Extracting Behaviour from an Executable Instruction Set Model

Extracting Behaviour from an Executable Instruction Set Model

Brian Campbell and Ian Stark
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh, UK
Email: Brian.Campbell@ed.ac.uk, Ian.Stark@ed.ac.uk

***Abstract*—Presenting large formal instruction set models as executable functions makes them accessible to engineers and useful for less formal purposes such as simulation. However, it is more difficult to extract information about the behaviour of individual instructions for reasoning. We present a method which combines symbolic evaluation and symbolic execution techniques to provide a rule-based view of instruction behaviour, with particular application to automatic test generation for large MIPS-like models.**

I. Introduction

It is a common practice to construct large formal models of instruction set architectures in the form of executable functions. Recent examples include an x86 model with system calls in ACL2 by Goel et al. [1] and the models constructed in Fox's L3 domain specific language [2], which can be translated into several systems. We have been using the latter in this work in the HOL4 theorem proving system.

There are several appealing aspects to such models. They can be used as simulators, existing tests suites can be run through them for validation, and they are in a form familiar to engineers. Indeed, L3 models are usually written in a form close to the pseudocode found in architecture reference manuals. However, they do not expose the structure normally found in a rule-based operational semantics, such as stating the conditions required for an instruction to have well-defined behaviour as explicit hypotheses.

Our goal is to extract this type of structure from the executable model for individual instructions, and use it to extend our previous automated test generation work [3] to new models. The core of that work used an existing verification support library [4] to provide a rule-based view of instructions and so obtain the constraints required to execute a randomly chosen sequence successfully, then express them in terms of the initial state and use an SMT solver to find such a state.

These *step* libraries have been constructed for several architectures and we have successfully used them for test generation with a model for the ARM Cortex-M0 microcontroller and a simple MIPS model. However, each library requires a considerable amount of effort, typically over 1000 lines of code per target, and ongoing maintenance when the model is altered.

We wanted to extend our MIPS testing to a much more complete model of the experimental CHERI processor [5]. CHERI features a hybrid capability system which can improve security by limiting access to resources while maintaining compatibility with existing code. The model includes a large number of new instructions for the additional security features, more complex representations of state and memory, and full memory management. It is over twice as large as the plain MIPS model and no *step* library has been written for it[1]. Moreover, we also wished to have the option of testing processor exception handling, which these libraries do not currently support.

We have constructed a new library to extract rules for individual instructions similar to those from the *step* libraries, but with much greater automation. To achieve this, and to deal with such a large model, we combine standard symbolic evaluation with a form of symbolic execution. The symbolic evaluation provides general computation using rewriting from the normal HOL4 libraries. The symbolic execution explores the different possible paths of execution, recording in the hypotheses the *path condition* which describes when each can be reached, and it treats the large state record carefully for reasonable performance.

Our contributions are to present our new library for extracting instruction behaviour from these executable models while maintaining a close, formal, connection to the model; to discuss its application to automatic test case generation; to demonstrate that a theorem proving system such as HOL is a practical setting for symbolic execution; and to show that these perform well enough for practical use on a realistic processor model, producing high test coverage. The close connection between the formal model and the generated tests is particularly important for CHERI, where some colleagues are now proving security properties about the model. Our code is available online[2].

In Section II we outline the form of the models, the desired form for expressing the extracted behaviour, and the testing process. Section III presents our combination of symbolic evaluation and symbolic execution, followed by a discussion of how sound and complete the process is in Section IV. Then Section V describes the application of the process to

[1] A basic *step* library for a simplified version of the model was produced after this paper was written, but it only covers a fraction of the behaviour that we are interested in testing.

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

```
dfn'ADDI (rs,rt,immediate) =
(λstate.
   (let s =
      if NotWordValue (FST (GPR rs state)) then
        SND
          (raise'exception
            (UNPREDICTABLE "ADDI: NotWordValue")
            state)
      else state
    in
    let v = (32 >< 0) (FST (GPR rs s))
              + sw2sw immediate
    in
      if word_bit 32 v ≠ word_bit 31 v
      then SignalException Ov s
      else write'GPR (sw2sw ((31 >< 0) v),rt) s))
```

Fig. 1. HOL4 version of 32-bit signed immediate add MIPS instruction

```
[s.CP0.Config.BE, ¬s.CP0.Status.RE,
 ¬s.exceptionSignalled,
 ¬if word_bit 31 (s.gpr 2w)
  then (63 >< 32) (s.gpr 2w) ≠ 0xFFFFFFFFw
  else (63 >< 32) (s.gpr 2w) ≠ 0w,
 s.MEM s.PC = 36w, s.MEM (s.PC + 1w) = 65w,
 s.MEM (s.PC + 3w) = 3w, s.MEM (s.PC + 2w) = 0w,
 (1 >< 0) s.PC = 0w, s.exception = NoException,
 s.BranchDelay = NONE, s.BranchTo = NONE]
⊢ NextStateMIPS s =
   SOME (s with
   <|BranchDelay := NONE; BranchTo := NONE;
    CP0 := s.CP0 with Count := s.CP0.Count + 1w;
    PC := s.PC + 4w;
    exceptionSignalled := F;
    gpr := (1w =+ sw2sw ((31 >< 0) (s.gpr 2w) + 3w))
           s.gpr|>)
```

Fig. 2. *Step* theorem for MIPS 32-bit unsigned immediate addition

testing, and in particular with the MIPS and CHERI models. We discuss related work in Section VI and possible further work in Section VII.

II. Background

Fox's L3 domain specific language [2] provides a natural environment for writing instruction set architecture specifications in the form of a function to compute successive states. The language features support for working with data at the bit level, including pattern matching for decoding and bit-field records for registers, exceptions to indicate undefined behaviour, incrementally defined global processor state and an instruction abstract syntax datatype automatically derived from the instruction definition functions. The L3 tool translates the language into the logics of several proof tools and also the SML programming language for simulation. L3 models have been constructed for a number of architectures, including ARMv7-A, ARMv6-M, MIPS, CHERI and partial models of ARMv8-A and x86-64, and there are several external users of these models.

The translation to HOL4 must transform several of L3's language features into the logic. Figure 1 shows the HOL4 translation of the MIPS 32-bit signed immediate addition instruction, where the global state has been implemented by threading a state record throughout the definition, in the `state` and `s` variables. The undefined behaviour when the value of the `rs` register cannot be represented in 32 bits is modelled by the `raise'exception` function which records the failure in the state record. Failures are then detected at the end of the instruction execution rather than using an exception monad to simplify certain types of reasoning (see [4, §2] for discussion about this design). The `SignalException` function is a normal function to set up a processor exception, rather than an L3 exception to indicate undefined behaviour. In this case it is for an arithmetic overflow, which is detected by performing a 33-bit addition (where the `(32 >< 0)` operator extracts the bottom 33 bits of the 64-bit register). The use of the `sw2sw` function in the last line extends the result to 64 bits, ready to be written to the destination register.

Each model has a main function which computes the next state, combining the parts of the model that perform instruction fetch, decoding and execution (such as the function in Figure 1). The return value of the main function is an option: either the next state is returned, or nothing is returned if the model is undefined on the input state.

A. L3 Support Libraries

In our previous work we used pre-existing libraries by Fox that are part of a system for verifying machine code with respect to an L3 model [4]. The main interface for this system is a program logic in the style of separation logic, but we are interested in the intermediate *step* library which provides a more equational view. This library presents the behaviour of instructions as theorems providing the result of the main next state function as a series of state updates when a number of hypotheses hold, roughly:

$$\frac{\begin{array}{c}\text{flags set correctly in state } s\\ s \text{ contains instruction in memory}\\ s \text{ contains data to be read}\end{array}}{\mathsf{Next}\ s = \mathsf{SOME}\ (s \text{ with sequence of state updates})}$$

These theorems cut across the model, including decoding, execution and memory accesses. A MIPS example is shown in Figure 2, which is a 32-bit immediate addition, but unlike Figure 1 it is unsigned to avoid showing the complexity of a potential processor exception. Some of the less relevant details are shown in light grey. The hypotheses include the processor flags, the unfolded `NotWordValue` test to avoid undefined behaviour, and the presence of the instruction in memory. The conclusion updates the state, and in particular the `(1w =+ ...) s.gpr` updates register 1 with the result.

In general, there may be multiple theorems for an instruction when there are several branch choices. This MIPS instruction actually has three variants due to the use of branch delay slots in the MIPS architecture.

The *step* libraries are primarily based around symbolic evaluation under sets of assumptions for each case of a class of instructions. The main evaluation procedure, which includes

setting up appropriate rewrites for the model's datatypes and definitions, and some specialised conversions, is in a common library used for all architectures. However, the per-model parts must still contain a substantial amount of information about the different classes of instruction present in the model, the different cases of each, and how to combine results about fetching, decoding and running instructions into a single result.

B. Test generation process

Our automatic test generation system [3] starts with random sequences of instructions, typically chosen to be long enough to exercise the pipeline. It finds an initial state which will run the sequence, avoiding undefined behaviour and (when undesired or unsupported) processor exceptions, by solving constraints derived from the *step* library information. It then compares the model's predicted final state with the result of an actual execution from the initial state. The process can be summarised as:

1) Generate instruction sequence
2) **Extract instruction behaviour from model**
3) Calculate sequence's constraints and behaviour in terms of the initial state
4) Solve constraints to build test with an SMT solver
5) Add test harness

The hypotheses from the extracted behaviour are the source of the constraints, while the conclusions are used to rewrite them in terms of the initial state of the whole sequence. We do not need there to be precisely one rule per branch choice to do this, so we can accept any reasonable partitioning of instruction behaviour into rules so long as the conclusion is in the form of a sequence of state updates.

We solve the constraints using an off-the-shelf SMT solver through an existing translation of a subset of HOL4 terms [6]. To adapt the testing to a new architecture the instruction generation, behaviour extraction and harness code must be adapted. This is routine for targets with a suitable *step* library, but for new targets such as CHERI we need a replacement for the behaviour extraction phase.

The *step* libraries have a few features that the testing does not require: there is support for partial instructions (where some operands are left as variables), it is compatible with the separation logic library which we do not use, and there is support for caching the resulting theorems.

III. Extracting Behaviour

One of our goals is to reduce the amount of user effort needed to extract instruction behaviour on a new target, so the new library must require much less model-specific information. Thus our replacement library discovers the different cases for each instruction rather than being provided with them, using the structure present in the model's definitions. This also removes the need to know about the different instruction classes. To increase automation we process the entire next state function at once, rather than building up a result from separate lemmas about the model's functions for fetching, decoding and executing instructions.

The threading of the global state record through the definitions by the L3 translation tool provides the structure used to discover the different cases of each instruction. To get results like Figure 2 which conclude with a sequence of state updates we need to break up any conditionals or pattern matches encountered in this threaded computation, producing a separate theorem for each path.

Symbolic execution techniques fit this view of the model; they follow the imperative structure of a program (in our case, the threaded state) and consider each path in the program independently, producing a separate result for each one. The parts of the computation which do not directly modify the state, such as the calculation of `v` in Figure 1, are left to symbolic evaluation, by which we mean rewriting the term under a set of assumptions producing a single equivalent term rather than a set of possible terms.

We avoid undesirable interactions between the evaluation and the execution by restricting the evaluation of **`let`** terms. There are beneficial interactions where conditionals and pattern matches can be simplified. For example, if we choose the always-zero register for `rs` in Figure 1 then evaluation will dispose of the `NotWordValue` test before execution even considers it. The implementation interleaves the symbolic evaluation and execution in a single recursive function.

A. Symbolic evaluation

For most of the symbolic evaluation of the model we use the `computeLib` call-by-value evaluation library included in HOL4 [7], appealing to rules for evaluating terms on bitvectors, arithmetic, pairs and other definitions from standard HOL4 theories. It also deals with operations on the model's datatypes and certain functions from the model, reusing some of the utility functions from Fox's libraries that can generate rewrites for any model. This is combined with some limited use of HOL4's simplifier for more complex rewrites, for example those which require higher-order matching.

In addition to avoiding evaluation of the **`let`** terms that the symbolic execution will explore, the evaluation must avoid expensive expansion of terms before there is sufficient information to reduce them properly. For example, the L3 translator uses a `FOR` combinator for loops, but if the number of iterations is not yet known (because some symbolic execution of the state is required first) then the standard evaluation rule will never terminate. We solved this by replacing the `FOR` rule by a restricted conversion that requires a concrete number for the loop bound. Similarly, most of the model definitions are only unfolded by the symbolic execution because they cannot be usefully evaluated before the state at the point of application is known.

Our symbolic evaluation also uses the current set of hypotheses as rewrite rules, including general user-specified assumptions about the particular target, such as the processor's endianness. The user can provide more specialised rewrite rules which introduce extra assumptions during evaluation. For example, we use this to restrict memory accesses to the small region of the address space that is used by our tests.

B. Symbolic execution

Traditional symbolic execution [8] requires a symbolic set of values, symbolic evaluation of expressions, a symbolic store, and a *path condition* to record the circumstances which lead to the part of the program currently being executed so that incompatible paths later in the execution can be avoided. The values and evaluation we get 'for free' by working in HOL with the evaluation procedure outlined above. Our initial treatment of the state was to substitute the entire symbolic value for the state record every time it was updated. However, we discovered that the performance was unacceptably poor for models with large state records such as CHERI. Instead, we maintain a rewrite for each field of the state which expresses its current value in terms of the initial state, and add these rewrites to the symbolic evaluation. In principle we could go further by using a separate rewrite for each entry of subrecords and maps in the state, for example, having one rewrite per register rather than the entire register map, but this has not been necessary in practice.

To maintain the path condition we add the appropriate assumption for each branch taken at a case split to the list of hypotheses. The symbolic evaluation will then use these assumptions to automatically eliminate incompatible branches later in the execution, and they may also be used for other simplification. For example, if a conditional takes one branch when a variable is zero, then in the execution which takes that branch the variable will be rewritten to zero throughout.

The symbolic execution procedure is summarised in Figure 3, where judgements of the form

$$H, S \vdash t \rightsquigarrow \overline{(H', t')}$$

mean that under the set of hypotheses H and the per-field state rewrites S, the execution of term t results in a set of terms t' paired with hypotheses H', which may extend H with path conditions and assumptions from special rewrite rules (such as limiting the range of memory addresses used). We also write u and v for terms, x for variables, and c for the names of constants in the rules.

The L3 translator always places the state record in the rightmost position of a tuple, so the PAIR and SND rules merely follow the state, then reconstruct the surrounding context. The LET rule propogates state updates: for each state s'_i found by executing t, we form a new set of rewrites, denoted $S \triangleleft s'_i$, which updates the rewrites in S with the changes in s'_i. This new set is then used for the symbolic execution of u.

Case splits are handled by the COND and CASE rules. In each branch we add a new hypothesis corresponding to the guard or pattern match, and then proceed with that branch in isolation. The actual implementation also replaces the variables which are bound in patterns with fresh ones to prevent clashes. In principle, case splitting at every conditional or pattern match would lead to an explosion in the number of cases to consider. In practice, many of the cases are eliminated by the symbolic evaluation due to existing assumptions or the path condition, and from the remainder most lead to some

$$\frac{H, S \vdash u \rightsquigarrow \overline{(H', u')}}{H, S \vdash (t, u) \rightsquigarrow \overline{(H', (t, u'))}} \text{ PAIR}$$

$$\frac{H, S \vdash t \rightsquigarrow \overline{(H', t')}}{H, S \vdash \texttt{SND}\ t \rightsquigarrow \overline{(H', \texttt{SND}\ t')}} \text{ SND}$$

$$\frac{\begin{array}{c} H, S \vdash t \rightsquigarrow \overline{(H', (t', s'))} \\ \forall i.\ H'_i, S \triangleleft s'_i \vdash u[t'_i/x] \rightsquigarrow \overline{(H''_i, u'_i)} \end{array}}{H, S \vdash \texttt{let}\ (x, s) = t\ \texttt{in}\ u \rightsquigarrow \bigcup_i \overline{(H''_i, u'_i)}} \text{ LET}$$

$$\frac{\begin{array}{c} (H, t), S \vdash u \rightsquigarrow \overline{(H', u')} \\ (H, \neg t), S \vdash v \rightsquigarrow \overline{(H'', v')} \end{array}}{H, S \vdash \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ v \rightsquigarrow \overline{(H', u')} \cup \overline{(H'', v')}} \text{ COND}$$

$$\frac{\forall i.\ (H, pt_i = t), S \vdash u_i \rightsquigarrow \overline{(H'_i, u'_i)}}{H, S \vdash \texttt{case}\ t\ \texttt{of}\ pt_1 \Rightarrow u_1 | \ldots \rightsquigarrow \bigcup_i \overline{(H'_i, u'_i)}} \text{ CASE}$$

$$\frac{}{H, S \vdash \texttt{raise'exception}\ t\ u \rightsquigarrow \emptyset} \text{ UNDEF}$$

$$\frac{\begin{array}{c} c\, x_1 \ldots x_{n+1} := t \qquad H, S \vdash v \rightsquigarrow \overline{(H', v')} \\ \forall i.\ H'_i, S \vdash t[u_1/x_1, \ldots, u_n/x_n, v'_i/x_{n+1}] \rightsquigarrow \overline{(H''_i, t'_i)} \end{array}}{H, S \vdash c\ u_1 \ldots u_n\ v \rightsquigarrow \bigcup_i \overline{(H''_i, t'_i)}} \text{ APP}$$

Fig. 3. Rules used in symbolic execution

form of undesirable behaviour which is discarded. The UNDEF rule does this for the L3 exceptions which indicate undefined behaviour. Extra rules can be added for any other function in the model; for example, when we are not interested in the handling of processor exceptions we discard paths where we reach the `SignalException` function.

Other functions involving the state are handled by the APP rule. The state is always passed in the final argument, so we process it first then unfold the function's definition. Functions which do not involve the state are unfolded by the symbolic evaluation.

Any term that does not fit one of the rules is only run through the symbolic evaluation.

C. Example

To illustrate the procedure we consider the main definition for a single instruction on fixed operands,

```
dfn'ADDI (2w,1w,3w) s
```

which is the 32-bit signed addition of 3 to the contents of register 2, placing the result in register 1. The APP rule unfolds the definition, which we saw in Figure 1. The first part of the **let**,

```
if NotWordValue (FST (GPR 2w state))
  then SND (raise'exception
              (UNPREDICTABLE "ADDI: NotWordValue")
              state)
  else state
```

is processed recursively, and COND examines each of the branches separately. The first is discarded by SND and UNDEF because the processor's behaviour on a value that cannot be represented in 32 bits is undefined. The second case is trivial, except that we now have an extra hypothesis,

```
¬if word_bit 31 (s.gpr 2w)
 then (63 >< 32) (s.gpr 2w) ≠ 0xFFFFFFFFw
 else (63 >< 32) (s.gpr 2w) ≠ 0w
```

which is the result of evaluating the guard, `NotWordValue ( FST (GPR 2w state))`. The same hypothesis is present in the unsigned case in Figure 2.

In the second part of the **let** the computation of `v` cannot change the state, so it is evaluated, leaving us with the final conditional:

```
if word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) ≠
   word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w)
then SignalException Ov s else
  write'GPR
    (sw2sw
      ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)),
     1w) s
```

Again, COND considers each branch. For the sake of brevity, we will not consider the overflow processor exception. In the second branch, the `write'GPR` definition is unfolded and it continues to the result

```
dfn'ADDI (2w,1w,3w) s =
  ((),
   s with c_gpr :=
     (1w =+ sw2sw
        ((31 >< 0) ((32 >< 0) (s.c_gpr 2w) + 3w)))
      s.c_gpr))]
```

which updates register 1 with the sum, under the two hypotheses,

```
¬if word_bit 31 (s.gpr 2w)
 then (63 >< 32) (s.gpr 2w) ≠ 0xFFFFFFFFw
 else (63 >< 32) (s.gpr 2w) ≠ 0w

¬word_bit 32 ((32 >< 0) (s.c_gpr 2w) + 3w) ≠
 word_bit 31 ((32 >< 0) (s.c_gpr 2w) + 3w)
```

which ensure the argument can be represented in 32 bits and that there is no overflow, respectively.

To illustrate how the state updates $S \lhd s$ are calculated, suppose that we start out with an unchanged initial state, `s0`, and want to update it with the result above. The initial set of field rewrites S will be:

```
s.c_gpr   = s0.c_gpr
s.c_state = s0.c_state
...
```

The new set of rewrites reflects the updates to each field. In our case only the register field `c_gpr` is affected:

```
s.c_gpr   = (1w =+ sw2sw (...)) s0.c_gpr
s.c_state = s0.c_state
...
```

Note that the previous rewrite is applied, replacing `s.c_gpr` with `s0.c_gpr`, so that it is expressed in terms of the initial state.

IV. SOUNDNESS, INCOMPLETENESS AND COMPLETENESS

The correctness of the extracted behaviour with respect to the model is ensured by construction because every stage of the process produces a theorem witnessing it. In particular, for each rule of the symbolic execution

$$H, S \vdash t \rightsquigarrow \overline{(H', t')}$$

the system generates a theorem for each result:

$$H'_i \vdash t = t'_i.$$

However, these theorems are generated dynamically, so any bug in the implementation will only be detected during symbolic execution.

This shows that the results will be sound, but we also wish to know whether they will be complete, that is whether the procedure finds all of the relevant behaviour. It must be incomplete in the sense that some behaviour is intentionally excluded; undefined behaviour is not useful for testing, the tests must respect restrictions on endianness and memory layout to run on the test system, and we only wish to explore processor exceptions in a controlled manner.

In principle we could codify all of the undesirable behaviour and construct an additional overall theorem stating that under only basic assumptions either one of the conclusions from the extracted behaviour will be reached, or one of the undesirable situations will. We believe it would be feasible to construct such a theorem because the intermediate results required would be of a similar size and number to the actual results we already compute. However, it would require considerable effort to add the code to compute this theorem, and, as before, it would only detect failure at runtime.

Less formally, we can compare our method with the *step* libraries. These are primarily intended for verification, and some of them deliberately restrict the supported behaviours even further. For example, the ARMv7-A step library requires all word loads to be aligned, even though the architecture permits unaligned loads in some circumstances. This is added by an explicit assumption in the model-specific part of the step library. While such restrictions can be useful for simplifying verification they also illustrate how easily behaviour can be accidentally excluded. In contrast, our library systematically explores the model with fewer user-provided assumptions, so accidentally missing behaviour is less likely. Thus we believe the procedure is complete for a model specialised to any particular processor, fixing details such as endianness and available memory.

V. APPLICATION TO TESTING

To use the library with an instruction set model we still need to provide some model-specific information, such as identifying the functions which raise L3 exceptions and processor exceptions, the standard set of assumptions, specialised rewrites, and the definitions which should not be unfolded (typically to enable a rewrite). For example, one use of model-specific rewrites is to avoid accessing hardware resources that are present in the model for simulation, such as serial ports.

There is a bootstrapping problem for the more complex models. In simple models such as the M0 microcontroller that we previously worked with, instructions can be injected into memory by adding hypotheses for each byte of each instruction. More complex models feature address translation and different memory representations. For example, in the CHERI model memory is represented in chunks that contain either a capability record or a raw capability-sized bitvector. Rather than writing a function to generate suitable assumptions for each model by hand, we use the procedure above to generate a theorem about the behaviour of the model's `Fetch` function, then build a rewrite from it which will ensure that the desired instruction is loaded. The behaviour of the instruction can then be extracted from the `Next` function if we add the rewrite to the set used by the symbolic evaluation. This approach has the advantage that it is relatively robust to changes in the model.

The first model the procedure was applied to was the plain MIPS model. This has a *step* library which we could use for comparison during development and for the performance comparisons below. We then moved on to our main target of interest, the large CHERI model. We had already tested the plain MIPS model against the CHERI hardware design, but CHERI has a considerable number of new instructions without a corresponding *step* library. Moreover, the instructions have a large number of security checks that result in processor exceptions, so in addition to checking fault-free instruction sequences we also generated tests where one of the instructions in the middle of the sequence exercised one of its exceptional behaviours, extracted in the same way.

Testing with the plain MIPS model had already detected bugs in the model and the hardware design. Extending testing to CHERI using our library not only produced tests that would detect those bugs, but also found problems in areas that were not supported by the *step* library; in particular, in the model store conditional instructions did not check enough of the supplied address, and several instructions wrote back results incorrectly when a processor exception was signalled.

We have been able to track changes to the model with very few adjustments. For example, a new instruction will need to be added to the instruction generator, but not the behaviour extraction library. Moreover, when we first targeted the test system at CHERI the most labour intensive model-specific work was adapting the test harness construction code which initialises the test state, because the model-specific part of our behaviour extraction library is so small.

A. Performance

Our main goal is the batch production of tests, so we could accept a large increase in the test generation time. We have not yet explored opportunities to accelerate the process, such as replicating the feature in the *step* libraries to extract the general behaviour of an instruction and cache it for future use with a range of different operands. Nonetheless, we generated 500 8-instruction tests for the plain MIPS model using both the *step* library and our library to compare the libraries and to determine whether behaviour extraction is a bottleneck in test generation.

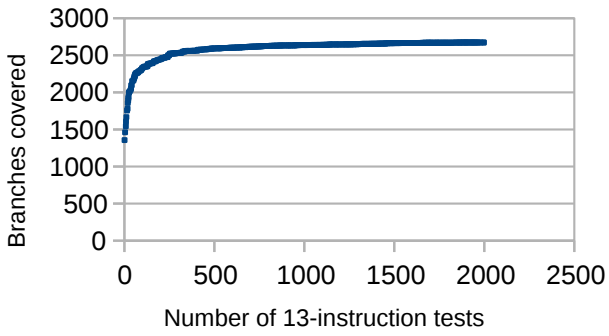

Fig. 4. Cumulative coverage graph

While the median behaviour extraction time increased considerably from 0.23 seconds to 3.16 seconds, it still represents a small fraction of the overall time for test generation, whose median increased from 16.98 seconds to 19.15 seconds. Thus improving our library would not make a huge difference to the rate of test generation unless substantial improvements were made elsewhere. Moreover, the model-specific code for plain MIPS for our library is an order of magnitude smaller than that for the original *step* library.

This is still true with the larger CHERI model. While the median behaviour extraction times are much greater, around 30 seconds for 8-instruction tests, they are still only a third of the total test generation time, and sufficient for batch test generation.

B. Coverage

Following the production of a large batch of CHERI tests for use against the hardware design, we wished to check how much of the model's instruction behaviour was actually exercised and whether any bugs in the behaviour extraction were causing cases to be missed. The testing system was set to produce 13 instruction tests, where the middle instruction raises a processor exception and all other instructions do not.

We measured the branch coverage by building the SML simulator version of the L3 model and using the MLton compiler's coverage support. To see if we had generated enough tests to demonstrate the available coverage we produced a graph showing the cumulative branch coverage of the model, shown in Figure 4. The flattening of the curve suggests that additional random tests would add little to the overall coverage.

To get a more qualitative idea of how good the coverage is we manually examined the branches in the instruction definitions that were not covered after 2000 tests[3]. The branches that were not covered fell into two groups: those which are impossible due to undefined behaviour or an unnecessary default case in a complete pattern match; and those which

[3]Examining the model as a whole is not appropriate due to the amount of code that is outside of the scope of the tests, such as serial ports, interrupts, full address translation, instruction encoding, and disassembly.

testing did not reach by pure chance, which were almost all due to the very large number of security checks in the new CHERI instructions. We know that the latter group were due to chance because the same checks in other instructions were tested; if we made the random test generation more targeted it should be possible to cover these cases without greatly increasing the number of tests. There was one other case amongst the branches that were not covered: a trap instruction that had accidentally been omitted in the instruction generation phase, which was easily corrected.

VI. Related Work

Automated conversions between functional semantics and more structured operational semantics has been studied in many forms. Note that the output of our system is not structured to the same extent as a Plotkin-style structured operational semantics [9]; while we do produce rules with stylised conclusions, separate hypotheses, and where we have one rule for each behaviour, we do not break the execution down into a set of intuitive judgements and build up behaviour in derivation trees. Instead, we provide monolithic rules where everything is described in basic terms, slicing across the entire model. For our automated testing this is quite reasonable because we will eventually present constraints derived from the rules to an SMT solver which has no direct knowledge of the model.

General purpose tools for converting or reasoning about functional semantics are more structured. The Function mechanism for Coq [10] generates an inductive relation for the graph of a function, together with induction and inversion principles. The relation does split out the different behaviours of the function, but there is a relation for every function, which would be difficult to use on a model with hundreds of definitions. Similarly, Owens et al. [11] recently advocated using functional definitions for programming language semantics when suitable induction principles are generated. In their case their principles came from HOL4's mechanism for defining recursive functions.

There are other functional semantics for instruction sets where rules have been manually specified for each instruction, and proved as lemmas. Srivas and Miller [12] did this when verifying the microcode for the AAMP5 processor, staying close to the pseudocode initially, then deciding to derive rules, saying:

> They were more readable, simpler to validate, and were closer to what a user wanted to know in the first place. They also made it possible to specify a small portion of the `next_macro_state` function, i.e., to specify one instruction or part of an instruction at a time.

However, this is exactly the type of work we wish to automate. Similarly, Jensen et al. [13] formalised a subset of the x86 architecture in Coq for verifying machine code programs where they proved manually specified separation logic rules.

Fox's libraries [4], which we described in Section II, fit in between these completely manual rules and our almost automatic system. The results differ from ours in several ways: they must correspond closely to the accompanying program logic library, work with partially specified operands, cache results, and produce one rule per branching choice, rather than partitioning the behaviour according to the structure of the definition. It may be possible to adapt our library to cover some of these points, but we leave that to future work.

There is also a body of work on translating from rule-based relational semantics to functions, such as Isabelle's predicate compiler [14] and a similar feature for Coq by Tollitte et al. [15]. This is an attractive way to animate semantics which have been presented relationally, as is common for many programming languages. Indeed, Lochbihler and Bulwahn have done this for a Java-like language [16]. However, if we were to rewrite our model like this we would lose the close correspondence to the designers' pseudocode. Transformations in this direction do have the advantage that they can produce several functions, depending on which parameters of the relation are chosen as inputs and outputs of the generated function.

Turning our attention to the techniques involved, symbolic evaluation is widely used with executable models. Fox's libraries provide one example. Moore [17] advocates the use of symbolic evaluation of a machine model for exploring the behaviour of assembly programs, which he calls *symbolic simulation*. Having demonstrated that an example program can be simulated in ACL2 despite only partially specifying the input, Moore suggests that this is reasonably accessible to engineers and that a special purpose user interface would aid adoption.

Symbolic execution has a long history of use in testing, early work by King [8] used it for interactive testing, while Boyer et al. [18] primarily generated test cases. It is now commonly used for automatic test case generation on large programs using *concolic* testing [19], where the symbolic execution follows the same path as the concrete execution of a test case, and part of the resulting path condition is negated to force the solver to find a test case which explores a new branch without searching through the full space of paths. Our library is closer to Boyer et al. because we explore all of the well-defined paths for a single instruction during symbolic execution, but leave the test case generation to a later phase of the testing process that uses the entire instruction sequence. In contrast, concolic methods have been used for single instruction simulator testing by Wagstaff et al. [20] for high coverage, and similar methods by Martignoni et al. [21] for cross-testing.

VII. Further work

The most straightforward area of possible work is to use the library in test generation for other L3 models; partly to test these models, and partly to identify any remaining aspects of the library that are too reliant on the particular models above. This would still require some manual effort for each model to construct the instruction generator and the production of harness code.

One danger with other L3 models is that they may make greater use of looping constructs. The plain MIPS and CHERI models use no recursion, have `FOR` loops with statically known iteration counts, and do not have more than one well-defined path inside a loop (which prevents the number of cases exploding). While other models may be similar, it is possible that difficult loops may appear occasionally. In fact, there is one in the CHERI model's address translation but it does not affect our testing because it is not used for the parts of memory that our tests run in. The loop tests each TLB entry, resulting in an exponential number of paths, although the surrounding code restricts the well-defined paths to one per entry. To tackle these issues we could manually prove that the loop can be replaced with a simpler form, or attempt a general solution, perhaps by analysing the loop body separately and adding more structure to the output.

We could also investigate adding more of the features from Fox's library, as described above. The exact requirements for interfacing with his program logic libraries are unclear, and it may be that our results are not sufficiently idiomatic to be compatible. Support for partially specified instructions and caching seem more feasible, and would improve performance.

A more ambitious task would be to apply the same approach to an instruction set modelling language which supports some weak memory model, both to test the sequential behaviour of such models, and to investigate automatic test generation for multicore architectures.

VIII. Conclusion

We have constructed a library which uses symbolic execution in a theorem prover to extract rule-based descriptions of processor behaviour from L3 executable instruction set models with minimal user-provided information about the model. The structure of the results can be used to drive an automatic test generation system, and the soundness of the procedure is ensured by the HOL4 theorem proving system. The resulting system has been successfully used with a large model of the CHERI experimental MIPS-like processor.

Acknowledgment

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/K008528/1], Rigorous Engineering for Mainstream Systems (REMS).

References

[1] S. Goel, W. A. Hunt, M. Kaufmann, and S. Ghosh, "Simulation and formal verification of x86 machine-code programs that make system calls," in *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '14. Austin, TX: FMCAD Inc, 2014, pp. 18:91–18:98.

[2] A. Fox, "Directions in ISA specification," in *Interactive Theorem Proving (ITP 2012)*, ser. LNCS, L. Beringer and A. Felty, Eds. Springer, 2012, vol. 7406, pp. 338–344.

[3] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using SMT-solver state generation," *Science of Computer Programming*, vol. 118, pp. 60–76, March 2016.

[4] A. Fox, "Improved tool support for machine-code decompilation in HOL4," in *Interactive Theorem Proving, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, 2015, pp. 187–202.

[5] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, 2014, pp. 457–468.

[6] T. Weber, "SMT solvers: New oracles for the HOL theorem prover," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 5, pp. 419–429, 2011.

[7] B. Barras, "Programming and computing in HOL," in *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, 2000, pp. 17–37.

[8] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[9] G. D. Plotkin, "A structural approach to operational semantics," Computer Science Department, Aarhus University, Tech. Rep. DAIMI FN-19, 1981.

[10] G. Barthe, J. Forest, D. Pichardie, and V. Rusu, "Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant," in *Functional and Logic Programming (FLOPS 2006)*, ser. Lecture Notes in Computer Science, M. Hagiya and P. Wadler, Eds., vol. 3945, 2006, pp. 114–129.

[11] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, "Functional big-step semantics," in *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016, pp. 589–615.

[12] M. K. Srivas and S. P. Miller, "Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods," *Formal Methods in System Design*, vol. 8, no. 2, pp. 153–188, 1996.

[13] J. B. Jensen, N. Benton, and A. Kennedy, "High-level separation logic for low-level code," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, 2013, pp. 301–314.

[14] S. Berghofer, L. Bulwahn, and F. Haftmann, "Turning inductive into equational specifications," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Springer Berlin / Heidelberg, 2009, vol. 5674, pp. 131–146.

[15] P. Tollitte, D. Delahaye, and C. Dubois, "Producing certified functional code from inductive specifications," in *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, 2012, pp. 76–91.

[16] A. Lochbihler and L. Bulwahn, "Animating the formalised semantics of a Java-like language," in *Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds. Springer Berlin / Heidelberg, 2011, vol. 6898, pp. 216–232.

[17] J. Strother Moore, "Symbolic simulation: An ACL2 approach," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and P. Windley, Eds. Springer Berlin / Heidelberg, 1998, vol. 1522, pp. 530–530.

[18] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 234–245.

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

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

[21] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: hi-fi tests for lo-fi emulators," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, 2012, pp. 337–348.