Test Program Generation for a Microprocessor

A Case-Study

Achim D. Brucker, Abderrahmane Feliachi, Yakoub Nemouchi and Burkhart Wolff

Introduction

- ÅCertifications of critical security or safety system properties are becoming increasingly important
- ÅCommon Creteria is an international stadard (ISO/IEC 15408) for computer security certification
- ÅThe Common Criteria requires for each level:
  - ✓EAL1: Functionally Tested
  - ✓EAL2: Structurally Tested
  - ✓EAL3: Methodically Tested and Checked
  - ✓EAL4: Methodically Designed, Tested, and Reviewed
  - ✓EAL5: Semi-formally Designed and Tested
  - ✓EAL6: Semi-formally Verified Design and Tested
  - ✓EAL7: Formally Verified Design and Tested

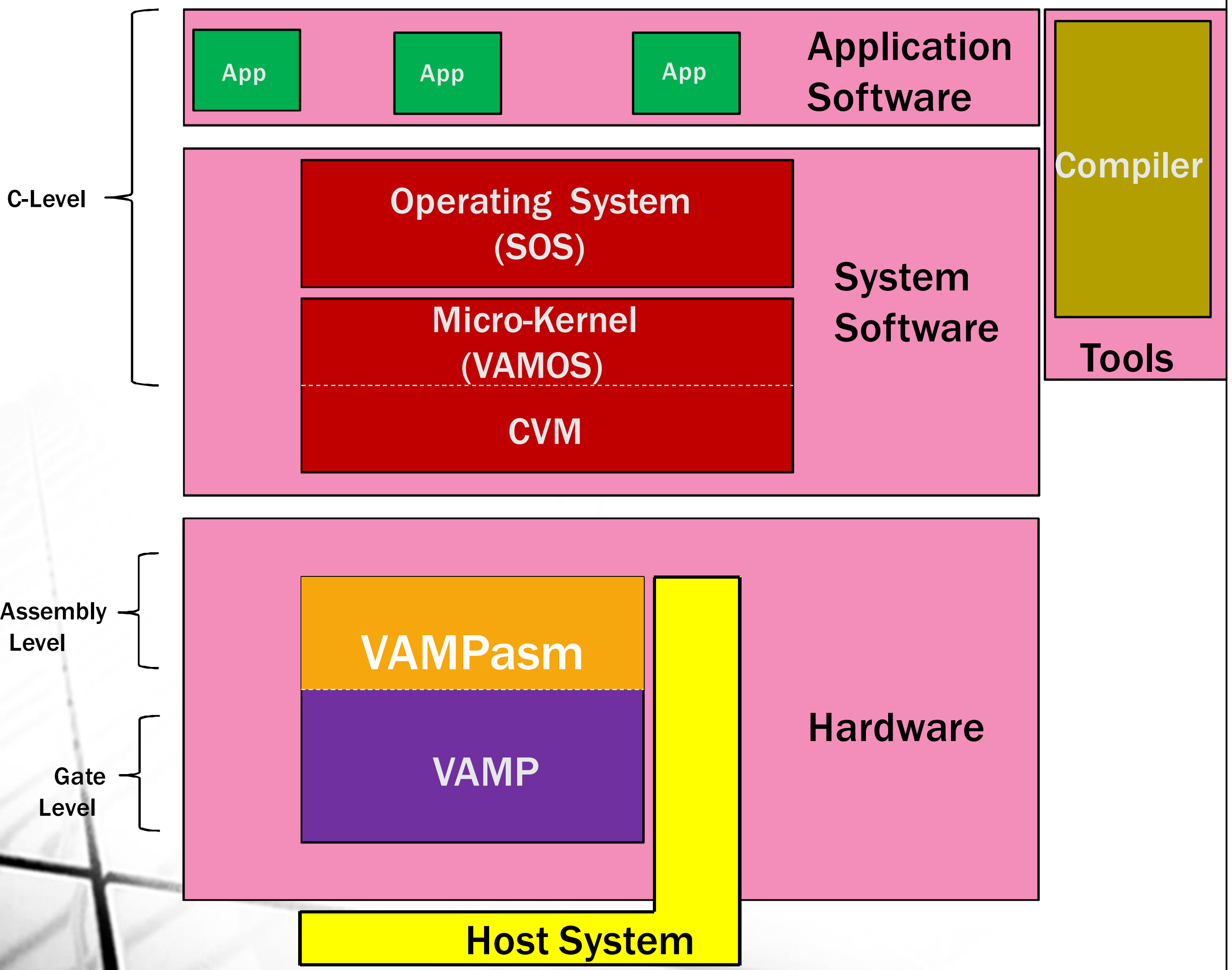
Introduction

- The principal goal of the EU IP Euro-MILS
  - ✓ Certification > EAL5 for the PikeOS real time operating system
    ➔ this requires the test of the underlying hardware of the entire system
  - ✓ In industry, this is done by special “test kits”
  - ✓ Our motivation: automatically generating test kits which check that a given Hardware meets the requirements imposed by PikeOS
- Our contributions :
  - ✓ Use a formal model of microprocessor (specified in Isabelle/HOL)
  - ✓ Generate test programs from this model (using HOL-TestGen)

Outline

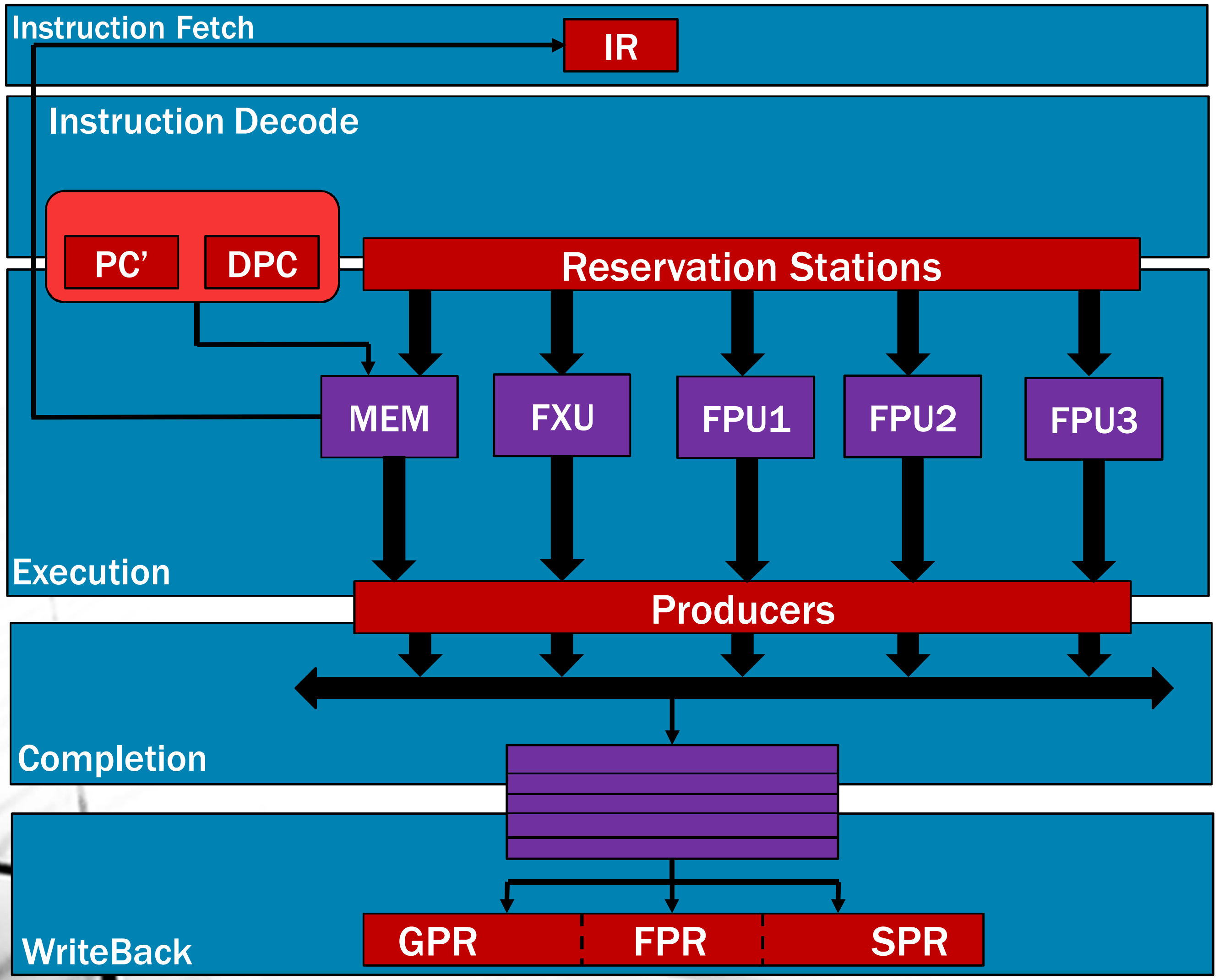
- Introduction
- Instead of PikeOS: The Verisoft system architecture
  - VAMOS, (gate-level) VAMP, abstract VAMP
  - Expressing Test-Scenarios in HOL-TestGen
  - Experimental Results
- Conclusion

Verisoft Layers



- The Context: The Verisoft Project
  - ✓ Main goal: pervasive formal verification of a computer system from application level to hardware (in Isabelle)
  - ✓ developed and verified in context of german BMBF project Verisoft
- We just use the VAMPasm for testing

VAMP Microarchitecture



Gate-level VAMP

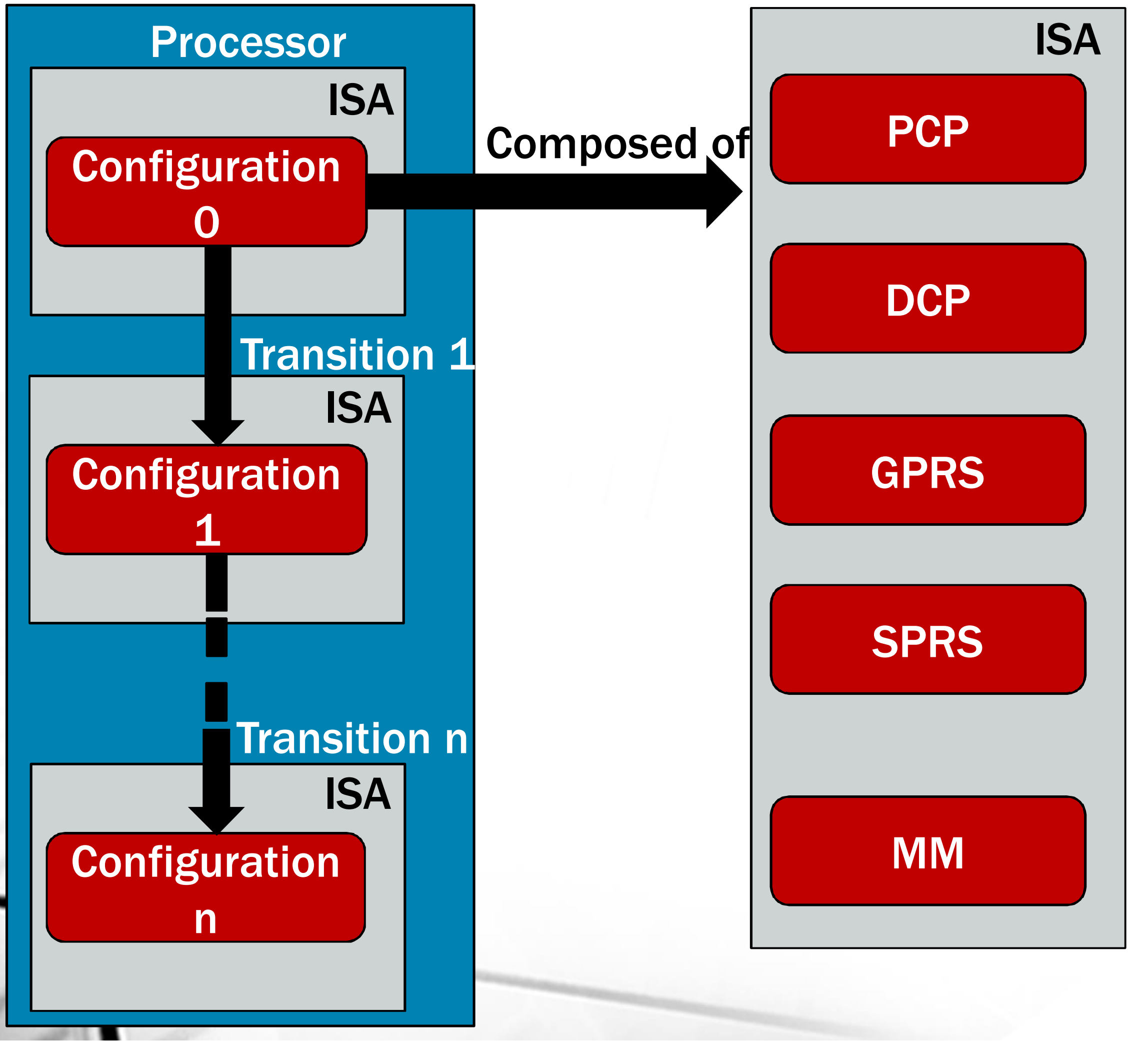
Å VAMP layers

- ✓ Consist of 5 layers
- ✓ Instruction Fetch Layer
- ✓ Instruction Decode Layer
- ✓ Execution Layer
- ✓ Completion Layer
- ✓ WriteBack Layer

Å VAMP Functional model

- ✓ IF and ID: realize a pipelined implementation
- ✓ EX, C, WB realize Tomasulo scheduler with
- ✓ Five functional units
- ✓ Fair scheduling policy on CDB
- ✓ ROB for precise interrupts

VAMP Abstract Model



The abstract VAMP

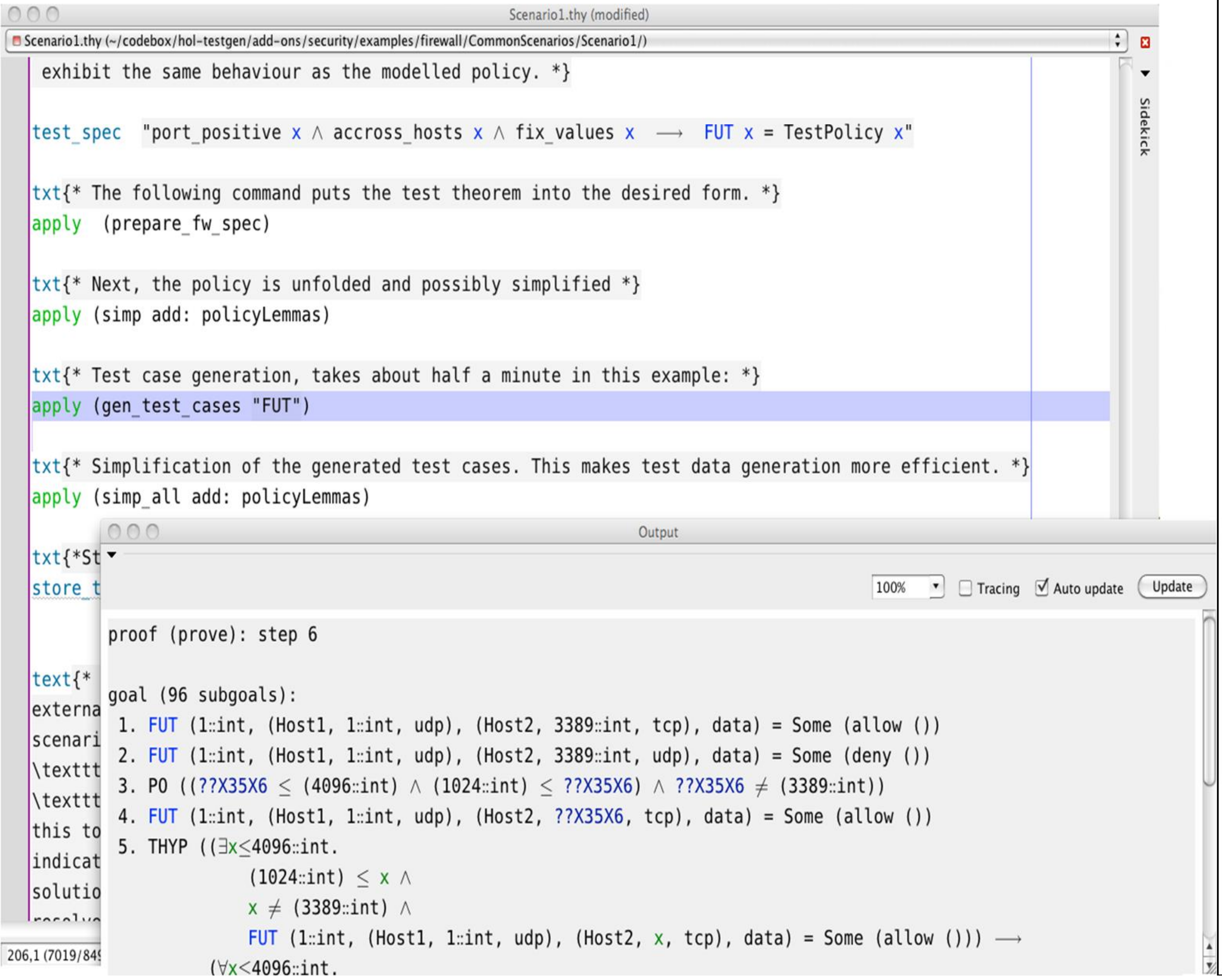
Å VAMP Model

- ✓ The processor consists of set of transitions
- ✓ Those transitions are defined over **I**nstruction **S**et **A**rchitecture configurations

Å A configuration consists of 5 elements

- ✓ **P**rogram **C**ounter : a 30 bit register
- ✓ **D**elayed **P**rogram **C**ounter : a 30 bit register
- ✓ **G**eneral **P**urpose **R**egister**s**: a register file consisting of 32 registers of 32 bits each
- ✓ **S**pecial **P**urpose **R**egister**s**: a register file consisting of 32 registers of 32 bits
- ✓ **M**emory **M**odel: a $2^{[illegible]}$ bytes addressable memory

HOL-TestGen session using Isabelle/jEdit Front-End



Scenario1.thy (modified)
Scenario1.thy (~/codebox/hol-testgen/add-ons/security/examples/firewall/CommonScenarios/Scenario1/)
Sidekick
Output
100%
Tracing
Auto update
Update

```
 exhibit the same behaviour as the modelled policy. *}

test_spec  "port_positive x ∧ accross_hosts x ∧ fix_values x  ⟶  FUT x = TestPolicy x"

txt{* The following command puts the test theorem into the desired form. *}
apply  (prepare_fw_spec)

txt{* Next, the policy is unfolded and possibly simplified *}
apply (simp add: policyLemmas)

txt{* Test case generation, takes about half a minute in this example: *}
apply (gen_test_cases "FUT")

txt{* Simplification of the generated test cases. This makes test data generation more efficient. *}
apply (simp_all add: policyLemmas)

txt{*St
store_t


text{*
externa
scenari
\texttt
\texttt
this to
indicat
solutio

206,1 (7019/849
```

```
proof (prove): step 6

goal (96 subgoals):
 1. FUT (1::int, (Host1, 1::int, udp), (Host2, 3389::int, tcp), data) = Some (allow ())
 2. FUT (1::int, (Host1, 1::int, udp), (Host2, 3389::int, udp), data) = Some (deny ())
 3. PO ((??X35X6 ≤ (4096::int) ∧ (1024::int) ≤ ??X35X6) ∧ ??X35X6 ≠ (3389::int))
 4. FUT (1::int, (Host1, 1::int, udp), (Host2, ??X35X6, tcp), data) = Some (allow ())
 5. THYP ((∃x≤4096::int.
              (1024::int) ≤ x ∧
              x ≠ (3389::int) ∧
              FUT (1::int, (Host1, 1::int, udp), (Host2, x, tcp), data) = Some (allow ())) ⟶
          (∀x≤4096::int.
```

HOL-TestGen

Å**Isabelle**

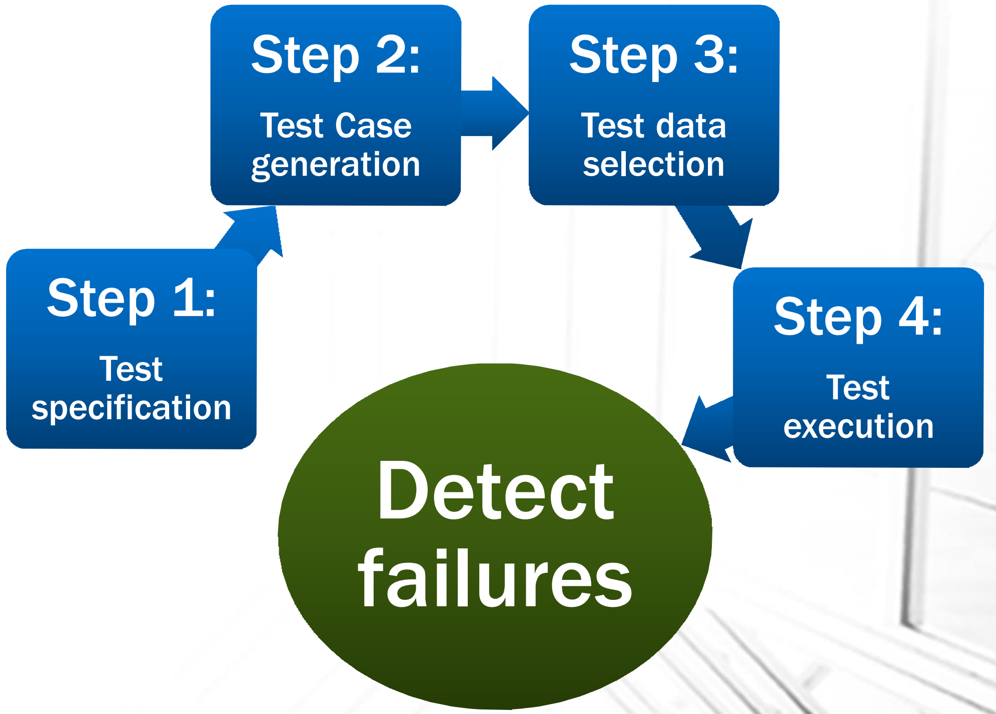
- ✓ An interactive proof Assistent based on kernel ensuring logical correctness
- ✓ Customizable to variety of logics (FOL,HOL, ZF)
- ✓ Supports many specification constructs
- ✓ Provide tools for automatic reasoning

Å**HOL-TestGen**

- ✓ Built on top of Isabelle/HOL
- ✓ Provide modeling environment for test theory
- ✓ For stating and tranforming a test specification
- ✓ For test generation using Isabelle's tactic procedures

The HOL-TestGen Workflow

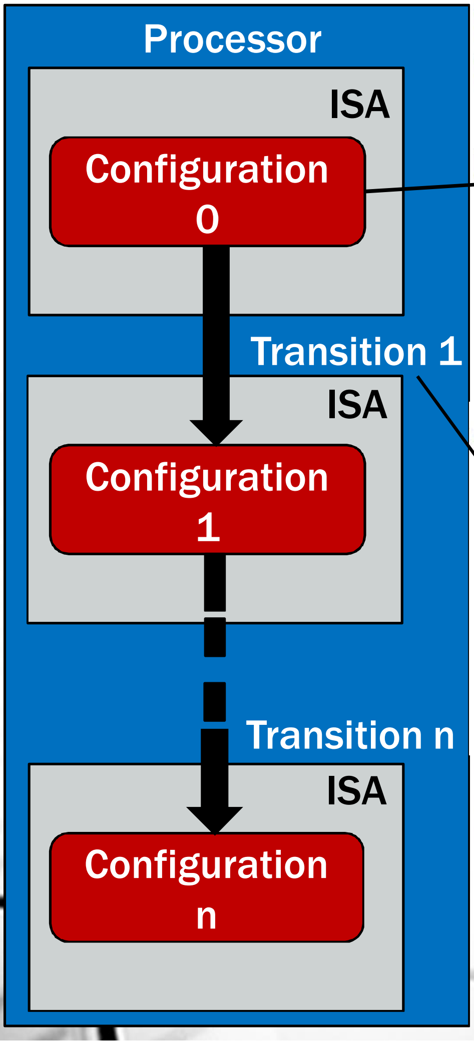
- The use of Hol-TestGen environment requires 4 major steps each step has its own specific tool
- Step 1 is performed on Hol-TestGen by unsing test_spec
- Step 2 is performed by gen_test_cases tactic
- Step 3 is performed by gen_test_data
- Step 4 is performed by generate_test_script



Outline

- ÅIntroduction
- ÅInstead of PikeOS: The Verisoft system architecture
  - ÅVAMOS, (gate-level) VAMP, abstract VAMP
  - ÅExpressing Test-Scenarios in HOL-TestGen
  - ÅExperimental Results
- ÅConclusion

VAMP Isabelle Model



```
type_synonym regcont = int              -- {* contents of register *}
type_synonym registers = regcont list -- {* register file *}

record ASMcore_t = dpc  :: nat
                   pcp  :: nat
                   gprs :: registers
                   sprs :: registers
                   mm   :: mem_t

definition is_ASMcore :: ASMcore_t ⇒bool where
 is_ASMcore st ≡asm_nat (dpc st) ∧
               asm_nat (pcp st) ∧
               length (gprs st) = 32 ∧
               length (sprs st) = 32 ∧
               (∀ ind < 32. asm_int (reg (gprs st) ind)) ∧
               (∀ ind < 32. asm_int (sreg (sprs st) ind)) ∧
               (∀ ad. asm_int (data_mem_read (mm st) ad))
definition Step :: ASMcore_t ⇒ASMcore_t
where   Step st ≡ exec_instr st (current_instr st)
```

Our Approach

Å VAMP assembly language

- ✓ An assembly language was introduced abstracting the VAMP ISA
- ✓ Addresses are represented by naturals
- ✓ Memory and registers contents represented by integers
- ✓ A configuration is introduced on Isabelle/HOL by record type ASMcore▯ with 5 fields
- ✓ The instructions are represented in an abstract datatype with readable names
- ✓ The transition relations between configurations is specified by Step on Isabelle
- ✓ Step execute the current program instruction given in DPC
- ✓ Those transition relations are used as the basis of test specification

Our Approach in different testing scenarios

Å4 types of instructions are concerned by test scenario:

- ✓Memory related load/store operations
- ✓arithmetic operations
- ✓logical operations
- ✓ control flow realted operations

Å1 test scenario is presented as an example:

- ✓Testing load/store operations

ÅThe scenario is applied with :

- ✓ Unit testing scheme
- ✓Sequence testing scheme

Testing Methodology

Å2 testing scenario schemes are applied :

✓Model-based unit testing

✓Model-based sequence testing

ÅUnit test scenario:

✓Unit-test specification has the following scheme:

$$\texttt{test\_spec}\ \texttt{pre}\ \sigma\ \iota \Longrightarrow \texttt{SUT}\ \sigma\ \iota =_k \texttt{exec\_instr}\ \sigma\ \iota$$

✓Where test_spec is used to state test specification

✓pre σ ι is precondition over input variables σ ι and SUT σ ι $=_k$ exec_instr σ ι is postcondition over results

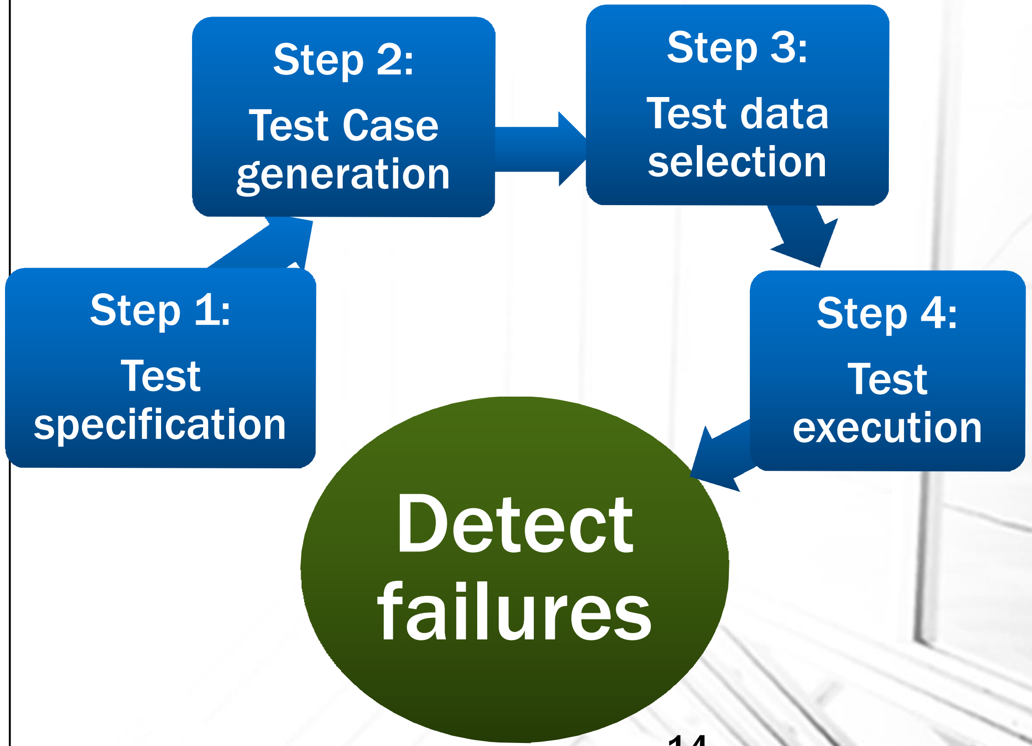
✓ $=_k$ is our conformance relation

ÅGoal of using Model-based unit testing scenario

✓Test individually each operation or instruction with different data

Steps for unit testing scenario of load/store operations

- **Step 1** reduce the domain of generated tests to load/store operations
- **Step 2** generation of test cases and uniformity hypotheses from TS
- **Step 3** instantiation of test data for each test case
- **Step 4** test execution



load/store unit testing scenrio on HOL-TestGen

```
test_spec is_load_store ι ⟹ SUT σ0 ι =k exec_instr σ0 ι
apply (gen_test_cases 0 1 SUT)
store_test_thm load_store_instr
 gen_test_data load_store_instr
```

```
1. SUT σ0(Ilb ??X7 ??X6 ??X5)
    (...)
2. THYP ((∃x xa xb. SUT σ0(Ilb xb xa x) (...)) ⟶
        (∀x xa xb. SUT σ0(Ilb xb xa x) (...)))
```

```
SUT σ0(Ilb 1 0 1) σ1
```

•. **Step 1** will be done by introducing the predicate **is_load_store** in precondition of TS

Å **Step 2** 8 test cases with symbolic operands for each instruction are generated

- ✓ A uniformity hypotheses is stated on each symbolic test case, which will allow to select one concrete witness for each sympbolic test

Å **Step 3** instantiation of test data with **gen_test_data**

Å Unit test scenario will reveal design faults or undesired state modification

Our Approach: Testing Methodology

Å2 testing scenario are applied :

- ✓ Model-based unit testing
- ✓ Model-based sequence testing

ÅSequence testing scenario:

- ✓ Sequence test specification has 2 schemes:

```
test_spec pre ιs::instr list ⟹
            (σ₀ ⊨(_ ←mbind ιs exec_VAMP; assert_SE (λσ. σ=_k SUT σ₀ιs)))
```

- ✓ Or

```
test_spec pre ιs::instr list ⟹
            (σ₀ ⊨(_ ←mbind (ιs@[load x 0]) exec_VAMP;
                     assert_SE (λσ. (gprs σ)!0 = (gprs (SUT σ₀ιs))! 0)))
```

- ✓ In both σ_0 is an initial state and ιs is the sequence of instructions that will be generated
- ✓ $\text{exec}_{\text{vamp}}$ is a lifting of exec_instr into state exception monad

ÅGoal of using Model-based sequence testing scenario

- ✓sequence of instructions up to a given length

load/store sequence testing scenrio on HOL-TestGen

```
test_spec list_all is_load_store (ιs::instr list) ⟹
          (σ_0 ⊨(s ←mbind ιs exec_VAMP; assert_SE (λσ. σ=_k SUT σ_0ιs)))
apply (gen_test_cases SUT)
store_test_thm load_stre_instr_seq
gen_test_data load_stre_instr_seq
```

```
1.σ_0 ⊨(s ←mbind [Isw ??X597 ??X586 ??X575, Ilbu ??X557 ??X546 ??X535,
                  Ilbu ??X517 ??X506 ??X595] exec_VAMP;
      assert_SE (λσ. σ=_k SUT σ_0[Isw ??X597 ??X586 ??X575,
                                  Ilbu ??X557 ??X546 ??X535,
                                  Ilbu ??X517 ??X506 ??X595]))
2. THYP ((∃x1 x2 x3 x4 x5 x6 x7 x8 x9. σ_0⊨(s ←mbind [Isw x1 x2 x3,
         Ilbu x4 x5 x6, Ilbu x7 x8 x9] exec_VAMP; (...))) ⟶
      (∀x1 x2 x3 x4 x5 x6 x7 x8 x9. σ_0⊨(s ←mbind [Isw x1 x2 x3,
         Ilbu x4 x5 x6, Ilbu x7 x8 x9] exec_VAMP; (...))))
```

```
σ_0 ⊨(s ←mbind [Isw 0 1 8, Ilbu 1 0 -3, Ilbu 3 2 8] exec_VAMP;
      assert_SE (λσ. σ=_k SUT σ_0[Isw 0 1 8, Ilbu 1 0 -3, Ilbu 3 2 8]))
```

- Å **Step 1 Filtering by is_load_store the entire input sequences, without filtring we have 178809 cases…**
- Å **Step 2 After filtring for a sequence of length < 4 we have 585 test cases:**
  - ✓ **585= 1 + 8 + 8*8 + 8*8*8**
- Å **Step 3 instantiation of test data for each test case**
- Å **Sequence testing scenario (« purpose») designed to reveal**
  - Å **Byte alignment errors**
  - Å **Memory errors caused by piplining**

Outline

- ÅIntroduction
- ÅInstead of PikeOS: The Verisoft system architecture
  - ÅVAMOS, (gate-level) VAMP, abstract VAMP
  - ÅExpressing Test-Scenarios in HOL-TestGen
  - ÅExperimental Results
- ÅConclusion

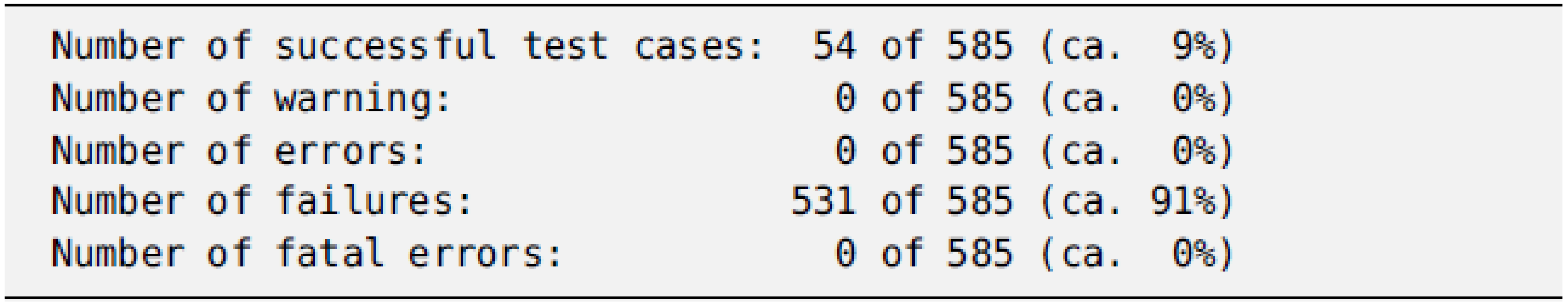
Experimental results

- In unit testing scenario
  - Strong assumption on testability
  - the test driver has actually access to registers and Memory
- In sequence scenario of load/store operations
  - 39 seconds in test partitionning phase
  - 42 in test data selection
  - 2 seconds of test program generation
  - 585 test programs for scenario in 83 seconds!!!

Experimental results

ÅIn this paper, we focussed on test generation method

- ÅNo experiments was done against ‘real’ hardware
- ÅHowever, we generated mutants from the generated code (using isabelle’s code generator) of the processor model
- Å585 test programs (of this scenario) were run against the mutant set
- ÅResults : 91% kills…



```
Number of successful test cases:  54 of 585 (ca.  9%)
Number of warning:                 0 of 585 (ca.  0%)
Number of errors:                  0 of 585 (ca.  0%)
Number of failures:              531 of 585 (ca. 91%)
Number of fatal errors:            0 of 585 (ca.  0%)
```

Outline

- Introduction
- Instead of PikeOS: The Verisoft system architecture
  - VAMOS, (gate-level) VAMP, abstract VAMP
  - Expressing Test-Scenarios in HOL-TestGen
  - Experimental Results
- Conclusion

Related Work

- Formal verification is widely used in industry since at least 10 years
  - 1997: Functional verification of the superscalar sh-4 microprocessor by P.Biswas, A.Freeman, K.Yamada, N.Nakagawa, K.Ushiyama
  - 2003: Formal verification at intel by John Harrison
- Formal models of complete micro-processors as well as verification approaches that provide verification from application layer to the hardware are rare.
  - 2010: Besides of VAMP we have Formal verification and verification of microkernel by Jan Dörrenbacher
  - 2003: We have Fox Formal Specification and Verification of arm6 by Anthony C.J.Fox
- Test program generation for microprocessor intruction sets have been known for long time
  - 2001: A new functional test program generation methodology by F.Fallah and K.Takayama.
  - 2005: A configurable random test program generator for micro-processors by Haihua Shen. Lin ma, and Hang Zhang.
- Only few works suggest to use model based or specification based test program generation algorithms
  - 2011: Reconfigurable model based test program generator for microprocessors by Alexander Kalkin, Eugene Kornykhin, Dmitry Vorobyev.

Conclusion

Å We introduced a model-based test generation technique for a realistic model of a RISC processor called VAMP. The technique is of particular interest for higher level certification (for example in higher EAL levels Common Criteria)

Å We adapted and reuse the formal model of the processor implemented on Isabelle/HOL to generate test cases

Å We automatically converted the test cases to test programs that can be used to check if a given hardware model conforms to the VAMP processor

Å We evaluated the technique by generating mutants from the generated code of the model and killing theme by the generated test sets.