Randomised testing of a microprocessor model using SMT-solver state generation

Randomised testing of a microprocessor model using SMT-solver state generation

Brian Campbell, Ian Stark

LFCS, School of Informatics, University of Edinburgh, UK

Abstract

We validate a HOL4 model of the ARM Cortex-M0 microcontroller core by testing the model's behaviour on randomly chosen instructions against real chips from several manufacturers.

The model and our intended application involve precise timing information about instruction execution, but the implementations are pipelined, so checking the behaviour of single instructions would not give us sufficient confidence in the model. Thus we test the model using sequences of randomly chosen instructions.

The main challenge is to meet the constraints on the initial and intermediate execution states: we must ensure that memory accesses are in range and that we respect restrictions on the instructions. By careful transformation of these constraints an off-the-shelf SMT solver can be used to find suitable states for executing test sequences. We also use additional constraints to test our hypotheses about the timing anomalies encountered.

Keywords: Randomised testing, microprocessor models, HOL, SMT

1. Introduction

Mechanised formal models of instruction set architectures provide a basis for low-level verification of software. Obtaining accurate models can be difficult; most architectures are described in large reference manuals consisting primarily of prose backed up by semi-formal pseudo-code. Once a model is produced it is common to test randomly chosen individual instructions against hardware to gain confidence in the model (for example, [1]). However, pipelined processors deal with multiple instructions simultaneously and some effects due to interactions in the pipeline may require a sequence of several instructions to appear.

These effects are relevant to the intended application of our model. We wish to extend existing low-level verification work using Myreen's decompilation [2] to include timing properties. The decompilation transforms raw machine code into a higher-level language (such as definitions in HOL), which can be easier to reason about. The proposed extension would also lift timing information about small sections of the machine code into the higher-level language, in a similar but simpler manner to previous work using a certified compiler [3]. We want to use a realistic processor with a relatively simple cost model, because our technique is largely orthogonal to the low-level timing analysis and we do not wish to spend resources recreating a complex worst-case execution time analysis (such as those surveyed in [4]).

While the ARM Cortex-M0 has a simple cost model which provides timing in terms of clock cycles per instruction, even this design has a non-trivial microarchitecture in the form of a three stage pipeline (fetch, decode and execute). Hence to build confidence in our model, and in particular the timing information contained in the model, we wish to test sequences of instructions in order to exercise the pipeline. In particular, we test randomly chosen instruction sequences because we do not have any knowledge of the internal implementation of the core, so we treat it as a black box.

Email addresses: Brian.Campbell@ed.ac.uk (Brian Campbell), Ian.Stark@ed.ac.uk (Ian Stark)

 The formal publication is in Elsevier's Science of Computer Programming, doi:10.1016/j.scico.2015.10.012

```
ldrsh   r0, [r1, r2]   ;  load r0 from r1+r2 (16 bits, sign-extended)
lsls    r0, r0, r2     ;  shift r0 left by r2
bcs     +#12           ;  branch if carry set
  (may jump due to branch)
add     r0, r0, r2     ;  add r2 to r0
ldr     r3, [r0, #0]   ;  load r3 from r0
```

Figure 1: An example test sequence of M0 code

However, finding a processor state in which an arbitrary sequence of instructions can be executed without faulting is not always easy. Consider the sequence in Figure 1. The final instruction reads a word from memory at an address stored in the `r0` register. We must ensure that this address is a valid location in memory, which makes up a tiny fraction of the processor's address space.

Moreover, `r0`'s value at the final instruction is a result of several operations: a half-word load, sign-extension, shift and bitvector addition, during which `r2`'s value is used in several different ways. If we were to express these operations as a term in a suitable logic, we could write a formula for the valid location problem as a constraint on the initial register and memory values. Finding solutions to such constraints is a natural application for an SMT (Satisfiability Modulo Theories) solver with good bitvector support.

The solutions provided by the solver will specify the relevant parts of the registers and memory, including the placement of instructions. For example, if the `bcs` branch is taken then the next instruction must be placed 12 bytes later. Whether the processor takes the branch is also decided by a non-trivial calculation due to the shift instruction; again, we leave this to the SMT solver.

To ensure successful execution of our test sequences we need to solve constraints from two sources; most, such as instruction alignment, come from the model, which describes the behaviour of individual instructions as theorems about the model. The hypotheses to these theorems are the requirements to avoid faulting, and the conclusions tell us how the state changes, which we use to express the requirements in terms of the initial state. However, the model is not intended to be comprehensive, especially where details about particular implementations are concerned. Thus we need to add additional constraints to capture these details, such as the size and layout of memory. The requirement for a valid address in the example above is an example of an additional constraint.

The testing system is outlined in Figure 2. The model is written in the L3, a specification language for processor instruction sets [5], and the generated HOL version is accompanied by tool libraries, shown in the dashed box. Instruction generation is separated from the main testing code, and can be overridden manually. The set of constraints for the instruction sequence is constructed in HOL, then an SMT solver is invoked to help find a pre-state using an adaption of the HolSmtLib library [6] to translate the constraints into the solver's logic, and the hardware is invoked over a USB link. The testing system is fully automatic—however there is no formal guarantee that it will be able to test every feasible instruction sequence. For example, the SMT solver could time-out on a particularly convoluted example, but the testing system would log the failure for manual examination, and continue with further tests. Our software is available online[1].

Our contributions are to demonstrate that symbolic evaluation with SMT constraint solving is an effective way to test a formalised microprocessor model over sequences of instructions, and that the ability to add additional constraints is useful for bridging the gap between a model and an implementation, and for checking hypotheses about deviations from the model.

Section 2 introduces the main pieces of software that we use, including the M0 model and accompanying tools, then the description of the testing system begins with the generation of instructions in Section 3 and continues in Section 4 with the construction of pre-states which satisfy the requirements for successful execution. Section 5 discusses the practicalities of running the tests on the hardware, and Section 6 presents the outcomes of testing. We consider the results and variations of the system in Section 7, then consider future and related work before concluding.

[1] `https://bitbucket.org/bacam/m0-validation`

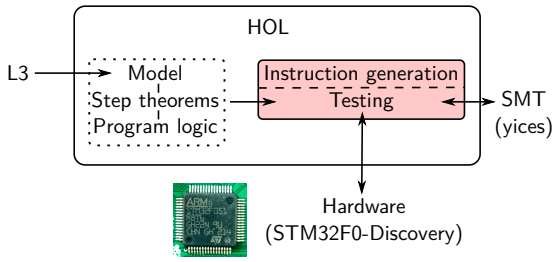

Figure 2: Outline of the testing system

This is a revised and extended version of an earlier conference paper [7]. In particular, much more detail is provided about the construction of the constraints that we need to solve, and more information is provided on why some sequences are impossible to test, our experience of testing a range of microcontrollers from several manufacturers and of testing the hypotheses we formed about the timing anomalies we observed.

2. Preliminaries

We begin by describing the tools that we use during testing, including the model itself. First, we outline the proof assistant environment that we work in, then the processor model, and finally the SMT solver and the proof assistant's interface to it.

2.1. The HOL4 proof assistant

HOL4 [8] is a proof assistant for higher-order logic. It is implemented in the Standard ML programming language, which also provides the main user interface to the system, continuing ML's original purpose as the 'Meta Language' to extend the system. It uses a small logical kernel protected by an abstract datatype to ensure the soundness of the system. On top of this kernel a suite of libraries provides a rich set of types and theories, including the ability to define new inductive datatypes and records.

Thus, Standard ML also provides a good setting to manipulate formal models described in HOL, and to interact with external components (in our case, a debugger interface to microcontroller hardware) to compare predictions obtained from a model to a real system. HOL4 supplements Standard ML with extendable parsing for producing terms in the logic. Functions can be defined in HOL by giving a set of defining equations (essentially equivalent to definitions by pattern matching in ML), allowing models based on functional programs to be embedded directly into the logic.

To support the examples in this paper, Figure 3 presents a small fragment of HOL4 syntax that is used in the examples. Logical statements, such as true, false and equality, have type `bool`. Thus theorems consist of a list of hypotheses and a conclusion, all of which are terms of type `bool`. In the following sections we denote free variables with italics, and free variables in theorems are implicitly universally quantified. Note that in addition to the natural numbers, HOL4 has a library of fixed-size bitvectors, and that the same syntax is used for the arithmetic operators of both types. The processor model uses maps to represent the register file and memory, which are implemented in HOL as ordinary functions together with the `=+` syntactic sugar to make map updates more readable.

2.2. The processor model

The ARM Cortex-M0 is the smallest processor core in ARM's range of microcontroller cores. It features 32-bit words and 32-bit addressing, although typical implementations only have tens of kilobytes of SRAM and hundreds of kilobytes of flash memory at most. Several manufacturers produce microcontrollers featuring the M0 core, adding their own buses, memory and peripherals.

The model of the Cortex-M0's instruction set and timing that we use was developed by Anthony Fox in his L3 domain specific language [5]. It is a greatly simplified adaption of his ARMv7 model, with the

Theorems: [$hypothesis_1$, ..., $hypothesis_n$] ⊢ *conclusion*

Types:

`bool`	booleans
α `option`	optional values of type α
α `->` β	functions from α to β
α_1 `#` ... `#` α_n	tuples
`num`	natural numbers
`word8` `word16` `word32`	bitvectors
identifier	inductive datatypes and records

Terms:

`T` `F`	boolean literals, true and false
term `:` *type*	type constraint
identifier	variables, constants, datatype constructors
(*term*$_1$, ..., *term*$_n$)	tuples
$\forall$*x* `:` *type*`.` *term*	universal quantification (*type* is optional)
λ*x* `:` *type*`.` *term*	function abstraction (*type* is optional)
term$_1$ *term*$_2$	function application
term$_1$ `=` *term*$_2$	equality
¬*term* *term*$_1$ ∧ *term*$_2$ *term*$_1$ ∨ *term*$_2$	boolean negation, conjunction, disjunction
`if` *term*$_1$ `then` *term*$_2$ `else` *term*$_3$	conditionals
`let` *x* `=` *term*$_1$ `in` *term*$_2$	local definition
`let` (*x*,*y*) `=` *term*$_1$ `in` *term*$_2$	break up a pair
`NONE` `SOME` *term*	optional values
`THE` *term*	strip `SOME` from an option value
`case` *term* `of` *identifier*$_1$ x_1 ... x_n `=>` *term*$_1$ `|` ...	case-split on inductive datatype
`<|` *field*$_1$ `:=` *term*$_1$`;` ...`;` *field*$_n$ `:=` *term*$_n$ `|>`	record literal
term `with` *field* `:=` *term*$_1$	record update (1 field)
term `with <|` *field*$_1$ `:=` *term*$_1$`;` ...`;` *field*$_n$ `:=` *term*$_n$ `|>`	record update (n fields)
term`.`*field*	record projection
`123`	natural number literal
`SUC` *term*	the natural number *term* + 1
`123w` `0x7Bw`	bitvector literal
`+` `-` `*` `DIV` `<` `>` `<=` `>=`	arithmetic operators (overloaded)
`~`*term*	bitvector inversion
term$_1$ `@@` *term*$_2$	bitvector concatenation
term$_1$ `<<~` *term*$_2$	bitvector logical shift
`sw2sw` *term* `w2w` *term*	signed and unsigned bitvector resizing
`w2n` *term* `n2w` *term*	convert bitvector to/from natural number
(*m* `><` *n*) *term*	extract bits n to m of bitvector *term*
(*term*$_1$ `=+` *term*$_2$) *term*$_3$	update map *term*$_3$ with *term*$_1$ ↦ *term*$_2$

Figure 3: HOL4 syntax used in this paper

```
dfn'LoadWord (t,n,m) =
(λstate.
   (let (v,s) =
          case m of
            immediate_form imm32 => (imm32,state)
          | register_form m =>
              Shift (FST (R m state),SRType_LSL,0,state.PSR.C)
                state
    in
    let (v,s) = MemU (FST (R n s) + v,4) s in
    let s = SND (IncPC () (SND (write'R (v,t) s)))
    in
      ((),s with count := s.count + 2)))
```

Figure 4: Core HOL4 function for the load register instruction, `ldr`

addition of instruction cycle timings from the Cortex-M0 reference manual [9, §3.3]. These timings only depend on the instruction and whether execution branches; no complex resource modelling is required. The only caveat the manual places on these timings is that the memory needs to be sufficiently fast ('zero wait-states'), which is true for the SRAM of all of the chips we consider, and also for the flash memory at lower clock speeds. We use the SRAM throughout to keep the model and testing process simple.

L3 provides a specification language with imperative features that allows models to closely follow the pseudo-code typically found in such manuals. In particular, it has some global state which is used to model the processor state, and a simple exception mechanism to indicate that the behaviour of the processor in the given state is not defined by the model. An automatic translation produces a version for HOL4, together with Standard ML versions of the instruction decoder and encoder. The imperative features are replaced by threading a record holding all of the global processor state, plus a field to record any L3 exception that is raised, through each of the function definitions.

Thus the HOL version of the model consists of a collection of executable function definitions. The main interface to the Cortex-M0 model is a next step function,

```
NextStateM0 : m0_state -> m0_state option
```

where the type `m0_state` is the global state record containing register values, memory content, flags, and other miscellaneous information about the processor state. The return value is optional because the model can raise an L3 exception when a step is unspecified. The memory is represented as a HOL function from 32-bit words representing addresses to 8-bit contents. This is an idealisation; it treats all of the address space as if it were writable memory, whereas actual implementations have small areas of SRAM, larger areas of flash, some memory mapped I/O and large amounts of unmapped address space.

As a running example of how the semantics of an instruction are represented in the model, consider the final instruction from Figure 1,

`ldr r3, [r0, #0]` load 32-bit word into register `r3` from the address in register `r0`.

The main function for loading a word in the HOL version of the model is shown in Figure 4. This function was automatically translated by the L3 tool into HOL and is shown verbatim, except for some changes to the layout. The first three arguments come from the instruction decoder, and denote the target register t, address register n and offset calculation m. Hence, for this instruction we have

t = `3w`, n = `0w`, m = `immediate_form 0w`

where `immediate_form` is a datatype constructor indicating that the value should be directly added to the address, rather than denoting a register containing an offset. The remaining argument, *state*, is the current

```
 [Aligned (s.REG RName_PC,2),      s.MEM (s.REG RName_PC) = 3w,
  Aligned (s.REG RName_0 + 0w,4), s.MEM (s.REG RName_PC + 1w) = 104w,
  ¬s.AIRCR.ENDIANNESS, ¬s.CONTROL.SPSEL, s.exception = NoException]
⊢ NextStateM0 s = SOME (s with
      <|REG := (RName_PC =+ s.REG RName_PC + 2w)
               ((RName_3 =+ s.MEM (s.REG RName_0 + 0w + 3w) @@
                            s.MEM (s.REG RName_0 + 0w + 2w) @@
                            s.MEM (s.REG RName_0 + 0w + 1w) @@
                            s.MEM (s.REG RName_0 + 0w)) s.REG);
        count := s.count + 2; pcinc := 2w|>)
```

Figure 5: Example step theorem for `ldr r3,[r0,#0]`

processor state represented by a record[2].

Even though the conversion has threaded the state through the definition, it remains close to the pseudo-code in the reference manual [10, §A6.7.28]. For example, the pseudo-code also has a `MemU` function to perform the memory access, including checking the alignment of the address and causing a processor fault if it is misaligned. The `write'R` function updates the register file in the state record with the newly-read value, and the `IncPC` function updates the program counter. The timing information has been added as the final state update in the last line, increasing a virtual clock in the `count` field by the 2 clock cycles the load would take according to the technical reference manual.

Additional tools for using the model are provided in two HOL libraries. The first, `stepLib`, provides symbolic evaluation of the step function for individual instructions. An example is shown in Figure 5. The result is presented as a theorem, which maintains a formal link between the original model and this presentation of the model's behaviour on a particular instruction. The hypotheses (the list of terms before the ⊢) assert that the program counter and source address are correctly aligned, that the instruction is present in memory and that certain control flags are set correctly. Note that many of these details did not appear in the main load function in Figure 4, because they are spread throughout other parts of the model, such as the instruction fetch and memory access functions. The `stepLib` library cuts across the model and gathers all of the requirements in one place.

The conclusion states that the model's step function will succeed with an updated state, where the program counter is moved forward, the result of the load is present and two ticks of the processor's clock have passed. By presenting the behaviour of the instruction in this state-update form it becomes much easier to reason about the effect of individual instructions without having to look at the implementation details of the actual model. Indeed, this presentation of instruction behaviour is much closer to giving an inductive-style operational semantics to the instruction set.

For branches, which are the only conditional instructions in the ARMv6-M architecture, `stepLib` returns two theorems: one for when the branch is taken, and one for when it is not.

The second library provides separation-logic-like specifications for instructions, and is built upon the first library. The principal intended use for the model is to provide low-level verification using Myreen's decompilation technique [2], and these specifications provide the interface between the model and the decompilation library by acting as a program logic.

We could attempt to build the testing system on the model directly, or using one of these two libraries. In order to determine what constraints on the state are necessary for successful execution we need to perform symbolic evaluation of the instructions, and the first library essentially does this for individual instructions. If we went further and tried to use the separation-logic specifications we would have to make memory aliasing decisions before we can obtain the preconditions (in addition to any problems we might have translating the preconditions into the SMT solver's logic). Thus by testing `stepLib` we obtain the symbolic evaluation and can leave the aliasing decisions to the SMT solver.

[2]The state variable is introduced by a λ rather than as a formal argument as a side effect of the way the transformation from L3 to HOL threads the state through the definition.

For most L3 models these libraries only consider a subset of the behaviour of the processor's instructions that has been used in verification projects. For example, the corresponding model for the ARMv7 architecture features unaligned loads, but the libraries leave these out because none of the code considered for verification uses them and it makes the library simpler. However, for the M0 we have unusually high coverage because of the small size and limited variety of the instruction set — unaligned loads are not present at all on the M0, for instance.

The model does not currently support interrupts and exceptions, which are not necessary for the verification work we intend to do in the near future. Hence we leave these for further work, briefly discussing them in Section 7.

2.3. SMT solving from HOL4

The model described above is a first-order, easily executable definition, as is the view of the model provided for individual instructions by `stepLib`. This suggests that automated theorem proving could be used to solve the constraints such as those that appear in step theorems. Moreover, the types of objects involved in the model—booleans, natural numbers, bitvectors, maps—and the operations used on those objects are all within the reach of many SMT solvers.

We build upon the existing HolSmtLib package by Weber [6], which supports Yices 1 and Z3. We chose to use Yices because HolSmtLib's Yices translation supports a wider variety of HOL types and terms, although we expect that a similar translation would work well with other solvers.

Normally, HolSmtLib proves goals where the free variables are universally quantified. To use an SMT solver in this way the library must negate the goal and check that that is *unsatisfiable*. We thus have to adapt the library, because we wish to check that the constraints are *satisfiable*, treating the free variables as *existentially* quantified and using the satisfying assignment returned for them to construct the pre-state for testing. Hence we adapted the library so that it does not negate the constraints, and to parse the satisfying assignments and translate them back into HOL terms.

The main translation from HOL terms to Yices terms is a straightforward syntax-directed replacement of HOL operations with their equivalents in Yices, together with a similar translation on types. There is no sophisticated encoding of terms which rely on the higher-order nature of HOL, instead the problem presented to the library should consist of supported first-order terms. A few complications arise during translation: parametrised types such as α `option` must be duplicated for every concrete type they are instantiated with, and a few Yices functions are declared to overcome differences in the semantics of the HOL and Yices operators; full details can be found in Weber [6].

Yices, and the translation from HOL, support theories for equalities, uninterpreted functions (such as the maps the model uses for memory), linear arithmetic, fixed-sized bitvectors, quantifiers, recursive datatypes and functions, and lambda expressions. Explicit quantifiers do not occur in our model, and we avoid introducing them because Yices is not complete on formulae with quantifiers and so they increase the chance that the solver will return 'unknown'. We also do not need lambda expressions, and the datatypes and functions from the model are not recursive.

3. Instruction sequence generation

We produce instruction sequences by randomly picking instructions that are supported by the model. The M0's instruction set is a subset of ARM's compact 'Thumb' instruction set. This subset is fairly small; the reference manual lists 77 instructions [10, §A6.7] of which the model only supports 63, user-level, instructions. Thus we take the opportunity to provide a fresh list of the instruction formats so that we can also cross check them against the model. In Section 7 we will consider alternative approaches.

Figure 6 gives a Standard ML datatype for fragments of M0 instruction formats and a list of a few sample formats. While the datatype has a few generic constructors for literal and immediate bit strings, the rest are specialised for targeting the M0. In particular, registers come in three-bit and four-bit representations[3], with several further variants for pairs of registers and specific instructions.

[3] Many instruction formats only use `r0` to `r7` to keep the instructions compact.

```
datatype instr_format =  Lit of int list      | Imm of int
    | Reg3             | Reg4NotPC            | Reg4NotPCPair
    | CmpRegs          | RegList of bool (* inc PC/LR *)
    | STMRegs          | Cond                 | BLdispl

val instrs = [
  ( 1, ([Lit [0,0,0,1,1,0,0], Reg3, Reg3, Reg3],          "ADD (reg) T1")),
  (14, ([Lit [1,1,0,1], Cond, Imm 8],                     "B T1")),
  ( 1, ([Lit [0,1,0,0,0,1,1,1,0], Reg4NotPC, Lit [0,0,0]], "BX")),
  ( 1, ([Lit [1,1,1,1,0], BLdispl],                       "BL")),
  ...
```

Figure 6: Language for instruction formats and sample formats

The entries in the `instrs` list are triplets of a weight, a format and a name. Each format consists of a list of fragments. For example, the `ADD` instruction format in Figure 6 has a constant prefix, the destination register and two source registers. The integer before the format is the weighting, which is used here to make the conditional branches appear more often than other instructions. A name is given to each format for debugging and logging purposes.

The Branch with Link (`BL`) instruction format is the most unusual. It is the only instruction supported by the model that is 32-bits long (the rest are system instructions) and the offsets for the jump are very large (up to 16MB) compared to the amount of memory available (kilobytes). Thus almost all `BL` instructions are unusable, and hence untestable, on the M0 because they must jump outside of the mapped region of address space and fault. To restrict our attention to the useful `BL` instructions, the `BLdispl` fragment picks and encodes branch displacements with magnitude bounded by the size of the available SRAM.

3.1. Consistency checks

We perform several automated tests of our generator. The first is an internal sanity check which ensures that every format is either 16 or 32 bits long to detect obvious format encoding errors early.

To cross check the set of instructions produced by the generator against the set supported by the model, we split the list of instruction formats into the instructions that we expect the model to support, and those that we believe the model does not. We then ask the model for theorems about several instances of each format, ensuring that they are present for all supported instructions and absent for all other instructions. Any deviation indicates that there is either an encoding error in the generator, a decoding error in the model, or an error in the model's behaviour—these can be diagnosed and resolved manually. We also check that only the conditional branches generate multiple theorems.

Following the discovery of an extra instruction in the model, we ensured that no further extra instructions were present by performing a manual check of the format list against `stepLib`. It may be possible to devise automated checks of this, but the small instruction set makes a manual comparison the most effective use of time. More details, and the other testing results, can be found in Section 6.

4. Generating pre-states for testing

Having chosen a sequence of instructions we can obtain theorems describing the behaviour of each one from `stepLib`. To keep the following process manageable we randomly pick whether to take each branch so that we only have one step theorem for each instruction[4]. Now we wish to combine them into a single theorem giving the model's requirements and prediction for the entire sequence of instructions. First we define a HOL function for executing n steps of the model:

[4]This will lower the rate of successful tests because sometimes only one direction of the branch is compatible with the other instructions in the sequence. However, both choices are equally likely, so it does not affect the potential coverage of testing.

```
NStates 0 s = s
NStates (SUC n) s = NStates n (THE (NextStateM0 s)))
```

In general `stepLib` will provide a theorem of the form

$$hyps_i \vdash \texttt{NextStateM0 } s \texttt{ = SOME } (s \texttt{ with <| } updates_i \texttt{ |>})$$

for the ith instruction. Suppose we already have

$$hyps \vdash \texttt{NStates } n \ s \texttt{ = } s \texttt{ with <| } updates \texttt{ |>}$$

then by unfolding the definition of `NStates (SUC` n`)` we can use the `stepLib` theorem to prove

$$hyps_i \cup hyps[s \mapsto s \texttt{ with <| } updates_i \texttt{ |>}] \vdash \\ \texttt{NStates (SUC } n\texttt{) } s \texttt{ = } (s \texttt{ with <| } updates_i \texttt{ |>}) \texttt{ with <| } updates \texttt{ |>}$$

where $[s \mapsto term]$ is the syntactic substitution of *term* everywhere that the free variable s appears. Note that we do not need to perform the proof interactively, but can write a Standard ML function to automatically construct it from the step theorem. Applying HOL's computation and simplification libraries will remove irrelevant parts of hypotheses and updates, and tidy up the conclusion into a single set of updates again. We can repeat this process for each instruction in the test sequence, starting from the final instruction and working backwards. In the resulting theorem, all of the hypotheses are in terms of the free variable s, which refers to the initial state, and so these form the set of constraints from the model that we have to solve. The conclusion gives the model's prediction of the final state after the sequence is executed, in terms of record updates applied to the initial state.

To give a more concrete example, recall the sequence in Figure 1. We saw the step theorem for an `ldr` instruction in Figure 5. After combining the step theorems the conditions become more complex due to the symbolic execution. Let us consider the requirement from the `ldr` instruction that the address for the load in `r0` is aligned, which is one of the hypotheses in the step theorem:

```
Aligned (s.REG RName_0 + 0w,4)
```

The preceding instruction is `add r0,r0,r2`. The conclusion of the step theorem for `add` updates `r0`:

```
... ⊢ NextStateM0 s =
  SOME (s with <|REG := (RName_PC =+ s.REG RName_PC + 2w)
                        ((RName_0 =+ s.REG RName_0 + s.REG RName_2) s.REG); ... |>)
```

So after substitution and simplification, our alignment hypothesis becomes

```
Aligned (s.REG RName_0 + s.REG RName_2,4)
```

where s is now the state before the execution of the `add` instruction. We can continue to do this for each of the other instructions. Once all of the instructions have been combined, the alignment hypothesis is expressed in terms of the initial state, reflecting how the final value of `r0` is calculated:

```
Aligned (s.REG RName_2 +
         sw2sw (s.MEM (s.REG RName_1 + s.REG RName_2 + 1w) @@
                s.MEM (s.REG RName_1 + s.REG RName_2))
         <<~ w2w ((7 >< 0) (s.REG RName_2)), 4)
```

where `(7 >< 0)` selects the low byte of `r2` and `<<~` performs the shift.

These hypotheses make up the bulk of the constraints the pre-state must satisfy, but we must also meet additional requirements that are imposed by the hardware, and translate the constraints into a form suitable for the solver.

4.1. Additional requirements

For successful execution on the device, the state must satisfy requirements about areas where the model is deliberately incomplete (often because they vary between implementations):

- self-modifying code must be forbidden;

- the memory map and its restrictions must be respected;
- a test harness is required to stop execution cleanly; and
- the model's implicit invariant that stack pointers are always aligned must be enforced.

The last point is the result of starting execution from a state loaded by the debugger; the model (and the manual's pseudo-code) establish the alignment of the stack pointers on reset and maintain it throughout execution. This invariant is implicit—the model does not check that it holds inside the main `NextStateM0` function, but the hardware will reject states that break it when we attempt to set them up using the debugger. As we start from a state chosen by constraint solving rather than the normal state after reset, we cannot use the part of the model for resetting the processor, but instead add constraints to ensure the invariant holds.

To implement the restrictions on self-modification and the memory map we need to know the symbolic positions of every instruction and memory access. Recovering this information after combining the step theorems would be difficult at best. For example, if we throw away the result of a load by using the same destination, `r0`,

```
ldr r0, [r1, #0]
ldr r0, [r2, #0]
```

then the use of the address in `r1` will disappear in the combined theorem. To capture and preserve this information, we add extra hypotheses to each step theorem that record the symbolic addresses for the set of instruction locations and accessed memory locations. These hypotheses take the form of an assertion that there is a map (called *instr_start* and *memory_address*, respectively) whose nth value is the location we wish to record. The symbolic value for the instruction location is given by the program counter, and we can find all memory accesses by finding the terms which consult the memory field of the state, s.MEM.

For example, for the final instruction in the example sequence we add a hypothesis to the step theorem in Figure 5 for the instruction location, two for the memory access for the instruction, and another four for the word that is loaded:

```
instr_start 4 = s.REG RName_PC,
memory_address 4 = s.REG RName_PC,
memory_address 5 = s.REG RName_PC + 1w,
memory_address 0 = s.REG RName_0,
memory_address 1 = s.REG RName_0 + 1w,
memory_address 2 = s.REG RName_0 + 2w,
memory_address 3 = s.REG RName_0 + 3w,
```

Combining the extended step theorems as before performs symbolic evaluation of the earlier instructions, restating these addresses in terms of the initial state:

```
instr_start 4 = s.REG RName_PC + 8w,
memory_address 4 = s.REG RName_PC + 8w,
memory_address 5 = s.REG RName_PC + 9w,
memory_address 0 = s.REG RName_2 +
                   sw2sw (s.MEM (s.REG RName_1 + s.REG RName_2 + 1w) @@
                          s.MEM (s.REG RName_1 + s.REG RName_2))
                   <<~ w2w ((7 >< 0) (s.REG RName_2)),
...
```

The remaining addresses are similar. We can then add constraints requiring all accesses to be in the range of the target device's SRAM. To eliminate self-modification we can discover the symbolic locations written to by examining the memory field of the prediction for the whole instruction sequence, then constrain them to be disjoint from the instructions by adding an extra hypothesis to the combined theorem.

By extracting the expression for the post-state program counter we can synthesise locations for the harness instructions, and so add constraints requiring them to be present and remain unmodified during execution. In most of our tests the harness consists of a single software breakpoint instruction[5].

[5]Hardware breakpoints would also work, but are slightly harder to use on our main target device.

Constraint	Source	Example
PC alignment	model	`Aligned (`s`.REG RName_PC,2)`
Instruction present	model	s`.MEM (`s`.REG RName_PC) = 3w` $\wedge$ s`.MEM (`s`.REG RName_PC + 1w) = 104w`
Alignment of memory accesses	model	`Aligned (`s`.REG RName_0 + 0w,4)`
Processor configuration	model	$\neg s$`.AIRCR.ENDIANNESS` (little endian)
Memory map constraints	tester	*memory_address* `1 >= 0x20000000w` $\wedge$ *memory_address* `1 < 0x20004000w`
No self-modifying code	tester	*write_address* `0 <` *instr_start* `0` $\vee$ *write_address* `0 >=` *instr_start* `0 + 2w`
Testing harness	tester	Similar to 'instruction present' and 'no self-modifying code' constraints
Stack pointer invariant	tester	`(1 >< 0) (`s`.REG RName_SP_main) = 0w`

Table 1: Sources of generated constraints

A summary of the types of constraint and their sources is in Table 1.

4.2. Impossible instruction sequences

Not all of the test sequences generated are actually useful tests. In Section 3 we noted that the instruction generator avoids Branch with Link instructions with large offsets because the branch target would be outside of memory and execution would fail, but not all impossible sequences can be ruled out this way.

An avoidable source of impossible sequences is our decision to pick whether to take branches during instruction generation rather than leaving the choice to the SMT solver. For example, if we generate two successive branches on the carry flag,

```
bcs  +#12
bcs  +#12
```

and the randomly generated choices for branching are to take the first, but not the second, then we will generate incompatible constraints on the carry flag, because there is no intervening code which could change the flag. As we will see in Section 6.1, impossible sequences from all causes were a small fraction of all generated sequences, so while it would be possible to avoid these particular cases we preferred to avoid the additional complexity required.

Other test sequences are impossible because the particular combination of instructions produce unsatisfiable constraints. Some instructions have restrictions on operands that preceding instructions prevent. For example, the sequence

```
ldrb  r1, [r0,#0]
str   r2, [r1,#0]
```

loads a single byte into the `r1` register, then uses it as an address to store a word at. However, there are no valid SRAM locations in the range of a byte (0–255 are mapped to flash memory) so the constraint that `r1` is a valid address is unsatisfiable. The same problem is also visible with instruction locations, for example by jumping to such an address. Moreover, while the generator rules out a single Branch with Link instruction with an offset too large to fit in memory, a sequence of them can still have the same effect:

```
bl  +#2048
...
bl  +#2048
```

Finally, some instructions impose restrictions on their operands. Unlike the addresses above, these constraints are provided by the model rather than added afterwards. For example, the sequence

```
lsls  r0, r0, #1
bx    r0
```

consists of a shift that always clears bit 0, and a Branch with Exchange instruction that requires bit 0 to be set (on other ARM architectures it selects which instruction set to use, but the M0 only has the Thumb instruction set). We reject this pair of instructions because of these contradictory constraints, and indeed they cannot be executed on an M0 processor without faulting.

4.3. Practical HOL

The technique outlined above of combining step theorems by instantiation and simplification is intended to keep the symbolic state manageable. Initial test runs revealed two further issues that we needed to address to achieve this, after which we were able to run large scale testing without further difficulty.

The main problem was that when a memory update is distributed to the uses of memory by simplification, subsequent memory lookups will be partially evaluated into aliasing checks. These grow quickly, especially when load-multiple and store-multiple instructions are involved; every address written to is compared against every address read. We could attempt to avoid exposing these aliasing checks by curtailing evaluation during simplification, but we chose the more compact solution of introducing free variables for the intermediate memory states. The updates from store instructions are moved to hypotheses which constrain the free variable to be exactly the intermediate memory state.

For example, storing a single byte with `strb r0, [r1,#0]` updates the memory field of the state record,

```
s with <|MEM := (s.REG RName_1 =+ (7 >< 0) (s.REG RName_0)) s.MEM;
         REG := (RName_PC =+ s.REG RName_PC + 2w) s.REG;
         count := s.count + 2; pcinc := 2w|>
```

so we add the new hypothesis

```
Abbrev (mem_step_0 = (s.REG RName_1 =+ (7 >< 0) (s.REG RName_0)) s.MEM)
```

using HOL's `Abbrev` mechanism to prevent the definition being used during simplification, and put the variable in the state record:

```
s with <|MEM := mem_step_0;
         REG := (RName_PC =+ s.REG RName_PC + 2w) s.REG;
         count := s.count + 2; pcinc := 2w|>
```

The second problem is that unconstrained evaluation will reveal implementation details about the model and HOL libraries, which it is important to avoid for the SMT translation described in the next section, and to keep the size of terms reasonable. Restricting the computation rules used during evaluation prevents this, along with careful choices of simplification and rewrite rules.

4.4. Constraint translation and SMT solving

We introduced the HolSmtLib-based interface to the Yices SMT solver in Section 2.3. However, the subset of HOL supported by the translation to Yices' input language is still rather small, and the definitions used in the model do not always fit within it. For example, the `Aligned` predicate is defined by:

```
Aligned (w,n) = (w = n2w (n * (w2n w DIV n)))
```

This is not well supported as it switches between bitvectors and natural numbers using `n2w` and `w2n`, which are not included in Yices. Thus during the process of combining the step theorems above we are careful not to unfold definitions like `Aligned`. Instead, we use HOL theorems about them to rewrite them into the supported subset. For example, the model's library already provides the result

```
⊢ (∀a : word32. Aligned (a, 1)) ∧
  (∀a : word32. Aligned (a, 2) = ~word_lsb a) ∧
  (∀a : word32. Aligned (a, 4) = ((1 >< 0) a = 0w:word2)) ∧
  (∀a : word8.  Aligned (a, 4) = ((1 >< 0) a = 0w:word2))
```

which provides a bitvector interpretation for all of the necessary cases. We prove simple results to ensure that bit selection, shifts and addition are also in the expected form, and slightly more complex ones to obtain the overflow and carry bits for addition.

By using HOL theorems we know that the transformations are *sound*; the worst case for an error is that we end up with constraints that the SMT solver cannot handle. We also extended the transformation itself for a few terms that are awkward to deal with in HOL, either due to the requirement for tedious correctness proofs, or because we want to use a Yices operator that the translation does not support. We do this for right rotation, map updates, 8-bit bitvector to natural conversion, and variable bit indexing. The latter two are defined in a brute-force fashion by large `if-then-else` trees, but perform well in practice. These do not benefit from any soundness guarantees, but are not critical to the soundness of the testing procedure as a whole because we check in HOL that the generated assignments satisfy the hypotheses from the model when we use them.

To find the above set of rewrites that are required to target the SMT-friendly subset of HOL we performed a survey of the step theorems produced by the model. For a randomly chosen set of instructions from each instruction format we attempted to translate the step theorem into Yices' input language. As each unsupported definition was discovered, we added a new rewrite, until no more were found.

One example of this is the carry bit from the `lsls` instruction in the example, where it decides whether the branch is taken. The step theorem from the model calculates it from the registers,

```
if w2n ((7 >< 0) (s.REG RName_2)) = 0
  then s.PSR.C
  else testbit 32 (shiftl (w2v (s.REG RName_0)) (w2n ((7 >< 0) (s.REG RName_2))))
```

saying that the old value is used if no shift is done, otherwise the correct bit is extracted from `r0`. However, the shift and test are done using *bitstrings* rather than bitvectors, a different but related HOL4 type which is not supported by the SMT translation. This was discovered during the survey of instruction formats described above. Thus we proved a small theorem,

```
∀x : word32. testbit 32 (shiftl (w2v x) n) =
               if (n > 0) ∧ (n <= 32) then x ' (32 - n) else F
```

that expresses the shift and test as a bit selection, and added it to the set of rewrites applied to the step theorems. The resulting theorems still need some of our extensions to the Yices translation. Note that these transformations can be sensitive to changes in the model; in particular, if more step theorems exposed bitstring operations like this, then we may need to perform more rewriting.

Once all of the constraints have been translated, Yices is invoked and if successful returns a satisfying partial assignment. We can represent this partial assignment in HOL as a series of updates over a free variable for each field of the state record. Each free variable represents all of the unspecified contents of that field. In this way we can form a partial pre-state from the solver's results; for our example this is (eliding some irrelevant details):

```
<| MEM :=
    (0x20000000w =+ 136w) ((0x20000001w =+ 94w) ((0x20000002w =+ 144w)
   ((0x20000003w =+ 64w)  ((0x20000004w =+ 4w)  ((0x20000005w =+ 210w)
   ((0x20000006w =+ 16w)  ((0x20000007w =+ 68w) ((0x20000008w =+ 3w)
   ((0x20000009w =+ 104w) ((0x2000000Aw =+ 0w)  ((0x2000000Bw =+ 190w)
   ((0x200002F0w =+ 7w)   ((0x200002F1w =+ 0w)  rand_mem))))))))))));
   PSR := rand_flags with C := F;
   REG := (RName_PC =+ 0x20000000w) ((RName_1 =+ 0xFFFFE9F8w)
         ((RName_2  =+ 0x200018F8w) rand_regs));
   count := 0 |>
```

To obtain a full pre-state we instantiate each of the *rand_* free variables with random data, essentially filling up all of the unspecified memory locations, registers and flags to create a complete state.

5. Test execution and comparison

Now that we have a full pre-state that the instruction sequence should run in, we can obtain a prediction of the post-state and compare it against the result of running the test on the hardware. The complete process is shown in Figure 7.

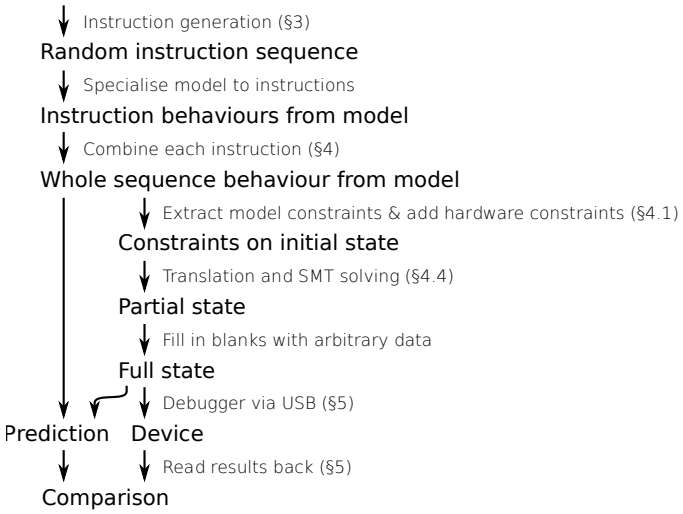

Figure 7: The complete testing process

The prediction can be constructed by instantiating the combined theorem that describes the behaviour of the whole instruction sequence with the pre-state we have discovered, which produces a concrete post-state to compare against the device. This step is complicated slightly by the abstraction of the intermediate memory values (Section 4.3) and the size of the term describing the whole memory. For the former we progressively instantiate each variable using the definition we recorded in the hypothesis, then use evaluation to make the next memory value concrete. For the latter we split up the state record before supplying the concrete values; this prevents duplication of the large memory term in several places where it is projected away.

At the end of this process we should obtain a theorem with no hypotheses giving the model's concrete prediction of the behaviour of the device. By checking that the hypotheses have been discharged we do not have to trust the SMT translation and solver.

To perform the test we used the OpenOCD debugging tool [11] to write the pre-state to the microcontroller on the development board and start execution. To encode the processor flags into the binary Application Processor Status Register (APSR) we use the HOL function provided by the model. If the test is successful, the device will halt at the breakpoint instruction in the test harness. The same tool is used to read the post-state back from the board.

Precise cycle timings are obtained by directly setting up and reading back the device's SysTick counter timer from the debugger. This timer only ticks while the processor is running, so it stops once the breakpoint is reached. We then have to correct for the time spent in the harness. However, the reference manual does not provide a cycle count for breakpoints, but experimentally we measured a consistent overhead of 3 cycles. (This matches the reasonable hypothesis that there will be 2 cycles overhead to fill the pipeline at the start, and one more at the end to execute the breakpoint.)

The post-state retrieved from the board is then compared to the model's prediction, checking the register contents (adjusting the PC for the extra instructions in the test harness), the SRAM, the APSR and the cycle count.

When a test fails due to an unexpected difference between the processor and the model that affects control flow (for example, by not taking a branch when expected, or a processor fault due to a bad memory access) then the execution may be derailed and fail to reach the breakpoint. These cases will still be detected because the debugger can read back the state of the processor regardless, and the comparison will fail because the program counter will be incorrect. Usually other parts of the state will also be wrong too; in particular, the cycle counter is likely to be extremely large because execution did not halt.

Manufacturer	Model	Main SRAM size	Timing Anomalies	
			end-of-memory	odd-halfword-store
ST	STM32F051R8T6	8kB	yes	no
NXP	LPC11U14	4kB	yes[a]	yes
Infineon	XMC1100	16kB	yes	no
Cypress	CY8C4125AXI-483	4kB	yes	yes

[a]Masked by undocumented memory mapping

Table 2: ARM Cortex-M0 based microcontrollers under test

5.1. Scaling up testing

To perform large test runs, and to debug the testing code, we designed logging with several features in mind:

- Ease of rerunning tests, if necessary with the same random background data for memory, flags and registers.
- Classification into successes and failures (impossible combination of instructions, SMT solver returned 'unknown', exception due to bug in model or test code, or genuine mismatch).
- Data about the differences between the post-state and prediction from the model.
- Text that can be read manually.

This is implemented in a straightforward manner, where there is one file per classification with one line per test. Each line contains a numerical identifier, the instruction sequence, branch choices, the test harness used, and details about the failure, if any. The random data used in each test case is stored in a file named using the identifier, so that it can be reproduced precisely when necessary. Instruction placement and profiling of the testing code were added to the logs later to provide more detail.

6. Outcomes from testing

First, the consistency tests for Section 3's instruction generation revealed a missing instruction pattern for `ldrsb` in `stepLib`, with the result that `ldrsb` instructions were present in the model but code using them could not be verified. A few minor bugs when generating step theorems were also found.

Recall that these consistency tests do not detect *extra* instructions in the model. Our manual check for these was prompted by Fox discovering one such instruction. Fortunately, as we can bypass the instruction generation we could immediately test for this instruction and confirm its absence from the device.

Similarly, there are alternative forms of the `bx` and `blx` instructions which the reference manual marks as UNPREDICTABLE, i.e., they might work but should not be used. L3 already features the syntax for indicating this, but doesn't act upon it, so the variants are present in the model. Again, if we bypass the instruction generator we can test these variants, and we found that a few of these instructions behave normally on the hardware we have.

Moving on to the results from the randomised testing, we tried sequences of 5 to 10 instructions long on four different microcontrollers based on the Cortex-M0, each from a different manufacturer. Details of each microcontroller can be found in Table 2; we will discuss the anomalies found below. We also varied the clock speed on some of the cores, observing that it made no difference to the results, as expected when running from SRAM.

Test failures with `bx` and `blx` instructions uncovered a bug in the model where the check on the Thumb mode bit was reversed. The lack of support for self-modifying code in the model can be seen if we turn off the additional constraints we generate to prevent it, as can the invariant about the alignment of the stack pointer.

Finally, we encountered two distinct anomalies with the timing behaviour of the chips. First, there is an extra single-cycle penalty whenever instructions in the last word of memory are executed. This is consistent with the processor's pipeline design: executing an instruction from the last word implies attempting to fetch the next word, but this does not exist. Presumably it is handling this corner case that consumes an extra processor cycle.

This effect was seen on all of the microcontrollers examined. However, on the NXP chip the SRAM is mapped into the address space several times consecutively, and the anomaly only appears in the last, unofficial, mapping. The first mapping, which is the only documented one, is not affected, presumably because the fetch of the next instruction will succeed with the first word of SRAM.

The other anomaly only affects the NXP and Cypress chips. Any store instruction which is located in an odd half-word of memory takes an extra cycle, so long as the write is to anywhere in the same block of memory. This is consistent with an interaction between the store and the subsequent instruction fetch, which occur on adjacent cycles. Indeed, the documentation for ARM's system design kit describes an SRAM controller which uses a one place buffer to avoid a wait-state in this kind of situation [12, §3.7].

To identify these anomalies we examined several test cases which produced them, including where the SMT solver had chosen to place instructions in each case, and looked for patterns that might explain the effects observed.

Our ability to constrain various aspects of the test generation helped investigate the hypotheses we formed about the anomalies. We were able to add extra constraints to avoid or force certain conditions, for example,

```
instr_start (some_instr:num) = 0x20001ffew : word32
some_instr < 5
```

to ensure that one of the 5 instructions is in the last word of memory on the ST chip over a large number of tests. Similarly we can constrain all of the instructions to be elsewhere. We can also specify a particular sequence of instructions to be tested, allowing us to narrow down test cases to minimal examples, and to test more elaborate constructions such as loops (see Section 7 for more details).

Our last method of confirming our hypotheses was to write scripts to examine the output from a test run and adjust the prediction from the model accordingly. This provided additional evidence that both of our anomalies are caused by instruction fetches: tests that feature *both* anomalies next to one another only incur *one* single cycle penalty, suggesting that the same instruction fetch is responsible.

6.1. Performance

To give a general indication of the performance of the system, we performed a test run of 1000 sequences of 5 instructions for this paper. We frequently used this configuration during testing, because it was large enough to execute several instructions with a full pipeline and good performance. Of the 1000, 105 had no possible pre-state, 882 matched the model's prediction, and 13 did not match; each instruction format was used in at least 34 viable tests. All of the non-matching cases differed only in execution time, and had an instruction in the last word. None of the successful tests did.

To compare the effects of increasing the sequence length, n, we present the per-test rate of impossible cases and mean time in seconds for each stage from runs of 200 instructions:

n	Impossible	Generation	Combination	Pre-state	SMT	Instantiation	Testing
5	0.105	0.037	0.769	2.418	0.313	4.795	3.272
7	0.145	0.053	1.585	5.909	1.101	8.662	3.437
9	0.320	0.038	2.917	11.239	5.114	12.956	3.516

The rate of impossible combinations increases as the probability of incompatible instructions and branch choices increases. The main time costs are the stages involving the full 8kB of memory; it may be possible to reduce this by restricting the memory to the test's footprint. All of the issues described above were discovered with sequences of 5 instructions, and increasing the sequence length did not uncover any further problems.

These were measured on a dual-core 8GB Intel Core i5-3320M using a development version of HOL4 from March 2014 running on PolyML 5.5.1. The generated HOL and SML for the M0 model and tools is present in the HOL4 distribution[6]. An STM32F0-Discovery board was used as the target. Testing with the other chips has similar performance, with minor variations due to differing amounts of SRAM.

7. Discussion

For a simple microprocessor like the Cortex-M0 we can ask whether testing sequences of instructions rather than individual instructions is worth the extra effort. Indeed, the mistakes in the model and tools could have been detected by single instructions, and even the timing anomalies could be detected despite appearing to be the result of the processor's pipeline.

However, it is only through testing sequences that we know this. While the timing anomalies may involve the pipeline they only depend on the location of the instruction, not on which particular instructions are in the decode and fetch stages. By way of contrast, the Cortex-M3 processor also has a three stage pipeline but has a pipelining feature for some consecutive loads and stores which reduces their total execution time [13, §3.3.2]. Our test sequences can demonstrate that the M0 does not have an undocumented version of this feature, whereas single instruction tests cannot because single instructions would never benefit from it.

Moreover, by adjusting the additional constraints that we add we can also witness the lack of support for self-modifying code, and if we wanted to extend the model to support that, or to support timing when executing from slower memory (such as the flash memory on the STM32F0), only sequences of instructions would explore the relevant behaviour.

The form of the generated code. We generate the sequences of instructions to be executed by picking each instruction individually. We do not expect this code to reflect real application code, but one feature that is worth considering in detail is the likelihood of generating loops. First, note that the execution cannot stop inside a loop because a breakpoint instruction is used to halt execution. More importantly, we must generate the same instructions several times with an appropriate branch. For example, we manually tested the following sequence to show that the extra timing penalty for placing instructions in the last word of memory could build up over time, only taking the `bcs` branch to the breakpoint at the end:

```
run_test_code debug
  ‘adds    r0, r0, r2     bcs     -#12     b       -#4
   adds    r0, r0, r2     bcs     -#12     b       -#4
   adds    r0, r0, r2     bcs     -#12‘
  (SOME [0, 1, 0, 0, 1, 0, 0, 0])
  (Basic Breakpoint) [‘‘instr_start 2 = 0x20001ffew : word32‘‘];
```

The SMT solver then picks a state in which the loop runs the correct number of times. However, the chances of generating a sequence like this randomly are vanishingly small due to the repetition of the loop body. One possible area for future work is to produce structural features like loops in conjunction with the instruction generation.

Generating instructions directly from the model. By writing our own instruction generator we duplicated some of the information contained in the model: the set and encodings of supported instructions. To apply the testing more widely it would be helpful to use the model directly. (Unsupported instructions are less of an issue because missing instructions do not affect the soundness of verification.)

L3 models feature a datatype for instructions, a decoding function and (optionally) an encoding function. An easy approach is to generate members of the datatype, then use the encoding function to produce binary code to test. However, we have few guarantees about these functions: the most we can expect is that decoding reverses encoding. The opposite may not be true.

[6] In the directory `examples/l3-machine-code/m0`.

For example, if we took this approach for the M0 model we would still not test the extra UNPREDICTABLE forms of the branch instructions, because they have the same representation in the datatype as the normal version, so the encoder will not produce them, but the decoder handles them.

We suggest two possibilities to consider: analysing the decoder function to guide generation of binary instructions; or generating the decoder and encoder functions from a single, more abstract, definition. An example of the latter approach would be a more principled version of our instruction format language.

There is another alternative which is suitable for some targets, including the M0: the Thumb instruction set is sufficiently dense that you can simply pick values at random, then check whether it is a valid instruction. The downside to this approach is that you cannot bias which instructions are chosen or the values used (such as the branch distance for `bl` instructions mentioned in Section 3).

The form of generated pre-states. The parts of the state most relevant to the execution are provided by the SMT-solver. These are certainly not randomly sampled, but can be pleasingly daemonic: for example, we have seen useful biases towards reusing locations and using the top and bottom of SRAM, which reveal self-modifying code (if allowed) and the timing anomaly we found. If we wished to generate a wider variety of pre-states we could investigate adding further constraints to force the solver to behave differently.

Potential extensions. More complete microprocessor models include system features such as interrupts and exceptions. We expect our model-driven approach would adapt well to these because we can generate tests where the event occurs on a particular instruction. For example, to produce a fault on a load instruction the SMT solver can be asked to ensure that the given address is invalid, and to interrupt at a given instruction the timing knowledge can be used to initialise a timer. Examining the behaviour of sequences of instructions would be vital in this context; even a relatively simple processor like the Cortex-M0 has complex interrupt handling features such as nesting and tail chaining.

8. Future work

In the preceding discussion we outlined some broad areas of possible work and alternative implementation decisions. Now we consider more concrete proposals.

We are particularly interested in how easily the testing code can be adapted to other instruction set architectures described by L3 models. Our intent is to isolate code specific to the M0 behind reasonable interfaces. The instruction generator is already in this form, so that it can be bypassed to test manually chosen sequences of instructions. Much of the rest of the M0 details are HOL types and terms, and practical information about hardware such as the SRAM location and size. It is less clear whether the HOL rewrites used to place constraints into an SMT-friendly form are closely tied to the M0 definition, but we can reuse the systematic survey of instructions from Section 4.4 to identify new problems.

However, in Section 2.2 we noted that we could only use the `stepLib` library to provide symbolic evaluation of individual Cortex-M0 instructions because the library covers all of the relevant behaviour. Other models generated by L3 have libraries which deal with a restricted set of instructions intended for use in verification, or may not even have such libraries at all. We have a range of options: merely test the supported behaviour, extend (or implement from scratch) a library for each architecture, or attempt to perform the symbolic evaluation directly from the model. The latter option may be the most demanding, but is less restricted than the step library because it only has to be useful for providing constraints to the SMT solver, whereas the existing libraries must fit into the other verification machinery.

We have carried out some preliminary experiments testing a MIPS model written in L3 against an experimental processor design on the supported instructions in its `stepLib` library. This has been successful, requiring only a few extra rewrites and special handling for MIPS' delay slots, and found problems in the model and implementation. To cover the whole processor, including the experimental features, we are currently investigating the latter approach of direct symbolic evaluation of the model.

A second area of work is to investigate the coverage of the model achieved by the generated tests. Simple measurements, such as how many pairs (or any n-way combination or n-gram) of instruction formats appear together in some test, could be performed independently of the test suite generation. We checked the

simplest form of this in Section 6.1, where we noted that every instruction format was used in at least 34 tests.

However, the number of instruction formats used is a poor proxy for coverage of the model, and we would like to apply standard code coverage measurements as well. Fortunately, the L3 tool can also produce SML code for a simulator as well as HOL4 definitions, so we intend to experiment with the support for coverage checking in MLton's profiler by rerunning our generated tests under the SML simulator.

Finally, we would like to investigate whether rejected test sequences can be automatically characterised to ensure that they were rejected for a valid reason (that is, those listed in Section 4.2), rather than (for example) a bug in the constraint generation or SMT solver. At present we check a randomly chosen sample of rejected sequences by hand, and checking coverage of the model would provide some additional confidence that we are not excluding good sequences, but these might not detect rarely triggered errors. One possible automated approach is to rerun the SMT solver with relaxed versions of the constraints to identify genuine problems. For example, we might omit some of the memory range checks to see if the instructions cannot generate a valid address for a load or store.

9. Related work

The combination of symbolic evaluation and constraint solving in automatic test case generation is already common, particularly in *concolic* tools [14]. These take a *con*crete trace of an existing test case, and perform sym*bolic* evaluation to find constraints on the input that will force the execution of some unexplored path in the program. A solution to these constraints, if one can be found, is combined with the old test to form a new test case, increasing the coverage of the suite. This is a form of 'white-box' fuzz testing.

Bounimova et al. [15] have described deploying their SAGE concolic fuzzer at scale to test Microsoft products, using the Z3 SMT solver for constraint solving. The tracing and symbolic execution is performed on x86 instructions, allowing them to test the final product code. The generated tests are then run against a runtime checker for various properties, including bad memory accesses. They also undertook a great deal of engineering to scale up to mass testing on a large variety of products, although we found that even the modest effort of Section 5.1 was extremely useful for understanding and debugging our relatively small number of tests. In particular, their principle of recording every run precisely enough to reproduce it quickly and exactly is also critical for us.

We cannot perform white-box testing techniques like these because we are concerned about the behaviour of the processors as well as the model, and do not have access to their design. However, just as concolic testing ensures that new test cases reach some new part of the program, we use symbolic evaluation and constraint solving to ensure that we reach the end of the sequence of instructions that make up our test case. In short, concolic testing is usually used to increase coverage, whereas we use similar techniques for fault-free execution.

Wagstaff et al. [16] have a test suite generation tool that is closer to our system. Like us, they start from a relatively formal description of a processor's instruction set architecture, but they wish to generate a small test suite with high coverage of the generated simulator. Their single instruction tests are generally sufficient for testing just the simulator, and by applying concolic techniques they can quickly generate a high coverage test suite for most of the architecture's instructions. However, the coverage of the processor implementations, and the effect of sequences of instructions on the processor, is outside the scope of their work.

IBM have made extensive use of randomised program generation during the validation of Power processors. In early work, Aharon et al. [17] considered highly randomised sequences like ours, but simplified the constraint solving by gradually extending the sequence by one instruction at a time. Hence the constraints on each instruction are solved individually, yielding a concrete partial state that the next instruction will be executed in. The trade-off for this simplification is that it is difficult to deal with complex situations such as the example in Figure 1, where fixing the value of a register early in the generation process may make it impossible to satisfy the constraints of a later instruction.

Later work moved towards template based testing in the Genesys-Pro system, where engineers use their knowledge of the processor design to direct the shape of the programs generated. Bin et al. [18]

describe the specialised constraint solver designed to support Genesys-Pro and some more specialised tools. They continue to use the instruction-at-a-time approach to simplify problems, but also incorporate 'soft' constraints to direct the solver towards test cases with interesting properties. Throughout the IBM work the models for the instruction set are built specifically for the tool.

More recently, Kamkin et al. [19] have developed a template-based system similar to Genesys-Pro, but which is easier to reconfigure due to the use of a common architecture description language for the model, and a general purpose SMT solver for constraint solving. This is similar to our use of L3 and Yices, although their process for generating constraints is unclear. They also suggest using Aharon et al.'s approach for testing with completely randomised sequences.

Fox and Myreen [1] validated a previous ARMv7 model in HOL4 using single instruction randomised testing, successfully covering a large portion of the instruction set and revealing several bugs. No constraint solving is necessary for single instruction testing when it is not model-driven, so they had more freedom to pick extreme values while searching for bugs.

Processor formalisation and testing is also a key part of Morrisett et al. [20], in which they built a formally verified static analysis. To construct a specification for the analysis they developed a formal model for a substantial subset of the x86 instruction set in the Coq proof assistant, and validated it in two ways: comparing execution traces from the model against traces from dynamic instrumentation of the code, and with black-box fuzz testing of instructions generated from their x86 instruction grammar. They noted that only the fuzz testing checked various unusual instructions, which is especially important for a security-related analysis where an attacker can choose to use atypical instructions.

Brucker et al. [21] performed a different form of model-based test generation for Verisoft's VAMP architecture, using the HOL-TestGen tool for the Isabelle/HOL proof assistant. The tool takes theorem-like specifications for tests and produces 'abstract' test cases, plus test input data which fills in the details (which can be selected randomly, or using Z3). In principle the whole pre-state could be generated by regarding it as an input, but they used an empty initial configuration because their representation of the model allows ill-formed states to be generated. Their work focused on test case generation and did not test against hardware.

The ability to predict the behaviour of test sequences and precisely eliminate those with faults or unpredictable behaviour is a key feature of our approach. Other randomised test generators take a more conservative approach. For example, Martignoni et al. [22] have produced the EmuFuzzer tool, which can test x86 instructions, including detecting undocumented extra instructions, but performs a static analysis before execution to remove 'dangerous' control flow instructions that might derail execution. We support these easily, because we obtain the next program counter from the model. If execution is derailed then we know that the model fails to match the processor.

A higher standard for models is a formal proof that they match a hardware design. There has been work in this area for decades; one particularly relevant example is Fox's verification of the ARM6 microarchitecture with respect to an ISA definition [23]. Proofs of processor designs can be difficult, and this is unusual for its high coverage of a commercial processor's instruction set. Similarly, Beyer et al. [24] have verified the VAMP gate-level design in the PVS system.

On a related note, Moore [25] promotes the usefulness of symbolic execution of executable processor models, and mentions the importance of testing even for low-level RTL models by recalling that testing an AMD processor model was crucial for persuading managers that the model was valid and worth investigating. This work involved executing existing test cases, whereas we have synthesised them by solving constraints revealed by symbolic execution.

10. Conclusion

Starting from an instruction set model that provides a reasonable, first-order view of the behaviour of individual instructions, we can automatically generate and test randomly chosen instruction sequences. The key parts of our process are to combine the model's requirements and predictions for each instruction into a single result for the whole sequence, essentially performing symbolic evaluation of the instructions, while

adding additional constraints for extra hardware-specific requirements, and then applying an off-the-shelf SMT solver to discover a state in which the test sequence will run.

By applying this testing at scale, we have gained confidence in our cycle-accurate ARM Cortex-M0 model, discovering some modelling errors and two timing anomalies. The SMT solver handled the constraints involved in forming a valid pre-state well, and we used additional constraints to validate our hypothesis about the timing anomalies found during testing. The testing system has few concrete dependencies on the M0 model itself, and should be adaptable to similar first-order models. Indeed, we are already applying it to another L3 model.

Acknowledgements. Support for this work was provided by the EPSRC Programme Grant EP/K008528/1, Rigorous Engineering for Mainstream Systems (REMS). Our thanks go to Anthony Fox, Magnus Myreen, Peter Sewell and the other members of the REMS project for their assistance.

References

[1] A. Fox, M. O. Myreen, A trustworthy monadic formalization of the ARMv7 instruction set architecture, in: M. Kaufmann, L. C. Paulson (Eds.), Interactive Theorem Proving (ITP 2010), Vol. 6172 of LNCS, Springer, 2010, pp. 243–258. doi:10.1007/978-3-642-14052-5_18.

[2] M. O. Myreen, M. J. C. Gordon, K. Slind, Decompilation into logic - improved, in: G. Cabodi, S. Singh (Eds.), Formal Methods in Computer-Aided Design (FMCAD 2012), IEEE, 2012, pp. 78–81.

[3] R. M. Amadio, N. Ayache, F. Bobot, J. P. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, P. Tranquilli, Certified complexity (CerCo), in: U. Dal Lago, R. Peña (Eds.), Foundational and Practical Aspects of Resource Analysis (FOPARA 2013), Vol. 8552 of LNCS, Springer, 2014, pp. 1–18. doi:10.1007/978-3-319-12466-7_1.

[4] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, The worst-case execution-time problem—overview of methods and survey of tools, ACM Trans. Embed. Comput. Syst. 7 (2008) 36:1–36:53. doi:10.1145/1347375.1347389.

[5] A. Fox, Directions in ISA specification, in: L. Beringer, A. Felty (Eds.), Interactive Theorem Proving (ITP 2012), Vol. 7406 of LNCS, Springer, 2012, pp. 338–344. doi:10.1007/978-3-642-32347-8_23.

[6] T. Weber, SMT solvers: New oracles for the HOL theorem prover, International Journal on Software Tools for Technology Transfer 13 (5) (2011) 419–429. doi:10.1007/s10009-011-0188-8.

[7] B. Campbell, I. Stark, Randomised testing of a microprocessor model using SMT-solver state generation, in: F. Lang, F. Flammini (Eds.), Formal Methods for Industrial Critical Systems (FMICS 2014), Vol. 8718 of LNCS, Springer, 2014, pp. 185–199. doi:10.1007/978-3-319-10702-8_13.

[8] HOL4 [online].

[9] ARM Ltd., Cortex-M0 Technical Reference Manual, r0p0 Edition, document DDI 0432C (2009). URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/index.html

[10] ARM Ltd., ARMv6-M Architecture Reference Manual, C Edition, document DDI 0419C (2010). URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0419c/index.html

[11] Open on-chip debugger [online] (2014).

[12] ARM Ltd., Cortex-M System Design Kit, r1p0 Edition, document DDI 0479C (2013). URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0479c/index.html

[13] ARM Ltd., Cortex-M3 Technical Reference Manual, r2p1 Edition, document DDI 0337I (2010). URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100165_0201_00_en/index.html

[14] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, in: M. Wermelinger, H. C. Gall (Eds.), Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, ACM, 2005, pp. 263–272. doi:10.1145/1081706.1081750.

[15] E. Bounimova, P. Godefroid, D. Molnar, Billions and billions of constraints: Whitebox fuzz testing in production, in: D. Notkin, B. H. C. Cheng, K. Pohl (Eds.), Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE, 2013, pp. 122–131.

[16] H. Wagstaff, T. Spink, B. Franke, Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators, in: K. S. Chatha, R. Ernst, A. Raghunathan, R. Iyer (Eds.), Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '14, ACM, 2014, pp. 15:1–15:10. doi:10.1145/2656106.2656113.

[17] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, V. Schwartzburd, Verification of the IBM RISC system/6000 by a dynamic biased pseudo-random test program generator, IBM Systems Journal 30 (4) (1991) 527–538. doi:10.1147/sj.304.0527.

[18] E. Bin, R. Emek, G. Shurek, A. Ziv, Using a constraint satisfaction formulation and solution techniques for random test program generation, IBM Systems Journal 41 (3) (2002) 386–402. doi:10.1147/sj.413.0386.

[19] A. Kamkin, E. Kornykhin, D. Vorobyev, Reconfigurable model-based test program generator for microprocessors, in: Software Testing, Verification and Validation Workshops (ICSTW 2011), IEEE, 2011, pp. 47–54. doi:10.1109/ICSTW.2011.35.

[20] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, E. Gan, RockSalt: better, faster, stronger SFI for the x86, in: J. Vitek, H. Lin, F. Tip (Eds.), Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI '12, ACM, 2012, pp. 395–404. doi:10.1145/2254064.2254111.

[21] A. Brucker, A. Feliachi, Y. Nemouchi, B. Wolff, Test program generation for a microprocessor, in: M. Veanes, L. Vigan (Eds.), Tests and Proofs (TAP 2013), Vol. 7942 of LNCS, Springer, 2013, pp. 76–95. doi:10.1007/978-3-642-38916-0_5.

[22] L. Martignoni, R. Paleari, A. Reina, G. F. Roglia, D. Bruschi, A methodology for testing CPU emulators, ACM Trans. Softw. Eng. Methodol. 22 (4) (2013) 29:1–29:26. doi:10.1145/2522920.2522922.

[23] A. Fox, Formal specification and verification of ARM6, in: D. Basin, B. Wolff (Eds.), Theorem Proving in Higher Order Logics (TPHOLs 2003), Vol. 2758 of LNCS, Springer, 2003, pp. 25–40. doi:10.1007/10930755_2.

[24] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, W. J. Paul, Putting it all together — formal verification of the VAMP, International Journal on Software Tools for Technology Transfer 8 (4-5) (2006) 411–430. doi:10.1007/s10009-006-0204-6.

[25] J. Strother Moore, Symbolic simulation: An ACL2 approach, in: G. Gopalakrishnan, P. Windley (Eds.), Formal Methods in Computer-Aided Design (FMCAD'98), Vol. 1522 of LNCS, Springer, 1998, pp. 530–530. doi:10.1007/3-540-49519-3_22.