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

Speaker: Wolfgang Kunz

Joint work with: Christian Bartsch, Carlos Villarraga, Dominik Stoffel

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

IWLS - 2017

Introduction

- 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

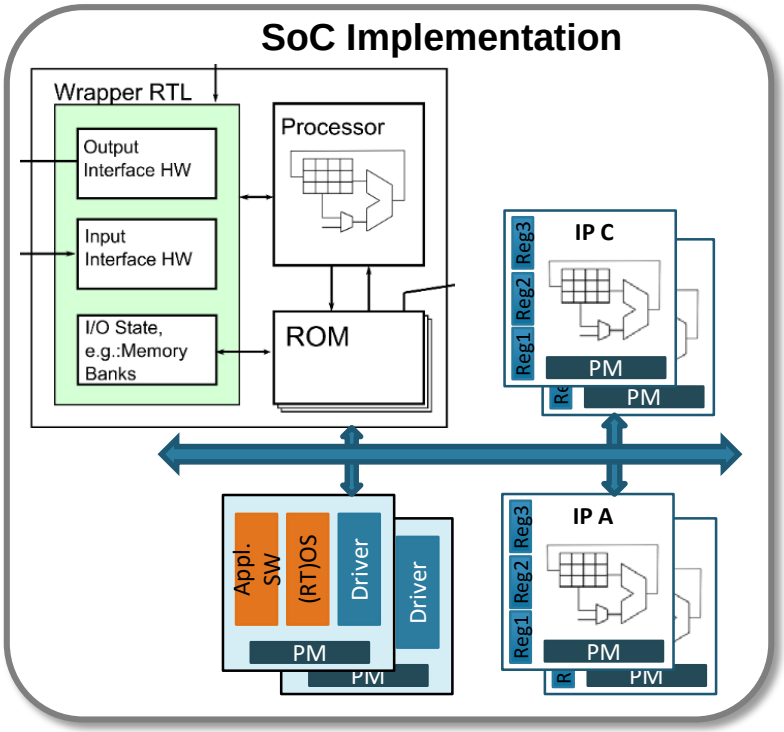
HW/SW Cross-Layer Analysis

Introduction

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)

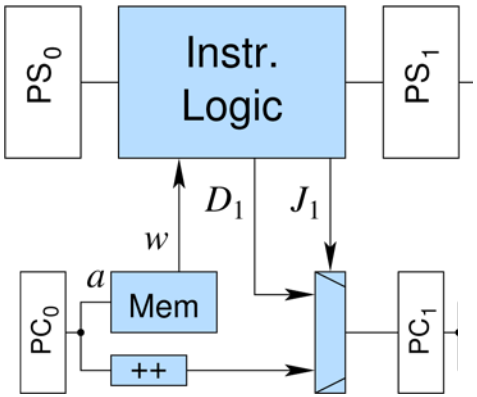


Modeling firmware

- 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

Unrolling the HW/SW Model



Abbreviations:

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

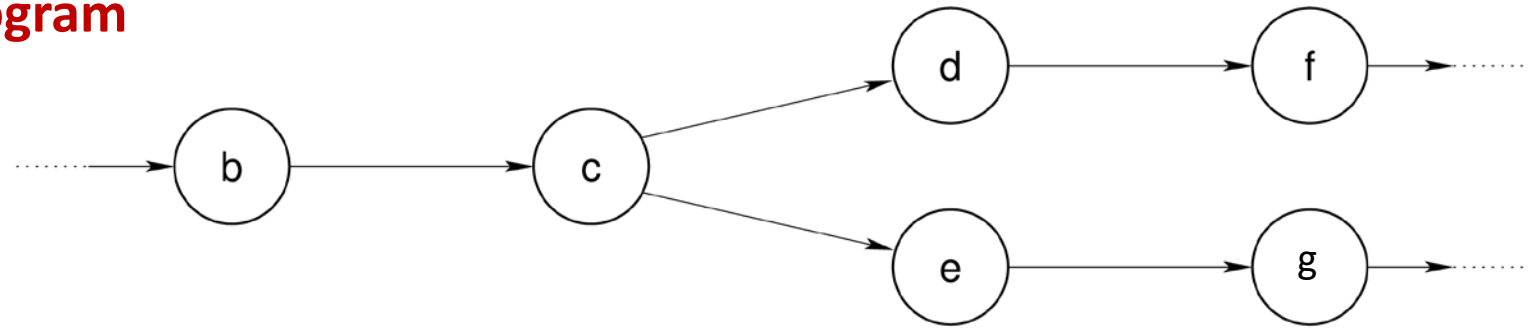
a: address in program counter

D: destination of a jump

J: jump

Unrolling the HW/SW Model

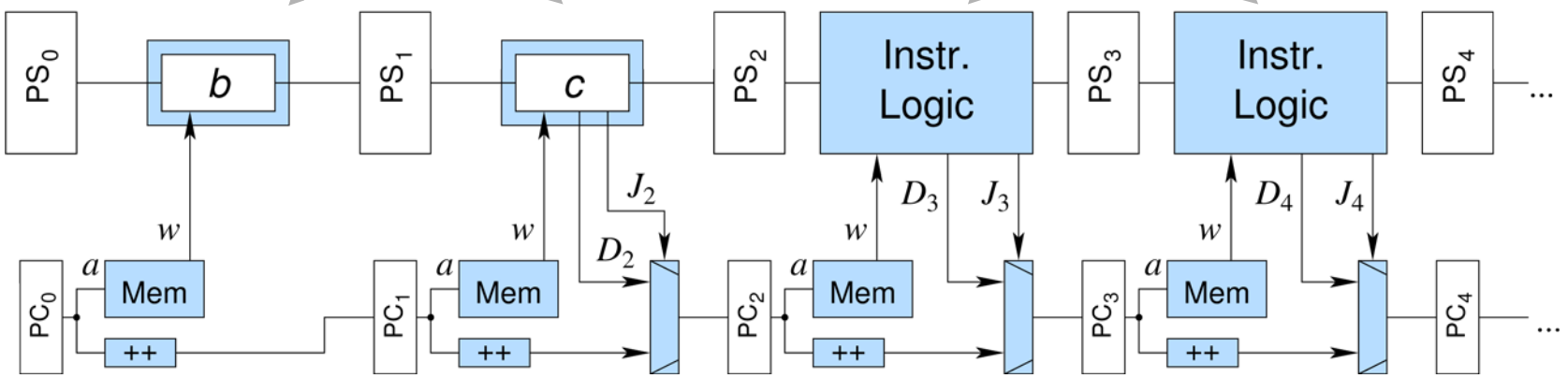
Program



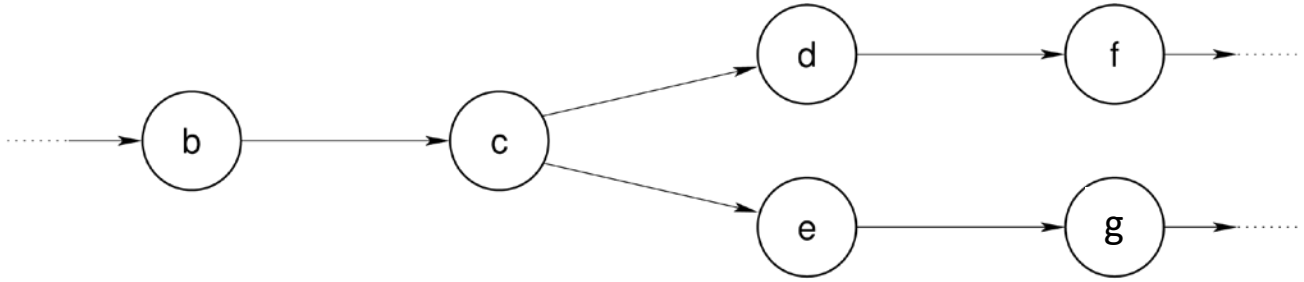
Unrolled Model

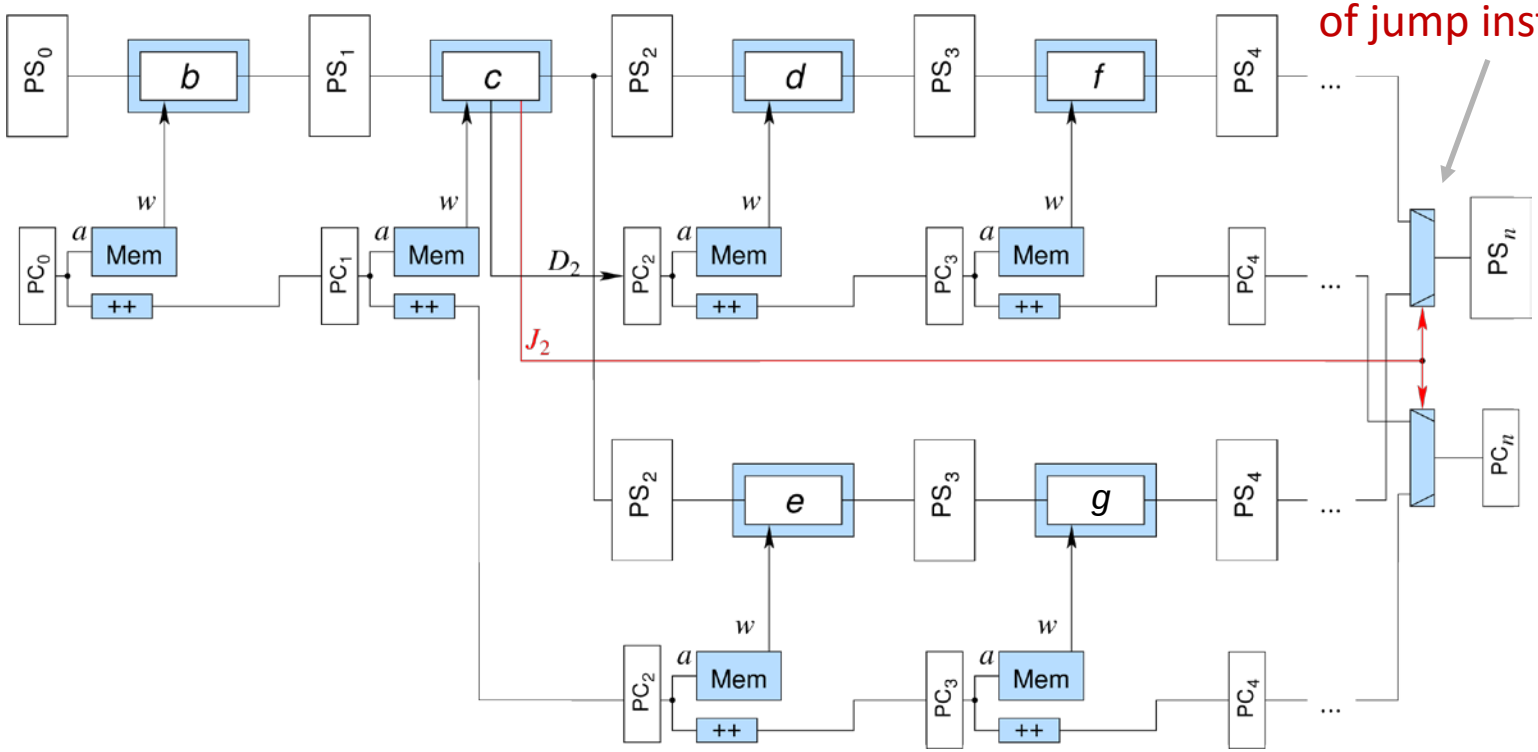
Boolean Constraint Propagation (BCP) simplifies instruction logic

BCP does not simplifies instruction logic



Unrolling with Conditional Jump

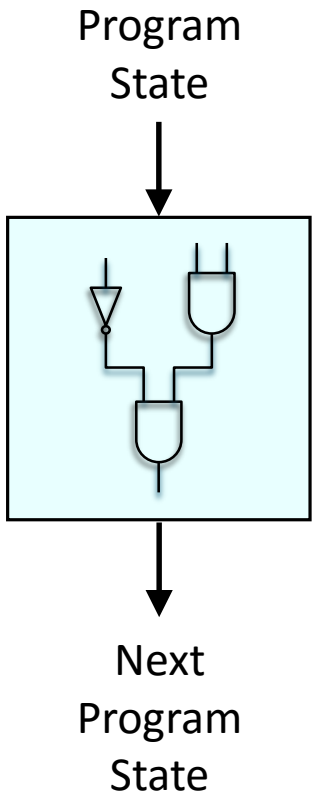




Modelling Processor Behavior

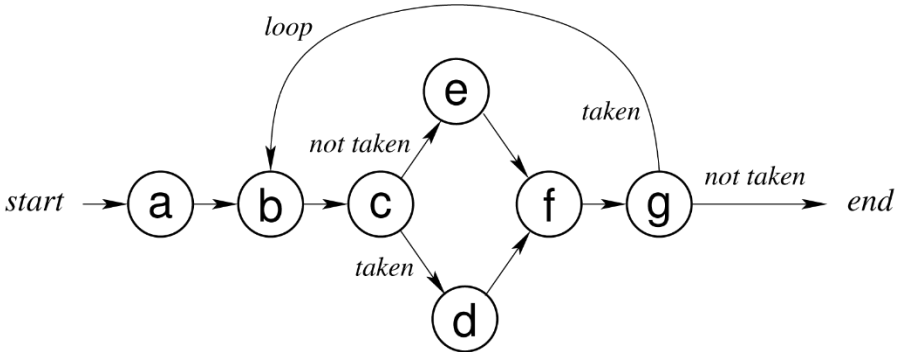
Instruction Cell

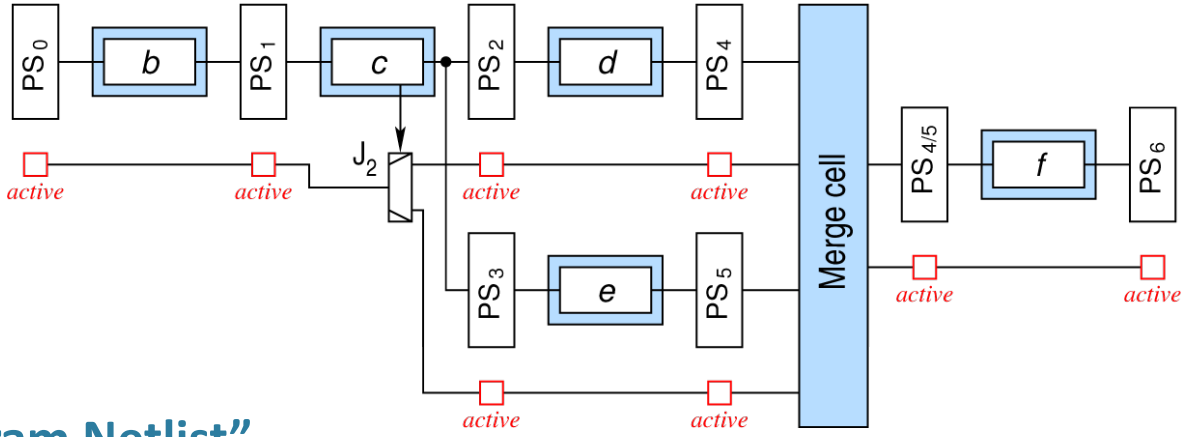
- Abstract model of a CPU instruction
  - combinational circuit
  - created manually (once for given ISA)
- Models the modification of the Program State (PS)
- Instruction cell represents
  - Program-visible registers
  - Can be directly mapped to gate-level registers



“Program Netlist”

- ↗ SAT analysis is interleaved with unrolling
  - Path pruning
  - Path merging
  - Unrolling loops





“Program Netlist”

- ↗ combinational circuit
- ↗ compactly represents all execution paths

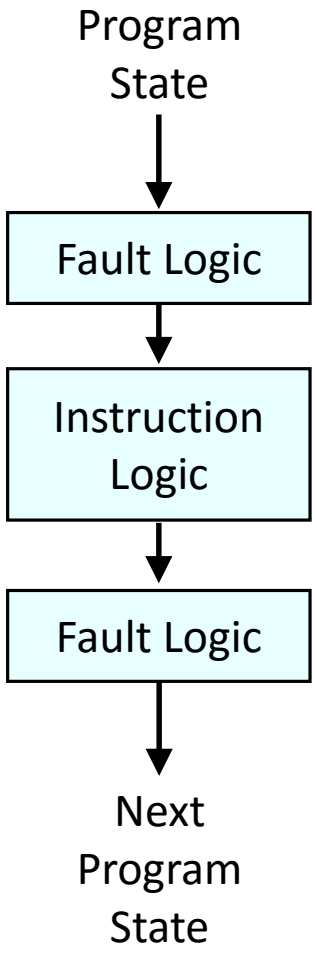
Program Netlist (PN)

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…

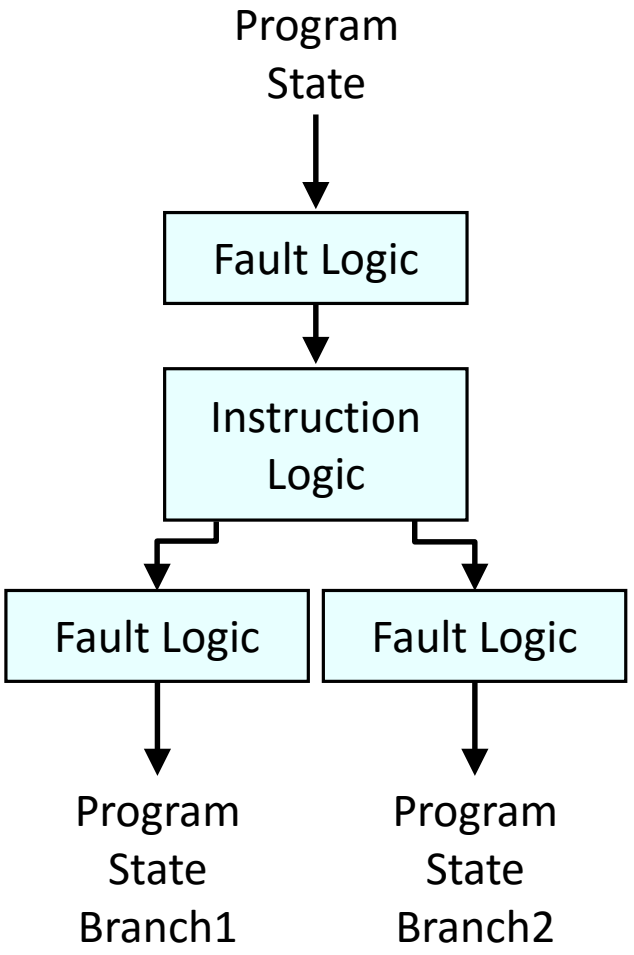
Fault Injection

- 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



PN Fault Injection

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

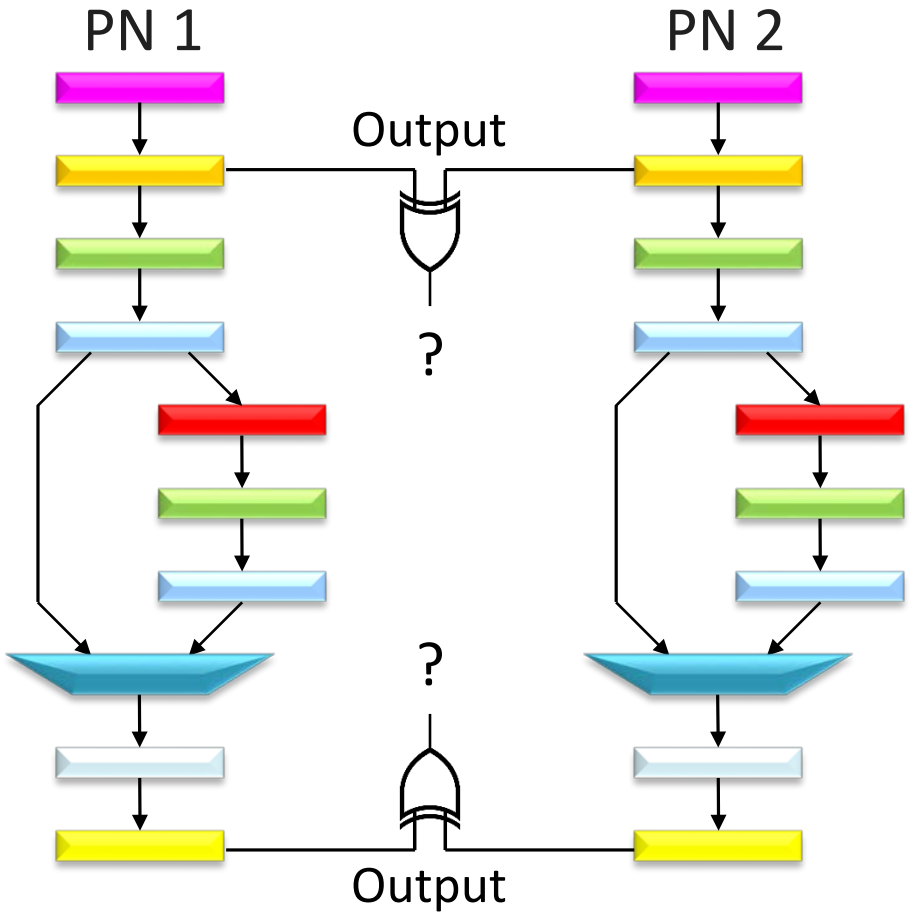
- ✓ Fault activation is configurable (single/multiple faults, permanent/temporary)
  - ✓ Modelling of a large fault list in a single PN
- ✓ SAT solver implicitly considers all possible single and multiple faults during unrolling

[C. Bartsch, C. Villarraga, D. Stoffel, W. Kunz: A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems, *IEEE Latin-American Test Symposium* (LATS-2016), Brazil, 2016.]

Fault Analysis

1. Duplicate generated PN
2. Compare outputs

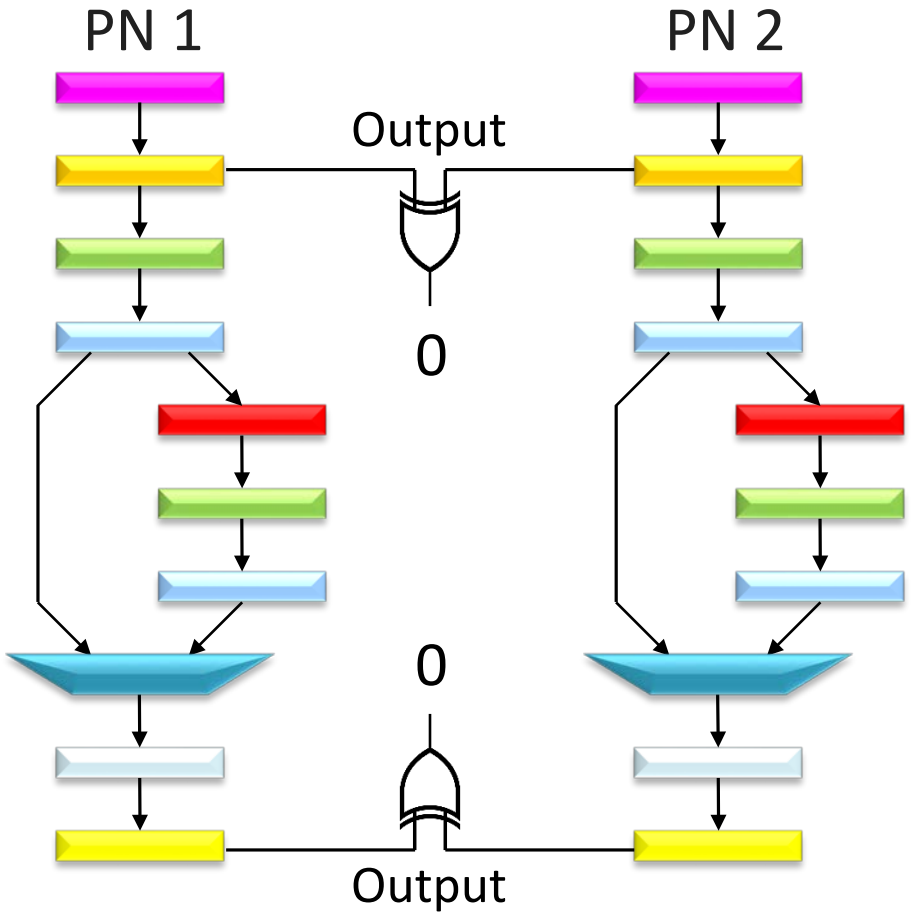
- PN1:
  - Faults are deactivated
    - Fault1: @0xABCD = 0
    - Fault2: @0xABCE = 0
- PN2:
  - Faults are activated
    - Fault1: @0xABCD = 1
    - Fault2: @0xABCE = 0



Fault Analysis

Results:

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

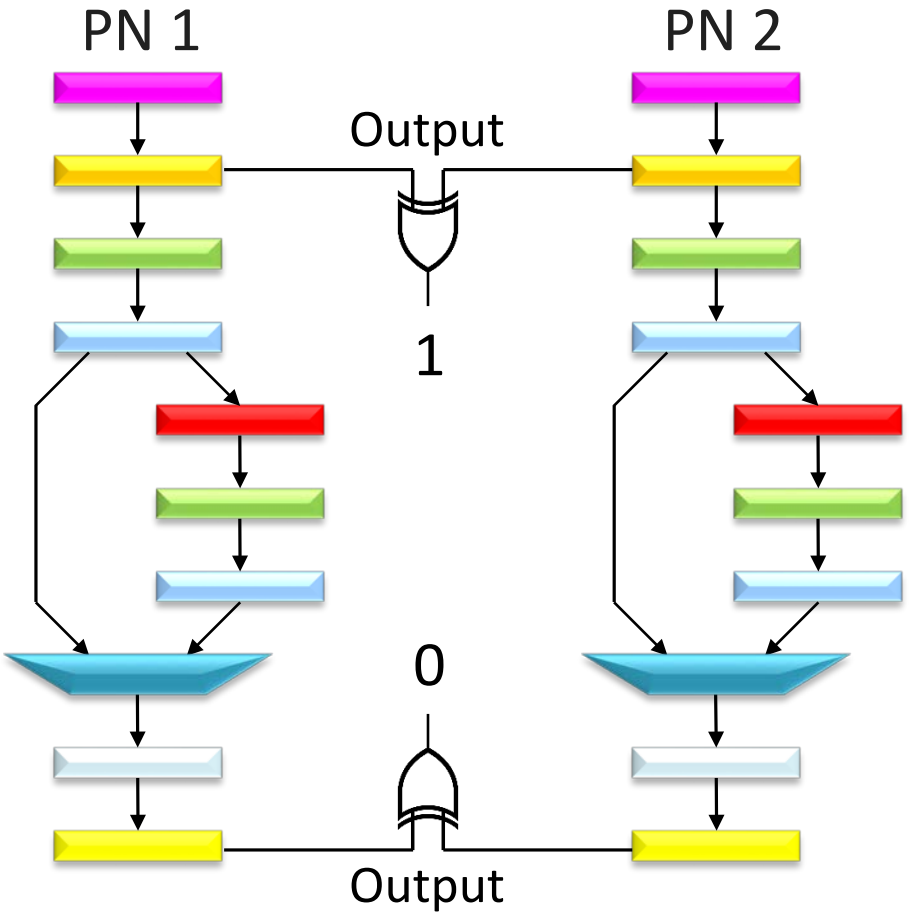


Fault Analysis

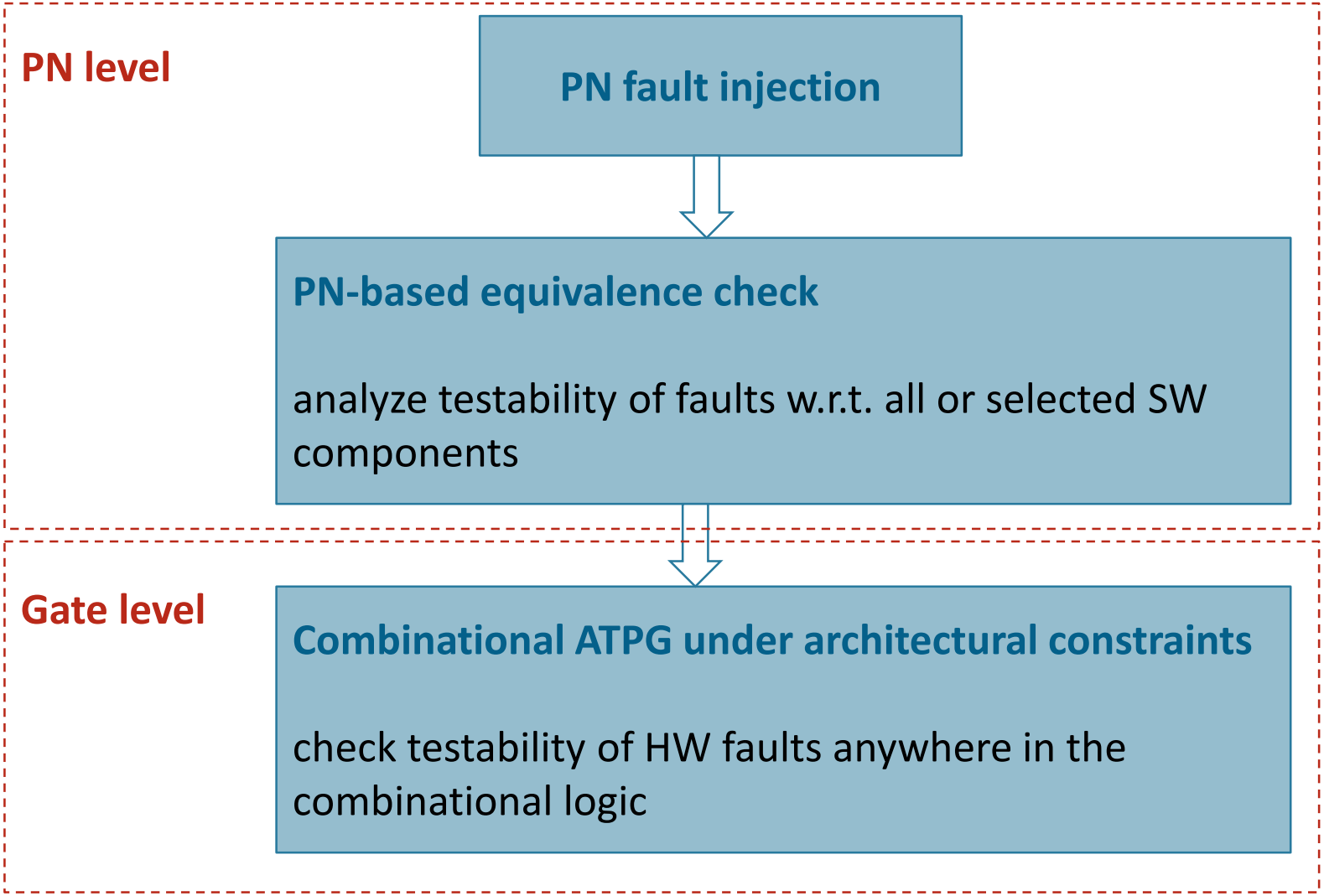
Results:

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**

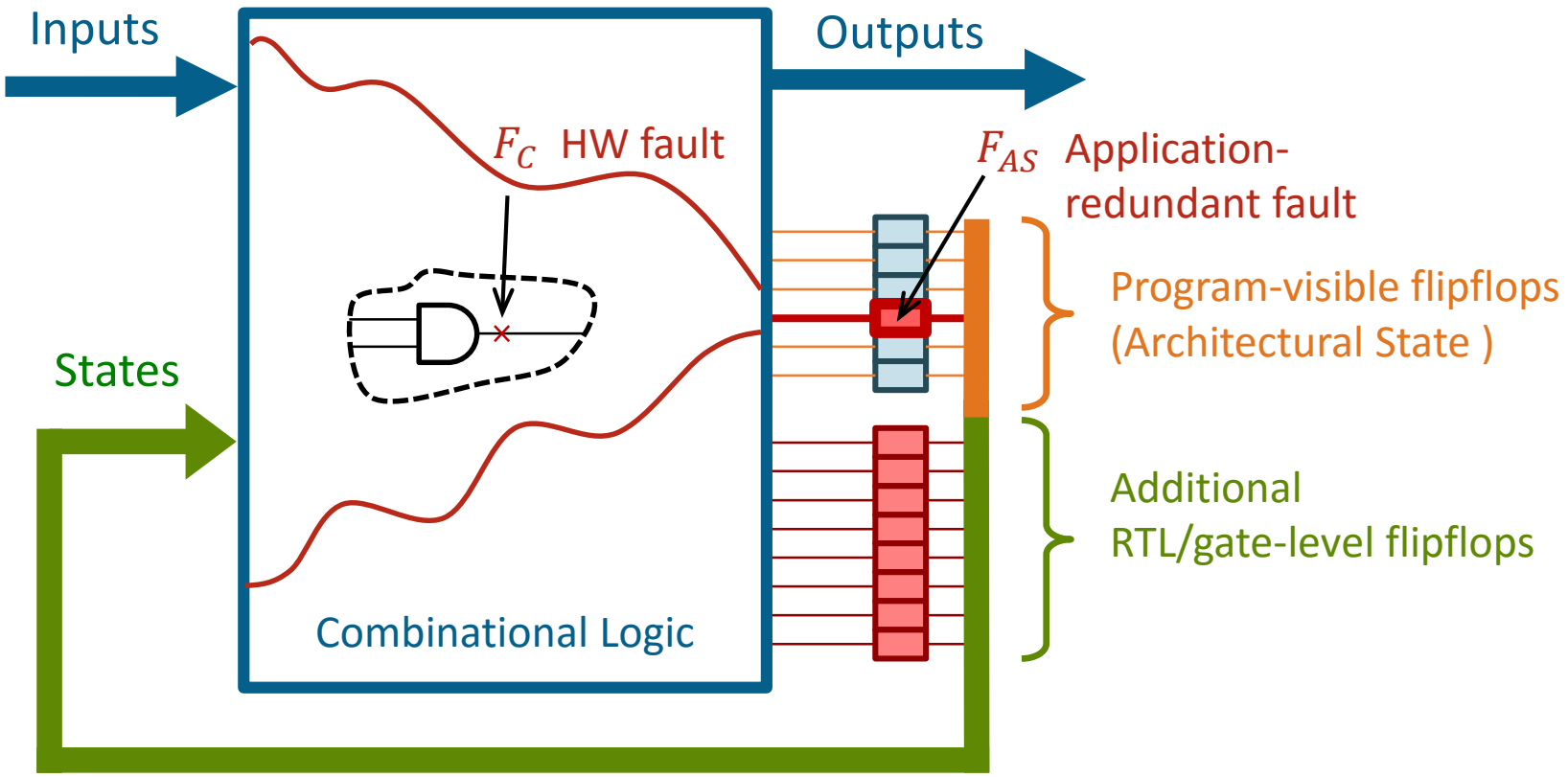


Cross-Layer Fault Analysis



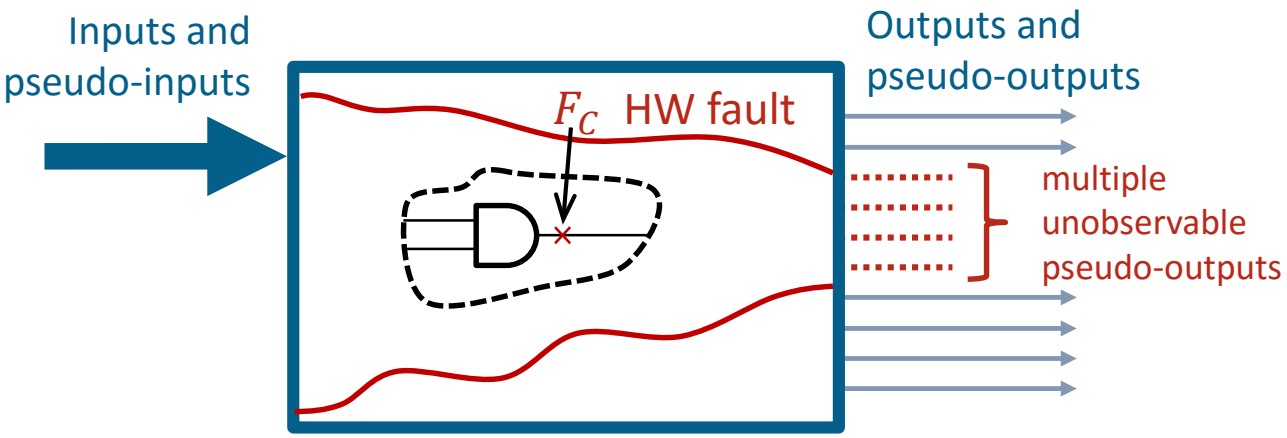
Gate Level Faults

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



Gate-Level Faults

Identifying gate-level redundancies



1. Identify **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

Experimental Results

- Computing platform Aquarius was synthesized to the gate level using Synopsys DesignCompiler

Design statistics for computing platform – gate level	
Primary inputs	51
Primary outputs	73
State bits	1217
Target stuck-at faults (single)	79184
Untestable stuck-at faults	3367

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

	PN Level			Gate Level		
Program	testable faults	untestable faults	CPU time [s] model gen.	testable faults	**untestable faults**	CPU time ATPG [s]
LIN	560	544	1081	50655	**28529**	2
TCAS	208	896	147	48234	**30949**	1.4

Experimental Results

- **Dependency Analysis**

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

PN-level dependency analysis – LIN driver	
# nodes in dependency graph	259
# syntactically critical state bits	289
# semantically critical state bits	200
# safety-function-redundant state bits	352
# program-visible state bits in Aquarius	552

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**

End

Questions?

[C. Bartsch, C. Villarraga, D. Stoffel, W. Kunz: A HW/SW Cross-Layer Approach for Determining Application-Redundant Hardware Faults in Embedded Systems, *Journal of Electronic Testing (JETTA),* Springer, Jan. 2017]

[C. Villarraga, B. Schmidt, B. Bao, R. Raman, C. Bartsch, T. Fehmel, D. Stoffel, W. Kunz. "Software in a Hardware View: New Models for HW-dependent Software in SoC Verification and Test" Proc. International Test Conference (ITC'14), 2014. Seattle, USA.]

[C. Villarraga, D. Stoffel, and W. Kunz. Formal System Verification, chapter Software in a Hardware View: New Models for HW-dependent Software in SoC Verification, Ed. R. Drechlser, Springer Int. Publishing, 2017.]