

ack1_with_tok = rule (tok1‘read \∧ req1‘read
    \∧ \neg (ack1‘read \lor ack2‘read \lor ack3‘read))
(ack1‘write (TRUE) >>
move_token)
```
move_token =
  tok1‘read >>= tok2‘write >>
  tok2‘read >>= tok3‘write >>
  tok3‘read >>= tok1‘write
A Monadic Embedding in PVS

Action move_token =
    (action tok1._write(tok3._read);
     tok2._write(tok1._read);
     tok3._write(tok2._read);
    endaction);

move_token =
    tok1\textquoteleft read >>= tok2\textquoteleft write >>
    tok2\textquoteleft read >>= tok3\textquoteleft write >>
    tok3\textquoteleft read >>= tok1\textquoteleft write
rule ack1_with_tok (token1._read && req1._read
    && !(ack1._read || ack2._read || ack3._read));
ack1._write (True);
move_token;
endrule

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

ack1_with_tok_primitive (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 #) ]
Experimental Results: Fair Arbiter

- Verified a 3 input fair arbiter
- 100 lines of Bluespec code (extracts provided in paper)
- Hand embedded Bluespec code in PVS
- Verified with the PVS model checker and proof strategies
- Verified deadlock freedom, mutual exclusion, liveness
Conclusion

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

• We are using PVS to experiment with proof strategies for Bluespec

• Today, I presented a strategy for embedding a subset of Bluespec in the PVS theorem prover