Correct Safety Critical Hardware Descriptions via Static Analysis and Theorem Proving

Nicholas Moore and Mark Lawford

May 27, 2017

FormaliSE: FME Workshop on Formal Methods in Software Engineering
Motivation

The Problem

- Safety Critical control applications require mathematical proof of correctness
- FPGA processing technology increasingly used in PLCs
- Proofs for Von Neumann architectures not applicable to FPGA architectures
- New proofs and techniques are needed to verify next generation of Safety Critical PLCs

Our Solution

- Hardware Descriptions written in semantically elegant languages are amenable to automatic translation to theorem proving environments
  - HDL - Bluespec SystemVerilog (BSV)
  - Theorem Prover - Prototype Verification System (PVS)
- This substantially reduces effort required for formal verification
Overall Project Direction

BSV Input Files → BSV Parser → BSV Syntax Tree → BSV Generator → BSV Output

BSV2PVS

BSV Syntax Tree

PVS Syntax Tree

PVS2BSV

Restricted Subset of PVS

PVS Parser

PVS Generator → PVS Output
Hardware Considerations

Programmable Logic Controllers (PLCs)

- Reputation for reliability
- “Ladder Logic” programs
- Safety critical applications

Field Programmable Gate Arrays (FPGAs)

- Reprogrammable networks of logic blocks
- Alternative to Von Neumann architectures

Figure: LIMITS_ALARM PLC FB
Bluespec SystemVerilog (BSV)

- An abstract, semantically elegant Hardware Description Language (HDL)
- Compiles to Verilog
- Modules are composed of state declarations, rules, and methods
- BSV uses a guarded action semantic for register writes
- All register writes are concurrent (unless declared otherwise)
- Rules and methods are atomic, it’s all or nothing!
- The user may specify an order of precedence for rules, resolving ambiguous behaviour when rule guards are not mutually exclusive.
Prototype Verification System (PVS)

- Open source emacs plugin
  - Specification language based in higher order logic
  - Proof environment with high mechanicity and legibility
- When prompted, PVS derives proof obligations from specifications. The user then specifies proof tactics to discharge obligations
- Track record of safety critical embedded systems verification
  - AAMP5 avionics microprocessor
  - Darlington nuclear power plant emergency shutdown system
Logical Basis for Translation

- The underlying logical model of a BSV module is a Kripke Structure
  \[ K = (S, s_0, T, L) \]
- \( S \) is the set of all program states
- \( s_0 \in S \) is the initial state
- \( T \) is a left-total transition relation:
  \[ T \subseteq S \times S \]
- \( L \) is a labelling function:
  \[ L : S \rightarrow 2^{AP} \]
- where \( AP \) is the set of atomic propositions

An Example Kripke Structure
Timed vs Untimed Semantics

BSV has two semantic properties governing execution.

- **Untimed (or execution step) semantics**
  - Relates actions to execution steps
  - In instances of ambiguous action precedence, one is arbitrarily but deterministically chosen to fire.

- **Timed (or clock cycle) semantics**
  - Relates execution steps to clock cycles
  - Composes a set of execution steps which can execute in parallel.
Comparison with Previous Approaches

- Previous logical description first proposed by Richards and Lester (2011)
  - Stated aim was syntactic similarity between source BSV and product PVS
  - Addresses untimed semantic, but not timed semantic.
  - Transition predicates are universal disjunction of actions
- Current work originally automated R&L method, which was found insufficient for certain problems
- We attempt to faithfully duplicate BSV’s action arbitration
  - Timed semantic is addressed
  - We require ambiguous behaviours to be constrained at the source code level.
  - Consequently, we can relate behaviours to clock cycles.
Comparison with Previous Approaches ctd.

Previous Approach

Current Approach

Nicholas Moore and Mark Lawford
Correct Safety Critical Hardware Descriptions
Generating a State Theory

BSV

Reg#(Int#(16)) foo <- mkReg(5);
Reg#(Bool) bar <- mkReg(False);
Generating a State Theory

**BSV**

\[
\text{Reg\#(Int\#(16)) foo <- mkReg(5);}
\]

\[
\text{Reg\#(Bool) bar <- mkReg(False);} 
\]

**PVS**

\[
\text{MyModule : type =}
\]

\[
\text{[# foo: Int(16)}
\]

\[
\text{, bar: bool #]}
\]

\[
\text{MyModule\_var : var MyModule}
\]

\[
\text{mkModule (MyModule\_var) : bool}
\]

\[
\text{= MyModule\_var\`foo = 5}
\]

\[
\text{AND MyModule\_var\`bar = False}
\]
Generating a State Transition Theory

BSV

(* descending_urgency =
"auto_stop, inc" *)

rule auto_stop (foo == 5);
    bar <= false;
endrule

rule inc (bar);
    foo <= foo + 1;
endrule

method Action start() if (!bar);
    bar <= true;
    foo <= 0;
endmethod
Generating a State Transition Theory

**BSV**

(* descending_urgency = "auto_stop, inc" *)

```plaintext
rule auto_stop (foo == 5);
  bar <= false;
endrule

rule inc (bar);
  foo <= foo + 1;
endrule

method Action start() if (!bar);
  bar <= true;
  foo <= 0;
endmethod
```

**PVS**

```plaintext
pre, post : var MyModule

MyModule_t (pre, post) : bool =
  ( post = pre with
    [ foo := if
      (bar AND (NOT (foo == 5)))
        then pre'foo + 1
        else pre'foo
      endif
    , bar := if (foo == 5)
      then False
      else pre'bar
    endif
    ]
  )
```
### Generating a State Transition Theory

#### BSV

```hs
(* descending_urgency = "auto_stop, inc" *)

rule auto_stop (foo == 5);
  bar <= false;
endrule

rule inc (bar);
  foo <= foo + 1;
endrule

method Action start() if (!bar);
  bar <= true;
  foo <= 0;
endmethod
```

#### PVS

```hs
pre, post : var MyModule

MyModule_t (pre, post) : bool =
  ( post = pre with
    [ foo := if
      (bar AND (NOT (foo == 5)))
      then pre'foo + 1
      else pre'foo
      endif
    , bar := if (foo == 5)
      then False
      else pre'bar
      endif
    ]
  )

pre, post : var MyModule

MyModule_t_start
  (pre, post) : bool =
  ( post = pre with
    [ foo := if (NOT bar)
      then 0
      else if (bar AND (NOT (foo == 5)))
        then pre'foo + 1
        else pre'foo
        endif
      endif
    , bar := if (NOT bar)
      then True
      else if (foo == 5)
        then False
        else pre'bar
        endif
      endif
    ]
  )
```

Nicholas Moore and Mark Lawford

Correct Safety Critical Hardware Descriptions
BAPIP: the Bluespec and PVS Interlanguage Processor

Figure: BAPIP Architectural Overview
Limits Alarm: General Overview

(* High limit *) REAL --| H   QH -- BOOL (* High flag *)
(* Variable value *) REAL --| X   Q -- BOOL (* Alarm output *)
(* Lower limit *) REAL --| L   QL -- BOOL (* Low flag *)
(* Hysteresis *) REAL --| EPS

\[ X \]

Nicholas Moore and Mark Lawford
Correct Safety Critical Hardware Descriptions
Limits Alarm: PLC Implementation
Case Study: Limits Alarm Tabular Specifications

<table>
<thead>
<tr>
<th>Condition</th>
<th>QH</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X &gt; H$</td>
<td>True</td>
</tr>
<tr>
<td>$(H - EPS) \leq X \leq H$</td>
<td>NC</td>
</tr>
<tr>
<td>$X &lt; (H - EPS)$</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th>QL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X &lt; L$</td>
<td>True</td>
</tr>
<tr>
<td>$L \leq X \leq (L + EPS)$</td>
<td>NC</td>
</tr>
<tr>
<td>$X &gt; (L + EPS)$</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>$QL \lor QH$</td>
<td>True</td>
</tr>
<tr>
<td>$\neg (QL \lor QH)$</td>
<td>False</td>
</tr>
</tbody>
</table>

assuming $(EPS/2) > 0$

**Figure:** Limits Alarm Tabular Specification for $QH$, $QL$, and $Q$
Proving Limits Alarm

- Both functional and Consistency proofs required.
- Consistency proofs automatically generated with tactics
- Functional proofs dischargeable with general-purpose strategy (\texttt{grind}).

\begin{verbatim}
LIMITS_ALARM_Req : THEOREM
LIMITS_ALARM_t_set_Alarms (s(t), s(next(t)), x(t), h(t), l(t), eps(t))
and (eps(t)/2 > 0)
and f_q(qh,ql,q)(next(t))
and f ql(x,l,eps,ql)(t)
and f_qh(x,h,eps,qh)(t)
and q(t) = LIMITS_ALARM_get_q(s(t))
and ql(t) = s(t)'low_alarm'q
and qh(t) = s(t)'high_alarm'q
implies qh(next(t)) = LIMITS_ALARM_get_qh(s(next(t)))
and ql(next(t)) = LIMITS_ALARM_get ql(s(next(t)))
and q(next(t)) = LIMITS_ALARM_get q(s(next(t)))
\end{verbatim}
A Brief Survey of Related Works

▶ PLC verification efforts
  ▶ Alonso et. al. [2009]
  ▶ Economakos & Economakos [2012]
  ▶ Pang et. al. [2013, 2015, 2016]

▶ Methods for Translating PLCs and BSV into COQ
  ▶ Blech & Biha [2013]
  ▶ Braibant & Chlipala [2013]
  ▶ Vijayaraghavin et. al. [2015]

▶ Other BSV verification Methods
  ▶ Oliver [2006]
  ▶ Singh & Shukla [2008]
Future Work

Nicholas Moore and Mark Lawford
Correct Safety Critical Hardware Descriptions
Any Questions?