Balancing Automation and Control for Formal Verification of Microprocessors

Shilpi Goel, Anna Slobodova(✉), Rob Sumners, and Sol Swords

Centaur Technology, Inc., Austin, TX, USA
{shilpi,anna,rsumners,sswords}@centtech.com

Abstract. Formal methods are becoming an indispensable part of the design process in software and hardware industry. It takes robust tools and proofs to make formal validation of large scale projects reliable. In this paper, we will describe the current status of formal verification at Centaur Technology. We will explain our challenges and our methodology—how various proofs and verification artifacts are interconnected and how we keep them consistent over the duration of a project. We also describe our main engine—a powerful symbolic simulator with rewriting capabilities that is integrated in a theorem prover and proven correct.

Keywords: Hardware verification · Microprocessor verification · Microcode verification · Formal methods · ACL2 · Symbolic simulation · Decision procedures

1 Introduction

The discussion of Formal Verification (FV) of software and hardware three decades ago was mostly about case studies or proofs of concept that required a lot of manual effort by researchers. Since then, FV has taken a transformational journey that has resulted in highly automated tools—equivalence checkers, model checkers, SMT solvers, and theorem provers. Large scale formal verification projects were first reported by hardware companies around ten years ago, e.g. Intel [28], IBM [36], ARM [34], and Centaur Technology [18,37]. Success stories of FV at software development companies followed. To name just a few, see Peter O'Hearn's keynote at *PLDI 2020* conference about incorrectness logic and static analysis his group applies at Facebook [30], David Dill's keynote at *CAV 2020* about the Libra project at Facebook [19] and their use of the Move Prover [44], or the invited talk by Byron Cook at *CAV 2018* about the application of formal methods at Amazon Web Services [16]. Formal methods are becoming a reliable and indispensable part of the design process in the commercial software and hardware industries. This newly elevated position of formal verification brings new responsibilities for those that develop tools and methods and those who build proofs. FV teams face various challenges:

© The Author(s) 2021
A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 26–45, 2021.
https://doi.org/10.1007/978-3-030-81685-8_2

- Tools and libraries used by FV teams are expected to be reliable and maintainable.
- FV teams get involved much sooner in a project cycle, often starting with an incomplete design, and they are expected to give feedback quickly.
- Designs under FV scrutiny are being continuously changed by several designers at a time.
- Specifications change during the process as designers get feedback from backend tools, or due to the changes in the target market.
- The scope and depth of proofs change as development continues.
- An FV team might be working on several proliferations of a project with overlapping schedules.

These challenges can only be solved by building robust expandable proofs. In this paper, we will describe the approach taken by our FV team at Centaur Technology. Centaur is a relatively small company, of about one hundred employees, that designs x86 compatible microprocessors, focusing on the low cost, low power market. It might surprise many that our formal verification tools are based on a theorem prover. This is only possible because the theorem prover we use, ACL2 [8], has been designed with industrial applications in mind [24]. ACL2 has been successfully used not only at our company but also at many others: e.g. ARM, AMD [35], IBM [36], Rockwell-Collins [22], and Oracle [32]. All our proofs are done within the ACL2 system. ACL2 is used to write specifications, models, tools, and tests, as well as to generate documentation. Two features of ACL2 that are crucial to our work are fast execution and extensibility. Our x86 model [20] is not only one of most complete of its kind, but is capable of executing application programs at a speed of around 3 million instructions per second.

We will start with a brief description of the ACL2 system and the features that make it a good choice for a verification framework (Sect. 2). The reflective features of ACL2 allow us to build verified tools within the system. One such tool is FGL [39], our symbolic simulator equipped with rewriting capabilities. FGL is completely integrated into ACL2 as a verified clause processor. It provides a desirable balance between automation and user guidance. We will describe its mechanism in detail in Sect. 4. We also explain its usability as a highly programmable solver that is capable of proving complex conjectures about Register-Transfer Level (RTL) design and microcode in Sect. 3. FGL and its use within our framework are primary contributions of the work presented in this paper. The challenges enumerated above are illustrated with the process of verification for a single x86 instruction. We explain the complex interconnection of the various parts of the proofs, and describe how they are built and maintained.

2 Our FV Tools

All formal verification at Centaur is done within the framework of ACL2 [8]. ACL2 is an untyped language (a subset of Common Lisp) and a theorem prover that supports first-order logic as expressed in this language. ACL2 also has

some limited support for higher-order style definitions [29]. ACL2 is an open source software project that has an active community contributing to an extensive library of proofs and utilities. Centaur has contributed to many libraries that support hardware verification, including support for translating Verilog and System Verilog to ACL2 expressions [7,10] and libraries that support bit operations. ACL2 provides an interface through which it can be connected to trusted tools such as SAT solvers. There is also an integration of Z3 in ACL2 [5] and an interface to the ABC model checker [1,14].

Besides interfaces to trusted tools, ACL2 has a mechanism for extending its reasoning by admitting verified clause-processors [2]. We use this feature in several ways, notably for SVL [43], a routine that automates verification of multipliers, and for FGL, the core tool that provides automation for our microoperation execution and microcode proofs.

FGL, briefly, is a term rewriter geared toward transforming expressions acting on fixed-sized data into Boolean formulas. For example, a specification for an x86 instruction may be written in high-level ACL2. Processing a call of this specification function on variable arguments in FGL yields a result that expresses each of the bits of the writeback data, flags, etc., as a Boolean formula (represented in an and-inverter graph) whose inputs are the symbolic bits of the input variables. Similarly, FGL processing of the ACL2 model of the microcoded implementation for that instruction yields Boolean formula representations of the implementation's outputs. Equivalence checking these two sets of Boolean formulas is then sufficient to show that the implementation result matches the specification. We describe the FGL system in more detail in Sect. 4, showing how it transforms terms into hybrid term/Boolean-function objects and how its behavior may be programmed with rewrite rules.

3 Challenges of Verifying a Single x86 instruction

An intuitive notion of the functional correctness of a microprocessor is that any sequence of bytes decoded as instructions either executes correctly or leads to an exception if byte sequence is illegal. For the x86 instruction set, parsing and decoding a sequence of bytes is a complex process due to the many instruction formats with varying lengths and field types. The Intel 64 and IA-32 Instruction Set Architecture (ISA) is defined by the Software Developer's Manuals [27], which have thousands of pages describing the expected impact of every instruction on the state of the machine. It is a living and growing specification, with new instructions and variants added constantly. The architectural specification does not dictate how the ISA is supposed to be implemented. Various implementation-specific choices, collectively called the *microarchitecture*, include:

- how memory is organized
- how an instruction is decoded into a sequence of microoperations
- the set of microoperations implemented in hardware
- the throughput and latency of microoperations and instructions

and various others features of the microprocessor. In our previous work [21], we described what it means for an x86 instruction to be decoded and executed correctly and how our proofs capture this property. For illustrative purposes, we use the same example that was described in that work. Table 1 describes the x86 double-precision shift right instruction `SHRD` and Table 2 shows the microoperations that implement it[1]. In this paper, we will recall the individual steps of the verification with a different purpose—to discuss the challenges in each step and how we deal with them. In particular, we will focus on increasing the automation and reducing the time required of engineers to catch and debug problems while maintaining the proofs.

In the process of verification, we refer to two sets of formal specifications: the architectural specification of x86 [20] and a microarchitectural specification, which is a proprietary IP of Centaur and unique to each project. We refer to the former as the *x86 model* and the latter as the *microcode model*. Both of these models are written in ACL2 following an interpreter-style operational semantics approach. The x86 model includes the specification of x86 instructions that operate on the ISA state, and analogously, the microcode model includes the specifications of microoperations that operate on the microarchitectural state. Thanks to the high execution speed of the x86 model, it can be validated by running extensive code. The microcode model is directly compared to the RTL implementation. In addition, for data-intensive operations like floating-point arithmetic, we have the ability to run our models against existing x86 hardware from Intel and AMD. Again, the efficient execution of ACL2 code is crucial for the validation of these models.

Our verification is done on the Register-Transfer Level (RTL) of microprocessor design. We have two goals: to confirm that the RTL behaves as specified by our microarchitectural specification and to show that it implements instructions correctly with respect to our architectural specification.

Table 1. `SHRD--Double Precision Shift Right`: irrelevant fields elided

Opcode	Instruction	Description
`REX.W + 0F AC /r ib`	`SHRD r/m64, r64, imm8`	Shift `r/m64` to right `imm8` places while shifting bits from `r64` in from left

3.1 Front-End and Microcode Verification

The front-end of a microprocessor fetches, decodes, and then translates a sequence of bytes into a sequence of microoperations. For a modern x86 processor, this is one of the more complicated parts of the design. Writing and

[1] Note that this is not the actual implementation of `SHRD` in our current design.

Table 2. `SHRD RCX, RDX, imm8`: a concrete run

Initial values	`RDX := 0x1122_3344_5566_7788`
	`RCX := 0x0123_4567_89AB_CDEF`
	`imm8 := 16`

Expected values	`RDX := 0x1122_3344_5566_7788` `RCX := 0x7788_0123_4567_89AB`
UOPs from front-end	**Concrete Run & Description**
`MOVSX G2, RCX` `(SSZ: 64; DSZ: 64)`	`G2 ← 0x0123_4567_89AB_CDEF` *Move* `RCX` *to internal register* `G2`
`MOVZX G3, <imm8>` `(SSZ: 8; DSZ: 64)`	`G3 ← 16` *Move immediate to internal register* `G3`
UOPs in ROM	**Concrete Run & Description**
`AND G3, G3, 63` `(SSZ: 8; DSZ: 64)`	`G3 ← 16` *Mask immediate operand*
`MOV G10, -1` `(SSZ: 64; DSZ: 64)`	`G10 ← 0xFFFF_FFFF_FFFF_FFFF` *Move -1 to internal register* `G10`
`JE G3, 0, ent_nop` `(SSZ: 16; DSZ: 16)`	`No jump taken` *Jump to routine* `ent_nop` *if* `G3` *== 0*
`SUB G5, 0, G3` `(SSZ: 32; DSZ: 32)`	`G5 ← 0xFFFF_FFF0; ZF ← 0` *Store* `-G3` *in internal register* `G5`; *clear the zero flag because result is non-zero*
`SHR`$_{<!ZF>}$ `G10, G10, G5` `(SSZ: 64; DSZ: 64)`	`G10 ← 0xFFFF` *Shift* `G10` *right by* `(G5 & 63)` *if* `ZF` *== 0*
`AND`$_{<ZF>}$ `G10, G10, 0` `(SSZ: 64; DSZ: 64)`	`G10 ← 0xFFFF` *Set* `G10` *to 0 if* `ZF` *== 1*
`AND G6, RDX, G10` `(SSZ: 64; DSZ: 64)`	`G6 ← 0x7788` *Store* `(RDX & G10)` *in internal register* `G6`
`SHR G7, G2, G3` `(SSZ: 64; DSZ: 64)`	`G7 ← 0x0000_0123_4567_89AB` *Store* `(G2 » G3)` *in* `G7`
`SHL G2, G7, G3` `(SSZ: 64; DSZ: 64)`	`G2 ← 0x0123_4567_89AB_0000` *Store* `(G7 « G3)` *in* `G2`
`OR G2, G2, G6` `(SSZ: 64; DSZ: 64)`	`G2 ← 0x0123_4567_89AB_7788` *Store* `(G2 | G6)` *in* `G2`
`ROR G7, G2, G3` `(SSZ: 64; DSZ: 64)`	`G7 ← 0x7788_0123_4567_89AB` *Rotate* `G2` *right by* `G3` *and store result in* `G7`
`OR RCX, G7, G7` `(SSZ: 64; DSZ: 64)`	`RCX ← 0x7788_0123_4567_89AB` *Store the result of* `G7 | G7` *in* `RCX`

maintaining a formal specification for it would be impractical. The readability and complexity of such a specification would be similar to that of the implementation itself. How, then, do we go about its verification? We have one methodology to verify the decoding of byte sequences into legal/illegal instructions (with appropriate exceptions), and another one to show that legal instructions are implemented correctly via microoperations.

Listing 1.1. `SHRD` entry in `inst.lst`

```
(xINST "SHRD"
       (OP :OP #xFAC)
       (ARG :OP1 '(:MODR/M.R/M :GPR :MEM)
            :OP2 '(:MODR/M.REG :GPR)
            :OP3 '(:IMM8))
       '(X86-SHLD/SHRD)
       '((:UD (UD-LOCK-USED))))
```

For illegal instructions, we make sure that all sequences of bytes that do not decode into a sequence of legal instructions are recognized as illegal and we verify that an appropriate exception is signaled. This is done by simulating the front-end on a symbolic sequence of bytes and proving that any input that does not map to a legal opcode (as defined by the decode specification in our x86 model) produces an exception. The decode specification in the x86 model relies heavily on `inst.lst`—a data structure defined by us that captures all the information needed to decode every x86 instruction. The initial version of `inst.lst` was mechanically extracted from the Intel manuals (Chaps. 3–5, Vol. 2) [27] by parsing the tables in the description pages of each instruction and transforming the contents into an ACL2-readable format. For instance, for the implementation in Table 2, the relevant entry in the Intel manuals is in Table 1 and that in `inst.lst` is in Listing 1.1. Since then, `inst.lst` has been inspected, enhanced, and validated against internal and external x86 decoders.

Next we focus on our process for verifying legal instructions. For each instruction, our goal is to prove that for any starting machine state and for any byte sequence representing a legal invocation of that instruction in that state, the front-end produces a sequence of microoperations which, when run on our microcode model, produce the same results as the instruction run on our x86 model[2]. To prove this, we simulate the front-end to generate the corresponding sequence of microoperations. Using FGL, we then prove that the sequence implements the instruction as defined by our x86 specification. FGL symbolically processes the sequence of microoperations as executed on our microcode model, resulting in a symbolic machine state where the bits of the written registers are represented as Boolean formulas in terms of the values read from the initial state. It likewise processes the instruction specification, reducing it to Boolean formulas as well. We can then show by Boolean equivalence checking that the front-end-generated sequence of microoperations has the same effect on the state as the x86 instruction specification. We discuss the process of symbolic simulation of microcode by FGL in Sect. 4. This FGL proof confirms that the front-end's operation is correct for this particular instruction.

This correctness has two caveats. First, it assumes that the individual microoperations are correctly implemented, i.e., in accordance with their specifications in our microcode model. Second, in the case of out-of-order processors, if the microoperations are executed in a different order, that sequence needs to be compared to the sequence generated by the front-end. Currently, we can

[2] Note that the microcode model is a proprietary formal model of the microarchitecture implemented by the design. Its validation is discussed later.

ensure only the former—for most microoperations, we have proved that their implementations in the processor's execution units matches their behavior in our microcode model; we discuss this further in Sect. 3.2. However, the latter—the correctness of reordering of the microoperations—is work planned for the future.

There is another part to the verification story. The front-end generates only sequences of microoperations of a limited length. Some instructions are complex and require much longer sequences (e.g. instructions performing transcendental or cryptographic functions). For these instructions, the sequence of microoperations generated by the front-end is just the beginning of the microcode program. The rest is stored in a ROM and the front-end generates the entry-point of this code. That means our verification has to account for those microcode routines.

ROM instructions are more complex than microoperations and they may also be compressed in order to save valuable ROM space. As in the front-end, the specification of this compression and decoding of ROM instructions into sequences of microoperations is complex and also changes during the design process. Even if we could define it formally, the maintenance of such a specification would be very time consuming. Instead, we do the same trick as with the front-end—we symbolically simulate the part of the design that fetches ROM instructions and translates them into a sequence of microoperations. The rest is done similarly as for the sequence of operations generated by the front-end. These proofs implicitly verify the correctness of fetching from ROM and ROM instruction translation. We call this *implicit verification* because we do not have an explicit specification of the translation and fetching. However, we do have formal specifications of the instructions implemented by the microcode. Therefore, the proof of correctness of the instructions implies the correctness of the underlying design, including ROM fetching and translation. In other words, we can verify some parts of the design as black-boxes, without knowing exactly how they work, by reasoning about the overall observable effect on the machine state. The main advantage of this type of verification of both the front-end and microcode translator is that the maintenance of the proofs does not require either deep understanding of the design or writing and maintaining cumbersome specifications.

The microcode sequences generated from x86 instructions that we encountered so far were in the style of straight-line code. We do not expect this to be the case for all of them. In the past, we worked on some microcode stored in ROM that served other purposes [18]. This code had loops and jumps between loops and we were able to do invariant-style proofs. Our main problem at that time was that the proofs were not robust enough and very hard to maintain. Now we are in a much better position, having FGL and a methodology that keeps the microcode model in sync with the design. Hence, we are optimistic about our ability to bring the verification of most, if not all, legal x86 instructions to completion.

Finally, we note that in our previous work [21], we used GL—the predecessor of FGL—as our core verification tool. The benefits of switching to FGL have been considerable. GL had limited support for term rewriting, as a result of which symbolic simulation of the microcode model was difficult and debugging failed proof attempts even more so. As such, instead of programming GL to deal with symbolic machine states, we usually used ACL2's rewriter to "open up" the microcode model and played to GL's strengths by using it for the final equivalence proof that often required non-trivial arithmetic reasoning. In other words, we obtained ACL2 formulas corresponding to the written registers in terms of the values in the initial microcode state, and then used GL to prove that those formulas were equivalent to our specification functions. FGL easily allows us to do these tasks (symbolic simulation and equivalence checking) along with others within a common environment and thereby reduces overhead in our methodology.

3.2 Verification of Execution Units

Everything that was said in Sect. 3.1 relies on the assumption that our microcode model is correct. Parts of that model—front-end decoding and ROM instruction fetch and translate—are implicitly verified. The other part, definitions of micro-operations that form the base of the model, were explicitly defined in ACL2 and need to be validated. A large portion of our work lies in the proofs that confirm that RTL executes the microoperations in compliance with those specifications. In order to achieve that, we build a formal model of the respective RTL module [7], unroll it with respect to the latencies of the microoperations to be verified [6] and check conformance with the specification using FGL.

These microoperations are executed in various units, the number, timing, and organization of which differs based on the specific microarchitecture. We might have separate floating-point add and floating-point multiply units, or one unit that executes both. There might be a unit that implements string operations, another that implements integer operations, and yet another one devoted to `SIMD` operations, etc. The scope of proofs that confirm correctness of execution of microoperations is dictated by the capacity of the tools we use. During the first years of FV at Centaur, we limited the proof for each microoperation to the specific unit where it was executed [25,26,37]. Since then, improvements to our RTL modeling and symbolic simulation (i.e., FGL) allow us to do the proofs in the scope of the module containing all those units (we refer to that module as the execution module or `EXE`) [21]. Migration to a higher scope has a huge advantage for the stability of the proofs. First, proofs are robust with respect to the changes of the interfaces of submodules in `EXE`. For instance, when an interface of a floating-point sub-unit changes to accommodate extra control signals that simplify its logic, very likely the change is transparent to the input-output behavior of `EXE` and will not effect our proofs. Second, if timing of an internal unit changes, but overall timing of the `EXE` module does not, that is transparent to the proofs.

Having all microoperation proofs in the same scope has another advantage—we can build just one formal model of the RTL, do one unrolling to the maximum latency, and store it as a constant that can be shared and loaded by individual proofs. A review of the assumptions about the interface and the maintenance of the assumptions is also simplified when all the proofs are done with respect to one module.

3.3 Regressions

Regressions have become an indispensable part of the continuous integration. There are several reasons why we need to re-run our proofs regularly. Since we start to build our proofs early in the design process, design changes occur regularly and can introduce bugs that we need to catch. But proofs can be broken not only due to changes in the design but also because of changes in the specifications, tools, and libraries. While the ISA specification is relatively stable, the microarchitecture specification might change during the project as a result of feedback from back-end tools or better ideas from the designers. Proofs might also change as the design becomes more mature and we add more thorough checks. While the core ACL2 theorem prover is very stable, ACL2 libraries are growing and may be modified by developers outside our team. All of these verification artifacts are tightly interconnected and regressions ensure that we keep them consistent.

When a proof of the correctness of a microoperation fails, there are several possible reasons:

- There is a bug in the RTL design.
- There was a change in the design (interface or timing) that our proofs need to take into account.
- The specification of the microoperation changed; e.g. some flags indicate a new intended use, or a portion of the result became "don't care".

We need to investigate the reason for failure and either report a bug to designers, adjust the proofs, or change the specification of the microoperation.

When we change the specification of a microoperation, the new definition will then be used by our microcode proofs. If those fail, it may indicate that the change affected some instruction implementations in an undesirable way. In other cases, the failure might be a result of missing rewrite rules. Microcode proofs might also fail due to the changes to front-end design or fetch and translate from ROM that introduced a new bug.

Regressions can be scheduled for a specific frequency (daily, weekly, etc.), run manually, or triggered by changes in the design, specification, or tool suite. We use open-source tools like `git` and Jenkins, and ACL2-specific scripts that compute dependencies on ACL2 files. Regressions also automatically generate a documentation manual from our ACL2 proof scripts [17]. This documentation includes information about which proofs failed and which succeeded and as the result of it, which microoperations and instructions are covered by the

successful proofs. This keeps the documentation in sync with the design as well as the proofs. We tag individual documentation topics to indicate their intended audience; e.g., the *General Audience* tag is used when an overview of a verification effort is presented, and the *FV Audience* tag is used when describing proof strategies and verification tools.

4 FGL

Since FGL is the core proof engine used in our microcode and execution unit proofs, we will describe here how it works and how it may be programmed.

FGL [4,39] is part of the ACL2 libraries and publicly available [9]. It is a significant rewrite and extension of GL ("G in the Logic") [38,41,42], which was itself a rewrite and extension of the G System of Boyer and Hunt [13]. The idea behind all of these is to recursively transform ACL2 terms into symbolic objects that represent the values of these terms and that consist mostly of structures containing Boolean function objects. When successful, the result of transforming the body of a conjecture is a single Boolean function, which may be checked for validity. The G System supported Boolean functions represented as binary decision diagrams [15], and operated on symbolic input objects using symbolic counterpart functions derived mechanically from function definitions. GL used an interpreter to capture function behavior rather than translating definitions, and added support for an and-inverter graph (AIG) representation for Boolean functions along with links to external SAT solvers for resolving Boolean function validity. Later changes in GL added preliminary support for rewrite rules and termlike symbolic objects so as to allow for some abstraction.

FGL continues the trend toward user-definable rules displacing built-in behavior. It is a rewriter at its core, so user-defined rewrite rules are the basis of its reasoning system, rather than an add-on. Nevertheless, it comes with an extensive library of rules that replicates the automation provided by GL. Rewrite rules supported by FGL offer powerful capabilities such as programmable binding of free variables and visibility into the syntax of the rewriting targets [39]. FGL also replaces built-in primitive function symbolic counterparts with *meta rules* similar in spirit to ACL2's [23], which similarly allow directly programmable manipulation of the syntax of objects but may also be added by users. FGL adds support for incremental SAT, allowing multiple SAT checks of related formulas to share learned clauses and heuristic information. It also allows global simplification of the entire AIG using combinational circuit simplification methods. Both of these features may be invoked from within rewrite rules; e.g., if the author of a rewrite rule judges that a hypothesis of the rule is unlikely to be solved by rewriting alone, they may specify that incremental SAT should be used to prove it.

Many other projects have also aimed to allow interactive theorem provers to call on automatic decision procedures; too many such efforts exist to list them all. In higher-order logic proof assistants, several tools collectively called *hammers* translate queries into the language of an automated theorem prover

Listing 1.2. Semantics of a machine instruction

```
(defun run-inst (inst st)
  (let* ((instname (first inst))
         (args (rest inst))
         (x (first args))
         (y (second args))
         (ans (case instname
                (const     y)
                (copy      (get-st-reg y st))
                (add       (+ (get-st-reg x st)
                              (get-st-reg y st)))
                (and       (bitwise-and (get-st-reg x st)
                                        (get-st-reg y st)))
                (rshift    (right-shift (get-st-reg y st)
                                        (get-st-reg x st))))))
    (set-st-reg x ans st)))
```

Listing 1.3. Semantics of a straight-line code block

```
(defun run-prog (insts st)
  (if (atom insts)
      st
    (let ((st (run-inst (first insts) st)))
      (run-prog (rest insts) st))))
```

and then translate the emitted proof back into a form acceptable by the original prover [12]. Several decision procedure integrations have also been carried out in ACL2. Reeber and Hunt [33] identified a decidable subclass of ACL2 list formulas and contributed a decision procedure that transforms such a formula into a SAT problem. Peng and Greenstreet [31] process a subclass of ACL2 formulas including integer and rational arithmetic, uninterpreted functions, and algebraic data structures, converting such problems to SMT queries. FGL differs by focusing on the efficient integration of user-extendible term rewriting and Boolean simplification and decision procedures.

4.1 Example

We describe how FGL works at a high level by running through an example, the code of which is publicly available [40]. We define a simple machine model (Listings 1.2, 1.3) that has 16 32-bit registers and a few instructions defined, and use those instructions to implement (in straight-line code) an optimized routine to count the number of bits set in a 32-bit input (Listing 1.4), similar to implementations in *Bit Twiddling Hacks* [11]. We also define a straightforward ACL2 specification `count-bits` for the bit count operation (Listing 1.5). We prove that for any initial state, if we run this program on the machine, then the resulting state has its register 0 value equal to the `count-bits` of the value that was in register 0 before running the program (Listing 1.6).

The invocation of `def-fgl-thm` in Listing 1.6 causes the FGL rewriter to be applied to the conjecture. It begins by descending into the term and applying rewrite rules to subterms from the inside out. In many cases, these rules are just the definitional formulas of the functions we have introduced; for example, the definitions of `run-prog`, `run-inst`, and `count-bits` are used as rewrite rules, so that calls of these functions are replaced by their bodies. Rewriting the term

Listing 1.4. BITCOUNT program listing

```
(defconst *bitcount*
  '((copy 10 0) ;; copy the operand to regs 10 and 11
    (copy 11 0)
    (const 5 #x55555555) ;; set reg 5 to the mask
    (and 10 5)  ;; bitand the operand with the mask
    (const 0 1)   ;; set reg 0 to 1
    (rshift 11 0) ;; right shift the operand by 1
    (and 11 5)  ;; mask the shifted operand
    ...
    (const 0 #x003f)
    (and 10 0)    ;; mask the relevant bits of the result
    (copy 0 10))) ;; move the result to reg 0.
```

Listing 1.5. `count-bits` specification function

```
(defun count-bits (x)
  (if (or (not (integerp x)) (<= x 0))
      0
    (+ (nth-bit 0 x)
       (count-bits (right-shift 1 x)))))
```

while opening such definitions effectively conducts a symbolic simulation of the program and its specification. For some functions, it is preferable to avoid opening the definitions and instead use rules that rely on particular properties to simplify combinations of calls; for example, Listing 1.7 shows a rule that simplifies a read of a write of the machine state's register file.[3]

Rather than producing a new term as the result of rewriting each subterm, the FGL rewriter produces hybrid structures we call *symbolic objects* that may (like terms) contain function calls, variable references, and constants, but (unlike terms) also may contain *symbolic Booleans*, represented by a reference into an AIG defining a Boolean function, and *symbolic integers*, represented by a list of references into the AIG giving the two's-complement bits. Table 3 lists the variants of symbolic objects.

In order to prove this conjecture, we aim for the result of rewriting the conjecture to be a symbolic Boolean, which can then be proved valid by encoding its negation as a SAT problem. We therefore want to compute a Boolean formula equivalent to the `equal` comparison of the specification and implementation results. Working backwards from this goal, we can obtain this if we can represent the specification and implementation results as symbolic integers; the `equal`

Listing 1.6. Correctness theorem for BITCOUNT

```
(def-fgl-thm bitcount-implements-count-bits
  (let* ((input (get-st-reg 0 st))
         (final-st (run-prog *bitcount* st))
         (result (get-st-reg 0 final-st)))
    (equal result (count-bits input))))
```

[3] Since ACL2 is an untyped language, functions have well-defined behavior even on ill-typed inputs. The uses of zero-extend in this rule reflect the choice of the definitions to coerce integers that don't fit in the allotted space into well-typed values by zero-extending them.

Listing 1.7. Read-over-write rule for `get-st-reg`

```
(def-fgl-rewrite get-st-reg-of-set-st-reg
  (equal (get-st-reg i (set-st-reg j v st))
         (if (equal (zero-extend 4 i) (zero-extend 4 j))
             (zero-extend 32 v)
           (get-st-reg i st))))
```

Table 3. Symbolic object variants

- (`g-boolean` *lit*) represents a Boolean, `t` or `nil`, as an AIG literal, *lit*
- (`g-integer` *lit0 lit1* ...) represents an integer as a list of AIG literals giving the two's-complement bits, least-significant first
- (`g-concrete` *obj*) represents the constant *obj* itself
- (`g-apply` *fn args*) represents a function application, where *fn* is a function symbol and *args* is a list of symbolic objects
- (`g-var` *name*) represents a variable named *name*
- (`g-ite` *test then else*) represents an if-then-else, where the three arguments are symbolic objects
- (`g-cons` *car cdr*) represents a cons pair, where the two arguments are symbolic objects
- (`g-map` *tag alist*) represents a table of key/value pairs with constant keys and symbolic values, supporting fast lookups (see ACL2 documentation on fast alists [3])

comparison of these is the conjunction of the Boolean equivalences between all the corresponding bits. Working further backwards, we'll find that we can similarly compute these values given the bits of the intermediate integer values from which they are computed, etc., back to the original values that are components of the free variables of the conjecture. That is, generally speaking, we wish to represent every intermediate integer value as a symbolic integer. In the next two sections we will describe how to extract Boolean variables from the initial variables of the conjecture (Sect. 4.2) and how to build up Boolean formulas to represent the bits of intermediate values (Sect. 4.3).

4.2 Extracting Boolean Variables

When rewriting a term in a Boolean context such as the test of an `if` expression, FGL will coerce the rewritten result to a symbolic Boolean object. The symbolic Boolean values of symbolic object types other than function calls and variables are easy to determine; for example, integers are non-`nil` and therefore considered true in ACL2. For function call and variable results, this coercion is accomplished by assigning a Boolean variable to the object, either a fresh one—a new primary input node in the underlying AIG— or an existing one when such an assignment has already been recorded for that object. These Boolean variables along with the constants `t` and `nil` are the base Boolean formulas. More complex formulas are built up from these variables by processing of `if` terms and by low-level meta-routines, introduced below.

The Boolean variables needed for the `bitcount` proof correspond to the bits of the accessed registers of the initial machine state `st`. We introduce rewrite rules that cause FGL to generate 32 Boolean variables for the bits of a 32-bit register when that register is accessed, composing these into a symbolic integer. The two rules involved are shown in Listing 1.8.

Listing 1.8. Rules for generating Boolean variables for initial register values

```
(def-fgl-rewrite get-st-reg-generate-bits
  (implies (syntaxp (fgl-object-case st :g-var))
           (equal (get-st-reg n st)
                  (zero-extend 32 (hide-get-st-reg n st)))))

(def-fgl-rewrite zero-extend-const-width
  (implies (syntaxp (integerp n))
           (equal (zero-extend n x)
                  (if (or (not (integerp n))
                          (<= n 0))
                      0
                    (intcons (intcar x)
                             (zero-extend (1- n) (intcdr x)))))))
```

The FGL rewriter will try to apply the first rule, `get-st-reg-generate-bits`, every time it encounters a call of `get-st-reg`, but due to its `syntaxp` hypothesis it will immediately fail if `st` is not syntactically a variable. In the case of the conjecture we're attempting to prove, this ensures that the rule will only apply to `get-st-reg` calls on the initial state. Such calls will be replaced by the `zero-extend` term of the right-hand side. In that term, `hide-get-st-reg` is an alias for `get-st-reg`; this avoids looping in the application of the rule. The construction of the 32-bit vector of Boolean variables is then accomplished by repeated application of the rule `zero-extend-const-width`. The functions `intcar`, `intcdr`, and `intcons` used here to access or construct bits of an integer as if it were a list of Booleans: `intcar` gets the Boolean value of the least-significant bit (LSB), `intcdr` right-shifts by 1 to remove the LSB, and `intcons` adds a new LSB to an integer, reversing the `intcdr` operation. The first argument to `intcons` is recognized by FGL as a Boolean context, so the rewriter will introduce Boolean variables corresponding to the terms that appear there, namely:

```
(intcar (intcdr ... (intcdr (hide-st-get k st)) ...))
```

The association of each such termlike object with the corresponding Boolean variable is stored in a hash table. Each time a termlike object is found in a Boolean context, it is looked up in the table; if it has an existing entry, the corresponding Boolean variable is returned, and if not, a new Boolean variable is generated and stored.

After generating the new Boolean variable, the `intcons` call becomes a new symbolic integer that now includes that bit. The final value produced by the zero-extend is therefore a symbolic integer consisting of 32 fresh Boolean variables. If the same register were to be accessed again, the same process would occur except that the objects associated with the Boolean variables would be recognized and the same Boolean variables returned again.

4.3 Composing Boolean Functions

The most basic way in which a new Boolean formula is computed from a previous one during FGL's rewriting process is by FGL's built-in handling of `if`. Specifically, if an `if` term occurs in which the two branches are both symbolic

Listing 1.9. Bitwise AND implementation rule

```
(def-fgl-rewrite fgl-bitwise-and
  (equal (bitwise-and x y)
         (if (int-endp-check x-endp x)
             (if (intcar x) (ifix y) 0)
           (if (int-endp-check y-endp y)
               (if (intcar y) (ifix x) 0)
             (intcons (and (intcar x)
                           (intcar y))
                      (bitwise-and (intcdr x) (intcdr y)))))))
```

Boolean objects, the result is the Boolean if-then-else of the test formula and the two branch formulas. This if-then-else formula is built in the AIG and a reference to the resulting node is returned as the Boolean formula resulting from the `if`. If the two branches are both integer values represented either as symbolic integers or integer constants, then the result is a new symbolic integer, the bits of which are the if-then-elses of the test with the corresponding bits from the two branches.

As a simple example, the rule used to expand calls of `bitwise-and` is shown in Listing 1.9. This rewrites a call of `bitwise-and` on a pair of symbolic integers, producing a new symbolic integer in which each bit's formula is the AND of the corresponding bits of the inputs.

The rule applies to any call of `bitwise-and`. It first checks each of the inputs with `int-endp-check`. This is true if it can be syntactically determined that the input must be either -1 or 0—in particular, if the input's symbolic integer representation has only one bit. (The syntactic check works by binding its result to the free variable `x-endp` introduced within the form. The technical details of this rewriter feature are described elsewhere [39].) If this is true of either input, then the result is based on the one relevant bit of that input (the `intcar`): if it is true, then the input's value is -1 and the result is the other input (coerced to an integer value using `ifix`, which replaces non-integer values with 0); if false, then the input's value is 0 and therefore the result is too. In many cases, the `intcar` value will be a (non-constant) Boolean formula; the result of this `if` is then a new vector of Boolean formulas, each of which is the conjunction of the `intcar` formula with the corresponding bit of the other input.

If the `int-endp-check` test is false on both inputs, then the rule creates the first bit of the result by creating the `and` of the first bits of the two inputs. (In ACL2, `(and x y)` is really shorthand for `(if x y nil)`, so this is actually another `if` merge operation.) It then makes another call of `bitwise-and` on the remaining bits of the two inputs, which will cause another application of this rule; this recurs until the bits of one of the inputs are exhausted.

The `bitwise-and` rule is a particularly simple example of how FGL can be programmed to compute complex Boolean formulas, but designing and proving these sorts of rules for other operations is a straightforward exercise in interactive theorem proving. FGL also includes a library of such rules which the user can safely extend with new rewrite rules as needed.

For some applications, the performance of stepping through iterative rules such as these using the rewriter is insufficient. For these cases, FGL supports

creating custom rewriting procedures analogous to ACL2's metafunctions [23] and invoking them via rules similar to ACL2's meta rules. Metafunctions operate directly on the syntactic forms to be rewritten—symbolic objects in FGL, terms in ACL2. They return a resulting term (and substitution in FGL, though not in ACL2) that is equivalent to the input object. To allow a metafunction to be applied during rewriting, a meta rule is admitted, which requires proving a theorem stating that the metafunction produces correct results. It is noteworthy that FGL itself is proven in ACL2 to produce correct results even with user extension via rewrite rules or custom rewriting procedures.

5 Conclusion

Over the past years, formal verification at Centaur has moved beyond its previous focus on data-path proofs for arithmetic modules. Our verification projects have expanded into the areas of front-end decoding and microcode, as well as the implementations of a rich set of microoperations. We engage with the design process in its early stages and maintain and expand our proofs throughout the whole life cycle of the project. Over the years, our tools have been improved and we have learned a few lessons.

We chose to use open-source tools and we are constantly contributing to ACL2 libraries. The ACL2 community has a tested way of collaboration between groups using `git`, peer reviewed commits, and a rich regression suite.

We write specifications that can be expanded and refined in response to design and microarchitectural changes. When the design is incomplete, the specifications are still useful when augmented by relevant assumptions. When a project requires additional flags or features, a modular style of specification allows for appropriate changes. We try to avoid complex specifications like those for the front-end decoder or ROM instruction decoder. These parts of the design are implicitly verified during microcode verification.

Scheduled, triggered, and manual regressions are an important safeguard to avoid breaking consistency among our proofs. They catch undesirable changes in the specifications, tools, and design.

A key to ensuring stability of the proofs is their scope—the bigger the scope, the more stable the proofs, because changes to interfaces of larger modules are less frequent than changes at lower levels. The transition from unit to cluster-level proofs led to substantially higher robustness and easier maintenance. This has been possible due to improvements in the process of building our formal models and enhancements in FGL. We also benefit greatly from enhancements in modern SAT solvers.

We still have considerable work to do towards achieving our verification goals. Some of these goals could be achieved with more man power, whereas for others we do not have the right technology yet. There is a lot of microcode left to be verified. We have not verified the mechanisms of out-of-order microoperation scheduling, but we believe it is possible with our tools. We do not have a complete methodology for verification of memory access instructions yet. Our plan is to work on all these fronts.

References

1. ACL2 Documentation: AIGNET-ABC-INTERFACE Interface to ABC. Accessed April 2021. http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/?topic=AIGNET___AIGNET-ABC-INTERFACE
2. ACL2 Documentation: CLAUSE-PROCESSOR. Accessed April 2021. http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/?topic=ACL2___CLAUSE-PROCESSOR
3. ACL2 Documentation: FAST-ALISTS. Accessed April 2021. http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/?topic=ACL2___FAST-ALISTS
4. ACL2 Documentation: FGL Bit-blasting Prover Framework. Accessed April 2021. https://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/?topic=FGL___FGL
5. ACL2 Documentation: SMTLINK Interface to Z3. Accessed April 2021. http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/?topic=SMT___SMTLINK
6. ACL2 Documentation: SV Hardware Verification Library. Accessed April 2021. http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/?topic=ACL2___SV
7. ACL2 Documentation: VL Verilog Toolkit. Accessed April 2021. http://www.cs.utexas.edu/users/moore/acl2/v8-3/combined-manual/?topic=ACL2___VL
8. ACL2 Home Page. Accessed April 2021. http://www.cs.utexas.edu/users/moore/acl2
9. FGL Library in the ACL2 Community Books. Accessed April 2021. https://github.com/acl2/acl2/tree/master/books/centaur/fgl
10. VL Verilog Toolkit. Accessed: April 2021. https://github.com/acl2/acl2/tree/master/books/centaur/vl
11. Anderson, S.E.: Bit twiddling hacks. Accessed: April 2021. https://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel
12. Blanchette, J., Kaliszyk, C., Paulson, L., Urban, J.: Hammering towards QED. J. Formaliz. Reason. **9**(1), 101–148 (2016). https://doi.org/10.6092/issn.1972-5787/4593
13. Boyer, R.S., Hunt, Jr., W.A.: Symbolic simulation in ACL2. In: Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and Its Applications, ACL2 2009, pp. 20–24. ACM, New York (2009). https://doi.org/10.1145/1637837.1637840
14. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
15. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
16. Cook, B.: Formal reasoning about the security of Amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part I. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3
17. Davis, J., Kaufmann, M.: Industrial-strength documentation for ACL2. In: Proceedings of the 12th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2014, Vienna, Austria, 12–13 July 2014, pp. 9–25 (2014). https://doi.org/10.4204/EPTCS.152.2

18. Davis, J., Slobodova, A., Swords, S.: Microcode verification – another piece of the microprocessor verification puzzle. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 1–16. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_1
19. Dill, D.L.: Formal Verification of Libra Blockchain Smart Contracts. Recording of the keynote (2020). https://www.youtube.com/watch?v=cYxxJU-Wt2U
20. Goel, S.: Formal Verification of Application and System Programs Based on a Validated x86 ISA Model. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin (2016). http://hdl.handle.net/2152/46437
21. Goel, S., Slobodova, A., Sumners, R., Swords, S.: Verifying x86 instruction implementations. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, pp. 47–60. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3372885.3373811
22. Greve, D., Wilding, M.: Evaluatable, high-assurance microprocessors. In: NSA High-Confidence Systems and Software Conference (HCSS), Linthicum, MD, March 2002. http://hokiepokie.org/docs/hcss02/proceedings.pdf
23. Hunt, W.A., Kaufmann, M., Krug, R.B., Moore, J.S., Smith, E.W.: Meta reasoning in ACL2. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 163–178. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_11
24. Hunt, Jr., W.A., Kaufmann, M., Moore, J.S., Slobodova, A.: Industrial hardware and software verification with ACL2. In: Verified Trustworthy Software Systems, vol. 375. The Royal Society (2017). https://doi.org/10.1098/rsta.2015.0399 (Article Number 20150399)
25. Hunt, W.A., Swords, S.: Centaur technology media unit verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 353–367. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_28
26. Hunt, Jr., W.A.A., Swords, S., Davis, J., Slobodova, A.: Use of formal verification at centaur technology. In: Hardin, D. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 65–88. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-1539-9_3
27. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, November, 2020, Order Number: 325462–070US. https://software.intel.com/en-us/articles/intel-sdm
28. Kaivola, R., et al.: Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 414–429. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_32
29. Kaufmann, M., Moore, J.S.: Limited second-order functionality in the first-order setting. J. Autom. Reason. **64**, 391–422 (2020). https://doi.org/10.1007/s10817-018-09505-9
30. O'Hearn, P.W.: Formal reasoning and the hacker way (keynote). In: Krishnan, P., Reichenbach, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP@PLDI 2020, London, UK, 15 June 2020, p. 1. ACM (2020). https://doi.org/10.1145/3394451.3401953
31. Peng, Y., Greenstreet, M.R.: Smtlink 2.0. In: Electronic Proceedings in Theoretical Computer Science, vol. 280, pp. 143–160, October 2018. https://doi.org/10.4204/eptcs.280.11
32. Rager, D.L., Ebergen, J., Nadezhin, D., Lee, A., Chau, C., Selfridge, B.: Formal Verification of Division and Square Root Implementations, an Oracle Report, pp. 149–160. ACM, IEEE, October 2016

33. Reeber, E., Hunt, W.A.: A SAT-based decision procedure for the subclass of unrollable list formulas in ACL2 (SULFA). In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 453–467. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_38
34. Reid, A., et al.: End-to-end verification of processors with ISA-formal. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part II. LNCS, vol. 9780, pp. 42–58. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_3
35. Russinoff, D.M.: Formal Verification of Floating-Point Hardware Design: A Mathematical Approach. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-95513-1
36. Sawada, J., Sandon, P., Paruthi, V., Baumgartner, J., Case, M., Mony, H.: Hybrid verification of a hardware modular reduction engine. In: Bjesse, P., Slobodova, A. (eds.) Proceedings of Formal Methods in Computer-Aided Design (FMCAD). ACM/IEEE CEDA (2011). https://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD11/
37. Slobodova, A., Davis, J., Swords, S., Hunt, Jr., W.A.: A flexible formal verification framework for industrial scale validation. In: Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 89–97. IEEE/ACM, Cambridge (2011). https://doi.org/10.1109/memcod.2011.5970515
38. Swords, S.: Term-level reasoning in support of bit-blasting. In: Slobodova, A., Hunt, Jr., W.A. (eds.) Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 22–23 May 2017. Electronic Proceedings in Theoretical Computer Science, vol. 249, pp. 95–111. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.249.7
39. Swords, S.: New rewriter features in FGL. In: Passmore, G., Gamboa, R. (eds.) Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications, Worldwide, Planet Earth, 28–29 May 2020. Electronic Proceedings in Theoretical Computer Science, vol. 327, pp. 32–46. Open Publishing Association (2020). https://doi.org/10.4204/EPTCS.327.3
40. Swords, S.: FGL example. Accessed April 2021. https://github.com/solswords/fgl-example
41. Swords, S., Davis, J.: Bit-blasting ACL2 theorems. In: Hardin, D., Schmaltz, J. (eds.) Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 3–4 November 2011. Electronic Proceedings in Theoretical Computer Science, vol. 70, pp. 84–102. Open Publishing Association (2011). https://doi.org/10.4204/EPTCS.70.7
42. Swords, S.O.: A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover. Ph.D. thesis, University of Texas at Austin, December 2010. http://hdl.handle.net/2152/ETD-UT-2010-12-2210
43. Temel, M., Slobodova, A., Hunt, W.A.: Automated and scalable verification of integer multipliers. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 485–507. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_23
44. Zhong, J.E., et al.: The move prover. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020, Part I. LNCS, vol. 12224, pp. 137–150. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_7

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.