Who Guards the Guards? Formal Validation of the Arm v8-M Architecture Specification

ALASTAIR REID, Arm Ltd, United Kingdom

Software and hardware are increasingly being formally verified against specifications, but how can we verify the specifications themselves? This paper explores what it means to formally verify a specification. We solve three challenges: (1) How to create a secondary, higher-level specification that can be effectively reviewed by processor designers who are not experts in formal verification; (2) How to avoid common-mode failures between the specifications; and (3) How to automatically verify the two specifications against each other.

One of the most important specifications for software verification is the processor specification since it defines the behaviour of machine code and of hardware protection features used by operating systems. We demonstrate our approach on ARM's v8-M Processor Specification, which is intended to improve the security of Internet of Things devices. Thus, we focus on establishing the security guarantees the architecture is intended to provide. Despite the fact that the ARM v8-M specification had previously been extensively tested, we found twelve bugs (including two security bugs) that have all been fixed by ARM.

CCS Concepts: • **Computer systems organization** → *Architectures*; Reduced instruction set computing; • **Hardware** → *Theorem proving and SAT solving*; • **Software and its engineering** → *Consistency*; *Software verification*; *Formal software verification*;

Additional Key Words and Phrases: ISA, Specification, Formal Verification

ACM Reference Format:
Alastair Reid. 2017. Who Guards the Guards? Formal Validation of the Arm v8-M Architecture Specification. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 88 (October 2017), 24 pages. https://doi.org/10.1145/3133912

1 INTRODUCTION

The last decade has seen formal verification techniques scaling to the point where it is possible to formally verify realistic compilers [Leroy 2009], operating system kernels [Klein et al. 2009], hypervisors [Dam et al. 2013] and processors [Reid et al. 2016]. These efforts are impressive but we must beware that the correctness of their proofs ultimately rests on the correctness of the specifications they depend on. This is worrying because these specifications are, themselves, large and complex artifacts with all the risks of bugs that we expect in large, complex software. This risk is only likely to increase as more effective formal verification techniques and tools allow larger, more complex projects to be verified against larger, more complex specifications.

Bugs in specifications are not just a theoretical possibility. In previous work [Reid 2016], we reported that correcting errors in the ARM v8-A Architecture Reference Manual [ARM Ltd 2013] resulted in changes to 12% of the lines of code in ARM's processor specification. The CompCert compiler [Leroy 2009] required a bugfix *despite being formally verified* and the bug can be traced to the architecture specification not describing the full behaviour of an instruction [CompCert 2016].

Author's address: first.last@arm.com, Arm Ltd, 110 Fulbourn Road, Cambridge CB1 9NJ, UK.
Author's address: Alastair Reid, Arm Research, Arm Ltd, 110 Fulbourn Road, Cambridge, CB1 9NJ, United Kingdom, alastair.reid@arm.com.

2475-1421/2017/10-ART88
https://doi.org/10.1145/3133912

In an empirical study of the correctness of three formally verified distributed systems [Fonseca et al. 2017], no protocol bugs were found in the verified systems but 16 bugs were found in the Trusted Computing Base (i.e., the unverified glue code, build system, and specifications used to build the code and in the proof) including two bugs in the specifications. If we are to trust the guarantees claimed for formally verified software, it is essential that we verify the large, complex specifications on which our formal correctness claims are founded.

Three common ways that bugs are found in specifications are by testing specifications against existing implementations [Flur et al. 2016; Fox and Myreen 2010; Goel et al. 2014]; by testing specifications using testsuites used to test implementations [Reid 2016]; or as a side effect of attempting to formally verify an implementation against a specification [Reid et al. 2016]. Unfortunately, these approaches can still miss bugs either because test suites are incomplete or because of common mode failure (i.e., the specification and the implementation do the same wrong thing).

This situation is bad for programmers relying on specifications because, no matter how careful they are, they are reliant on the quality of the specification available to them. For example, Dunlap [Dunlap 2012] describes a bug in the Xen hypervisor that arose because of an inconsistency between the Intel and AMD specifications of the SYSRET instruction allowing a privilege escalation when run on Intel processors. The problem is that neither AMD's specification of x86-64 nor Intel's specification of x86-64 fully captures the range of implementations of the architecture.

The situation is also bad for architects extending specifications. Reliance on testsuites or verification against implementations creates a "chicken and egg" problem because implementors and test writers do not want to work on unstable, incomplete specifications but architects want to test changes to the specification while they are still developing the changes.

Our solution to this problem is to write high-level properties about the specification and to formally verify that the specification satisfies those properties.

One of the most important specifications that formal verification of software depends on is the processor specification that defines the boundary between software and hardware and on which formal proofs about the entire software and hardware stack are founded. In this paper, we consider properties about ARM's v8-M architecture specification [ARM Ltd 2016] that extends ARM's microcontroller specification with additional security features that software can use to improve the trustworthiness of Internet of Things devices.

We focus on cross-cutting features of the architecture and specify properties of the architecture as a whole involving exceptions, privilege and security.

Cross-cutting features [Kiczales et al. 1997] are difficult for humans because they require understanding of interactions between many disparate parts of the architecture and so subtle errors can slip through the cracks. The flip side of this is that writing cross-cutting properties can also scale well: a single cross-cutting property can catch an error in many parts of the architecture. We do not attempt to state properties about more cleanly decomposable parts of the architecture such as whether an ADD instruction performs addition. We believe that these properties are adequately served by existing techniques and that they would not give the same degree of leverage as our cross-cutting properties.

We developed sets of properties by examination of natural language text in the ARM architecture reference manual, by examining recently discovered bugs in the specification, and by discussion with the architects of the specification.

The central design challenge we face is to create a set of properties that:

- express the major guarantees that programmers depend on;
- are concise so that architects can easily review and remember the entire set of properties;
- are stable so that architecture extensions don't invalidate large numbers of rules;

- and that describe the architecture differently from the existing specification to reduce the risk of common-mode failure.

The ARM formal specification is split into many functions all rooted in a single top-level transition function that specifies any state change that the processor can make due to executing instructions, taking interrupts, etc. The classic compositional approach would be to tackle the problem hierarchically: stating and proving properties about the little functions at the bottom of the call tree of the specification, then using these properties as the basis for proofs about the functions in the next layer up the call tree and gradually working our way up the tree until all the properties of the specification have been stated and proved.

This conventional approach has a number of problems:

- *It is not how the architects view the architecture.* The architects describe the architecture at a high level in terms of the net effect of the architecture using statements like "if X happens then Y will happen" or "Y cannot happen unless X is enabled." Walking through the call tree or the individual lines of code in the specification is a secondary activity during their internal discussions. This observation is reflected in the natural language part of ARM's architecture specification and, we believe, reflects the way the architects think about the specification. Our experience is that, *when dealing with domain experts, there is significant benefit from formalizing their view of the domain instead of forcing them to use a different view.*
- *It does not aid understanding.* The problem we face in gaining confidence that the ARM specification is correct stems from all the fine detail in the existing specification: it is hard to see the forest for the trees. Adding properties to each function continues this problem: we better understand each tree but we still cannot see the forest.
- *It increases the maintenance burden.* Annotating every function with what would typically be multiple preconditions and postconditions would require significant specification and review effort. In addition, the internal structure of the specification is less stable than the boundary of the specification: functions are refactored, function arguments and results are added or removed, etc. The more that is written and proved about each individual function, the more there is to update as the specification evolves.

Accordingly, we adopt an "end to end" approach to writing properties: we only write properties that apply to the whole system. To make this practical, we extended conventional specification techniques based on predicates over the state with a novel kind of property inspired by coverage-based testing techniques.

Our verification is based on translating the specification plus the properties into verification conditions that an SMT solver can check. This allows the verification process to be entirely automated and requires no expert intervention.

The specification had previously been extensively tested [Reid 2016; Reid et al. 2016] but, despite this, we found a dozen bugs including two security bugs. Due to extensive testsuites used in ARM processor development and to redundancy between the natural language part of the specification and the formal part of the specification, these bugs had not impacted processors but they are important to anyone verifying software or hardware against the formal part of the specification.

To our knowledge, no realistic architecture specification has been subjected to this degree of formal verification before.

The remainder of this paper is structured as follows: Section 2 describes the coverage properties we use to write end-to-end properties; Section 3 sketches the ARM microcontroller architecture, describes how ARM writes architecture specifications and provides an introduction to how we write end-to-end properties; Section 4 further illustrates our approach with examples; Section 5 describes the design and implementation of our system; Section 6 describes our experience of using

the system; Section 7 describes related work; Section 8 describes limitations and future work; and Section 9 concludes.

2 COVERAGE PROPERTIES

The classic approach to writing properties is to write invariant properties and function properties using predicates that refer only to the state of the system before and after the state transition function. Our decision to limit ourselves to writing end-to-end properties makes it very hard to capture key properties only in terms of states. For example, some properties will only apply if the system takes a certain kind of transition such as taking a reset or an exception but these are hard properties to observe from the state alone. In principle, we could reverse engineer the initial conditions under which these events could occur but then we would be reasoning about when we *think* certain transitions occur instead of what the specification actually says.

Our solution to this problem takes its inspiration from the approach used in coverage-based testing techniques that use measurement of the coverage of the system under test to determine whether the tests are hitting the relevant parts of the system. For example, many programmers have added an ad-hoc "debug printf" to a program to confirm that a test hits some line of code or some condition. More rigorous applications of this approach are built into hardware design languages such as System Verilog [IEEE 2013] that provides a rich set of functional coverage mechanisms for tracking how many tests hit each case or combination of cases. Inspired by these test-based mechanisms, we augment the traditional Hoare-style properties about states with the ability to observe execution paths. The property *Called*(f) is satisfied for any execution of the function under test that calls the function f. For example, to state that an exception causes register R[0] to be set to zero, one could write the following (where ExceptionEntry is the name of the function that is called when an exception is taken).

Called(ExceptionEntry) ⇒ R[0] = 0

In some cases, finer-grained observation is important and we specify an additional predicate P that tests the values of the parameters when a function is called. For example, the function ExceptionEntry has a boolean parameter isSecure that specifies whether a secure or non-secure exception should be taken so the property

Called(ExceptionEntry **when** isSecure) ⇒ R[0] = 0

weakens the statement to say that R[0] is set to zero for *secure* exceptions.

Similarly, it is useful to write properties about function return and the values returned. We write *Returned*(f **when** P) to say that a function f returned successfully with values that satisfy the predicate P. (ARM's specification language includes exceptions so it is possible for a function to be called but not to return.)

We feel that observing execution paths in this way is a satisfactory compromise on our commitment to writing end-to-end properties: we mostly focus on the overall properties of the architecture but we allow references to some of the inner structure of the specification when required. In practice, we find that only a small fraction of the functions need to be observed in this way.

3 FORMALIZING ARM SPECIFICATIONS

The Internet of Things (IoT) adds network access to microcontroller-based systems: a class of devices that are small, cheap and energy efficient but not previously required to be secure. To meet this challenge, ARM created the "M-class" of processor that retains the positive characteristics but adds extensive security features. This does not eliminate the IoT security challenge but it gives software developers a sound foundation to build on.

The challenge in designing security features is that security is asymmetric: the designer has to get everything right but the attacker only has to find a single weakness to gain access. This makes the design and programming of Internet of Things devices appealing targets of formal verification research.

This section gives a brief introduction to the essential aspects of ARM's microcontroller specification referred to in this paper, describes how ARM writes architecture specifications and provides an introduction to their formalization.

3.1 The ARM v8-M Security, Privilege and Exception Model

ARM's v8-M architecture specification applies to ARM's 32-bit microcontrollers such as the recently announced Cortex-M23 and Cortex-M33 processors that are designed for embedded, low-cost devices at different performance points with a focus on security. Our focus in this paper is on writing and proving properties about specifications but, in order to explain the examples, it is necessary to describe a few key architectural concepts.

- An *exception* can be triggered by memory protection faults, security faults, interrupts, etc. If the cause of the exception was a fault then an appropriate field of the *Fault Status Register* is set to 1. Each exception has a priority and the processor selects the highest priority exception to work on. On taking an exception, the processor automatically saves the current context (user registers) onto the current stack and reads the address of the exception handler from an exception vector table in memory.
- A *derived exception* occurs if the act of taking an exception triggers a further exception. Two particular ways that derived exceptions occur are if saving the current context to the stack triggers a fault such as a stack overflow or if reading the exception vector triggers a fault. A cascade of up to two derived exceptions can result from an initial exception.
- *Lockup* occurs if a derived exception has lower priority than the original exception since there is then no way to report the derived exception. When in lockup, the processor sets the program counter to a distinctive lockup address and stops executing instructions.
- A processor can be in *Privileged* or *Unprivileged* mode. *Privilege* corresponds to the traditional protection mechanisms used by operating systems: only privileged execution mode is allowed to access system registers (including the memory protection registers).
- Orthogonally, a processor can be in *Secure* or *NonSecure* mode. The two modes share user registers but the stack pointer and many of the system control and status registers are *banked*: there are two copies and which copy is accessed depends on the current mode. *Security* goes beyond traditional processor-based protection and enforces access checks in peripherals and memory devices so that when a DMA controller or processor is executing in non-secure mode they cannot access secure peripherals or memory containing secrets such as crypto keys.
- An external debugger may request that the processor *halt* and can then examine and modify the processor and memory state. When the processor is halted, it is said to be in *Debug State*. It is possible to disable debugging of the processor when it is in Secure mode.

Inclusion of all these features makes the architecture more complex than a classic RISC architecture. There are multiple motivations for these features ranging from optimisations that can be performed in stacking/unstacking registers that make interrupt response faster and more deterministic; enabling interrupt handlers to be written in plain C code; and adding security features. It also means that some of the corner cases arising from the interaction of features only have to be handled correctly once by the hardware designers instead of having to be handled in many different software stacks. However, the combination of four different privilege/security modes, priority,

derived exceptions, debug, lockup and security adds considerable complexity to the architecture that makes testing and formal verification of the architecture specification desirable.

3.2 ARM's Specification Language

ARM's architecture specifications consist of two parts: a detailed, executable formal specification and a natural language part.

The formal part of ARM's specifications is written in ARM's Architecture Specification Language (ASL) that grew out of the pseudocode used in earlier versions of architecture reference manuals. At a high level, ASL is an indentation-sensitive, imperative, strongly-typed, first-order language with type inference, exceptions,[1] enumerations, arrays, records, and no pointers. All integers in ASL are unbounded and there is direct support for N-bit bitvectors and functions are allowed to be polymorphic in the width of a bitvector. For example, memory read returns a value of type **bits**(8*size) where size is constrained to be 1, 2, 4 or 8.

To make this more concrete, here is a small example of an ASL function that is called when a processor exception is triggered. The function pushes the current state onto the stack and, if this does not trigger a memory access fault, it calls the function ExceptionTaken that adjusts registers, swaps stacks, reads the address of the exception handler from memory and branches to it.

```
ExcInfo ExceptionEntry(integer exceptionType, boolean toSecure, boolean commitState)
    // PushStack() can abandon memory accesses if a fault occurs during the stacking sequence.
    exc = PushStack(commitState);
    if exc.fault == NoFault then
        exc = ExceptionTaken(exceptionType, FALSE, toSecure, FALSE);
    return exc;
```

The ARM v8-M formal specification is over 15,000 lines of code consisting of over 300 instructions and over 250 functions. This makes it one of the largest formal specifications we are aware of. The most important functions in the specification are: (1) The function that defines the initial state of the system. This function is called TakeColdReset and it specifies how the processor performs a "cold" reset (i.e., when first powered up). (2) The transition function. This function is called TopLevel and it specifies all types of transition that the specification can make: instruction fetch, instruction execute, entering and returning from processor exceptions, warm reset, entering/leaving Debug State, etc.

3.3 Rule Based Specification

ARM's architecture reference manuals also contain natural language statements about the architecture. Starting with the v8-M Architecture Reference Manual [ARM Ltd 2016], these are structured into a number of labelled "rules." Labels begins with the letter "R" for normative statements and with the letter "I" for informative statements and are followed by four randomly chosen letters.

We found that many of these rules simply repeated information found in the formal specification in much the same structure as the formal specification. These were not very useful for our purposes because they were somewhat low level and, worse, they were prone to common-mode failure wrt the formal specification. However, a small number of the rules stated high level properties about the architecture. For example, the following rule describe properties of how a processor can exit the Lockup state.

[1] The presence of "exceptions" in both the processor architecture and in ASL can lead to confusion as to which kind of exception we are referring to. In the remainder of this paper, we use "processor exception" and "ASL exception" to avoid confusion.

R$_{JRJC}$

Exit from lockup is by any of the following:

- *A Cold reset.*
- *A Warm reset.*
- *Entry to Debug state.*
- *Preemption by a higher priority processor exception.*

We interpret this to mean that the *only* way to exit from a lockup state is if one of the four listed conditions occurs. By examining the ASL specification, we found tests that could be used to formalize this statement:

- The variable LockedUp indicates whether the processor is in lockup.
- A cold reset is specified by TakeColdReset and a warm reset is specified by the function TakeReset;
- The variable Halted indicates whether the processor is in Debug state.
- Taking a processor exception is initiated by the function ExceptionEntry.

Based on this, one could choose to formalize the original statement in the following Hoare-triple

$$\{\ \mathsf{Invariants} \wedge \mathsf{LockedUp}\ \}$$
$$\mathsf{TopLevel}();$$
$$\{\ \neg\ \mathsf{LockedUp} \Rightarrow \mathit{Called}(\mathsf{TakeColdReset}) \vee \mathit{Called}(\mathsf{TakeReset}) \vee (\neg \mathsf{Halted'} \wedge \mathsf{Halted}) \vee \mathit{Called}(\mathsf{ExceptionEntry})\ \}$$

where Invariants is the conjunction of all the invariants for the system and Halted' represents the value of Halted before the function is called. (This omits the requirement that preemption must be by a higher priority exception. This is a general requirement on all processor exceptions and we chose to specify it in a separate property.)

Even for this simple rule, we found this notation to be quite unwieldy so we introduced some syntactic sugar to let us write properties in a more structured way.

- Each property is labelled for ease of reference.
- Instead of using the v' convention for accessing the old value of a variable v, we provide an operator $\mathit{Past}(e)$ that refers to the value of an expression e before the function under test was called.
- Following the example of System Verilog Assertions [IEEE 2013], we define syntactic sugar for some common uses of the *Past* operator.

$$\begin{aligned} \textsc{Stable}(e) &\triangleq \textsc{Past}(e) = e \\ \textsc{Changed}(e) &\triangleq \textsc{Past}(e) \neq e \\ \textsc{Rose}(e) &\triangleq \textsc{Past}(e) < e \\ \textsc{Fell}(e) &\triangleq \textsc{Past}(e) > e \end{aligned}$$

  By abuse, *Rose* and *Fell* can also be applied to boolean expressions.
- We separate the assumptions from the consequences of those assumptions to improve readability.
- We omit the name of the function under test because we wish to test the same invariants on both the reset function and the transition function and because there is only one transition function.

In our notation the above property is written as follows.[2]

[2]To avoid distraction, we have simplified the ASL language slightly in this paper: omitting explicit type conversions and using mathematical symbols such as =, ≠ and ∧ where the concrete syntax uses conventional programming notation such as "==", "!=" and "&&".

property R_JRJC
  assume *Fell*(LockedUp);
  Called(TakeColdReset) ∨ *Called*(TakeReset) ∨ *Rose*(Halted) ∨ *Called*(ExceptionEntry);

We feel that this is a reasonably close match to the structure of the original natural language statement.

Invariants: Invariants are properties that should initially be valid and then their validity should be preserved by any action the processor takes. An example invariant is that a processor can be in Lockup or it can be Halted but it cannot be both. In our notation, this property is written like this.

invariant dbg_lockup_mutex
  ¬(Halted ∧ LockedUp);

Unpredictable Behaviour: ARM's specifications are deliberately incomplete and do not specify what a processor should do in all circumstances. ARM labels these gaps in the specification as *UNPREDICTABLE* and the processor is free to do anything that can be achieved at the current or a lower level of privilege using instructions that are not *UNPREDICTABLE* and that does not halt or hang the processor or parts of the system.

Most unpredictable behaviour in the ARM specification is associated with attempting to do something that is nonsensical and that can be easily avoided by the programmer. In ASL, unpredictable behaviour is marked by the statement *UNPREDICTABLE;*. To let us distinguish executions that do not execute this statement, we add a new property Predictable that is true for executions that do not execute *UNPREDICTABLE;*.

Implicit assumptions: All of the properties that we wish to prove about the transition function only hold under the restrictions that the initial state satisfies the invariant, and the execution is Predictable. These restrictions could be added to each individual property by adding the following assumptions:

  assume *Past*(Invariants);
  assume Predictable;

where Invariants represents the conjunction of all the invariant properties. Such assumptions would be the same for all properties and would only serve to add noise to our properties so, instead, we choose to leave these restrictions implicit and add them in our proof tool (see Section 5.3).

4 EXAMPLES

This section illustrates the use of the notation introduced in the previous section to write further properties about the architecture and it will look at the challenges in formalizing rules found in the natural language part of the specification.

4.1 The Exception Entry Bug

One of our motivating examples in this work was trying to detect a bug that had recently been found in the v8-M specification and to prove that any bugfix does, indeed, fix the bug.

In order to write a property that would detect what the specification did wrong we asked the v8-M architects how they could tell that the bug had occurred (but not to describe the bug itself). They told us that the bug involved what state is saved on taking a processor exception. From testing, they knew that the state was usually saved correctly but, under some circumstances, the specification was not saving information about which stack the interrupted context was saved on.

The current stack selection is recorded in the field SPSEL of the register CONTROL. On entry to a processor exception handler, the current stack selection that was active before the exception is recorded in bit two of the register LR. Using our notation, we formalized the property like this.

```
property exn_entry_spsel
    assume CALLED(ExceptionEntry);
    assume ¬CALLED(TakeReset);
    assume ¬CALLED(ExceptionReturn);
    PAST(CONTROL.SPSEL) = LR<2>;
```

Having specified the required property, we used our tool to attempt to rediscover the bug. Our tool generated a counterexample that the architecture development team confirmed as a possible symptom of the bug they had previously found. In particular, the bug occured if the attempt to save the state of the processor on the original stack failed and the processor exception could not be escalated to an appropriate handler (e.g., because the processor was already in the highest priority exception handler). In this case, the processor enters the Lockup state and stops execution but even in this desperate circumstance, it is required that the originating mode and security level are saved correctly in LR to enable the problem to be diagnosed through the debugger.

After confirming that the properties could detect the original bug, we applied a bugfix proposed by the architecture team and repeated the check. To our relief, all of the processor exception entry properties were found to hold: our first formal verification that a bugfix actually fixed a specification bug.

4.2 Property Groups

Properties often share a number of assumptions and triggering conditions so we find it useful to group multiple properties together to allow them to share common antecedents.

For example, when a processor exception is taken, the processor doesn't just save the current stack selection, it also saves the current security state, the exception mode, whether the floating point state is "dirty", etc. We provide some syntactic sugar for writing sets of related properties sharing a common set of antecedents. The first sub-property in the following is equivalent to the exn_entry_spsel property above.

```
rule exn_entry
    assume CALLED(ExceptionEntry);
    assume ¬CALLED(TakeReset);
    assume ¬CALLED(ExceptionReturn);

    property spsel: PAST(CONTROL.SPSEL) = LR<2>;
    property secure: FELL(IsSecure()) ⇔ (LR<0> = '1');
    property mode: PAST((CurrentMode() = PEMode_Handler)) ⇔ LR<3> = '0';
    property ftype: PAST(CONTROL.FPCA) = NOT LR<4>;
```

4.3 Entry to Lockup

One of the more challenging rules to formalize was the following rule that describes entry to Lockup.

R_{VGNW}

Entry to lockup from a processor exception causes:

- *Any Fault Status Registers associated with the exception to be updated.*
- *No update to the exception state, pending or active.*
- *The PC to be set to 0xEFFFFFFE.*
- *EPSR.IT to be become UNKNOWN.*

In addition, HFSR.FORCED is not set to 1.

Each bullet in this rule required careful interpretation/debugging in order to understand it.

- Other rules detail which fields of the Fault Status Registers should be set but one consequence of the first bullet is that at least one bit in Fault Status Register CFSR should be set after entry to lockup (in configurations that provide the CFSR register).
- The second bullet turned out to be false: pending and active processor exception state should be updated to reflect the attempted exception entry. The statement had been true in the previous version of the architecture but had not been updated. We filed a bug against the documentation.
- The third bullet suggested that we check that PC = 0xEFFFFFFE but this property failed with counterexamples where PC was equal to 0xF0000002. On investigation, we found that the rule was implicitly referring to the "debug view" of the program counter that, for historical reasons, reads as four less than the "program view" that is accessed as PC. We filed a clarification request against the documentation.
- The fourth bullet is untestable because setting a register to *UNKNOWN* is allowed to choose any value — including the current value of the register. We are currently unable to formalize this statement.
- The final sentence seemed to allow multiple intepretations including HFSR.FORCED must become 0 or may become 0 or must not be changed. After consulting the architects, we learned that it meant that HFSR.FORCED is not modified. We filed a clarification request against the documentation.

Of course, the task of determining which interpretation to use is not quite as direct as suggested above and in practice, we followed a more experimental methodology. We would typically formalize several different interpretations of each clause of a rule; we test which interpretations hold for the specification; and we consult the architects to confirm that the winning interpretation is the intended interpretation. This lead to the following set of properties

```
rule lockup_entry
    assume Rose(LockedUp);
    assume ¬Called(TakeReset);

    property R_VGNWa: HaveMainExt() ⇒ CFSR ≠ 0;
    property R_VGNWc: _RName[RNamesPC] = 0xEFFFFFFE;
    property R_VGNWe: Stable(HFSR.FORCED);
```

4.4 Exit from Lockup

The example rule in Section 3 described when a processor could exit the Lockup state. The following rule describes one part of what a processor should do when that happens.

> R_{SPPN}
>
> *On an exit from lockup by entry to Debug state, or by preemption by a higher priority processor exception, the return address is 0xEFFFFFFE.*

We initially formalized the debug part of this rule with the following property.

```
property R_SPPNa
    assume Fell(LockedUp);
    assume Rose(Halted);
    LR = 0xEFFFFFFE;
```

Our tool reported that this property did not hold and, on investigation, we realized that we had misinterpreted the phrase "return address." When an ARM processor is executing instructions, the return address is normally held in register LR but when an ARM processor is in Debug state, the return address is held in the program counter and when an exception is taken, the return address is held on the stack. The amended formalization read as follows for the debug case

```
property R_SPPN
    assume Fell(LockedUp);
    assume Rose(Halted);
    _R[RNamesPC] = 0xEFFFFFFE;
```

We filed a clarification request against the documentation and recommended splitting these two cases.

4.5 Lockup Invariants

Lockup occurs when a fault occurs and it is not possible to report the fault because the appropriate fault handler is lower priority than the current execution priority. A consequence of this is that, under normal circumstances, Lockup can only occur in the highest priority processor exception handlers: Non-maskable Interrupt NMI and HardFault. We formalized this as follows using the Interrupt Program Status Register IPSR to read the current processor exception handler and adding the additional assumption that execution priority had not been boosted using the FAULTMASK register.

```
invariant lockup_IPSR
    assume LockedUp;
    assume FAULTMASK.FM = 0;
    IPSR ∈ {NMI,HardFault};
```

A further rule about Lockup states

> R_{MBTM}
>
> *When the PE is in lockup:*
> - *DHCSR.S_LOCKUP reads as 1.*
> - *The PC reads as 0xEFFFFFFE. This is an execute never (XN) address.*
> - *The PE stops fetching and executing instructions.*
> - *If the implementation provides an external LOCKUP signal, LOCKUP is asserted HIGH.*

We formalized this as follows.

```
rule R_MBTM
    assume LockedUp;
    invariant a DHCSR.S_LOCKUP = 1;
    invariant b PC == 0xEFFFFFFE;
    property c
        assume Past(LockedUp);
        ¬ Called(FetchInstr) ∧ ¬ Called(DecodeExecute);
```

Attempting to prove these properties found that the DHCSR register was only partially implemented in the specification and that PC referred to the debug view of the program counter and we filed bugs against the specification and the documentation.

4.6 Preemption by Processor Exceptions

As a final example, an important property of the processor exception mechanism is that execution can only be preempted by higher priority exceptions. Since higher priority is represented by smaller numbers, this says that if a processor exception is successfully taken then the priority value must be lower than it was before the transition. In formalizing and proving this statement, we found a counterexample: the priority need not increase if the program triggers a derived exception while attempting to perform a processor exception return (e.g., because of a memory fault while popping the exception frame off the stack). Our amended statement is as follows.

```
property priority_increase
    assume Called(ExceptionEntry);
    assume !Called(ExceptionReturn);
    ExecutionPriority() < Past(ExecutionPriority());
```

Interestingly, the complementary property does not always hold: exception return does not always lead to a decrease in priority (that is, an increase in priority number) because an exception handler can dynamically change the priority of an interrupt before returning.

4.7 Summary

This section described several properties we created by talking to the architects or by translating natural language "rules" to our property notation. The process of formalizing and of attempting to prove the properties found several bugs in both the formal part of the specification and in the natural language part.

The bugs we found in the formal specification typically involved corner cases that trigger cascades of derived processor exceptions, exceptions triggered when returning from an exception, exceptions triggered because the vector table is in an unreadable part of the memory space, etc. These bugs tend to creep into a specification because humans find it hard to think about all of the corner cases and because it is natural to focus on your current task when extending the architecture and to forget about all of the cross-cutting issues.

It is not surprising to find ambiguous, misleading and erroneous statements in natural language specification — even one as heavily reviewed as the ARM specification. It took a process of experimentation to find the correct interpretation of some statements although, a bit like a good crossword puzzle, our final solution was obvious once we knew what it was. Our property language and checker allows us to perform those experiments and to confirm that those results are consistent with the formal specification; and the act of formalizing the statements helps us formulate clearer, more accurate natural language statements.

```
⟨definition⟩ ::= 'rule' ⟨ident⟩
                 { 'var' ⟨ident⟩ ':' ⟨type⟩ ';' }
                 { 'assume' ⟨prop⟩ ';' }
                 { ('property' | 'invariant') ⟨ident⟩ ⟨expr⟩ ';'
                 }
               | ('property' | 'invariant') ⟨ident⟩
                 { 'var' ⟨ident⟩ ':' ⟨type⟩ ';' }
                 { 'assumes' ⟨expr⟩ ';' }
                 ⟨expr⟩ ';'

⟨expr⟩       ::= 'Past' '(' ⟨expr⟩ ')'
               | 'Rose' '(' ⟨expr⟩ ')'
               | 'Fell' '(' ⟨expr⟩ ')'
               | 'Stable' '(' ⟨expr⟩ ')'
               | 'Changed' '(' ⟨expr⟩ ')'
               | 'Called' '(' ⟨ident⟩ ['when' ⟨expr⟩] ')'
               | 'Returned' '(' ⟨ident⟩ ['when' ⟨expr⟩] ')'
               | 'PREDICTABLE'
               | 'Invariants'
```

Fig. 1. Property Syntax Extensions

5 DESIGN AND IMPLEMENTATION

This section describes the semantics and implementation of the property notation described in earlier sections.

5.1 Property Language

Our notation for specifying invariants and properties extends the ASL specification language with the ability to refer to the values of expressions before execution of the code under test; to test whether an execution performs an action such as calling a function; and to name properties for ease of reference. It also adds some syntactic sugar for defining groups of larger properties. The grammar for these extensions is shown in Figure 1 which defines additional productions for the ⟨definition⟩ and ⟨expr⟩ non-terminals.

Our property language blends two different notions: conditions involving the *state* of the processor before and after a processor transition; and conditions involving the execution path taken while executing the state transition function. Defining the semantics of this combination requires two steps:

- We extend the stateful semantics of the ASL language with generation of a trace during execution. The details of this extension are unsurprising and results in a trace of function call and return events. Function call events are represented by $C\langle f, \bar{a}\rangle$ consisting of the name f of the function and a binding $\bar{a}$ of the function's formal parameters to the values of each actual parameter of the call. Function return events are represented by $R\langle f, \bar{a}, \bar{r}\rangle$ consisting of the name f of the function and bindings $\bar{a}$ and $\bar{r}$ of the function's formal parameters and results to the names of each function argument and return variable in the function.
- We define the semantics of the CALLED and RETURNED operators in terms of this trace. For any terminating execution producing a trace T, the CALLED(f **when** P) operator is satisfied if T contains an element $C\langle f, \bar{a}\rangle$ such that $[\![P]\!]_{\bar{a}}$ is satisfied where $[\![\_]\!]_\rho$ is the semantics of evaluating a expression wrt a binding ρ. Similarly, the RETURNED(f **when** P) operator is satisfied if T contains an element $R\langle f, \bar{a}, \bar{r}\rangle$ such that $[\![P]\!]_{\bar{a}\cup\bar{r}}$ is satisfied.

5.2 Implementation

A key requirement for practical deployment is that all proofs should be performed automatically without needing to train the authors of the architecture in the use of an interactive proof assistant. Our implementation therefore is based on translating the architecture specification and the properties to be checked into verification conditions suitable for SMT solvers. This translation consists of three major steps: converting property specifications to ASL; a number of "lowering passes" that convert complex language features into simpler language features; and converting the simplified ASL specification to a verification condition expressed in the SMT-Lib language [Barrett et al. 2016].

5.2.1 Converting Properties to ASL. Properties are written using an extension of ASL so we convert each extension into the original ASL language. This is done by introducing "ghost variables" to collect information needed by the properties and adding code to initialize, update and test these variables that should execute before, during and after the function under test. We introduce two new functions SMTPre and SMTPost to hold the statements that execute before and after execution of the function under test.

- The *Past*(e) operator is implemented by introducing a fresh global variable v and adding an assignment v = e; to the SMTPre function. Occurrences of *Past*(e) are replaced by v.
- The *Called*(f **when** P) and *Returned*(f **when** P) operators are implemented by introducing a fresh global boolean variable v initialized to FALSE and instrumenting each function with an assignment v = v ∨ P;. The assignment is placed at the start of the function for *Called* and before each Return statement for *Returned*. Occurrences of the operator are replaced by v.
- Invariant properties are evaluated before and after the function under test by creating two fresh global variables pre and post adding assignments pre = P; and post = P; to SMTPre and to SMTPost, respectively. Our proof frontend uses the conjunction of all the pre-variables and (separately) all the post-variables when proving that properties hold.
- All function properties are evaluated after the function under test by creating a fresh global variable v and adding an assignment v = P; to SMTPost. Our proof frontend replaces the property name with v.

With this conversion, testing whether a property holds for some function f consists of checking whether the corresponding global variable is TRUE after executing the sequence SMTPre(); f(); SMTPost();.

5.2.2 Simplifying ASL. The challenge in translating the rich, expressive ASL language to an SMT problem is that SMT-Lib [Barrett et al. 2016] is a pure expression language and lacks polymorphic types, dependent types, function calls, control flow, assignments, exceptions and structured data types.

Before starting translation, we apply a number of "lowering passes" that convert complex language features into simpler language features. The primary transformations performed in these passes are

- Eliminating dependent types and polymorphism by specializing all instructions and creating monomorphic instances of all polymorphic functions. For example, the memory load instruction can perform an 8, 16, 32 or 64-bit memory access based on a 2-bit size field of the instruction encoding. This results in many intermediate variables and function arguments whose width is dependent on the value of the size field. The specialization pass creates 4 separate instances of the instruction each of which accesses a single data width.
- Unrolling all loops. In our application, we were fortunate that it was always possible and often trivial to find an appropriate loop bound. There was one use of recursion but the architects were easily persuaded that rewriting it would make it easier to understand. Had this not been the case, we would have resorted to bounded unrolling and bounded recursion depths as is common practice elsewhere [Clarke et al. 2004].
- Eliminating unstructured control flow using a simplified form of if-conversion [Allen et al. 1983]. ASL does not have goto but it provides functions that return in the middle of a function and provides exception throwing that can exit in the middle of a function. This is converted to structured control flow by introducing an additional control variable into each function. This variable is initially true but it is set to false in the event of function return or an ASL exception and the variable is used as a guard to disable actions of statements if the variable is false.

- Global context-insensitive, flow-insensitive, structure-insensitive constant propagation and dead code elimination to exploit the large number of constants introduced by the previous passes. The choice of global/local and sensitive/insensitive propagation is based on our understanding of the structure of the specifications we wish to reason about.

These preprocessing steps reduce the ASL specification to a simple monomorphic, imperative language with functions, structured control flow and no loops or recursion.

5.2.3 Converting ASL to Verification Conditions. The remainder of the transformation is performed by symbolically executing the specification using SMT expressions as symbolic values and with each step of the evaluation extending a graph of SMT expressions representing the data/control flow of the program. When control-flow splits, the control expression is remembered, both control paths are executed using separate copies of the current execution environment, and when control-flow joins, the two execution environments are merged by introducing if-then-else nodes to select values from one path or the other. Function calls are handled in the usual way for an interpreter: a fresh environment is created containing the values of the function arguments and the function body is evaluated in that environment. Uninitialized variables and *UNKNOWN* expressions are handled by introducing oracles (that is, fresh variables that are unconstrained).

Unfortunately, this conventional translation resulted in excessively large SMT problems and we were unable to generate SMT problems for even the smallest architecture configuration.

To overcome this, we implemented four important optimisations:

- When merging environments, we omit the if-then-else node if neither environment has changed the value of a variable.
- We perform "hash-consing" to avoid creating nodes that are identical to a previously constructed node. This increases the effectiveness of the first optimization in the case that both branches set a variable to the same value.
- When evaluating an if-statement, if the control expression is definitely true or definitely false then we avoid exploring the dead branch. This is a significant optimization.
- We perform a limited amount of constant folding to catch constant propagation opportunities that were missed during preprocessing. Our primary goal in doing this is to evaluate boolean conditions to make the third optimization more effective.

After implementing these optimisations, the generated SMT expression was still large: approximately 30,000 terms for TakeColdReset and between 360,000 terms and 860,000 terms for TopLevel depending on the architecture configuration tested. We found that the Z3 SMT solver [De Moura and Bjørner 2008] was able to handle problems of this size but we found that even proving very shallow properties took 30-60 minutes: this put the feasibility of tackling interesting properties in question. To resolve this, we consulted one of the Z3 developers [Wintersteiger 2017] who suggested that we further simplify the SMT expression by avoiding use of high-level constructs such as enumerated types and arrays whenever possible. Replacing enumerated types with small bitvectors was an easy change but to avoid arrays we had to construct expressions that closely resemble the way that register files are typically implemented in hardware using address decoder trees to write array elements and using trees of multiplexors to read array elements. These additional optimisations reduced the need to switch between different theories when solving problems and resulted in a performance improvement of approximately 5x. Solution times with the above optimisations are detailed in Section 6.3.

5.3 Proof Frontend

The final part of our implementation is a proof frontend that uses the Z3 solver to prove that invariant properties and function properties hold.

For each invariant I, we check two properties (expressed here as Hoare triples)

$$\{\}\ \textsc{TakeColdReset}()\ \{I\}$$

$$\{\textsc{Invariants}\}\ \textsc{TopLevel}()\ \{\textsc{Predictable} \Rightarrow I\}$$

For each function property P, we check the property

$$\{\textsc{Invariants}\}\ \textsc{TopLevel}()\ \{\textsc{Predictable} \Rightarrow P\}$$

For each assertion or bounds check P, we check the properties

$$\{\}\ \textsc{TakeColdReset}()\ \{P\}$$

$$\{\textsc{Invariants}\}\ \textsc{TopLevel}()\ \{P\}$$

5.4 Debugging Properties

Given the large size of the state space, we found it hard to debug failing properties just by examining the initial and final states. To help us understand counterexamples, we added the ability to emit code that would set the processor registers to the final state.

A minor challenge in doing this is that the generated SMT problem loses several type distinctions that were present in the original ASL: we solved this by emitting a file containing the ASL-level type of every SMT variable that our proof tool could use to generate type-correct ASL code. This was used with an interpreter for ASL that provides useful debugging features such as displaying the call tree of an execution, displaying register reads/writes, an interactive mode, etc. Using the interpreter to animate counterexamples proved to be essential for understanding bugs in the specification and when testing speculative properties and invariants. It was also useful while developing the transformation from ASL to SMT for identifying differences between the transformation and the interpreter that indicated bugs in the transformation.

A more significant challenge is that ASL allows underspecification (i.e., the specification does not completely constrain the behaviour in some circumstances). Our ASL interpreter handles this by choosing just one possible behaviour whenever the specification provides a choice. In contrast, the SMT solver explores all possible behaviours and may find a counterexample that is allowed by the specification but that is not the behaviour chosen by the ASL interpreter. When the underspecification affects the control path in the specification we can see significant divergence between the interpreter and the SMT solver. This has prevented us from debugging some of the failing properties found by our tool (Section 6) and is the subject of future work.

6 EXPERIENCE

We subjectively feel that our properties closely reflect the rules we formalized and, hence, the way that architects view the architecture. More objectively, we evaluate the effectiveness of our approach based on the ability of our properties to find bugs, and the efficiency of proof.

6.1 Formalizing Natural Language Specifications

Our original intention in this project was to focus on verifying properties that would be useful to programmers or that the architects identified as having been hard to get right (e.g., based on bugfixes to the specification). When we realized that some of the natural language rules in ARM's existing architecture specification could also be formalized using our tools, we shifted our focus to formalizing those rules and added more structure to our notation to better match the style of those rules.

We are also working with the team responsible for creating and maintaining the natural language part of ARM's architecture documents about two improvements to the rule style. The first

improvement is adoption of a more structured approach where rules are categorized according to the type of constraint expressed and each type is then written in a consistent way. For example, responses to exceptional events might be written in a sentence structure like this:

> $R_{\langle label \rangle}$
>
> *IF* ⟨*optional preconditions*⟩ ⟨*trigger*⟩, *THEN the* ⟨*system name*⟩ *shall* ⟨*system response*⟩

(This approach is inspired by the "EARS" requirements specification style [Mavin et al. 2009] used by Rolls Royce PLC to write the requirements of their avionics engine control systems.) The second improvement is adoption of a standardized terminology to describe the triggers, actions and responses. For example, there should be only one way to describe a signal becoming '1', only one way to describe a signal changing from '0' to '1' and only one way to describe a signal remaining unchanged. These changes to improve consistency are motivated by a desire for clarity and ease of understanding and is especially valuable for customers who are not native English speakers.

The more formal notation described in this paper is also a structured way of capturing rules: the notation for properties and the assumptions they rely on provides a high level structure while the ASL notation coupled with temporal operators such as *Rose* and *Stable* provides a standard way to describe signal values and their changes.

We are starting to look at extending the formal notation with structures and operators directly corresponding to the sentence structures and terminology used in the natural language rules. Our hope is that we can narrow the gap between the two notations so that our formal properties are "eyeball close" to the corresponding informal rule: that is, identically structured and using corresponding terminology/notation so that humans can easily see that they have the same meaning. We don't expect this to be possible for all rules but the experience reported in Section 4 suggests that it should be possible.

An obvious further step would be to write rules in a style that can always be directly translated to formal properties or, conversely, to write properties that can be automatically converted from our formal notation to English sentences [Burke and Johannisson 2005, for example]. Our current feeling is that this would be a step too far: it is possible and desirable to narrow the gap between natural language and formal notation but there is a tension between the best way to express rules so that humans from different technical backgrounds can understand them and the best way to express properties to enable machine proofs. This tension is especially strong when the specification has to deal with new concepts for which we do not have a good mathematical theory and may not yet know how to formalize or prove a property. For example,

- We cannot currently formalize statements about *UNKNOWN* (see Section 4.3).
- The best way to formalize memory concurrency semantics is still an active area of research with no clear agreement between an operational approach (e.g., [Flur et al. 2016]) and an axiomatic approach (e.g., [Alglave et al. 2014]).
- It is not clear how to formalize statements about security properties of the architecture.

In such cases, we must start with a natural language specification, then formalize rules as techniques and understanding develop and only then hope to find a way of structuring both the rules and the properties to be "eyeball close."

6.2 Bugs Found

We checked the properties on two configurations of the v8-M architecture: one with security extensions enabled and one with security extensions disabled. The configuration with security extensions had previously been heavily tested [Reid 2016] but the configuration without security

extensions was relatively untested. In addition, debug features had only recently been implemented and only partially tested.

We checked all properties on both configurations and, in the process, we found twelve bugs in the formal part of the specification and 9 issues in the natural language part including:

- *Trivial Bugs*: Array bounds failures, a guarding test that was placed after the action it was meant to guard instead of before the action, and an uninitialized variable.
- *Unimplemented/untested functionality*: Some parts of the debug specification were ignoring a "debug disable" control signal;
- *System register problems*: We found several bugs in the machine-readable description of the system registers. Some status fields in system registers gave incorrect information about the state of the processor while others should have masked the status information based on a control field but did not.
- *Ambiguity of Natural Language*: Some bugs were not in the ASL part of the specification but in the natural language part that specifies the rules.
- *Imprecision of Natural Language*: Section 4 contains several examples where the specification referred to the PC but meant the "debug view of the program counter." These lead to considerable confusion and attempts at bugfixes failed until we understood the subtle distinction between the two.
- *Processor exception entry*: The example discussed in Section 4.1 was already known to the architects but we were able to detect it using very high level properties without knowledge of the details of the bug.
- *Mixed logic polarity*: The security parts of the specification use boolean variables where TRUE indicates that something *is* secure and they also use variables where TRUE indicates that something *is not* secure. That is, it uses both positive logic and negative logic. We found a bug where a variable of one polarity was passed to a function that expected a variable of the opposite polarity.
- *Secure accesses from NonSecure processor*: The most serious of the bugs we found was in the configuration with security extensions disabled. In this configuration, the processor should behave as though it was in the NonSecure state: all accesses should be non-secure. Our tool found a case where the processor was treating accesses as secure.

The most difficult and tedious part of this process was in creating invariant properties. Many of the invariant properties were added in response to puzzling counterexamples involving processor states that seemed to be nonsensical. After some time staring at these examples, we would convince ourselves that these nonsense states were unreachable and we would add and prove another invariant.

6.3 Proof Time

We wrote a total of 59 function properties and invariants. In addition the specification already contained assertions and we added additional array bounds checks during SMT generation. We test each of the invariants on both the initial function TakeColdReset and on the transition function TopLevel and we test the function properties on TopLevel as detailed in Section 5.3. We applied our tool to two architecture configurations: "NS" with security extensions disabled and "S" with security extensions enabled. Our SMT generator omits checks for assertions and bounds checks that are provably satisfied at generation time so the number of assertions and of bounds checks varies slightly between configurations.

For this experiment, we ran all properties on an Intel(R) Xeon X5670 at 2.93GHz equipped with 48GB memory. We attempted to prove 315 verification conditions. Each proof attempt was run

Table 1. Number of properties of different classes for the TakeColdReset and TopLevel functions and number of proofs that pass, fail or timeout in 30000s.

	TakeColdReset			TopLevel			
	Asserts	Bounds	Invariant	Asserts	Bounds	Invariant	Properties
Configuration = NS							
Total	21	2	38	36	2	38	23
Passed	21	2	38	36	2	36	21
Failed							1
Timeout						2	1
Configuration = S							
Total	18	3	38	32	3	38	23
Passed	18	3	38	29	3	33	19
Failed							2
Timeout				3		5	2

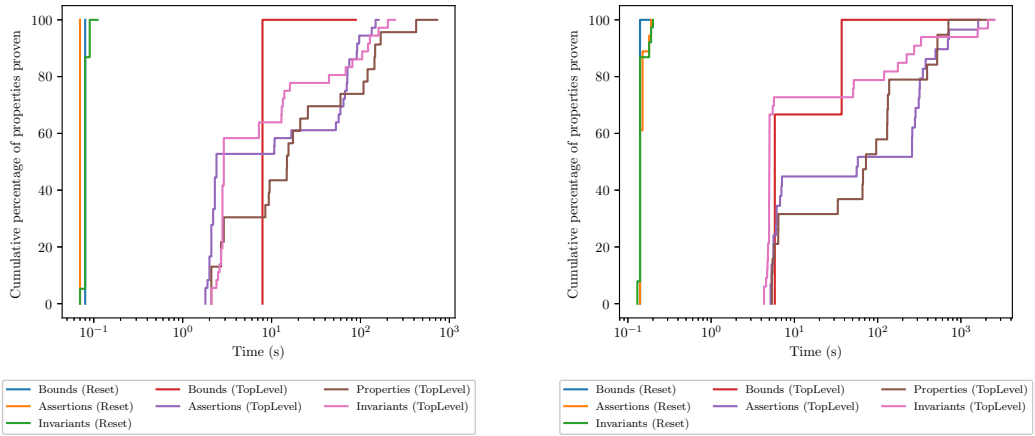
with a timeout of one day and we were able to prove 299 of them within the timeout. The results are summarized in Table 1. The 3 failing properties are still being diagnosed but are probably due to missing invariants. The presence of 7 timeouts on the invariants means that our proofs are not yet sound but this has not prevented us from using the tool for finding bugs. The presence of 6 timeouts on assertions and properties means that some of those properties could yet fail if given enough time to run the checks. Though less worrying, we hope to reduce the size of the SMT problem we generate and that that will allow these proofs to terminate one way or the other in an acceptable time.

To help understand the timeouts in Table 1, Figures 2a and 2b summarize what fraction of properties can be proved in a given time interval for the "NS" and "S" configuration, respectively. As one might expect, proofs about the reset function are fairly trivial and take just a fraction of a second while the amount of choice present in the transition function makes proofs about TopLevel take longer. The graphs show that the properties for the "NS" configuration are typically proved 3-10x times faster than the properties for the "S" configuration but even for the "S" configuration most proofs are generated within 1000 seconds. The total time taken for all passing proofs is under 5 hours and using a 1000 second timeout would result in the tests that fail or timeout taking another 4 hours. In practice, the proof effort should parallelize nicely so the total elapsed time is primarily bounded by the number of properties that fail or timeout.

6.4 Notation

As Section 5.2.2 shows the ASL language used in the main specification is a little awkward for the work described in this paper: it would be easier to translate ASL to an SMT problem if all loops had explicit bounds, if ASL did not support exceptions and did not allow return from the middle of a function, if ASL did not provide unbounded integers, etc. On the other hand, simplifying ASL in such ways would make the language less readable, less robust or require a more subtle semantics (whose finer details might be lost on some readers).

ASL is a compromise specification language intended to be useful to multiple communities inside and outside ARM: OS engineers, compiler engineers, hardware engineers, hardware verification engineers, authors of tests, JIT writers, creators of simulators, documentation teams and formal verifiers of software. Enabling new user groups and applications increases the utility of the specification, detects previously undiscovered bugs in the specification and, through successful use, increases our confidence in the specification. The cost of these benefits is that each individual use



(a) Time to prove properties about configuration NS

(b) Time to prove properties about configuration S

Fig. 2. Cumulative time to prove properties

is more difficult than if the specification and tools were optimized for that individual purpose. For example, ARM's hardware engineers engineers would generally find an unoptimized processor implementation written in Verilog to be easier to understand and easier to verify against than the ASL specification. This would lead to fragmentation of the specification since a specification written in Verilog would only be useful to hardware engineers and other groups would use one or more separate and incompatible specifications.

7 RELATED WORK

There are two areas of closely related work: formal specification of processors and formal validation of requirement specifications.

This paper is concerned with the general problem of trusting large specifications but its particular focus is on specifications of processors. Most recent papers on creating processor specifications describe how they tested their specification. The most extensively tested processor specifications are the executable ARM specifications described in our previous work [Reid 2016; Reid et al. 2016] and the executable x86-64 specifications created by Goel et al. [Goel et al. 2014]. Both have been verified using substantial programs: Reid uses ARM's architecture conformance testsuite while Goel runs real programs including (amusingly) a SAT solver. ARM has publicly released their v8-A processor specification in machine readable form. Reid et al. also formally verify ARM processor pipelines against the instruction set part of the specification: this increases confidence in the instruction set part of the specification but it says little about the system architecture part of the specification that is our primary concern in this paper.

Other notable processor specifications are the Fox/Myreen ARM v7-A ISA specification in HOL [Fox and Myreen 2010] and Flur et al.'s ISA and concurrency specification in SAIL [Flur et al. 2016] both of which were tested against actual processors using random and directed tests (8400 tests in Flur et al., 281,307 tests in Fox/Myreen). The other major ARM ISA specification that we are aware of is embedded in the CompCert compiler and is used in the proof that the compiler faithfully translates the input C program to ARM assembly code. This specification is limited to a subset of the user-mode ARMv6 specification and there is no published statement of how it was validated.

It is clear that testing of processor specifications is becoming standard practice but all of the above work consists of testing or formally verifying the specification against implementations of that specification. They are therefore vulnerable to the problems we identified in the introduction: (1) Testing of the specification cannot begin until the first implementation or a test suite is produced; and (2) Testing against an implementation of the specification is vulnerable to common mode failure. Our approach avoids these pitfalls by relying on formalization of high level properties that the specification is intended to meet and it avoids the well known limitations of testing by using formal verification techniques.

We believe that our *Called* operator is a novel feature in formal specifications but it can be seen as a adaptation of the coverage measurement features found in System Verilog [IEEE 2013] that allow verification engineers to annotate Verilog programs with specific coverage goals. Verilog simulation tools generate reports of how many times each coverage goal was hit allowing verification engineers to confirm that their test harness is exercising the desired behaviour. Alternatively, in software testing, it is common to add debug printf statements to a program to confirm that a certain path is being tested.

Another important part of processor architecture is the specification of the memory concurrency semantics [Alglave et al. 2014; Flur et al. 2016; Sarkar et al. 2011]. These specifications are tested extensively against commercial processors. More recently, the MemAlloy tool has been created for automatically comparing memory consistency models [Wickerson et al. 2017]. Although different in almost every detail, we see this as solving a similar problem: understanding if a specification is correct without the need to wait until an implementation is available.

Our work can also be seen as a variant on formal validation of requirements specifications[3]. The Alloy language and analyzer [Jackson 2002] is closest to the system described here. Alloy is a simplified and improved descendant of the Z notation that allows definition of a model consisting of some state and operations on those states states and one can verify expected properties using a SAT solver. In some ways, Alloy is considerably more general and sophisticated than the system described here: it provides a simple, mathematically clean language for specification. In other ways the ASL language is more powerful because it provides specialized concepts for the task of defining processor semantics such as bitvectors and dependent types, concepts like instructions and bitfields of registers, etc. and it is imperative: these features allow the creation of detailed specifications of large, complex architectures and proofs about specifications with very large state spaces.

The Formal Tropos language [Fuxman et al. 2001] is specifically designed to allow the form of loose specification that characterizes the early stages of requirements engineering: it focusses on entities and the relationships between them and allows the addition both of hard goals specified using first order linear temporal logic and soft goals that might be subjective (e.g., a company may have a goal of attracting new customers). The language provides a number of high level abstractions of events such as notions of object creation, fulfillment of a goal, etc. that could be expressed in temporal logic but whose inclusion improved readability; specifications can be animated to check understanding; and model checking can be used to formally verify that properties of the specification are true or can be satisfied. Support for temporal logic is the most obvious difference from our system but we are not sure that model checkers would be able to cope with the large state spaces of our specification because, even with explicit invariants, some of the properties we wish to check are barely provable by an SMT solver. However, the rich set of abstractions for

[3]Strictly speaking, ARM says that the natural language rules and the formal ASL specification in ARM's specifications have equal weight: they are both part of the specification and both must be satisfied by an implementation. However, in this work we have treated the natural language rules as a loose specification of the properties that the more precise ASL specification is required to satisfy — much as a requirements specification can be seen as a loose specification for more refined specifications developed as design and implementation proceeds.

describing events appears useful and we are considering whether they can be adopted without requiring the use of model-checking.

More broadly, it is common practice when creating specifications in a theorem prover to prove that a new definition satisfies sets of properties such as commutativity, associativity, etc. [Pierce et al. 2016]. Like our properties about specifications, such properties may not completely characterize the functions being developed but they give increased confidence that the functions are correct and they are often useful when using those functions. The difference is that our specifications are somewhat larger and that our properties make use of the ability of our property language to restrict the set of execution paths taken by the function being checked.

8 LIMITATIONS AND FUTURE WORK

This is part of a long program of creating a complete and precise specification of the ARM architecture. The most significant limitation is that it has not yet been integrated with parallel work on concurrent memory semantics [Alglave et al. 2014; Flur et al. 2016; Sarkar et al. 2011]. This limitation shows up most clearly in situations where TopLevel performs multiple memory accesses in a single transition (either because of executing ARM's "load/store multiple" instructions or because of pushing/popping context on/off the stack during exceptions). Our reasoning treats the entire execution/exception as a single atomic transition while an external observer would see multiple independent memory accesses.

We see this work as a step towards creating a set of properties that can be used to verify low-level system code such as interrupt handlers, memory protection, etc. We hope that this could be used to plug the gaps in formal proofs of software such as the seL4 OS kernel [Klein et al. 2009, Section 4.4] that rely on manual inspection and thorough testing of a few pieces of low-level code instead of providing a formal proof.

The current performance of our tool is adequate for daily or weekly checking of the specification but it is currently too slow to use as a check on every commit to the specification repository. We plan to implement a variation on DAG inlining [Lal and Qadeer 2015] to improve scalability.

We are considering how we could formalize the statement in Section 4.3 that says "EPSR.IT to be become UNKNOWN." This property cannot be checked in our current implementation because it is a 2-safety property: detecting a violation would require comparing the result of traces from two program traces [Clarkson and Schneider 2010].

9 CONCLUSION

Formal verification of programs is becoming more and more practical but, if the verification is to be meaningful, it must be based on correct architecture specifications for the hardware that the programs run on. That is, the specifications are a critical part of the Trusted Computing Base. Unfortunately, the size and complexity of architecture specifications is such that it seems inevitable that specifications will contain bugs and our previous work confirms this supposition [Reid 2016].

While it is common to debug specifications by *testing* the specification, this paper proposes a different approach: we define a set of formal properties that should hold for the specification and we *formally verify* that the architecture specification satisfies these properties. We think of the relationship between the properties and the specification as being like the relationship between a nation's constitution and a nation's laws: the constitution is concise enough that everyone can read them while the laws are too large for effective review; the constitution can be used to test whether existing or proposed laws are compatible with high level goals; and the constitution is stable and changes very, very slowly.

We have extended ARM's Architecture Specification Language with a property language that is able to concisely express many of the properties currently written in natural language. Our

extension's power comes from a novel coverage operator that lets us express cross-cutting, end-to-end properties. We are able to check that these properties hold by converting both the specification and the properties to verification conditions that can be checked in a push-button manner using an SMT solver. We have used this system to check ARM's v8-M specification. Despite the fact that the ARM v8-M specification had previously been extensively tested and reviewed, we found twelve bugs in it, that have all been fixed by ARM.

To our knowledge, no realistic architecture specification has been subjected to this degree of formal verification before.

ACKNOWLEDGMENTS

We wish to thank Christoph Wintersteiger for his suggestions for using Z3 more effectively. We are grateful to John Regehr, Peter Sewell and Nathan Chong and to the anonymous referees for their comments and suggestions about the content and presentation of this paper.

REFERENCES

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. DOI: http://dx.doi.org/10.1145/2627752

J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '83)*. ACM, New York, NY, USA, 177–189. DOI: http://dx.doi.org/10.1145/567067.567085

ARM Ltd. 2013. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (DDI0487)*. ARM Ltd. https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

ARM Ltd. 2016. *ARMv8-M Architecture Reference Manual (DDI0553)*. ARM Ltd. https://developer.arm.com/docs/ddi0553/latest/armv8-m-architecture-reference-manual

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org. (2016).

D. A. Burke and K. Johannisson. 2005. Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach, J. Busquets P. Blace, E. Stabler and R. Moot (Eds.). *Logical Aspects of Computational Linguistics (LACL 2005)* 3402 (2005), 51–66. DOI: http://dx.doi.org/10.1007/11422532_4

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. *A Tool for Checking ANSI-C Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176. DOI: http://dx.doi.org/10.1007/978-3-540-24730-2_15

Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210. http://dl.acm.org/citation.cfm?id=1891823.1891830

CompCert. 2016. Release Notes for CompCert 2.7 (Bugfixes). (29 June 2016). http://compcert.inria.fr/release/Changelog

Mads Dam, Roberto Guanciale, and Hamed Nemati. 2013. Machine Code Verification of a Tiny ARM Hypervisor. In *Proc. Workshop on Trustworthy Embedded Devices (TrustED '13)*. ACM, 3–12. DOI: http://dx.doi.org/10.1145/2517300.2517302

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

George Dunlap. 2012. The Intel SYSRET Privilege Escalation (Xen Project Blog). (2012). https://blog.xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proc. Principles of Programming Languages, POPL 2016*. 608–621. DOI: http://dx.doi.org/10.1145/2837614.2837615

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 328–343. DOI: http://dx.doi.org/10.1145/3064176.3064183

Anthony C. J. Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Proc. Interactive Theorem Proving ITP 2010 (LNCS)*, Vol. 6172. Springer, 243–258. DOI: http://dx.doi.org/10.1007/978-3-642-14052-5_18

Ariel Fuxman, Marco Pistore, John Mylopoulos, and Paolo Traverso. 2001. Model checking early requirements specifications in Tropos. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. IEEE, 174–181. DOI: http://dx.doi.org/10.1109/ISRE.2001.948557

Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and formal verification of x86 machine-code programs that make system calls. In *Formal Methods in Computer-Aided Design, FMCAD*. 91–98. DOI: http://dx.doi.org/10.1109/FMCAD.2014.6987600

IEEE. 2013. IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. *IEEE Std. 1800-2012* (2013). DOI: http://dx.doi.org/10.1109/IEEESTD.2013.6469140

Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (April 2002), 256–290. DOI: http://dx.doi.org/10.1145/505145.505149

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer, Berlin, Heidelberg, 220–242. DOI: http://dx.doi.org/10.1007/BFb0053381

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. DOI: http://dx.doi.org/10.1145/1629575.1629596

Akash Lal and Shaz Qadeer. 2015. DAG Inlining: A Decision Procedure for Reachability-modulo-theories in Hierarchical Programs. In *Programming Language Design and Implementation (PLDI)*, Vol. 50. ACM, New York, NY, USA, 280–290. DOI: http://dx.doi.org/10.1145/2813885.2737987

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. DOI: http://dx.doi.org/10.1145/1538788.1538814

Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. 2009. Easy Approach to Requirements Syntax (EARS). In *17th IEEE International Requirements Engineering Conference (RE'09)*. IEEE. DOI: http://dx.doi.org/10.1109/RE.2009.9

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2016. *Software Foundations*. Electronic textbook. http://www.cis.upenn.edu/~bcpierce/sf Version 4.0.

Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *Proceedings of Formal Methods in Computer-Aided Design, (FMCAD 2016), Mountain View, CA, USA*. 161–168. http://doi.acm.org/10.1109/FMCAD.2016.7886675

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of ARM Processors with ISA-Formal, In Proceedings of the 2016 International Conference on Computer Aided Verification (CAV'16), S. Chaudhuri and A. Farzan (Eds.). *CAV 2016, Part II, Lecture Notes in Computer Science* 9780 (July 2016), 42–58. DOI: http://dx.doi.org/10.1007/978-3-319-41540-6_3

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186. DOI: http://dx.doi.org/10.1145/1993498.1993520

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 190–204. DOI: http://dx.doi.org/10.1145/3009837.3009838

Christoph M. Wintersteiger. 2017. private communication. (2017).