An Equivalence Checker for Hardware-Dependent Software

C. Villarraga, B. Schmidt, C. Bartsch, J. Bormann, D. Stoffel, W. Kunz
Agenda

- Motivation
- Related work
- Program Netlist (PN)
- Memory Model
- Interface Model
- SW Miter
- Experiments
- Conclusions
HW verification methods are widely applied in industry

- Equivalence/Property Checking

Need for verification methods also for HW-dependent SW

- Particularly in safety critical systems
Hardware-dependent Software (HWDS):

- HWDS is a critical component in embedded systems
- Error prone, e.g., HW/SW interface exhibits complex interactions
- Hard to test

Examples: drivers, firmware, SW-implemented buses, boot code, test code
Equivalence Checking of HWDS

- Notion of equivalence:
  - I/O functionality of two different programs is shown to be equal
  - Interaction sequences between SW and HW must also be compared (reactive behavior)

- Applications scenarios:
  - Optimizations: size, speed, power, etc.
  - Code porting
  - Backwards compatibility
Related Work

- Current approaches for equivalence checking of low-level SW are based on symbolic execution
  - Specialized verification algorithms need to be developed for exploring individual execution paths

- Communication takes place at specific points (as in transformational software)
  - [Currie et al., DAC00]. Programs should have same CFG structures
  - [Arons et al., CAV05]. Communication can take only place at end points

- Our approach: Communication can take place at arbitrary points (as in reactive software)
  - No specialized algorithms need to be developed
Naïve HW-dependent approach

- Bounded Model Checking (BMC) approach
- Accurate level of abstraction
- Implicit representation of control flow and computation
- Complexity problem:
  - Size of the model
  - Verification runtime

**HW-style BMC unrolling**

1010110101
1011011110
1011101010
1111001111
- Unrolling: use information provided by the control flow graph (CFG)
- Include new control logic to make execution paths visible
- This produces a program netlist [see ASP-DAC 2013]
  - Control flow is represented explicitly: facilitates SAT reasoning
  - Program computation is represented implicitly: compact model
Abstract model for a CPU instruction

Combinational model:
- Can be formally verified against RTL description of the CPU

Models the modification of the Program State (PS)
PS includes signals for:
- CPU registers
- Program variables
- **Active Bit**

**Active Bit:**
- Set to “true” if the instruction belongs to an execution path which is active
Program Netlist Generation

CFG

EXG

 Execution Graph (EXG): DAG containing all possible execution paths

 Path pruning:
  • SAT check on each branch

 Merging: join successors if
  • They correspond to the same instruction
  • No loops are introduced
Replace every EXG node by its corresponding instruction cell

Insert additional control structures

- branch logic
- merge cells

PN is combinational model
- Represents compactly all execution paths
- Compositional
Memory Model for the PN

- Avoid representing memory logic for all possible data memory locations

- An approach that scales much better:
  - Build logic only for memory locations reachable by the instruction cell

- Issue:
  - Find for every instruction cell in the PN the memory addresses it can access

**Instruction Cell reading from memory**

Program State ($PS$) → $maddr_k$ → $addr_1$, $addr_m$ → $mdata_k$ → Next Program State ($PS'$)
Find all possible values for each signal: $maddr_k$

For safety-critical embedded software this can be simplified, because:

- Number of solutions are restricted by design
- Static memory allocation
- Most addressing results in a single constant value

Use a mixed simulation/formal approach:

- Ternary simulation for constant cases
- All-SAT enumeration for remaining cases

$$\varphi = \bigvee_{addr_i} (maddr_k = addr_i)$$
I/O Model for the PN

Equivalence is formulated in terms of I/O sequences of the programs

- Exploit path-oriented representation of the PN

Let $LOC$ be the set of environment locations (memory addresses, dedicated ports) accessed by a given program

$LOC = \{loc_1, loc_2, \ldots, loc_m\}$
I/O Model for the PN (2)

- $I^{loc_j}, O^{loc_i}$ denote access sequences for input and output locations

- For each input location (load access):
  - $loc_j \in LOC: I^{loc_j} = (i^{loc_j,1}, i^{loc_j,2}, \ldots, i^{loc_j,lj})$

- For each output location (store access):
  - $loc_i \in LOC: O^{loc_i} = (o^{loc_i,1}, o^{loc_i,2}, \ldots, o^{loc_i,li})$

- Every access point $o^{loc_i,k}, i^{loc_j,k}$ is characterized by the global signals:
  - *data* and *active*
Instruction cells access the interface through specific signals (here called **ports**).

A port includes the signals:

- \( p_{data} \): data being written/read
- \( p_{active} \): inst. cell’s active signal
- \( p_{loc} \): location being accessed
I/O Model for the PN (4)

- Additional logic is required to model individual contribution to sequence points
- For each location (loc) a new time point variable (loc.t) is propagated through the program netlist
- Instruction cells implement new logic based on incrementers

Example:

\[
\text{loc}_{k}, t = 0 \quad \text{port}_o
\]

\[
\text{loc}_{k}, t = 0 \quad \text{port}_o
\]

\[
\text{loc}_{k}, t = 0 \quad \text{port}_o
\]
Global model for I/O access sequences

Data ($o_{loc,i}.data$)

$$\bigwedge_{k=1}^{m} (pactive_k) \land (ploc_k \leftrightarrow loc) \land (lock.t \leftrightarrow i) \rightarrow (o_{loc,i}.data = pdata_k)$$

Active ($o_{loc,i}.active$):

$$o_{loc,i}.active = \bigvee_{k=1}^{m} (pactive_k) \land (ploc_k \leftrightarrow loc) \land (lock.t \leftrightarrow i)$$

This global model encapsulates the interface of the program with its environment as a set of Boolean formulas.
User provides a bijective mapping of environment locations: $o_{locG} \leftrightarrow o_{locR}$

Comparison functions express equivalence between mapped sequence points

Two conditions for equivalence: control (active signals) and data

\[
equiv(o^{o_{locG}}, o^{o_{locR}}, k) = (o^{o_{locG}, k}.active \leftrightarrow o^{o_{locR}, k}.active) \\
\land (o^{o_{locG}, k}.data \leftrightarrow o^{o_{locR}, k}.data)
\]
Exploit compositionality of PN

Use an incremental SAT solver

Logic for proving $\text{equiv}(o^{\text{loc}G}, o^{\text{loc}R}, k + 1)$ contains the logic of $\text{equiv}(o^{\text{loc}G}, o^{\text{loc}R}, k)$
Experiments (1)

Case 1: LIN Driver
- 1350 lines of HW-dependent C code (Infineon Automotive)
- Transformations:
  - Compiler optimizations: Gcc levels l0, l1 and l2
  - Manual refactoring

Case 2: Serial interface
- Original implementation for the SuperH2 architecture
- Ported to an ARM7 processor

<table>
<thead>
<tr>
<th>Program</th>
<th>Model generation time (s)</th>
<th>Instruction Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sat</td>
<td>Sim/Sat(*)</td>
</tr>
<tr>
<td>LIN (l0)</td>
<td>6828</td>
<td>27</td>
</tr>
<tr>
<td>LIN (l1)</td>
<td>1087</td>
<td>12</td>
</tr>
<tr>
<td>LIN (l2)</td>
<td>1016</td>
<td>11</td>
</tr>
<tr>
<td>LIN (modif.)</td>
<td>1323</td>
<td>13</td>
</tr>
<tr>
<td>LIN (error)</td>
<td>1298</td>
<td>13</td>
</tr>
<tr>
<td>SER (orig.)</td>
<td>787</td>
<td>698</td>
</tr>
<tr>
<td>SER (ported)</td>
<td>723</td>
<td>645</td>
</tr>
</tbody>
</table>

(*) percentage of cases solved by ternary simulation: 87.3%
Global I/O model:

<table>
<thead>
<tr>
<th>Program</th>
<th>No. of locations</th>
<th>Sequence points</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input</td>
<td>Output</td>
</tr>
<tr>
<td>LIN</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>SER</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Equivalence proofs:

<table>
<thead>
<tr>
<th>Program</th>
<th>Miter size (no. of Inst. Cells)</th>
<th>Proof time (s.)</th>
<th>Memory Usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIN (l0) LIN (l1)</td>
<td>13764</td>
<td>692.3</td>
<td>777.5</td>
</tr>
<tr>
<td>LIN (l0) LIN (l2)</td>
<td>13198</td>
<td>766.2</td>
<td>698.1</td>
</tr>
<tr>
<td>LIN (l1) LIN (l2)</td>
<td>11520</td>
<td>419.5</td>
<td>343.0</td>
</tr>
<tr>
<td>LIN (l1) LIN (modif.)</td>
<td>11904</td>
<td>500.5</td>
<td>470.5</td>
</tr>
<tr>
<td>LIN (l1) LIN (error)</td>
<td>11915</td>
<td>295.1</td>
<td>336.2</td>
</tr>
<tr>
<td>SER (orig.) SER (ported)</td>
<td>11760</td>
<td>188.2</td>
<td>404.6</td>
</tr>
</tbody>
</table>

Incremental SAT strongly reduces verification runtimes: by 64% (LIN) and by 73% (SER)
Conclusions

- SW miter is an efficient computational model to perform equivalence checking of HW-dependent SW
  - Facilitates formulation of equivalence checking for reactive HW-dependent SW
  - Highly automated verification flow

- Explicit representation of control flow is combined with implicit representation of program’s computation
  - All intelligence is in the model and SAT solver: no need for specialized verification algorithms

- Successful experiments on industrial software
Questions?
Largely automated flow

- **Machine Code** → **Control Flow Graph** → **Execution Graph** → **Program Netlist** → **Formal Verification: SAT engines**