Safety across the HW/SW interface – can Formal Methods meet the challenge?

**Speaker:** Wolfgang Kunz

**Joint work with:** Christian Bartsch, Carlos Villarraga, Dominik Stoffel
Embedded Systems take over more and more tasks in safety-critical application domains.

**Functional Safety** is key concern in industry (ISO-26262, DO-178B/C)

- functional correctness + robustness w.r.t. HW faults
- testing techniques, on-chip test and diagnosis infrastructures for SoCs are becoming increasingly complex
- determining good trade-offs between effectiveness and costs requires an *application-dependent* approach

**HW/SW Cross-Layer Analysis**
Low Level Software

- does not change much during the system’s life time
- often safety-critical:
  - implements control and communication infrastructure
  - implements “safety functions”

(ISO 26262: Safety functions are specified traces of behavior by which the system must respond under certain inputs or in case of internal errors to ensure the overall safety)
Most previous work on formal SW verification operates at the level of source code.

Our approach: HW-dependent software model:
- works at binary level
- precisely describe behavior of a program in terms of its effect on the underlying hardware
- models reactive behavior with HW environment
- assesses criticality of HW faults at the firmware level
Abbreviations:

**PS**: program state (CPU registers, program variables in memory)

**a**: address in program counter

**D**: destination of a jump

**J**: jump
Unrolling the HW/SW Model

Program

Unrolled Model

Boolean Constraint Propagation (BCP) simplifies instruction logic

BCP does not simplify instruction logic
Unrolling with Conditional Jump

Shifted multiplexer of jump instr. here
Modelling Processor Behavior

**Instruction Cell**

- Abstract model of a CPU instruction
  - combinational circuit
  - created manually (once for given ISA)

- Models the modification of the Program State (PS)

- Instruction cell represents
  - Program-visible registers
  - Can be directly mapped to gate-level registers
SAT analysis is interleaved with unrolling
• Path pruning
• Path merging
• Unrolling loops

“Program Netlist”
• combinational circuit
• compactly represents all execution paths
Program Netlist (PN)

**Hardware-dependent software model**

- PNs include explicit information for a given program on:
  - all possible execution paths (unlike traditional symbolic execution)
  - the address spaces reached by every instruction
  - all possible input/output access sequences to peripheral hardware components and to shared memory
  - all possible effects of the program on the program-visible hardware registers

- **Applications:** property checking, equivalence checking...
Fault Injection

- Append fault behavior to instruction logic

- During model generation:
  - Insert faulty instruction cell instead of correct one into PN
Fault Injection

- Append fault behavior to instruction logic
- During model generation:
  - Insert faulty instruction cell instead of correct one into PN
Challenges

- Occurrence of a fault can change program behavior and thus the PN model
- How to avoid repeated PN generation in large fault lists?

Our approach

- Fault activation is configurable (single/multiple faults, permanent/temporary)
  - Modelling of a large fault list in a single PN
- SAT solver implicitly considers all possible single and multiple faults during unrolling

1. Duplicate generated PN

2. Compare outputs

- PN1:
  - Faults are deactivated
  - Fault1: @0xABCD = 0
  - Fault2: @0xABCE = 0

- PN2:
  - Faults are activated
  - Fault1: @0xABCD = 1
  - Fault2: @0xABCE = 0
Fault Analysis

Results:

1. Fault free and fault injected PN are equivalent
   - Fault(s) have no effect
   - Fault(s) are “application-redundant”
Fault Analysis

Results:

1. Fault free and fault injected PN are equivalent
   - Fault(s) have no effect
   - Fault(s) are "application-redundant"

2. Fault free and fault injected PN are not equivalent
   - Fault(s) have an effect

Weaker notions of "equivalence" model selected fault effects, e.g., effect only on data but not on control
Cross-Layer Fault Analysis

**PN level**

- **PN fault injection**
- **PN-based equivalence check**
  - analyze testability of faults w.r.t. all or selected SW components

**Gate level**

- **Combinational ATPG under architectural constraints**
  - check testability of HW faults anywhere in the combinational logic
Gate Level Faults

PN-level redundancies from the gate-level point of view

Inputs

States

Combinational Logic

Outputs

$F_C$ HW fault

$F_{AS}$ Application-redundant fault

Program-visible flipflops (Architectural State)

Additional RTL/gate-level flipflops
Identifying gate-level redundancies

1. Identify \textbf{sets of unobservable pseudo-outputs} at the gate level: all multiple bit flips in the corresponding PN-level state bits must be untestable

2. Run combinational ATPG at the gate level under the unobservability constraints: all faults identified as redundant in the gate-level circuit are actually "application-redundant"
Experimental Results

Test Programs

- Traffic Alert and Collision Avoidance System (TCAS)
  - Developed by Siemens

- SW-implemented interrupt-based driver for master node of automotive protocol LIN
  - Developed by Infineon

HW Platform

- Aquarius
  - 32-bit architecture
  - SuperH2 ISA
Computing platform Aquarius was synthesized to the gate level using Synopsys DesignCompiler.

### Design statistics for computing platform – gate level

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary inputs</td>
<td>51</td>
</tr>
<tr>
<td>Primary outputs</td>
<td>73</td>
</tr>
<tr>
<td>State bits</td>
<td>1217</td>
</tr>
<tr>
<td>Target stuck-at faults (single)</td>
<td>79184</td>
</tr>
<tr>
<td>Untestable stuck-at faults</td>
<td>3367</td>
</tr>
</tbody>
</table>
Experimental Results

- Considering stuck-at faults
- Unobservability constraints resulting from PN-level analysis were applied to gate level netlist
- Synopsys TetraMAX TM was used for running combinational ATPG on the synthesized design HW platform

<table>
<thead>
<tr>
<th></th>
<th>PN Level</th>
<th>Gate Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>testable faults</td>
<td>untestable faults</td>
</tr>
<tr>
<td>Program</td>
<td>CPU time [s]</td>
<td>model gen.</td>
</tr>
<tr>
<td>LIN</td>
<td>560</td>
<td>544</td>
</tr>
<tr>
<td>TCAS</td>
<td>208</td>
<td>896</td>
</tr>
</tbody>
</table>
Dependency Analysis

identify safety-critical code components and reason backwards to identify what faults are relevant

<table>
<thead>
<tr>
<th>PN-level dependency analysis – LIN driver</th>
</tr>
</thead>
<tbody>
<tr>
<td># nodes in dependency graph</td>
</tr>
<tr>
<td># syntactically critical state bits</td>
</tr>
<tr>
<td># semantically critical state bits</td>
</tr>
<tr>
<td># safety-function-redundant state bits</td>
</tr>
<tr>
<td># program-visible state bits in Aquarius</td>
</tr>
</tbody>
</table>
Conclusion

- **Formal analysis** on the effects of HW faults on SW
- Highly **configurable fault injection framework**
  - Single and multiple faults
  - Permanent and temporary faults
- Injected faults can be mapped to faults at the gate level
- A high percentage of gate-level faults turns out to be application redundant: valuable information in **safety analysis and certification**
Questions?

