A Computational Model for SAT-based Verification of Hardware-Dependent Low-Level Embedded System Software

Bernard Schmidt, Carlos Villarraga, Jörg Bormann, Dominik Stoffel, Markus Wedler, Wolfgang Kunz

Yokohama, 1/25/2013
Motivation

Related Works

Model Generation
  - Basis: Abstract HW/SW Model
  - Flow

Advantages of Model

Experiment

Conclusion / Future Work
Motivation

- Embedded System
  - Close interaction between HW and SW
  - Examples: drivers, communication structures

- Goal
  - Formal verification (FV) of combined HW/SW behavior

- Objective of this work
  - Computational model and algorithms for FV of hardware-dependent, low-level software
Related Works


- Large research body on SW verification
- Our focus: New computational model for hardware-dependent SW
State of the Art: Bounded Model Checking (BMC)

E.g.,


Basic approach:

Unroll CPU + software in memory

- Program flow (instruction sequencing) as given by the program is represented only *implicitly* (by PC + memory)
- Program computation also is represented only *implicitly* (by hardware implementation of CPU)
- High computational complexity for realistic designs
State of the Art: Symbolic Execution

E.g.,

Basic approach

- Enumerate program execution paths and check conditions for verification along these paths by specialized verification algorithms
- Symbolic formulas explicitly represent all possible input scenarios (along path)

⇒ program flow is enumerated *explicitly* (by path enumeration)
⇒ program computation is represented *explicitly* (through symbolic formulas)
⇒ high complexity of symbolic formulas
State of the Art: CBMC and related work

E.g.,

Basic approach
- Build a SAT formula for the program computation based on CFG unrolling
- SAT formula contains information about control flow only implicitly in terms of functional dependencies between variables of the formula
- *HW-independent* paradigm (high-level programming language)
State of the Art: CBMC and related work

E.g.,

⇒ Extension of CBMC for HW-dependent software is difficult!

The computational model (SAT formula):
- lacks information about temporal relationships between statements/instructions as is needed when relating program behavior to HW periphery
- cannot be integrated with environment hardware model
- is not compositional
Our Approach

New **hardware-dependent** computational model

- Program **computation** should be represented **implicitly**
  - \( \Rightarrow \) compact model

- Program **flow** should be represented **explicitly**
  - \( \Rightarrow \) facilitates SAT reasoning (since global information about global execution paths is explicitly available)

- Representation of the temporal dependencies between instructions and I/O
  - \( \Rightarrow \) enables HW-dependent verification
  - \( \Rightarrow \) enables compositionality of model
Basis: Abstract HW/SW Model
Basis: Abstract HW/SW Model

- Instruction Logic:

```
ADD  ROT  MUL  MOV  LOAD  BEQ
...
PS'  PS
a
w
```

Diagram:

- Input Ports: (PC), (icode), w
- Output Ports: D, J
- PS
- PS'
- Instruction Logic:
  - ADD
  - ROT
  - MUL
  - MOV
  - LOAD
  - BEQ
Instruction Cell

- Abstract model for a CPU instruction
- Hardware-dependent
- Describes the modification of the program state
- Can be formally verified against RTL implementation of CPU
Program State

- Registers
- Control signals in the CPU
- Program data in RAM
- “Active” signal (program flow variable)
**Active Signal**

- Additional 1-bit signal in program state
- Signal is asserted if corresponding program state belongs to active execution path
- Additional logic in instruction cells to handle active signal:
  - Datapath and load/store instructions: propagate *active* flag from input state to output state (no extra logic required)
  - Branch instructions: propagate *active* flag depending on branch condition from input state to branch target output state. All other output states are set inactive.
Execution graph:

- Models all possible execution paths
- Optimized by pruning and merging
- Not unique
Verifying if a branch is active

- **Branch instruction**
  - Propagates “active” signal to exactly one of the branches

- **Generated property:**
  - **Claim:** “active” signal is *never* asserted
  - **Property holds:** branch never taken
  - **Property does not hold:** branch is taken in some executions
Program Netlist (PN)

- **PN generation**
  - Replace every EXG node by an instruction cell
  - HW-dependent description of program computation
  - Insert merge cells

- **Merge cell**
  - Merges two paths into one
  - Path is selected by incoming *active* signal
Advantages of PN Model

Replacing EXG nodes by instruction cells

- PN is a combinational circuit
  - Supports SAT-based FV

- CFG-based unrolling
  - Explicitly represented program flow

- Concatenation of instruction cells
  - Implicitly represented program computation
Advantages of PN Model (2)

Explicit program flow:
program paths are considered separately

⇒ Simplification of program computation
  - CPU logic is simplified to single instruction cell
  - Machine instruction parameters are constant: individual instruction cell is further simplified by Boolean constraint propagation
  - Simplification takes place during PN generation (not during property verification)
Advantages of PN Model (3)

- **Active flags:**
  - Small set of signals captures global reachability information of entire program
  - Valuation on active signals selects possible execution paths or sets of execution paths
  - Helps SAT solver to take into account reachability without detailed reasoning on all intermediate instructions

- Example:

```
Property:
A: \[ \text{true} \]
C: if b.active then b=a
```
Experiment – Software LIN Master (1)

- Industrial automotive design
- 1309 lines of C code
- Send/receive message frames via UART
- Interrupt-driven
- Compiled for SuperH2 based 32-bit CPU, RISC architecture, 5-stage pipeline
Model generation:

- Interrupt Service Routine (ISR), main program (scheduler) and initialization generated automatically
- Composed model: assembled manually

<table>
<thead>
<tr>
<th>Program component</th>
<th>#instructions</th>
<th>CPU (s)</th>
<th>mem. (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIN-Init</td>
<td>225</td>
<td>1.32</td>
<td>36</td>
</tr>
<tr>
<td>LIN-Scheduler</td>
<td>85</td>
<td>0.13</td>
<td>27</td>
</tr>
<tr>
<td>LIN-ISR</td>
<td>790</td>
<td>11.00</td>
<td>102</td>
</tr>
</tbody>
</table>
Property verification:

- Composed model: 24001 instructions, 62592 primary inputs
- Verify correct generation of LIN frame (header, data, checksum and control signal for UART)
- Commercial HW property checker (OneSpin 360MV)

<table>
<thead>
<tr>
<th>Property</th>
<th>CPU(s)</th>
<th>MEM(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX frame 4or8 data bytes incl. checksum</td>
<td>17</td>
<td>1641</td>
</tr>
<tr>
<td>RX frame 4or8 data bytes wrong checksum</td>
<td>28</td>
<td>1545</td>
</tr>
<tr>
<td>TX frame 4or8 data bytes incl. Checksum</td>
<td>15</td>
<td>1584</td>
</tr>
<tr>
<td>Wrong PID or not matching ID (8bytes)</td>
<td>14</td>
<td>1566</td>
</tr>
</tbody>
</table>
Conclusion

- Program netlist (PN)
  - New hardware-dependent model for low-level software verification combining the advantages of HW-style bounded model checking and SW-style symbolic execution
  - Automatic PN generation successful for industrial automotive software

- Future Work
  - Equivalence checking
  - Integrate PN with hardware for FV of firmware
Thank you!

- Questions?

bschmidt@eit.uni-kl.de