Verifying Parametrised Hardware Designs via Counter Automata

A. Smrčka[1] T. Vojnar[1]

[1]Faculty of Information Technology
Brno University of Technology

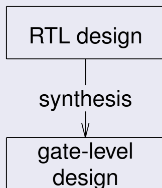
Haifa Verification Conference 2007, Haifa, IL

Outline

1. Motivation
2. Basics of RTL design and VHDL
3. Counter automata
4. Transformation VHDL $\longrightarrow$ counter automata
5. Bad state specification
6. Experiments + conclusions

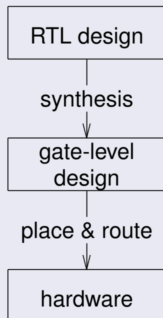
A Common Process of FV of HW Design

A generic RTL design (VHDL, Verilog, ...) transformed to a design using a given kind of gates.



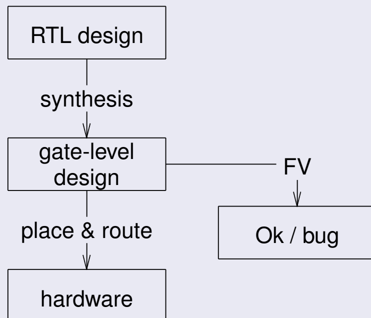
A Common Process of FV of HW Design

Transforming gate-level design into a hardware description.



A Common Process of FV of HW Design

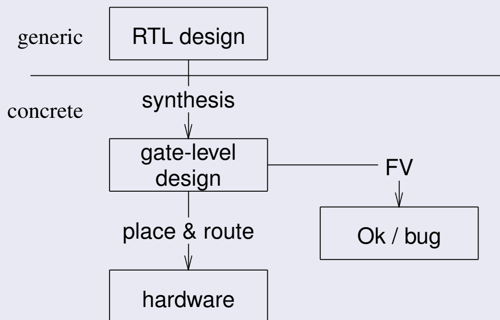
Given a specification and a gate-level design we can perform formal verification (MC on a finite state systems).



A Common Process of FV of HW Design

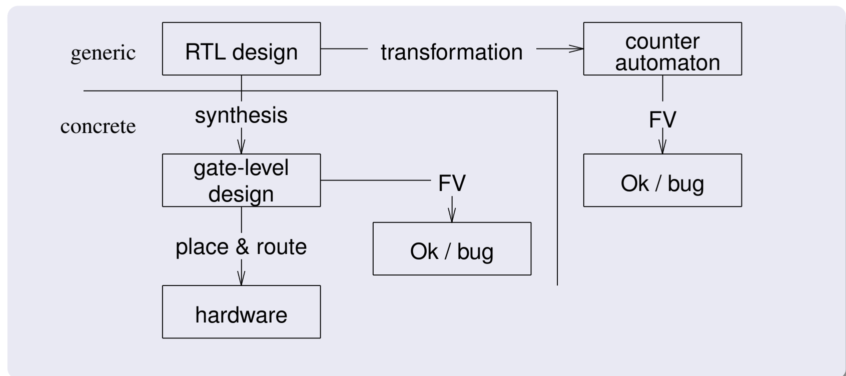
Problem with parametric/generic properties of design.

Motivation: How to verify parametrised hardware design



The Main Idea

Transform RTL design to a counter automaton, specify the set of bad states, model check the generated automaton.



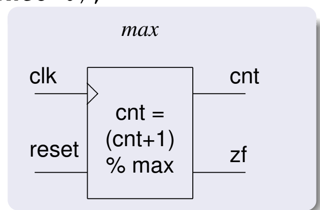
RTL Hardware Design

Register Transfer Level – RTL design provides a high-level description of a circuit behaviour by defining the data transfer between hardware registers and logical operations.

- Logic gate – implements one of logical operations $\neg, \wedge, \vee, \ldots$ (output depends only on the current input combination)
- Register (sequential gate) – a hardware element with memory (output depends on the current and the past inputs)
- Signal – a wire that transfers data among combinational and sequential gates
- Component (module, entity) – an encapsulation of digital circuit + I/O control

Hardware design is described in a modular way.

```
entity counter is
  generic ( max: integer );
  port (
    reset, clk: in std_logic;
    cnt: out std_logic_vector(32 downto 0);
    zf: out std_logic;
  );
end entity;
architecture cnt_arch of counter is
begin
  process (reset, clk)
  begin
    if (reset='1') then cnt <= 0;
    elsif (clk'event and clk='1') then
        if (cnt = max-1) then cnt <= 0;
        else cnt <= cnt+1;
        end if;
    end if;
  end process;
  comp: entity comp32 port map ( cnt, 0, zf );
end;
```

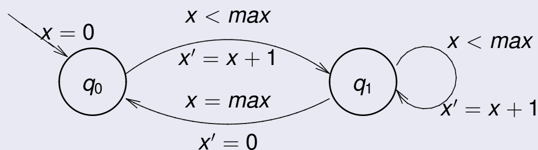


Counter Automata

A counter automaton $A = (X, Q, q_0, \varphi_0, \delta)$

- X – a finite set of variables (counters)
- Q – a finite set of control locations
- $q_0 \in Q$ – a designated initial location
- φ_0 – an arithmetic formula of initial assignment
- $\delta \subseteq Q \times \Phi(X) \times Q$ – a transition relation
  $\Phi(X)$ – a formula over X and X' (future references of counters)

Example: Increment *x* modulo *max* + zero test



Intermediate Behavioural Model

A direct mapping from VHDL source code to counter automaton is too complex $\longrightarrow$ introduce an intermediate behavioural model as the “bridge”.

Steps to produce a counter automaton:

1. Normalize the VHDL code of an RTL design
2. Refine an intermediate behavioural model
3. Build the counter automaton

RTL Design :: Normalization of VHDL Code

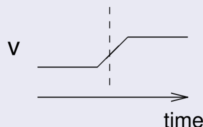
Goal: obtain code using registered signals (1-bit type, bit vectors), behavioural description, assignment statement + `if` stmt only

1. Variables of user defined structural types → more variables of simple types (single bit, bit vectors).
   Bit vectors (constant size) accessed bitwise → more 1-bit signals.
   Bitwise operations over parameter sized bit vectors → more complex arithmetic operations over bit vectors
2. Structural description → behavioural description (only process and assignment statements are allowed)
3. Non-registered signals → expressions over registered signals
4. Conditional assignments (`with`, `case`) → `if` statements

Intermediate Behavioural Model :: Definition

$M = (V, T, B)$, where:

- V – a set of variables, we use $\overline{V}$ as "references" to V:



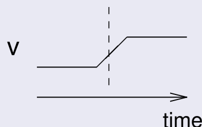
$v = 0$	current value of v
$v' = 1$	future value
$\uparrow v = 1$	positive edge $\uparrow v = \neg v \wedge v'$
$\downarrow v = 0$	negative edge $\downarrow v = v \wedge \neg v$

- $T : V \rightarrow \{\texttt{bool}, \texttt{int}\}$ – the type of a variable
- Let E be a set of expressions over $\overline{V}$ $(+, -, =, \neq, \geq, \neg, \wedge, \ldots)$,
- Let $C \subseteq E$ be the set of boolean valued expressions.
- $B \subseteq C^* \times V \times E$ is a set of behavioural rules representing conditioned assignments ($b \in B,\ b :\ c \rightarrow v := e$)

Intermediate Behavioural Model :: Definition

$M = (V, T, B)$, where:

- V – a set of variables, we use $\overline{V}$ as "references" to V:



$v = 0$	current value of v
$v' = 1$	future value
$\uparrow v = 1$	positive edge $\uparrow v = \neg v \wedge v'$
$\downarrow v = 0$	negative edge $\downarrow v = v \wedge \neg v$

- $T : V \rightarrow \{\texttt{bool}, \texttt{int}\}$ – the type of a variable
- Let E be a set of expressions over $\overline{V}$ $(+, -, =, \neq, \geq, \neg, \wedge, \ldots)$,
- Let $C \subseteq E$ be the set of boolean valued expressions.
- $B \subseteq C^* \times V \times E$ is a set of behavioural rules representing conditioned assignments ($b \in B$, $b : \; c \rightarrow v := e$)

$$
\begin{array}{rcl}
res' & \rightarrow & addr := 0 \\
\neg res', \uparrow clk, (addr \neq max - 1) & \rightarrow & addr := addr + 1 \\
\neg res', \uparrow clk, (addr = max - 1) & \rightarrow & addr := 0 \\
\neg res', \neg\uparrow clk & \rightarrow & addr := addr
\end{array}
$$

Model :: Extracting from Source Code (1/2)

$M = (V, T, B)$

- Variables V contain all registered signals + parameters
- $T(v) = \texttt{bool}$ if v is 1-bit signal,
  $T(v) = \texttt{int}$ if v is a bit vector or a parameter
- A VHDL expression: `(clk'event and clk = '1')`
  InterBM expression: $\uparrow clk \in E$

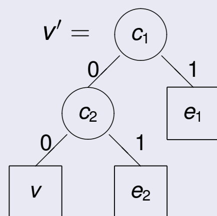
Model :: Extracting from Source Code (2/2)

Behavioural rules: $B \subseteq C^* \times V \times E$

- Process `if` conditional statements such that every `if` sets the value of only one state variable.
- From every `if` statement, create a tree of preconditions and expressions of the next values (missing branches represent no change of the variable value: $v' = v$).

Example

```
if c1 then
  v := e1;
else
  if c2 then
    v := e2;
  end if;
end if;
```



$c_1 \rightarrow v := e_1$

$\neg c_1, c_2 \rightarrow v := e_2$

$\neg c_1, \neg c_2 \rightarrow v := v$

Model :: Environment

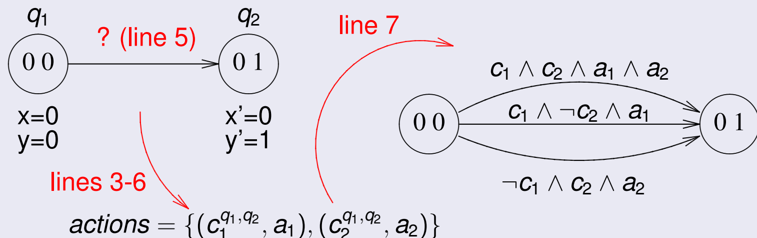
For model checking, we need to model the environment too. Environment = input signals of a component. Behaviour of the environment can be completely random representation of an input:
$\varepsilon \rightarrow v := random \in B$ for $v \in V$ representing input signal.

InterBM: $M = (V, T, B)$

Counter automaton: $A = (X, Q, q_0, \varphi_0, \delta)$

- the set of counters $X = \{v \mid v \in V, T(v) = \texttt{int}\}$
- the control locations $Q = \{0, 1\}^{V_q}$ - set of all possible evaluations of boolean variables $V_q = \{v \mid v \in V, T(v) = \texttt{bool}\}$.
- initial state q_0, φ_0: one must provide an additional input of the verification process (or can be guided by heuristics)
- the transition relation $\delta \subseteq Q \times \Phi(X) \times Q$: (cont.)

```
1   δ := {}
2   for q1, q2 in Q do
3         actions := {}
4         for (b : c → v := e) in B do
5               if c^{q1,q2} ≠ 0 then
6                     actions := actions ∪ {(c^{q1,q2}, v' = e^{q1,q2})}
7         δ := δ ∪ transitions(q1, actions, q2)
```



Formal Verification :: Set of Bad States

Only safety properties are taken into account $\rightarrow$ specification of bad state(s):

1. One defines the propositional formula over component signals defining the bad state – e_{bad}
2. A control location $q_{bad} \in Q$ representing the bad states is created
3. From every control location, create a transition to q_{bad} if there is possible true evaluation of e_{bad}

```
for q in Q do
    if e^q_bad ≠ 0 then
        δ := δ ∪ {(q, e^q_bad, q_bad)}
```

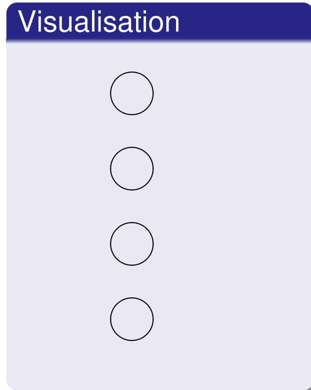
Counter automaton $A = \langle X, Q, q_0, \varphi_0, \delta \rangle$

1. Divide variables to 1-bit variables and multiple-bit ones (X)
2. Extract the behaviour from the VHDL source code
   1. Create the set of control locations (Q)
   2. Create the transition relation (δ)
3. Specification of the initial state and the initial evaluation
   - Evaluation of 1-bit variables represents the initial location (q_0)
4. Create the bad state and the transitions to that state

Transformation :: Summary

Counter automaton $A = \langle X, Q, q_0, \varphi_0, \delta \rangle$

1. Divide variables to 1-bit variables and multiple-bit ones (X)
2. Extract the behaviour from the VHDL source code
   1. Create the set of control locations (Q)
   2. Create the transition relation (δ)
3. Specification of the initial state and the initial evaluation
   - Evaluation of 1-bit variables represents the initial location (q_0)
4. Create the bad state and the transitions to that state

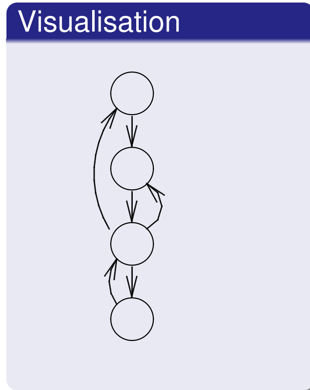


Transformation :: Summary

Counter automaton $A = \langle X, Q, q_0, \varphi_0, \delta \rangle$

1. Divide variables to 1-bit variables and multiple-bit ones (X)
2. Extract the behaviour from the VHDL source code
   1. Create the set of control locations (Q)
   2. Create the transition relation (δ)
3. Specification of the initial state and the initial evaluation
   - Evaluation of 1-bit variables represents the initial location (q_0)
4. Create the bad state and the transitions to that state

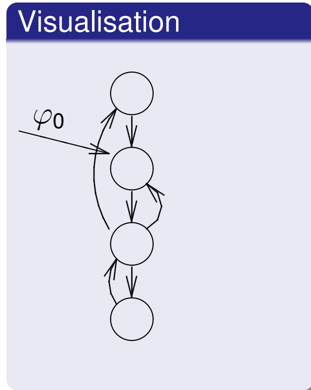
Visualisation



Counter automaton $A = \langle X, Q, q_0, \varphi_0, \delta \rangle$

1. Divide variables to 1-bit variables and multiple-bit ones (X)
2. Extract the behaviour from the VHDL source code
   1. Create the set of control locations (Q)
   2. Create the transition relation (δ)
3. Specification of the initial state and the initial evaluation
   - Evaluation of 1-bit variables represents the initial location (q_0)
4. Create the bad state and the transitions to that state

Visualisation

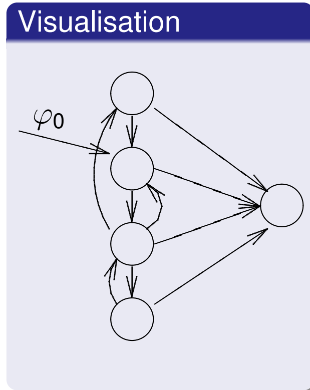


Transformation :: Summary

Counter automaton $A = \langle X, Q, q_0, \varphi_0, \delta \rangle$

1. Divide variables to 1-bit variables and multiple-bit ones (X)
2. Extract the behaviour from the VHDL source code
   1. Create the set of control locations (Q)
   2. Create the transition relation (δ)
3. Specification of the initial state and the initial evaluation
   - Evaluation of 1-bit variables represents the initial location (q_0)
4. Create the bad state and the transitions to that state

Visualisation



Formal Verification :: Experiments

Pentium4 2.8Ghz, 512MB, Python interpreter for translator.

Component	$\|Q\|$	$\|\delta\|$	$\|X\|$	Trans.	ARMC [1]
Counter	5	13	1	$<$ 1s	$<$ 1s
Register	9	45	1	1s	$<$ 1s
SynLIFO	65	985	1	1m13s	2.7s
AsFIFO (Status)	65	5484	11	3m51s	26m58s
AsFIFO (FE)	65	5075	11	3m31s	1h17m

Safety properties like:
$bad = \neg RESET \wedge CLK \wedge EN \wedge (OUT \neq DATA)$

[1] A. Podelski, A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. PADL 2007.

Conclusions

- We have proposed the method of FV of parametrised HW components through the counter automata.
- Future work/ideas:
  - Reduction of a size of generated automaton
  - Allow bit-wise operations on integers (often used in HW design)
  - Experiments with more components and more tools

Model :: Appendix A

- Environment: In the case of $T(v) = \texttt{bool}$: $\begin{array}{l} \varepsilon \rightarrow v := v \\ \varepsilon \rightarrow v := \neg v \end{array}$
- For a simpler construction of CA from InterBM, if $T(v) = \texttt{bool}$:
  $$c \rightarrow v := e \qquad \Longrightarrow \qquad c,\ v' = e \rightarrow v := e$$
- e_{bad}: using references to current variable values (with no v', $\uparrow v$, or $\downarrow v$ reference)

Example of Behavioural rules

$$\begin{array}{rcl} res' & \rightarrow & addr := 0 \\ \neg res', \uparrow clk, (addr \neq max - 1) & \rightarrow & addr := addr + 1 \\ \neg res', \uparrow clk, (addr = max - 1) & \rightarrow & addr := 0 \\ \neg res', \neg\uparrow clk & \rightarrow & addr := addr \end{array}$$

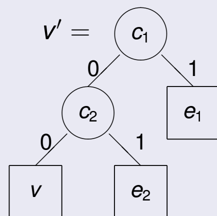
Model :: Appendix B

Behavioural rules: $B \subseteq C^* \times V \times E$

1. Transparent $\times$ synchronous mode: For the path from root to leaf: If there is no $\uparrow v$ (transparent mode): all variable references point to their future values. If there is $\uparrow v$ (synchronous mode): all variable references in expressions of subsequent nodes point to their current values
2. Transform such a tree to the set of behavioural rules

Example

```
if c1 then
  v := e1;
else
  if c2 then
    v := e2;
  end if;
end if;
```



$c_1 \rightarrow v := e_1$

$\neg c_1, c_2 \rightarrow v := e_2$

$\neg c_1, \neg c_2 \rightarrow v := v$