Test Program Generation for a Microprocessor
A Case-Study

Achim D. Brucker[1], Abderrahmane Feliachi[2], Yakoub Nemouchi[2], and Burkhart Wolff[2]

[1] SAP AG, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com
[2] Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France
{feliachi, nemouchi, wolff}@lri.fr

Abstract. Certifications of critical security or safety system properties are becoming increasingly important for a wide range of products. Certifying large systems like operating systems up to Common Criteria EAL 4 is common practice today, and higher certification levels are at the brink of becoming reality.
To reach EAL 7 one has to formally verify properties on the specification as well as test the implementation thoroughly. This includes tests of the used hardware platform underlying a proof architecture to be certified. In this paper, we address the latter problem: we present a case study that uses a formal model of a microprocessor and generate test programs from it. These test programs validate that a microprocessor implements the specified instruction set correctly.
We built our case study on an existing model that was, together with an operating system, developed in Isabelle/HOL. We use HOL-TestGen, a model-based testing environment which is an extension of Isabelle/HOL. We develop several conformance test scenarios, where processor models were used to synthesize test programs that were run against real hardware in the loop. Our test case generation approach directly benefits from the existing models and formal proofs in Isabelle/HOL.
Keywords: test program generation, symbolic test case generations, black box testing, white box testing, theorem proving, interactive testing.

1 Introduction

Certifications demonstrating that certain security or safety requirements are met by a system are becoming increasingly important for a wide range of products. Certifications play an increasing role in industrial applications including operating systems and embedded systems. While the certifications of large systems, including fully functional operating systems up to Common Criteria EAL 4 are common practice today, higher-levels involve the use of formal methods and combined test and proof activities, covering various layers of a system including soft and hardware-components. To reach EAL7 [8] one has to formally verify properties on the specification as well as test the implementation thoroughly.

M. Veanes and L. Viganò (Eds.): TAP 2013, LNCS 7942, pp. 76–95, 2013.

The certification of systems combining software and hardware, such as modern avionics systems, requires to test microprocessors in the context of the developed system. Thus, an isolated verification (and certification) by the chip manufacturer is not enough. *Test program generation*, i. e., generating test cases in terms of low level programs for the microprocessor under test, is a well-established technique for validating processor designs. As it allows to validate processors on the instruction set or assembly level, it is well suitable for validating commercial off-the-shelf (COTS) processors for which, usually, implementation details are not available. Microprocessor vendors that want to support their customers in certification processes can provide them with the necessary test programs. Such a set of test cases is called a *certification kit* and selling usually manually developed certification kits is a profitable business, as is the case for, e. g., avionics certifications according to DO-178 and DO-245 [16].

We present a case study for the model-based generation of test programs (i.e, the basis for a certification kit) for a realistic model of a RISC processor called VAMP. VAMP is inspired by IBM's G5 architecture. In the Verisoft project (see `http://www.verisoft.de`), a formal model for both the processor and a small operating system has been developed in Isabelle/HOL. We will adapt and reuse the processor model to generate test cases that can be used to check if a given hardware conforms to the model of the VAMP processor. The presented test scenario is of particular interest for the higher levels of certification processes as imposed by Common Criteria EAL 7. Even if the transition from C programs to the processor models has been completely covered by deductive verification methods as in CompCert [18], certification bodies will require test sets checking the conformance of the underlying processor model to real hardware.

At present, specification-level verification and the development of test sets are usually two distinguished tasks. Moreover, test sets for certification kits are usually developed manually. In contrast, our model-based test case generation approach uses the design model that was already used for the verification task. In particular, we are using HOL-TestGen to generate test sequences generated from the VAMP model. As HOL-TestGen is built on top of Isabelle/HOL, i. e., test specification are expressed in terms of higher-order logic (HOL), we can directly benefit from the already existing verification models. In fact, the tight integration of a verification and a test environment is a distinguishing feature of HOL-TestGen.

2 Background

2.1 The Verified Architecture Microprocessor (VAMP)

The Verified Architecture Microprocessor (VAMP) as well as the micro-kernel VAMOS [10] has been developed and verified in the context of the German research projects Verisoft (`http://www.verisoft.de`) and VerisoftXT (`http://www.verisoftxt.de`). The goal in particular of the former project was the pervasive formal verification of computer systems from the application level down to the silicon, i. e., the hardware design.

On the *Application Software Layer*, this includes foundational proofs justifying a verification approach for system-level concurrent programs that are running as user processes on the micro-kernel VAMOS [10]. On the *System Software Layer*, VAMOS provides an infrastructure for memory virtualization, for communication with hardware devices, for process (represented as a sequence of assembly instructions), and for inter-process communication (IPC) via synchronous message passing that need to be verified. On the *Tools Layer*, the correctness of the compiler needs to be verified and, finally, on the *Hardware Layer*, the functional correctness of the hardware design is formally verified.

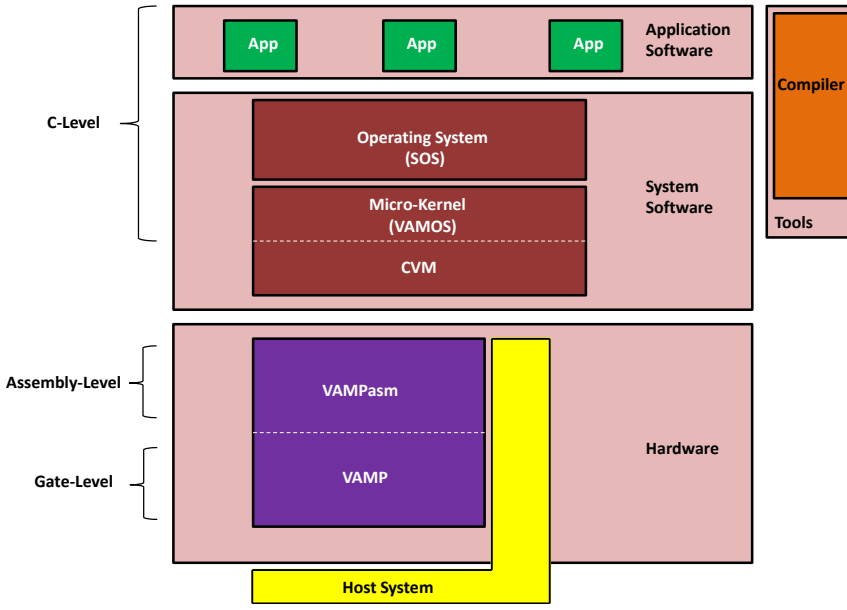

Fig. 1. The Verisoft System Layers

These four layers comprise the Verisoft Architecture (see Figure 1); each of the layers is in itself structured in several sub-layers.

Our work focuses on the hardware layer, more precisely the assembly-level (VAMPasm), i. e., the instruction set of the Verified Architecture MicroProcessor (VAMP) [3]. VAMP is a pipelined reduced instruction set (RISC) processor based on the out-of-order execution principle (see Hennessy and Patterson [15] for details). The VAMPasm (Section 3 presents the formal model we are using in our work) includes 56 instructions: 8 instructions for memory data transfer, 2 instructions for constant data transfer, 2 instructions for register data transfer, 14 instructions for arithmetic and logical operations, 16 instructions for test operations, 6 instructions for shift operations, 6 instructions for control operations as well as 2 instructions for interrupt handling.

In our unit and sequence test scenarios, we generate tests from a formal model of the instruction set, i. e., we test the conformance of the gate level (which corespondents to the implementation in traditional model-based testing) to assembly-level (which corespondents to the model in traditional model-based testing).

2.2 Isabelle/HOL and the HOL-TestGen Framework

Isabelle [20] is a proof assistant based on a kernel ensuring logical correctness. It is highly customizable to a variety of logics, among them first order logic (FOL), Zermelo-Fraenkel set-theory (ZF) and most notably higher-order logic (HOL). The HOL instance is well equipped with a number of components that support for specific specification constructs such as type definitions, (recursive) function definitions involving termination proofs and inductive set definitions. Isabelle is an interactive development environment providing immediate feedback in formal proof attempts and symbolic computations, as well as tools for automatic reasoning such as an term rewriting engine and various decision procedures. Beyond a verification environment, Isabelle can also be understood as a framework for building formal methods tools [24].

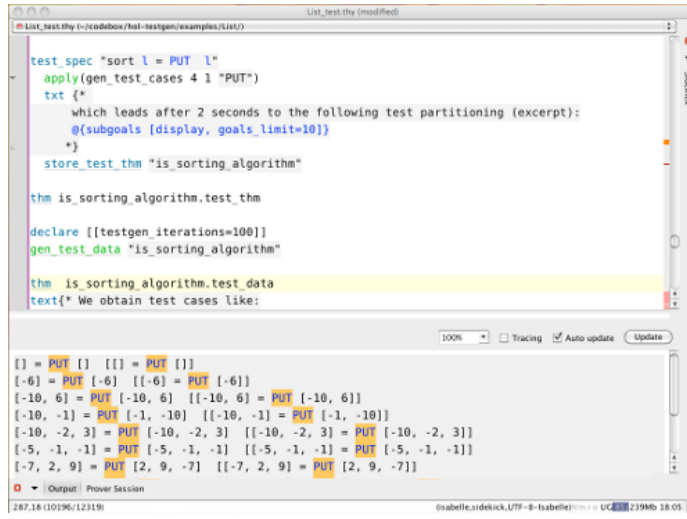

Fig. 2. A HOL-TestGen Session Using the Isabelle/jEdit Front-End

HOL-TestGen [5–7] is such a formal tool built on top of Isabelle/HOL. While Isabelle/HOL is usually seen as "proof assistant," HOL-TestGen (see Figure 2) is used as a document centric modeling environment for the domain specific background theory of a test (the *test theory*), for stating and logically transforming test goals (the *test specifications*), as-well as for the test generation

method implemented using Isabelle's tactic procedures. In a nutshell, the test generation method consists of:

1. a *test case generation* phase, which is essentially a process intertwining bounded case-splitting on variables (i. e., applying rules of the form: $x :: \alpha\text{list} = x = [] \wedge \exists al'.\, x = a\#l'$), simplification with respect to the underlying theory (using, e. g., $|a\#l'| = |l'| + 1$, etc.) and in a CNF-like normal form that leads to partitioning of the input/output relation.
2. a *test data selection* phase, which essentially uses a combination of constraint solvers using random test generation and the integrated SMT-solver Z3 [9] to construct an instance for each partition,
3. a *test execution* phase converts the instantiated test cases ("test oracles") to test driver code that is run against the system under test (SUT).

A detailed account on the symbolic computation performed by the test case generation and test selection procedures is described by Brucker and Wolff [6].

Several approaches for the generation of test cases are possible: while unit-test oriented test generation methods essentially use pre-conditions and post-conditions of system operation specifications, sequence-test oriented approaches essentially use temporal specifications or automata-based specifications of system behavior. In the case of test program generation, the state of the processor is an important element of the test description. The tests describe then sequences of state transitions that the processor may perform when executing the program instructions.

As HOL is a purely functional specification formalism, it has no built-in concepts for states and state transitions. To support sequence test specifications, HOL-TestGen uses the well-known notion of monads. The state-exception monad is, in fact, well fitted for this purpose, which is modeling partial state transition functions of type

```
type_synonym (o, σ) MONSE = σ⇀(o ×σ)
```

Using monads, programs under test can be seen as *i/o stepping functions* of type $\iota \Rightarrow$ `(o, σ)MONSE`, where each stepping function may either fail for a given state σ and input ι, or produce an output `o` and a successor state.

The usual concepts of *bind* (representing sequential composition with value passing) and *unit* (representing the embedding of a value into a computation) are defined for the case of the state-exception monad as follows:

```
definition bindSE :: (o, σ) MONSE ⇒(o ⇒(o', σ) MONSE) ⇒(o', σ) MONSE
where bindSE f g = λσ. case f σ of None ⇒ None
                         | Some(out, σ') ⇒g out σ'

definition unitSE :: o ⇒(o, σ) MONSE
where unitSE e = λσ. Some(e, σ)
```

`x ←f; g` is written for $\texttt{bind}_{SE}\,\texttt{f}(\lambda\texttt{x. g})$ and `return` for $\texttt{unit}_{SE}$. On this basis, the concept of a *valid test sequence* (no exception, P yields true for observed output) can be specified as follows:

$$\sigma \models \texttt{o}_1 \leftarrow \texttt{SUT i}_1\texttt{; ...; o}_n \leftarrow \texttt{SUT i}_n\texttt{; return (P o}_1 \cdots \texttt{o}_n\texttt{)}$$

where $\sigma \models \texttt{m}$ is defined as $\texttt{(m }\sigma \neq \texttt{None} \wedge \texttt{fst(m }\sigma\texttt{))}$. For iterations of i/o stepping functions, an `mbind` operator can be used, which takes a list of inputs $\iota\texttt{s} = [\texttt{i}_1, \ldots, \texttt{i}_n]$, feeds it subsequently into `SUT` and stops when an error occurs. Using `mbind`, valid test sequences for a program under test `SUT` satisfying a post-condition `P` can be reformulated by:

$$\sigma \models \texttt{os} \leftarrow \texttt{mbind } \iota\texttt{s SUT; return(P os)}$$

which is the HOL-TestGen's standard way to represent sequence test specifications. For cases, where a post-condition depends explicitly on the underlying state, we use the state-exception primitive:

```
definition assert_SE :: (σ ⇒bool) ⇒(o, σ) MON_SE
where assert_SE P = (λσ. if P σthen Some(True,σ) else None)
```

instead of `return(P)`.

3 The VAMP Model

The Verified Architecture MicroProcessor (VAMP) [3] is a 32-bit RISC CPU with a DLX-instruction set including floating point instructions, delayed program counter, address translation, and support for maskable nested precise interrupts. The VAMP hardware contains five execution units: the Fixed Point Unit, the Memory Unit, and three Floating Point Units. Instructions have up to six 32-bit source operands and produce up to four 32-bit results. The memory interface [2] of the VAMP consists of two Memory Management Units that access instruction and data caches, which in turn access a physical memory via a bus protocol.

In the context of the Verisoft project, an Isabelle/HOL specification (programmer's model) of the VAMP processor was introduced. The processor consists of a set of transitions defined over the Instruction Set Architecture (ISA) configurations. A configuration is composed of five elements:

1. *Program counter (pcp):* a 30 bit register containing the address of next instruction to be executed, this register is used to fetch an instruction without altering the execution of the current one. This pipelining mechanism is called *delayed pc.*
2. *Delayed program counter (dcp):* a 30 bit register for delayed program counter, containing the currently executed instruction. While the fetch of the next instruction is performed in the *pcp* register, the *dcp* is kept unchanged until the end of the execution of the current instruction.
3. *General purpose registers (gprs):* a register file consisting of 32 registers of 32 bits each. These registers are used in different operations, and can be addressed by their index (0–31). The first register is always set to 0.

4. *Special purpose registers (sprs):* a register file consisting of 32 registers of 32 bits each, used for particular tasks. The first register for instance is the *status* register, containing the interrupts masks. Some registers are used as flags registers or as condition registers. Each special purpose register is addressed directly by its name.
5. *Memory model (mm):* a 2^{32} bytes addressable memory. Different caching and virtual memory infrastructures are implemented in the VAMP system.

The transition relation is defined by the execution of the program instructions defined in the initial configuration. The VAMP implements the full DLX instruction set from Hennessy and Patterson [15]. This set includes load and store operations for double words, words, half words and bytes. It includes also different shift operations, jump-and-link operations and various arithmetic and logical operations.

To avoid the complex and inconvenient bit vector representation of data and instructions, an assembly language was introduced abstracting the VAMP ISA. In this case addresses are represented by natural numbers and registers and memory contents by integers. Our test specifications and experiments are based on this instruction set (assembler) model.

The Isabelle theory of the assembler model is an abstraction of the instruction set architecture. In addition to the representation of addresses as naturals and values as integers, some other ISA features are abstracted. The instructions are represented in an abstract datatype with *readable* names. The address translation is not visible at this level, assembler computations live in linear (virtual) memory space. Interrupts are not visible at this level as well. The assembler configuration is an abstraction of the ISA configuration, defined as a record type with the following fields:

- *pcp:* a natural number representing the program counter,
- *dcp:* a natural number representing the delayed program counter,
- *gprs:* a list of integers representing the general purpose register file,
- *sprs:* a list of integers representing the special purpose register file,
- *mm:* a memory model represented by a mapping from naturals to integers.

The HOL definition of the configuration is given by the $\texttt{ASMcore}_t$ record type. The register file type is defined as a list of integers representing the different registers.

```
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
```

Since the assembler representation of addresses and values is less restrictive than the bit vector representation, some conversion functions and restriction

predicates were defined to reduce the domain of addresses and values to only meaningful values. This was the case also for the configurations, since the number of registers is not mentioned in the definition of the registers type. The *well-fomedness* of assembler configurations is given by the `is_ASMcore` predicate. This predicate ensures that register files contain exactly 32 registers each. It also checks that all register and memory cells contain valid values.

```
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))
```

The instruction set of the assembler is defined as an abstract datatype `instr` in Isabelle. All operations mnemonics are used as datatype constructors, associated to their corresponding operands. Different types of instructions can be distinguished: data transfer commands, arithmetic and logical operations, test operations, shift operations, control operations and some basic interrupts.

```
datatype instr =
  -- {* data transfer (memory) *}
    Ilb regname regname immed
  |...
  -- {* data transfer (constant) *}
  | Ilhgi regname immed
  | ...
  -- {* data transfer (registers) *}
  | Imovs2i regname regname
  | ...
  -- {* arithmetic / logical operations *}
  | Iaddio regname regname immed
  | ...
  -- {* test operations *}
  | Iclri regname
  | ...
  -- {* shift operations *}
  | Islli regname regname shift_amount
  | ...
  -- {* control operations *}
  | Ibeqz regname immed
  | ...
  -- {* interrupt *}
  | Itrap immed
  | ...
```

An inductive function is defined over the assembler instructions to provide the semantics of each operation. This function returns for each configuration

and instruction, the configuration resulting from executing the instruction in the initial configuration.

```
fun exec_instr :: [ASMcore_t, instr] ⇒ASMcore_t
where
 -- {* Arithmetic Instructions *}
 exec_instr st (Iaddo RD RS1 RS2) =
      arith_exec st int_add (reg (gprs st) RS1)
                            (reg (gprs st) RS2) RD
| ...
 -- {* Logical Instructions *}
| exec_instr st (Iand RD RS1 RS2) =
      arith_exec st s_and (reg (gprs st) RS1)
                            (reg (gprs st) RS2) RD
| ...
 -- {* Shift Instructions *}
| exec_instr st (Isll RD RS1 RS2) =
      arith_exec st sllog (reg (gprs st) RS1)
                            (reg (gprs st) RS2) RD
| ...
```

The transition relation is defined as a function that takes a configuration and returns is successor. The transitions are defined by the execution of the current program instruction given in the delayed program counter.

```
definition Step :: ASMcore_t ⇒ASMcore_t
where   Step st ≡  exec_instr st (current_instr st)
```

These transition relations are used in our study as the basis of test specifications. The assembler model is more abstract than the processor model, consequently, different complex details are made transparent. Examples are interrupts handling and virtual memory and caching, pipelining and instruction reordering. In a black-box testing scenario, an abstract description of the system under test is used as a basis for test generation. This will be the case in our study, where the processor model is used to extract abstract test cases for the processor. The aim of this testing scenario is to check that the processor behaves as described in the assembler model, independently of the internal implementation details.

4 Testing VAMP Processor Conformance

As motivated earlier, we will apply essentially two testing scenarios: model-based *unit testing* and *sequence testing*. In a unit testing scenario, the test specification is described by pre- and post-conditions on the inputs and results produced by the system under test. This scenario assumes control over the initial state and the access to the internal states of the SUT after the test. In sequence testing scenario, only the control of the internal state initialization is necessary, and in some cases the reference to the final state. In principle, the test result is inferred from a sequence system inputs and observed outputs. For any given inputs and state, the system—defined as an i/o stepping function—may either

fail or produce outputs and a successor state. The unit testing scenario can be seen as a special form of (one step) sequence testing, where the output state is more or less completely accessible for the test.

In our case study, both testing scenarios are useful. The unit testing scenario will be used to test individually each operation or instruction with different data. Sequence testing will be used to test any sequence of instructions up to a given length. We will address subsets of related instructions separately, a combination of different instruction types is possible but not explored here. We studied four types of instructions: 1. memory related load and store operations, 2. arithmetic operations, 3. logic operations and 4. control-flow related operations.

4.1 Generalities on Model-Based Tests

A general test specification for unit instruction testing would be the following:

```
test_spec pre σ ι ⟹ SUT σ ι =k exec_instr σ ι
```

where $\_ =_k \_$ is a specially defined executable equality that compares the content of the registers and just the top k memory cells (instead of infinite memory). $\_ =_k \_$ is our standard conformance relation comparing the state controlled according to the model and the state controlled by the `SUT`; here, we make the testability assumption that we can trust our test environment that reads the external state and converts it to its abstraction. Note that `SUT` is a free variable that is replaced during the test execution with the system under test.

Each test case is composed of an instruction, an initial configuration and the resulting configuration after the execution of the instruction. From this test specification, HOL-TestGen will produce tests for all possible instructions.Subsets of instructions are isolated by adding a pre-condition in the test specification, specifying the type of the instruction.

For instruction sequence testing, based on the combinators from the state-exception monad (see Section 2.2) `mbind`, bind $\_ \leftarrow \_;\ \_$ and the assertion $\mathtt{assert}_{\mathtt{SE}}$ a test specification can be given specifications of *valid test sequences* from initial state σ_0. In general, there are two kinds of sequence test scenarios: those who involve just observations of the executions of the local steps and those who involve a test over the final state. The former class is irrelevant in our application domain since the local steps are just actions not reporting a computation result. However, the latter scenario may just involve a conformance on the entire state:

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

or just a bit of it, e. g., where a computation is finally loaded into register `0` which is finally compared:

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

which requires that the last load action(s) are tested before, but makes less assumptions over the execution environment (i. e., a trustworthy implementation

of $\_ =_{k\_}$). In both schemes σ_0 is the initial state and ιs is the sequence of instructions that will be generated and $\texttt{exec}_{\texttt{VAMP}}$ is a lifting of exec_instr into the state exception monad:

```
definition execVAMP where execVAMP ≡(λ i σ. Some ((), exec_instr σi))
```

The pre-conditions pre of our test specifications—also called test purposes—are added to the test specifications to reduce the generated instruction sequences to any given subset.

The initial configuration can also be generated as an input of the test cases. This may produce ill-formed configurations due to their abstract representation in the assembler model. We choose for our study to define and use an empty initial configuration σ_0 that is proved to be well-formed.

4.2 Testing Methodology

Common analysis techniques such as stuck-at-faults [14] are based on the idea that a given circuit design—thus, an implementation—is modified by mutators capturing a particular fabrication fault model, e. g.: one or n wires connecting gates in the circuit are broken. This can be seen conceptually as a white-box mutation technique and has, consequently, all advantages and all draw-backs of an implementation-based testing method compared to all draw-backs and all advantages to its specification-based counterparts. Stack-at-faults are very effective for medium-size circuits and use the structure of the given design to construct equivalence classes tests incorporating directly a fault model. This type of testing technique, however, will not reveal design flaws such as a write-read error under the influence of byte-alignments in the memory.

While we have a VAMP gate-level model in our hands and could have opted for testing technique on this layer, for this paper, we opted to stay on the design level of the VAMP machine. This does not mean that we can not refine with little effort the equivalence classes underlying our tests further: instead of assuming in our test hypothesis that "one write-read of a memory cell successful, thus all write-reads in this cell successful," one could force HOL-TestGen to generate finer test classes, by exploring the byte-or the bit-level representations of registers and memory cells.

4.3 Testing Load-Store Operations

To formalize a test purpose restricting our first test scenario to load and store operations, the test purpose is_load_store is used. This predicate returns for each instruction, if it is a load/store operation or not. It is defined just as a constraint over the syntax of the VAMP assembly language:

```
abbreviation is_load_store_byte' :: instr ⇒bool
where is_load_store_byte' iw ≡
    (∃ rd rs imm. (is_register rd ∧is_register rs ∧is_immediate imm) ∧
                iw ∈{Ilb rd rs imm, Ilbu rd rs imm, Isb rd rs imm})
```

```
definition is_load_store :: instr ⇒bool
where   is_load_store iw ≡is_load_store_word' iw
                         ∨is_load_store_hword' iw
                         ∨is_load_store_byte' iw
```

(the analogous test cases for `is_load_store_word'` and `is_load_store_hword' iw` are ommited here for space reasons).

Introducing this predicate in the pre-condition of the test specification reduces the domain of the generated tests to load/store operations. The resulting test specification formally stating the test goal for unit test scenario is given by the following:

```
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
```

The test case generation procedure defined in HOL-TestGen is used to preform an exhaustive case splitting on the instructions datatype. Symbolic operands are generated for each instruction to give a set of symbolic test cases. The test generation produced 8 symbolic test cases, corresponding to the different load and store operations. A uniformity hypothesis is stated on each symbolic test case, which will allow us to select one concrete *witness* for each symbolic test case. The final generation state contains 8 *schematic* test cases, associated to 8 uniformity hypotheses. The conjunction of the test cases and the uniformity hypotheses is called a test theorem.

An example of a generated test case and its associated uniformity hypothesis is given in the following. The variables starting with `??X` (e. g., `??X4`,`??X5`,) are schematic variables representing one possible witness value.

```
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) (...)))
```

The second phase of test generation is the test data instantiation. This is done using the **gen_test_data** command of HOL-TestGen. One possible resulting test case is given by the following:

```
SUT σ0(Ilb 1 0 1) σ1
```

where σ_1 is the expected final state after executing the given operation. With this kind of test cases, each operation is tested individually, in a unit test style. This kind of test will reveal design faults i. e.if the result of the operation is not correct. It also detects any undesired state modification, like changing some flags or registers.

In a similar way, load and store instruction sequences are characterized using the same predicate `is_load_store` which is generalized to entire input sequences to the combinator `list_all` from the HOL-library. Rather than using a fairly difficult to execute characterization in form of an automaton or an extended finite state-machine that introduce some form of symbolic trace, we use monadic

combinators of the state-exception monad directly to define valid test sequences constrained by suitable test purposes.

```
test_spec list_all is_load_store (ιs::instr list) ⟹
          (σ0 ⊨(s ←mbind ιs execVAMP; assertSE (λσ. σ=k SUT σ0ιs)))
apply (gen_test_cases SUT)
store_test_thm load_stre_instr_seq
```

Note that step two is just the call to the automatic test case generation method (declaring the free variable `SUT` as the system under test of this test case), and while the third command binds the results of this step to a data-structure called *test environment* with the name `load_store_instr_seq`. The experimental evaluation of this scenario is discussed in the next section.

One possible generated test case of length 3 is given by the following subgoal:

```
1.σ0 ⊨(s ←mbind [Isw ??X597 ??X586 ??X575, Ilbu ??X557 ??X546 ??X535,
                 Ilbu ??X517 ??X506 ??X595] execVAMP;
      assertSE (λσ. σ=k SUT σ0[Isw ??X597 ??X586 ??X575,
                               Ilbu ??X557 ??X546 ??X535,
                               Ilbu ??X517 ??X506 ??X595]))
2. THYP ((∃x1 x2 x3 x4 x5 x6 x7 x8 x9. σ0⊨(s ←mbind [Isw x1 x2 x3,
           Ilbu x4 x5 x6, Ilbu x7 x8 x9] execVAMP; (...))) ⟶
       (∀x1 x2 x3 x4 x5 x6 x7 x8 x9. σ0⊨(s ←mbind [Isw x1 x2 x3,
           Ilbu x4 x5 x6, Ilbu x7 x8 x9] execVAMP; (...))))
```

where the first subgoal gives the schematic test case, and the second subgoal states the uniformity hypothesis for this case.

The generation of test data is done similarly using the `gen_test_data` command, which instantiate the schematic variables with concrete values.

```
σ0 ⊨(s ←mbind [Isw 0 1 8, Ilbu 1 0 -3, Ilbu 3 2 8] execVAMP;
      assertSE (λσ. σ=k SUT σ0[Isw 0 1 8, Ilbu 1 0 -3, Ilbu 3 2 8]))
```

this corresponds to the following assembly code sequence:

```
ISW  0 1  8
LLBU 1 0 -3
LLBU 3 2  8
```

This test programs will eventually reveal errors related to read and write sequences. Even if each operation is realized in a correct way, the sequencing may contain errors, like errors due to byte alignment or information loss due to pipelining.

In this testing scenario, we consider test post-conditions expressed on the final state of the automaton. This post-condition is expressed using the state-exception primitive $\texttt{assert}_{\texttt{SE}}$. This scenario is not very realistic in hardware processors, because the final state, in particular the internal processor registers, will not be directly observable. An alternative scenario would be to consider the state-exception primitive `return` that introduces a step by step checking of the output values. This output value might be, e. g., retrieved from the updated memory cell. Test specification for this kind of scenarios is as follows:

```
test_spec list_all is_load_store ιs ⟹
        (σ0 ⊨(s ←mbind ι execVAMP'; return (SUT ι s)))
```

which require a modified VAMP where individual steps were wrapped into trusted code that makes, e. g., internal register content explicit.

4.4 Testing Arithmetic Operations

Similarly, we set up a unit test scenario, where we constrain by the test purpose `is_arith` the operations to be tested to arithmetic ones:

```
test_spec σ= exec_instr σ0i ⟹is_arith i ⟹SUT σ0i σ
apply (gen_test_cases 0 1 SUT)
store_test_thm arith_instr
```

At this stage, each arithmetic operation is covered by one generated test case, an example is given in the following:

```
1. SUT σ0(Iaddi ??X277 ??X266 ??X255) (...)
```

which contains a test case for he addition operation.

A note on the test granularity is at place here: as such, the granularity that HOL-TestGen applies to test arithmetic operations is fairly coarse: just one value satisfying all constraints over a variable of type integer is selected. This is a consequence of our model (registers were represented as integers and not as bitvectors of type: `32 word` which would be (nowadays) a valuable alternative) as well as the HOL-TestGen heuristics to select for each variable just one candidate. The standard workaround would be to introduce in the test purpose definitions more case distinctions, e. g., by `x ∈{MinInt} ∪{-50 .. -100} ∪{0} ∪{50 .. -100}` which result in finer constraints for each of which a solution in the test selection must be found.

The sequence scenario is analogously:

```
test_spec list_all is_arith (ι::instr list) ⟹
            (σ0 ⊨(s ←mbind ιs execVAMP; assertSE (λσ. σ=k SUT σ0ι)))
apply (gen_test_cases SUT)
store_test_thm arith_instr_seq
```

A possible generated sequence is given in the following, resulting from the `gen_test_data` command.

```
σ0 ⊨(s ←mbind [Isub 2 1 0, Iadd 1 5 2, Iadd 1 0 4] execVAMP;
      assertSE (λσ. σ=k SUT σ0[Isub 2 1 0, Iadd 1 5 2, Iadd 1 0 4]))
```

which corresponds to the following assembly code sequence:

```
ISUB 2 1 0
IADD 1 5 2
IADD 1 0 4
```

This sequence corresponds to a subtraction followed by two addition operations.

4.5 Testing Control-Flow Related Operations

Also with branching operations we are following the same theme:

```
test_spec is_branch i ⟹SUT σ0i =k exec_instr σ0i
apply (gen_test_cases 0 1 SUT)
store_test_thm branch_instr
```

This generates unit test cases for branching operations starting from the initial sate σ_0. One example of the generated schematic test cases is given by:

```
1. SUT σ0(Ijalr ??X27X7) (...)
```

The problem with this scenario is that the initial state is fixed, while the branching operations behavior depends essentially on the flag values. A more interesting scenario would be to consider different initial states, where the flags values are changed for each test case.

In the test sequence generation, the test specification is given as follows:

```
test_spec list_all is_branch (ιs::instr list) ⟹
            (σ0 ⊨(s ←mbind ιs execVAMP; assertSE (λσ. σ=k SUT σ0ιs)))
apply (gen_test_cases SUT)
store_test_thm branch_instr_seq
```

The test sequence and test data generation returns, e. g., this concrete test sequence:

```
σ0 ⊨(s ←mbind [Ij 1, Ijalr 0] execVAMP;
       assertSE (λσ. σ=k SUT σ0[Ij 1, Ijalr 0]))
```

which corresponds to the following assembly code sequence:

```
IJ    1
IJALR 0
```

The test data generation in all the considered scenarios is performed by constraint solving and random instantiation. This leads to test sequences with coarsely grained memory access. As such, an underlying fault-model is somewhat arcane (i. e., interferences of operations in distant memory areas). If one is interested in such faults, a more dense test method should be chosen.

Rather, one would adding additional constraints to reduce the uniformity domain again. One could simply bound the range of addresses to be used in test sequences, or define a used-predicate over input sequences that computes the set of addresses that store-operations write to, and constrain the load-operations to this set, or the like. This kind of constraints can also be used to improve the coverage of our selected data, by dividing the uniformity domain into different interesting sub-domains.

5 Experiences and First Experimental Data

Methodologically, we deliberately refrained in this paper to modify the model—we took it "as is," and added derived rules to make it executable in test scenarios

where we assume a reference implementation running against the SUT. For example, the model describes padding functions for bytes, words, and long-words treating the most significant bit differently in certain load and store operations; in the semantic machine model as it was developed in the Verisoft Project, there are comparisons on these padding *functions* themselves—this is possible in HOL, but in no functional executable language, had therefore to be replaced by equivalent formulations exploiting the fact there are only three variants of padding functions, thus a finite number, were actually *used* in the VAMP machine. Another issue is the linear memory in the machine (a total, infinite function from natural numbers to memory cells, i. e., long words); comparisons on memory, as arising in tests where the real state has to be compared against the specified state, had to be weakened to finitized conformance relations.

While as a whole, our approach is done in a pretty generic model-based testing framework, a few adaptions had to be made due to some specialties of this model. For example, since the assembly language has 56 variants, case-splitting over the language explodes fast over the length of test sequences. While sequence tests are methodologically and pragmatically more desirable (less control over the state is assumed), they are therefore more vulnerable to state-space explosion: sequences of length 3 generate at some point of the process $56+56^2+56^3 = 178808$ cases. In this situation, a few heuristic adaptions (represented on the tactic level) and more significantly, constraints on the level of the test purposes had to be imposed with respect to state-space explosion, test purposes like `list_all is_logic` ι helps to reduce the test sequences to $7+7^2+7^3$, i. e., a perfectly manageable size (see discussion in the next section).

5.1 Test Generation

As mentioned earlier, we opted for a combination of unit and sequence test scenarios. Unit tests have the drawback of imposing stronger assumptions on testability: it is assumed that the test driver has actually access to registers and memory (which essentially boils down to the fact that we trust code in the test driver that consists of store-operations of registers into the memory). Sequence tests rely on the observed behavior of tests and make weaker assumptions on testability, for the price of being more vulnerable to state-space explosion.

The sequence scenarios on load-and store operations in Section 4.3 uses 39 seconds in the test partitioning phase and 42 seconds in the test data selection phase (measurements were made on a Powerbook with a 2.8 Ghz Intel Core 2 Duo). 1170 subgoals were generated, where one third are explicit test hypothesis and two third are actual test cases. The other scenarios in Section 4.5, Section 4.4 and the more basic Section 4.1 use considerably less time (between two and twenty seconds for the entire process).

5.2 Test Execution

Nevertheless, compile time for the model (as part of the test drivers) was less than a second; compilation of the entire test driver in SML depends, of course,

drastically on the size of finally generated tests. Since we restrained via test purposes the test cases in each individual scenario to about 1000, the compile time for a test remained below 3 seconds. Scaling up our test plan is essentially playing with a number of control parameters; however this is usually done only at the end of the test plan development for reasons of convenience.

Our study focuses for the moment on *test generations*; we did not do any experiments against hardware so far. However, there is a hardware-simulator in the sources of the Verisoft-project; in the future, we plan to generate mutants of this simulator and get thus experimental data on the bug-detection capabilities on the generated test sets.

To give an idea on how the test cases will be executed, we did some experiments using the generated executable model. Starting from the abstract model, an executable translation of it in SML is performed using the Isabelle's code-generation facilities. This generated code contains all the type and constant definitions that are needed to execute the different assembler operations on an executable state. A sketch of the generated SML code for the VAMP processor is given in the following:

```
structure VAMP : sig
  datatype num = One | Bit0 of num | Bit1 of num
  datatype 'a set = Set of 'a list | Coset of 'a list
  datatype instr = Ilb of IntInf.int * IntInf.int * IntInf.int |
    ...
    Ijr of IntInf.int | Itrap of IntInf.int | Irfe
  val int_add : IntInf.int -> IntInf.int -> IntInf.int
  val int_sub : IntInf.int -> IntInf.int -> IntInf.int
  val cell2data : IntInf.int -> IntInf.int
  val exec_instr : unit aSMcore_t_ext -> instr -> unit aSMcore_t_ext
  val sigma_0 : unit aSMcore_t_ext
  val execInstrs : unit aSMcore_t_ext -> instr list
                                      -> unit aSMcore_t_ext
   ...
```

where the datatype definition `instr` is generated from the instruction type definition introduced in Section 3. the functions definitions are generated from their corresponding constants and functions defined in the model.

Our fist experiment was the application of the generated test cases on this executable model. Using the HOL-TestGen test script generation, two test scripts were generated for load/store and arithmetic operations sequence. For both cases, 585 test cases were generated and then transformed to executable testers. Running all these tests did, obviously, not reveal any error, since the same model was used for test generation and execution.

To evaluate the quality our generated test cases, we introduced some changes to the executable model, producing a mutant model. Three changes were introduced in the `int_add`, `int_sub` and `cell2data` operations of the generated SML code. In this case, a majority of tests detected the errors. For testing the arithmetic operations, we obtained:

```
Number of successful test cases: 303 of 585 (ca. 51%)
Number of warning:                 0 of 585 (ca.  0%)
Number of errors:                  0 of 585 (ca.  0%)
Number of failures:              282 of 585 (ca. 49%)
Number of fatal errors:            0 of 585 (ca.  0%)
```

For testing the load/store operations, we obtained:

```
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%)
```

6 Conclusion and Related Work

6.1 Related Work

Formal verification is widely used in the hardware industry since at least ten years (e. g., [4, 12, 13, 21, 23]). Nevertheless, formal models of complete processors as well as verification approaches that provide an end-to-end verification from the application layer to the hardware design layer are rare. Besides VAMP [10], notably, exceptions are Fox [12] and Appenzeller and Kuehlmann [1]. The closest related work with respect to the processor model is Fox [12] to which our approach should be directly applicable.

Similarly, test program generation approaches for microprocessor instruction sets have been known for a long time (e. g., [11, 17, 19, 22]). Among them manual approaches based on informal descriptions of the instruction set such as Fallah and Takayama [11] or random testing approaches such as Shen et al. [22]. Only a few works suggest to use model-based or specification-based test program generation algorithms, e. g., Kamkin et al. [17] and Mishra and Dutt [19]. These works have in common that they are based on dedicated test models that are independently developed from the verification models. Mishra and Dutt [19] is the most closely related work; the authors are using the explicit state model checker SMV to generate test programs from a dedicated test model for SMV that concentrates on pipelining faults. In contrast, our approach seamlessly integrates the test program generation into an existing verification tool chain, re-using existing verification models.

6.2 Conclusion and Future Work

We presented an approach for testing the conformance of a processor with respect to an abstract model that captures the instruction set (i. e., the assembly-level) of the processor. This abstraction level is particular important as, first, it is the level of detail that is usually available for commercial off-the-shelf (COTS) processors and, second, it is the target level of high-level compilers.

Thus, our approach can, on the one hand, support the certification of the COTS processors for which the manufacturer is neither willing to certify the processor itself or to disclose the necessary internal details. Moreover, our approach helps to bridge the gap between the software layer (e. g., in avionics requiring certification according to DO-178 [16]) and the hardware layer (e. g., in avionics requiring certification according to DO-254 [16]).

As (embedded) systems combining hardware and software components for providing core functionality in safety critical systems (e. g., "fly-by-wire") are used more and more often, we see an increasing need for validation techniques that seamlessly bridge the gap between hardware and software. Consequently, we see this area as the utterly important one for future work: providing a test case generation methodology that can be applied end-to-end in the development process and allows for validating each development step. These test cases, called *certification kits*, are required even if compilers and processors are formally verified: The system builders require them for proving, as part of their certification process, that their are applying the tools correctly (i. e., according to their specification).

Acknowledgement. This work was partially supported by the Euro-MILS project funded by the European Union's Programme [FP7/2007-2013] under grant agreement number ICT-318353.

References

[1] D.P. Appenzeller and A. Kuehlmann. Formal verification of a powerpc microprocessor. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD '95. Proceedings., 1995 IEEE International Conference on*, pages 79–84, oct 1995. doi: 10.1109/ICCD.1995.528794.

[2] Sven Beyer. *Putting it all together - Formal Verification of the VAMP*. PhD thesis, Saarland University, SaarbrÃijcken, Germany, 2005.

[3] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together – formal verification of the vamp. *Int. J. Softw. Tools Technol. Transf.*, 8(4):411–430, August 2006. ISSN 1433-2779.

[4] P. Biswas, A. Freeman, K. Yamada, N. Nakagawa, and K. Uchiyama. Functional verification of the superscalar sh-4 microprocessor. In *Compcon '97. Proceedings, IEEE*, pages 115–120, feb 1997. doi: 10.1109/CMPCON.1997.584682.

[5] Achim D. Brucker and Burkhart Wolff. HOL-TestGen: An interactive test-case generation framework. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, 2009. doi: 10.1007/978-3-642-00593-0_28.

[6] Achim D. Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing (FAC)*, 2012. ISSN 0934-5043. doi: 10.1007/s00165-012-0222-y.

[7] Achim D. Brucker, Lukas Brügger, Matthias P. Krieger, and Burkhart Wolff. HOL-TestGen 1.7.0 user guide. Technical Report 1551, Laboratoire en Recherche en Informatique (LRI), Université Paris-Sud 11, France, April 2012.

[8] Common Criteria. Common criteria for information technology security evaluation (version 3.1), Part 3: Security assurance components, September 2006. Available as document CCMB-2006-09-003.

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.

[10] Jan Dorrenbacher. *Formal Specification and Verification of Microkernel*. PhD thesis, Saarland University, SaarbrÃijcken, Germany, 2010.

[11] F. Fallah and K. Takayama. A new functional test program generation methodology. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 76–81, 2001. doi: 10.1109/ICCD.2001.955006.

[12] Anthony C. J. Fox. Formal specification and verification of arm6. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 2003. ISBN 3-540-40664-6.

[13] John Harrison. Formal verification at intel. In *LICS*, pages 45–. IEEE Computer Society, 2003. ISBN 0-7695-1884-2. doi: 10.1109/LICS.2003.1210044.

[14] John P. Hayes. Fault modeling for digital mos integrated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 3(3):200–208, 1984. ISSN 0278-0070. doi: 10.1109/TCAD.1984.1270076.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123704901.

[16] Vance Hilderman and Tony Baghai. *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)*. Avionics Communications Inc., 2007. ISBN 978-1-885544-25-4.

[17] Alexander Kamkin, Eugene Kornykhin, and Dmitry Vorobyev. Reconfigurable model-based test program generator for microprocessors. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:47–54, 2011. doi: 10.1109/ICSTW.2011.35.

[18] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814.

[19] Prabhat Mishra and Nikil Dutt. Specification-driven directed test generation for validation of pipelined processors. *ACM Trans. Design Autom. Electr. Syst.*, 13 (3), 2008.

[20] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002. doi: 10.1007/3-540-45949-9.

[21] David M. Russinoff. A mechanically checked proof of correctness of the amd k5 floating point square root microcode. *Formal Methods in System Design*, 14(1): 75–125, 1999.

[22] Haihua Shen, Lin Ma, and Heng Zhang. Crpg: a configurable random test-program generator for microprocessors. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4171–4174 Vol. 4, may 2005. doi: 10.1109/ISCAS.2005.1465550.

[23] Sudarshan K. Srinivasan and Miroslav N. Velev. Formal verification of an intel xscale processor model with scoreboarding, specialized execution pipelines, and impress data-memory exceptions. In *MEMOCODE*, pages 65–74. IEEE Computer Society, 2003. ISBN 0-7695-1923-7. doi: 10.1109/MEMCOD.2003.1210090.

[24] Makarius Wenzel and Burkhart Wolff. Building formal method tools in the Isabelle/Isar framework. In Klaus Schneider and Jens Brandt, editors, *TPHOLS 2007*, number 4732 in Lecture Notes in Computer Science, pages 352–367. Springer-Verlag, Heidelberg, 2007. doi: 10.1007/978-3-540-74591-4_26.