PHILOSOPHICAL TRANSACTIONS A

rsta.royalsocietypublishing.org

Research

Article submitted to journal

Subject Areas:

xxxxx, xxxxx, xxxx

Keywords:

xxxx, xxxx, xxxx

Author for correspondence:

J Strother Moore

e-mail: moore@cs.utexas.edu

Industrial Hardware and Software Verification with ACL2

Warren A. Hunt, Jr.[1], Matt Kaufmann[1], J Strother Moore[1] and Anna Slobodova[2]

[1]Department of Computer Science
University of Texas at Austin
e-mail: {hunt,kaufmann,moore}@cs.utexas.edu

[2]Centaur Technology, Inc.
7600-C N. Capital of Texas Hwy, Suite 300
Austin, TX 78731
e-mail: anna@centtech.com

ACL2 has seen sustained industrial use since the mid 1990s. Companies that have used ACL2 regularly include AMD, Centaur Technology, IBM, Intel, Kestrel Institute, Motorola/Freescale, Oracle, and Rockwell Collins. This paper focuses on how and why ACL2 is used in industry. ACL2 is well-suited to its industrial application to numerous software and hardware systems because it is an integrated programming/proof environment supporting a subset of the ANSI standard Common Lisp programming language. As a programming language ACL2 permits the coding of efficient and robust programs; as a prover ACL2 can be fully automatic but provides many features permitting domain-specific human-supplied guidance at various levels of abstraction. ACL2 specifications and models often serve as efficient execution engines for the modeled artifacts while permitting formal analysis and proof of properties. Crucially, ACL2 also provides support for the development and verification of other formal analysis tools. However, ACL2 did not find its way into industrial use merely because of its technical features. The core ACL2 user/development community has a shared vision of making mechanized verification routine when appropriate and has been committed to this vision for the quarter century since the Computational Logic, Inc., Verified Stack. The community has focused on demonstrating the viability of the tool by taking on industrial projects (often at the expense of not being able to publish much)

THE ROYAL SOCIETY PUBLISHING

1. Introduction

ACL2 is a logic and programming language for modeling computer systems, together with an integrated verification environment. "ACL2" stands for "*A* Computational *L*ogic for *A*pplicative *C*ommon *L*isp."

ACL2's logic is a first-order logic that is based on an applicative programming language, in particular, an extended applicative subset of Common Lisp [1]. The ACL2 system, including its theorem prover and system development environment, is largely coded in that same applicative programming language.[1] ACL2 is available in open source form, without cost, under the terms of a 3-clause BSD license. See the ACL2 home page [2].

Extensive documentation is available online.[2] In this paper we sometimes refer to it by writing "see :DOC x," which means: go to the ACL2 home page [2], click on The User's Manuals link, then click on the ACL2+Books Manual link and type x into the **Jump to** box.

We also refer to the *ACL2 Community Books*. An ACL2 *book* is a file of prover commands created by the user, including definitions and theorems. These are verified by ACL2 when the book is "certified." Books may include other books and are thus incremental and hierarchical. The ACL2 Community Books reside on GitHub[3] and are contributed, documented, and maintained by the ACL2 user community. As of February, 2016, there were 5,780 books in the repository, containing approximately 62,242 user-supplied definitions and 123,804 user-supplied theorems.[4]

This paper is about how the ACL2 system came to be used in industry. As such, the paper is more of a historical narrative than a technical article. The story is complicated by the fact that ACL2's adoption owes much to the close-knit Nqthm/ACL2 user community whose members share a common vision of making mechanized proof a routine industrial practice when appropriate. That community, in turn, owes much of its coherence and vision to the place where the ACL2 project began, Computational Logic, Inc. (CLI). We start, in Section 2, by describing the origin of the ACL2 project in 1989 at CLI. In Section 3 we describe the early industrial demonstration projects. In Section 4 we briefly mention the closing of CLI, the dispersion of most of the community into industry, and continued industrial demonstrations. Section 5 describes how ACL2 was integrated into the design workflow at one particular company, Centaur Technology, Inc. (abbreviated "Centaur" below). Section 6 describes how ACL2 is used today at that company. Section 7 briefly lists some other ongoing industrial uses of ACL2. Then in Section 8 we discuss some strengths and weaknesses relevant to the industrial use of ACL2. In Section 9 we discuss a "softer" feature: the attitude of the developers toward "maintenance" and the needs of industrial users. We summarize the lessons learned in Section 10. Finally, we acknowledge our sponsors – and our debt to the companies and people who have made ACL2 as successful as it has become.

2. The Origin Story

The ACL2 project dates back to 1989 at Computational Logic, Inc. (CLI), when Robert S. Boyer and J Strother Moore (a co-author of this paper) decided that their Nqthm [3] prover was inadequate for many of the projects in which it was being used. Within a few years, ACL2 was being used

[1] Of the 5.7MB of code (not counting blank lines or comments) in the ACL2 source files as of February, 2016, 87% of it is written in ACL2's applicative language; the remaining 13% is written in Common Lisp outside the ACL2 subset. This "raw" code primarily implements the single-threaded state, fast applicative arrays and property lists, hash consing, memoization, some parallelization, fast serialization, and tracing.

[2] A text version of the web-based manual for ACL2 plus the books has size exceeding 68MB.

[3] The top of the directory structure is `https://github.com/acl2/acl2`. A particular file (book) may be found by clicking your way down the directory hierarchy. For example, to find `books/projects/codewalker/` start on the GitHub page above and click on `books`, then `projects`, etc.

[4] These counts must be taken with several grains of salt First, because ACL2 is first order the user might define several functions where a single higher order function would suffice. Second, and more significantly, these counts were derived from the source text of the books. Macros make it possible for the user to introduce definitions and theorems without explicitly using the standard `defun` and `defthm` commands. In addition, some macros introduce multiple definitions and theorems. Finally, while no book was scanned twice, there are some duplications of `defun`s and `defthm`s between different books.

in-house at CLI and Matt Kaufmann (a co-author of this paper) had joined the development team. In 1994, before ACL2 was released outside of CLI, Kaufmann and Moore wrote "Design Goals for ACL2" [4], the abstract for which best describes the situation.

> ACL2 is a theorem proving system under development at Computational Logic, Inc., by the authors of the Boyer-Moore system Nqthm, and its interactive enhancement, Pc-Nqthm, based on our perceptions of some of the inadequacies of Nqthm when used in large-scale verification projects. Foremost among those inadequacies is the fact that Nqthm's logic is an inefficient programming language. We now recognize that the efficiency of the logic as a programming language is of great importance because the models of microprocessors, operating systems, and languages typically constructed in verification projects must be executed to corroborate them against the realities they model. Simulation of such large scale systems stresses the logic in ways not imagined when Nqthm was designed. In addition, Nqthm does not adequately support certain proof techniques, nor does it encourage the reuse of previously developed libraries or the collaboration of semi-autonomous workers on different parts of a verification project. Finally, Nqthm is implemented in an informally specified programming language (Common Lisp) and hence is not subject to mechanical verification. ACL2 is our response to these perceived inadequacies. While the logic of Nqthm is based on pure Lisp, the logic of ACL2 is based on the applicative subset of Common Lisp. By adding to the applicative subset of Common Lisp a single-threaded notion of state, fast applicative arrays and property lists, and efficiently implemented multiple values, an efficient and practical applicative programming language is produced. By axiomatizing the primitives and introducing appropriate rules of inference and extension principles, that language can be turned into a logic.

The "Design Goals for ACL2" paper discusses some stressful applications of Nqthm, including the CLI Stack (discussed further below), the porting of the stack from one verified microprocessor (FM8502) to another (FM9001) [5,6], the latter of which was fabricated, and the modeling of the Motorola MC6820 and the verification of the `gcc`-produced MC68020 binary for 21 of the 22 programs in the Berkeley Unix C String Library [7].

The CLI Stack [8] (sometimes called the "Verified Stack") was a hardware/software system consisting of a gate-level description of a microprocessor (Hunt, a co-author of this paper), an instruction set architecture, an assembler, linker, and loader (Moore with help from Kaufmann), an operating system (W. R. Bevier), compilers from a mini-Pascal subset (W. D. Young) and a pure Lisp subset (A. D. Flatau) to the assembly code, and simple applications written in those high level languages (M. M. Wilding). The stack was verified in the following sense: Nqthm was used to prove that the formal semantics of each abstraction was preserved by the implementation on the next layer down. For example, it was proved that the gates implemented the ISA, that the assembly/linker/loader implemented the assembly language in terms of the ISA, that the compilers implemented their respective high-level languages in terms of the assembly code, etc. All of the theorems "stacked" in the sense that they fit together to prove top-level theorems: the behavior of an application running under the semantics of its high-level language was preserved by the composition of transformations down to the gates. Resource constraints were tracked and were explicitly handled at each level.

The implementation of each layer of the stack was tested by running the lower level model on the code generated by the compiler, etc., for three reasons: (i) to "debug" the implementations, (ii) to help formulate the exact statement of "equivalence," including the necessary side conditions, and (iii) to gain the confidence necessary to invest the effort in trying to prove correctness mechanically. This testing required simulating thousands of cycles on these various abstract machines.

CLI similarly tested the FM9001 microprocessor model against the fabricated gate array and tested the MC68020 model against a Sun 3 workstation. Since all of these models were moderately large Lisp systems, execution efficiency was very important. To quote again from [4]:

> Note that the post-fabrication testing of verified devices changes the role of model execution. Heretofore execution was merely an efficient way to avoid "premature" proof attempts. ... But the post-fabrication testing of a verified device is not a purely mathematical question. On the one hand one has a physical object. On the other one has a mathematical expression. The question is whether the behavior of the object is accurately predicted by the mathematical expression. The behavior of the object can only be manifested by giving it concrete data and observing its concrete output. Thus, one is forced to give concrete data to the mathematical expression and derive its concrete output. ... Often, at least for devices for which verification is needed, the required computations are so large and the test cases so numerous that we expect the efficiency with which the model can be executed becomes an important issue.

The performance challenges raised by the use of Nqthm in the Stack and MC68020 projects prompted Boyer, Moore, and Kaufmann to begin creating ACL2 in the late 1980s.

3. Initial Industrial Demonstrations

By 1993, CLI was ready to try ACL2 on industrial verification projects. Finding suitable projects was primarily the responsibility of Warren Hunt, CLI's Vice President for Hardware Engineering.

The first project was the modeling and verification of a Motorola digital signal processor, the Motorola Complex Arithmetic Processor (CAP) DSP. CLI's involvement in the project started in late 1993, with Bishop Brock relocating from CLI's home in Austin, TX, to Phoenix, AZ, to work with the Motorola Government Systems' DSP design team. The formal methods project lasted 31 months. In the end, ACL2 was used to model the processor and its microcode and to verify that the microcode semantics was correctly implemented. The most significant contribution was the identification of over 50 "pipeline hazards," the coding of a Lisp (ACL2) function for recognizing those hazards, and the proof that the absence of those recognized hazards implied accurate execution of the microcode. The compiled hazard detection predicate was used to identify hazards even after the proof part of the project ended. For reasons not related to formal methods, the CAP DSP was never released as a product [9–11].

The second important industrial verification project was the verification of the pseudocode for floating-point division on the AMD K5 microprocessor of Advanced Micro Devices, Inc. This proof by Moore and Kaufmann with help from the lead FPU designer at AMD, Tom Lynch, was done before the K5 was fabricated. That project lasted about 2 months, starting in June, 1995, and involved formalizing in ACL2 the relevant part of the IEEE 754 floating-point standard [12], modeling the microcode for the `FDIV` operation, and proving that it satisfies the specification [13].

Hidden in this discussion of the emergence of ACL2 is a more subtle aspect of the experience of working at CLI during that period. Prior to the founding of CLI, most of the people using Nqthm and later ACL2 at CLI had been located together in the Institute for Computing Science and Computer Applications at the University of Texas at Austin. When CLI had acquired space and contract support, the group was again located together off campus. Many of the researchers were among the first generation of PhD and Masters students of Boyer and Moore. The Verified Stack project brought a close-knit group even closer together because it required cooperation and collaboration to negotiate the interfaces and make adjustments up and down the stack as required by difficulties encountered at individual layers. Moving from Nqthm to the untried ACL2 for contract work increased the risk and the team spirit because those "in the trenches" needed to be supported. These endeavors, and the leadership of Don Good, President of CLI, helped produce

a team spirit and cemented a shared vision among all the researchers. This vision dominated not just formal meetings and proposals but hallway and coffee-pot conversations.

Such "soft" features of ACL2's history are important to the subject of this paper because ACL2's subsequent success in industry may owe as much to that shared community vision as it does to ACL2's technical features. We list the names of the researchers using Nqthm and ACL2 extensively at CLI during this period because many will occur later in this paper: Larry Akers, Ken Albin, Bill Bevier, Bob Boyer, Bishop Brock, Rich Cohen, Art Flatau, Warren Hunt, Matt Kaufmann, J Moore, David Russinoff, Larry Smith, Mike Smith, Matt Wilding, and Bill Young.

4. Dispersion

By 1997, it was increasingly hard for the researchers at CLI to get contracts to verify industrial designs. Instead, clients wanted to verify their designs in-house to protect their intellectual property. By 1999, CLI closed its doors and the ACL2 users at CLI had scattered to several companies: AMD, EDS, IBM, Motorola, and Rockwell Collins. Boyer and Moore returned to the University of Texas at Austin to train more students and to keep ACL2 non-proprietary.

Coincidentally, the US economy was booming in the late 1990s and companies were more than normally disposed towards experimental projects that might give them a competitive edge. Some dispersed CLI researchers therefore continued to use ACL2 and formal methods in many of their initial industrial projects.

By 1998, all of the elementary floating-point arithmetic for the AMD Athlon was verified to be IEEE compliant using ACL2 [14]. This proof was done at AMD by Russinoff before the Athlon was fabricated and was based on a mechanical translation of the Register-Transfer-Level (RTL) description of the design implemented primarily by Flatau. Before the ACL2 model was trusted to predict the behavior of the RTL it was run against AMD's own RTL simulator on a hundred million floating-point test vectors.

At approximately the same time at Rockwell Collins, an ACL2 microarchitectural simulator for the first silicon-implemented JVM (the design became the JEM1 of aJile Systems, Inc.) was produced [15].

Hunt, who left CLI in 1997 and joined IBM's Austin Research Laboratory, but who also served as an Adjunct Professor at UT Austin, supervised the next major step in ACL2 modeling of microarchitectures, namely J. Sawada's modeling and verification of the FM9801, an out-of-order microprocessor with speculative execution, exceptions, and program modification capability [16].

Another important step was the publication, in 2000, of two books about ACL2 [17,18], the first about the ACL2 programming language, logic, and prover, and the second about case studies. Both were written by Kaufmann, Manolios (a student of Moore), and Moore. The case studies included such industrially-relevant chapters as "High-Speed, Analyzable Simulators," (by Greve, Wilding, and Hardin, of Rockwell Collins), "The DE Language" (a hardware description language) (by Hunt, of IBM), "RTL Verification: A Floating-Point Multiplier," (by Russinoff and Flatau, of AMD), as well as chapters on graph theory proofs, calculus and real analysis, proof checkers, model checkers, and other topics.

Thereafter, properties of the AMD Opteron and other desktop microprocessors were verified, and a verified ACL2 BDD package was built [19] that achieved about 60% of the speed of the CUDD package. At IBM, the algorithms used for floating-point division and square root on the IBM Power 4 were verified [20]. At Rockwell Collins, the instruction equivalence of different implementations of a commercial microprocessor [21] was proved by Greve and Wilding; in addition they proved that the microcode for the Rockwell Collins AAMP7 implements a given security policy having to do with process separation [22]. That proof allowed the AAMP7 to be certified for Multiple Independent Levels of Security (MILS). At UT Austin, under Moore's supervision, H. Liu formalized the Sun Java Virtual Machine and verified certain properties of the Sun bytecode verifier as described in JSR-139 for J2ME JVMs [23] and the class loader [24].

In 2002, Hunt left IBM's Austin Research Laboratory and moved to the University of Texas at Austin. Hunt's arrival at UT added a hardware focus to the UT ACL2 group, which had largely

focused on low-level software like the JVM and higher-level verification issues like bi-simulation [25].

Then in 2005, Kaufmann (who left CLI in 1995 to join Motorola, then joined EDS in 1997 and AMD in 1999) joined UT Austin.

Aside from their ACL2 expertise, Hunt and Kaufmann brought extensive industrial experience into the UT ACL2 group. For example, Hunt had been involved in the IBM Power 4 design and was instrumental in the creation of IBM's highly successful PERCS project. Meanwhile, Kaufmann had worked at Motorola on a model-checker and associated BDD-based tools, at EDS on analysis tools to assist the mitigation of Y2K errors in COBOL code and then, at AMD, on ACL2 proofs (mainly about floating-point RTL) and on various C++ tools used in the RTL simulation environment, in particular one for checking multiprocessor memory ordering rules.

So the end of 2005 found a critical mass of "first generation" Nqthm/ACL2 users working in major microprocessor companies: Greve, Hardin, and Wilding at Rockwell Collins; Bevier, Flatau, Russinoff, and Sumners at AMD, soon to be joined by Liu; and Sawada at IBM. In addition, the ACL2 development team (Kaufmann and Moore) was at UT Austin, together with the leader of the ACL2 hardware verification team (Hunt), all training students in the use of ACL2, in particular to model microprocessors.[5]

The successful industrial applications (Motorola CAP, AMD K5 FDIV, Athlon and Opteron floating-point proofs, the Rockwell Collins AAMP7 information flow proofs, and the IBM Power 4 floating-point proofs) served as powerful testimony that ACL2 was useful in industry, and the UT team was positioned to supply both technical support and manpower.

Independently, about this same time, Boyer and Hunt were exploring integrating hash cons [26,27] into ACL2. Hash cons, which they called `hons`, is a memoized [28,29] version of the `cons` pairing function; `hons` produces a pair just like `cons`, but if the necessary pair is already allocated then a pointer to the existing pair is returned.[6]

Making `hons` efficient required coding directly in the Common Lisp layer below ACL2. The addition of `hons` supported the memoization of other functions. In addition, Boyer and Hunt used their modified ACL2 to implement a fast and mechanically verified BDD package in only 15 lines of ACL2. In April, 2007, an experimental extension of ACL2 Version 3.2 supporting `hons`, called ACL2(h), was released.[7]

5. Integrating ACL2 into the Microprocessor Design Workflow

As of 2007, industrial ACL2 projects were still more or less demonstration verification projects even if they were often done repeatedly for different products as at AMD. What was missing was the complete integration of ACL2 and ACL2-backed formal methods into the design workflow. But the microprocessor industry's workflow includes many tools, such as equivalence checkers and symbolic simulation[8] tools as well as less formal tools such as linters, synthesizers, etc.

In May of 2007, Hunt was asked by Terry Parks, a co-founder of Centaur Technology[9], to make a presentation to Centaur personnel about the technical feasibility of using formal verification techniques to check the correctness of arithmetic circuits. Also attending this meeting were Glenn

[5]It is beyond the scope of this paper to trace impacts of students and ACL2 users after the first generation, except as relevant to our focus on the infrastructure built at Centaur described in the next sections. Suffice it to say that the first generation was not the last to make significant impact in building tools, interfaces, and useful books, and enlarging the community by teaching still more students.

[6]Since ACL2's programming language is side-effect free, this is a correct behavior.

[7]Jared Davis and Sol Swords, of Centaur, made additional contributions. After extensive field testing by the whole community and further integration by the ACL2 authors, the `hons` extension became the default release of ACL2 Version 7.0 in January, 2015. See :DOC hons-and-memoization for details.

[8]By *symbolic simulation* we mean a process whereby objects from a given finite set are represented using nested structures whose leaves are Boolean constants and variables. The process computes related objects from definitions or other equations using Boolean decision methods ("bit blasting") typically based on binary decision diagrams (BDDs) or Boolean satisfiability procedures (SAT). Symbolic simulation is just a special case of the commonly understood notion of (symbolic) simplification.

[9]Centaur designs high performance, low-cost x86 compatible microprocessors; it is an independent subsidiary of VIA Technologies, Inc., a Taiwanese chipset manufacturer.

Henry (President of Centaur) and Rodney Hooker. Hunt, having unsuccessfully tried to inject ACL2 into IBM's design flow during his time there, was now better prepared to recommend an approach that had a chance of succeeding. Before the end of this one-hour meeting, Glenn Henry said that he would like Hunt to attempt to prove the correctness of the VIA Nano floating-point adder design. Henry made it clear that the ACL2 team was not to talk to any of his engineers – he didn't want some "academics" bothering his very busy engineers.

The effort began a few weeks later (after the end of the spring semester in 2007). Hunt enlisted his PhD student, Sol Swords, in the project. The unit to be verified was actually composed of four adders so that the unit could simultaneously add four pairs of single-precision numbers, two pairs of double-precision numbers, or one pair of extended-precision numbers. The unit could complete four floating-point additions in two steps (clock cycles), and it was pipelined so it could produce four results every clock cycle. The unit's description involved 33,700 lines of Verilog in 680 modules; its implementation involved 432,322 transistors. The floating-point adder was part of a larger media unit that had 1074 inputs, including 26 clock inputs, and 374 output signals. The unit could perform over 100 other operations – and there were many (> 1000) inputs that had to be set properly to force it to perform the additions to be verified.

The hardest analysis task was to show that the extended-precision (80-bit) addition was correct since this case involved the largest number of input bits.

Hunt and Swords first attacked the problem of translating the Verilog description into a form that ACL2 could use. Hunt's DE hardware description language, developed in ACL2 at IBM and published in 2001 [30], formed the basis of a new hierarchical language named E adequate to capture the small subset of Verilog used in VIA Nano FPU adder. The syntax and operational semantics of E were expressed in ACL2. With the help of Parks, Hunt and Swords developed a mechanical translator from Verilog into E.

To test their translation, they ran the Verilog simulator used by the Centaur designers to simulate the entire VIA Nano design and collect suitable inputs and outputs for the unit to be analyzed. ACL2's execution engine was used to test the E model against the Verilog simulator and gave Hunt and Swords the confidence to try to prove the model correct.

The E semantics included a mechanism to translate a circuit representation into several different symbolic forms, one being a BDD. Hunt and Swords realized that it would make more sense if any finite conjecture in ACL2 could be so represented. Boyer and Hunt addressed this problem and implemented, in Common Lisp, a symbolic simulation mode for the ACL2 logic. Their previous work on `hons` was crucial to performance and scaling.

This prototype symbolic simulator for ACL2 (called G for "Generalize") was hand-coded and its correctness was not obvious. Swords, in his capacity as Hunt's PhD student, eventually dealt with this problem in his dissertation research which concluded (in 2010) with an ACL2 symbolic simulator written in ACL2(h) and proved correct with ACL2(h). He named the resulting tool GL for "Generalize in the Logic" [31]. (See also :DOC gl.) GL has since become a key component in ACL2-based hardware verification.

Let us return to the 2007 attempt by Hunt and Swords to verify the floating-point adder. This was too large a problem to tackle directly with (the prototype) symbolic simulator, so Hunt and Swords developed a method to split a large symbolic simulation into a set of smaller ones that could be glued together by ACL2 proofs.

The result of translating the VIA Nano FPU Verilog into E and symbolically simulating it helped perform many simple checks on the resulting netlist, for example, a check that the output at a given cycle is a function of appropriate inputs. And, using various simplification techniques, they were able to represent the functionality of the adder circuit alone, rather than representing all operations (e.g., comparison) performed by its floating-point module.

Hunt and Swords next turned to the problem of formalizing floating-point arithmetic in ACL2. They did not use the floating-point library available at that time in the ACL2 Community Books (see :DOC rtl) because they wanted a specification more closely matched to the style of

their HDL.[10] Furthermore, they wanted the specification to include some Centaur-proprietary microarchitectural features beyond the IEEE 754 standard.

About five months after they started on the project, they were able to check the single-precision, floating-point addition. About two months later, they were able to confirm that when the adder unit was configured to add two 64-bit (double-precision) floating-point numbers, it worked correctly.

They spent several more months trying to confirm that the unit worked correctly for two extended-precision (80-bit) numbers. However, they discovered one pair of numbers that seemed to produce an incorrect result. They gave this pair of numbers to the VIA Nano simulation team and a few days later that team confirmed that the "official" model also produced the incorrect sum. The floating-point adder design team then agreed that an error had been found by the ACL2 team. The error was subtle, and required a mask change to fix.

This bug discovery is the event that triggered Centaur's investment in formal verification. Hunt helped Centaur hire an experienced formal verification expert, Anna Slobodova (a co-author of this paper). He also encouraged Centaur to permanently hire Jared Davis and Sol Swords, both ACL2 experts and PhD graduates (2009 and 2010, respectively) from the ACL2 group at UT Austin. Slobodova became the leader of the formal verification team at Centaur. Boyer and Hunt continued to work with Centaur, but not as intensely as they did during the first 18 months. Hunt remains involved with Centaur, but Boyer retired soon after the initial effort concluded.

6. ACL2 at Centaur Today

Since 2008, the use of ACL2 at Centaur transitioned from a proof of concept project to a mature verification framework that has become an indispensable part of the design process [32,33]. ACL2 proved to be a stable and reliable software system with a responsive support team. It allows a high customization of design automation tools with a solid formal base as well as fast prototyping.

Centaur's Formal Verification (FV) team has three full-time employees and regularly hires one or two graduate students as interns for three to six months. We do not have any numbers supporting financial savings that the company made by establishing the team. However, the company's management feedback lead us to believe that the team's contributions are highly valued. While its main focus is the validation of the RTL design, the FV team's wider goal is to provide design and verification support at various abstraction levels. The team routinely discovers new areas, where an analyzable design model offers a big advantage compared to simulation-based design and verification approaches. On many occasions, the FV team was able to offer quick solutions to unexpected problems. As a consequence, the team's goals are often changing and expanding, based on the current needs of Centaur engineers. The contributions of the ACL2 team include the validation of new algorithms, microcode, and custom design elements; timing analysis; and late changes in the design known as engineering change orders. It has developed a number of ACL2 tools discussed below. Some of these are proprietary but most are available in the ACL2 Community Books repository. These tools have extensive online documentation [34] and are distributed under a very permissive MIT-style license. Some of these tools have been successfully used in other companies (Intel and Oracle).

In the following Subsections we discuss how ACL2 is used at Centaur for formal specification ((a)), formal modeling ((b)), proofs that models satisfy their specifications ((c)) and the creation of a wide variety of custom tools that help in the design, debugging, and validation of microprocessor designs ((d)). In Section 8 we discuss some strengths and weaknesses of ACL2 from the industrial perspective with a focus on Centaur's usage.

(a) Formal Specification of Microprocessors

Centaur Technology, as a subsidiary of VIA Technologies Inc., designs microprocessors that are based on the Intel® 64 and IA-32 Instruction Set Architecture (ISA) [35]. The current series of

[10] Oracle does however use that library with the Verilog tools contributed by Centaur.

microprocessors is called VIA Eden. Centaur developed an ACL2 specification of a subset of the x86 architecture. That specification is validated by routinely running millions of tests against existing x86 machines from Intel, AMD and Centaur. This is one example of the importance of high-speed execution of ACL2 code. The x86 ISA specification is sometimes used in post-silicon validation. Another use is correlating ISA-level specification to the specification of the microarchitectural level.

The ISA can be viewed merely as an interface to the machine. Each ISA-level instruction is decoded internally into a sequence of microoperations and these microoperations comprise the true language of the VIA Eden microprocessor. The intended behavior of microoperations is a part of the microarchitectural specification. This specification serves a dual purpose – to verify that the units responsible for the execution of individual microoperations are correct, and as an operational semantics for the formal microcode model (see [36] for microcode verification).

(b) Formal Models of Hardware Design

The Centaur System Verilog model is considered to be the golden model of the VIA Eden design. Most functional validation is performed at this level of abstraction. While small parts of the VIA Eden design are refined into custom design blocks, most implementation elements are automatically synthesized into a transistor-level design. The FV team has built a verification flow that translates the synthesizable subset of the RTL design into a formal model that can be both executed and formally analyzed. Most of this translator is written in ACL2. The tool can translate the entire VIA Eden design (which is more than 700,000 lines of hierarchical System Verilog code) into a formal model in a few minutes. Since the RTL model is changing several times a day, the speed at which the corresponding formal model can be built is important. Design translation into a formal model is based on the IEEE Verilog specification. The ACL2 model is validated by comparing the output of its execution with that of commercial simulators on carefully crafted test suites. Centaur uses NC Verilog (Cadence) and VCS (Synopsys).[11]

Building a formal model of VIA Eden requires several non-trivial steps. The process of loading the design into ACL2 starts with a lexer and parser. After that the design can be statically analyzed by a linter. Then, the design representation is subjected to a sizing analysis which determines the width of individual expressions. Next, all modules are unparameterized, the hierarchical design is flattened, and each signal is assigned a so-called *S*ymbolic *V*ector *Ex*pression (SVEX). At this point the design has been rendered into a truly formal model, because each SVEX expression has a formal operational semantics as defined by the SVEX evaluator. The SVEX-based representation is finally transformed into a finite state machine model.[12] This entire tool chain is written in ACL2 although a few low-level utilities have been implemented "under the hood" (see Section 8). Some parts of the tool chain, e.g., all SVEX transformation rules, have been verified with ACL2. This tool chain is distributed as part of the ACL2 Community Books as the Verilog Toolkit and Symbolic Vector hardware verification library. See :DOC vl and :DOC sv. Symbolic simulation [31] of the formal RTL model (in SVEX form) is at the core of the Centaur formal verification method.

There is a big advantage of the multi-purpose use of the ACL2 language for both specification and modeling. Centaur can move from one validation task to another without additional translations that might introduce discrepancies in the verification process. That goes for moving from RTL verification to transistor-level verification, and from RTL verification to microcode verification. In typical design flow, RTL is refined into custom or synthesized transistor-level blocks. Centaur can extract an SVEX representation of the post-synthesis VIA Eden design and that model can be compared to the formal RTL model. The formal RTL model also plays a role in microcode verification [32]. In particular, a translator that generates microoperations from ROM instructions is part of the formal microcode model. At the same time specifications of microoperations that are used to verify the RTL provide operational semantics of the microcode

[11] Those two commercial simulators do not always agree, and Centaur chooses the more conservative interpretation in its formal model.
[12] Any potential combinational loops would have already been discovered.

model. Definitions, specifications, validations and formal verification of these models are carried out within the ACL2 system.

(c) Formal Proofs of Hardware Design Models

The two main areas where Centaur applies formal verification are for proving correctness of parts of the RTL design [32,33,37–40] and proving the correctness of microcode [41]. When talking about correctness of RTL, we mean proving that each individual microoperation is executed correctly with respect to its behavioral specification in ACL2. Most microoperations have a finite latency and are assigned to a specific execution unit. The FV team creates a formal model of such a unit and shows that symbolic simulation of this unit, for a fixed number of steps, matches the specification. This is achieved by translating SVEX expressions into And-Inverter-Graphs (AIGs) and then translating AIGs to either to Conjunctive Normal Forms (CNFs) or Binary Decision Diagrams (BDDs). Translation and the AIG and BDD "bit-blasting" are done with ACL2 functions that have been proven correct (some of the translations can be found in [39,42,43]; also see :DOC aig and :DOC gl). The CNF mode requires a SAT solver. SAT solvers used by Centaur are trusted but unverified external tools called via the use of ACL2 trust tags (see :DOC `defttag`), but their answers can be checked, and some checkers are formally verified [44].

In the case of microcode verification, the process is less automatic and the FV team focuses on the most critical code (no longer than several hundred lines). The key to Centaur's method is to symbolically simulate the microcode for a few steps, proving invariants about small blocks of code (with support for loop invariants [36]), and then compose those lemmas into a proof of an entire microcode routine. An important point is that the microcode model is based on the very same specifications that were used to verify the RTL model. So there is confidence that the model matches the actual behavior of the design.

An important feature of the ACL2 proofs at Centaur is that they can be re-run effectively. Most proof scripts are robust with respect to minor design changes. These proof scripts have been assembled into a proof regression suite that runs on a nightly and weekly schedule, and this process assures that no new bugs have been introduced into the design ("bugs introduced today are detected tonight and fixed tomorrow"). See the discussion of "proof maintenance" in Section 8.

(d) Custom Tools

Besides formal models and formal specifications, the Centaur FV team uses ACL2 for various custom built tools. We already mentioned its ACL2 linting tool (see :DOC lint), which helps to catch bugs before any model simulation is done. Each time a new design model is built (several times per day), the linter runs and reports warnings by sending email to the logic designers whose commit caused new warnings; the linter also updates a Lint web page where designers can explore all warnings for their modules. Another example is an RTL design browser that allows to explore RTL design (its hierarchy, module interfaces, critical expressions, etc.)

The VL-Mangle tool has been developed to help with reverse engineering of the design [45]. It allows transformations to be made that lead to a simplified, more readable design. It can also be used to extract a relevant part of the design (e.g., the clock tree) for further analysis.

One of the big problems Centaur engineers face is the interpretation of reports produced by timing analysis tools run on synthesized designs. Such reports on critical timing paths are delivered in terms of the synthesized gate names, which never appear in the source RTL. The FV team has developed an ACL2 tool that builds a formal model of the synthesized design and maps it to the RTL model by finding the gate-to-RTL signal correspondences. This can be used not only for interpretation of timing reports, but also when performing engineering changes at the mask level very late in the design process. Whenever the design needs to be changed (e.g., as a result of finding a bug), this tool helps to find the corresponding change to the synthesized design.

7. Other Ongoing Industrial Projects

We now turn to use of ACL2 at other companies. If one limits oneself to traditional academic publications it can be difficult to determine how any tool is used in industry as there is often little incentive to publish. This partially explains the relative dearth of ACL2-based papers in automated reasoning journals and conferences. (Another part of the explanation is that much of the work on the ACL2 system itself is of an engineering nature and explaining the interesting logical and heuristic issues that arise is often impractical.[13])

On the other hand, the industrial user community has been extraordinarily cooperative in contributing to the ACL2 Community Books repository. Furthermore, the user community holds a workshop every 18 months or so. The workshops offer a glimpse of other industrial applications.

The most recent workshop was held in October, 2015, in Austin, TX, in conjunction with FMCAD. The schedule, talks, some supporting material, and proceedings are publicly available. See the ACL2 Workshop webpage, `www.cs.utexas.edu/users/moore/acl2/workshop-2015` for the online schedule, which includes the slides for the talks, rump sessions, and panel discussions. See also the proceedings [46].

Among the industrially relevant presentations are "Software Synthesis with ACL2" and "Verifying Android apps with ACL2" both by Eric Smith of Kestrel Institute, "$2^{255} - 19$ is Prime: Toward Verified Elliptic Curve Cryptography" and "A Formal Theory of RTL and Computer Arithmetic" both by David Russinoff now of Intel (previously of AMD), "ACL2 as Verification Ecosystem" by Sol Swords of Centaur, "What's New in the Community Books" by Jared Davis of Centaur, "Industrial Use of ACL2: Present and Future" and "Reasoning About LLVM Code Using Codewalker" both by David Hardin of Rockwell Collins, "Formal Verification at AMD... then and now... ACL2 or not..." by Rob Sumners of AMD, and "A Brief Introduction to Oracle's Use of ACL2 in Verifying Floating-Point and Integer Arithmetic" by David L. Rager, Jo Ebergen, Austin Lee, Dmitry Nadezhin, Ben Selfridge, and Cuong K. Chau, of Oracle.

The presentation by Sumners of AMD makes it clear that the ACL2 team at AMD has suffered severe losses of manpower and a consequent reduction in the scope of formal methods at AMD. However, Sumners writes "Additionally, we have started using ACL2 again for defining and verifying high-level transaction protocol properties. For this work, ACL2 is primarily a tool development and target language for specification and proofs of properties used in the tool."

Russinoff's presentation on elliptic curve cryptography includes in the abstract "Curve25519 is an elliptic curve Diffie-Hellman function that has achieved new speed records and is used in a variety of high-security encryption applications. As a very first step in the formalization and proof of correctness of a proposed hardware implementation of this function, I'll describe an ACL2 proof of the primality of $2^{255} - 19$ based on Pratt certification." Approximately a month later Russinoff sent an email to the ACL2 community saying "I gave a rump session at the recent ACL2 workshop on a proof-in-progress of the Abelian group properties of the addition operation on the elliptic curve known as Curve25519, defined by the equation $y^2 = x^3 + Ax^2 + x$, where $A = 486662$, over the Galois field of order $p = 2^{255} - 19$. I'd like to report that the proof is done."

Hardin's talk on the verification of LLVM code mentions an ACL2 Community Book called Codewalker (`books/projects/codewalker/`), a decompiler that takes a formal ACL2 operational semantics model and a piece of code in the ISA described by that model and decompiles the code into an ACL2 function. Codewalker proves the generated function correct with respect to the semantics of the code. Codewalker is similar to the decompiler for HOL described in [47].

The workshop proceedings include some industrially relevant work in the ACL2 group at the University of Texas at Austin: "Stateman: Using Metafunctions to Manage Large Terms Representing Machine States" by Moore, "What's New in the ACL2 System" by Kaufmann and Moore, "CCL Compiler Work" by Hunt, and "The x86isa Books" by Hunt's PhD student, Shilpi

[13]However, it bears noting that the ACL2 online documentation is always current, extensive, and detailed. Furthermore, the source code is freely available and contains about 4MB of comments discussing motivating examples, rationales and justifications, alternative heuristics or implementations, performance results, etc.

Goel. Hunt's talk described improvements being made to the CCL compiler by Gary Byers with help from Bob Boyer, to better support ACL2 array access; that work is sponsored by Hunt's company, ForrestHunt, Inc. The Goel talk concerned a major effort within the group to formalize the x86 ISA in a way that allows both formal analysis and efficient execution of x86 machine-code.

8. Strengths and Weaknesses of ACL2 from an Industrial Perspective

In this section we discuss a few very high-level aspects of ACL2 that are important to industry, especially Centaur.

As noted, one of the main features making ACL2 suitable for industrial use is that it provides an integrated programming and verification environment for a competitive ANSI standard programming language, Common Lisp. ACL2 inherits Lisp's convenience for rapid prototyping, implementing other languages, and building powerful extensions.[14] ACL2 also inherits Lisp's syntax, which we believe is an advantage. Every well-formed expression in the external syntax corresponds to an object in the semantics that is displayed the same way, easing the tasks of extending the syntax with powerful macros and of writing other symbolic manipulation tools.

ACL2 is enormously complicated by the commitment to stay true to the *Common Lisp the Language* specification [1] aka "CLtL." The logic would be simpler if the ACL2 developers could change some things about CLtL — the confusion of `NIL` and false, the difference between "it is an error" and "an error shall be signaled," the complexity of the symbol package system, the restrictions on what can be in a compiled file, and the handling of macros, to name but a few. But by adhering to the CLtL standard ACL2 users can take advantage of six independent implementations (GCL, CCL, SBCL, Allegro, CMU Lisp, and LispWorks) that run on a wide variety of hardware and OS platforms. These provide a wide range of development, debugging, and browsing tools, along with a community of compiler implementors who strive (indeed, compete) to provide exceptional execution efficiency, robustness, and capacity.

The fact that ACL2's logic is "just" a programming language may have an unexpected impact on its perceived effectiveness: users do not try to do things that are wildly outside its strengths.

Common Lisp is syntactically untyped. But CLtL is riddled with runtime restrictions on the domains of the primitives (e.g., the function `+` "requires that its arguments all be numbers"); violations of these restrictions result in unspecified behavior. ACL2's axioms complete the specifications of the primitives. Thus, ACL2 cannot trust Common Lisp to evaluate an expression according to the axioms unless the runtime restrictions are satisfied. This issue is handled by ACL2's notions of *guards* (see :DOC guard) which may be used to annotate code via Common Lisp's compiler declarations. ACL2 *guard verification* is essentially type checking via proof; a function whose guards have been verified is run directly in Common Lisp, otherwise it is interpreted in accordance with the axioms. Centaur and other companies routinely use guards extensively both to type check their ACL2 routines and to attain Common Lisp execution speeds in their tool chains.

Of course, ACL2 also provides conventional logical facilities to reason about Lisp objects: a conservative definitional facility for total first order recursive functions, induction up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$, and the usual first order rules of inference. ACL2 provides logical extensions for such things as introducing constrained functions satisfying demonstrably consistent axioms, scoping primitives, second order instantiation, and the ability to extend the prover by defining and verifying new proof procedures. The latter, which may seem esoteric, is actually quite commonly used in industrial projects where special domain-specific simplifiers are built, verified, and used.[15]

[14] Not least among Lisp's important features is automatic storage allocation and garbage collection. In Centaur's environment hundreds of millions of conses are created in typical runs.

[15] The Centaur SVEX book contains six such verified simplifiers.

ACL2 includes such proof techniques as rewriting with equivalence relations [48,49] (see :DOC equivalence, :DOC congruence, and :DOC patterned-congruence), support for non-linear arithmetic [50] (see :DOC non-linear-arithmetic), extended metafunctions and clause processors allowing the use of external proof tools like SAT solvers [51,52] (see :DOC meta and :DOC clause-processor), proof debugging tools [53] (see :DOC debugging), hash cons and memoization [54] (see :DOC hons-and-memoization)[16], and many system programming facilities to support faster execution[17], faster loading of files, and the use of ACL2 both as a prototyping language and a language in which efficient, verified verification tools can be produced [55–57].

The ACL2 book mechanism allows the user to extend the theory in a session by including the definitions and theorems in a previously certified file. The ACL2 Community Books repository allows sharing among users. Among the most widely used books are ones for elementary arithmetic (see `books/arithmetic-5`), ordinal arithmetic (:DOC ordinals), floating-point arithmetic (:DOC rtl), bit-blasting (:DOC gl), Verilog and System Verilog (:DOC vl), symbolic vector hardware verification (:DOC sv), and various clause-processing tools (:DOC clause-processor-tools).

In a shallow sense, the prover is fully automatic: once it begins a proof attempt the user can offer no guidance except to abort the search. The various proof techniques mentioned above are integrated into a single engine which uses hundreds of heuristics to control their applications automatically. The prover can print a running commentary of its progress, which reads quite like a detailed but informal proof description.

When a proof attempt fails or is aborted by the user, the user must decide "what went wrong," usually by inspecting unproved *checkpoints* printed at the end of every failed proof attempt. Experienced ACL2 users have learned how to look at these checkpoints and diagnose strategic or tactical mistakes in the prover's search strategy. Most often the appropriate response to a proof failure is to formulate one or more lemmas (to be used automatically as rewrite rules, as discussed below), sometimes requiring the definition of new concepts to express the general properties.

That brings us to the deeper sense of the prover as an interactive proof assistant.

By default, every time a new definition is admitted or a new theorem is proved, one or more *rules* are created from it. These rules are henceforth and by default automatically used by the prover. The user can tag a definition or theorem to specify the kinds of rules to be built (see :DOC rule-classes). The most common are conditional rewrite rules designed to normalize expressions. But, depending on the form of the definition or theorem, it might also be used to suggest induction schemes, to restrict the range of variables introduced by automatic generalization, to extend the linear arithmetic procedure, install a new verified term or clause simplifier, etc. Thus, many users actually think of themselves as "programming" the prover by the selection of definitions and theorems and their tags. The definitions and theorems found in books also have such tags, so the inclusion of a book actually configures the system in ways designed by the book's author.

The user can also attach hints to a definition or theorem to help the prover admit the definition or prove the theorem (see :DOC hints). A typical hint for a definition is to tell the prover what measure is supposedly decreasing in the recursion. A typical hint for a theorem might be that on a particular subgoal the prover should not use certain rules or that the prover should appeal to a particular lemma or proof technique; indeed, it is possible for a hint to specify low-level proof commands. But goal-specific hints can be cumbersome, tedious, disruptive to the user's train of thought, and fragile in the face of modifications, so ACL2 also allows the user to develop "computed hints" that add hints when the goal is of a recognized form. These facilities are often exploited in sophisticated books to code proof strategies for particular domains.

The upshot of this user interface is that users tend not to focus on particular proofs but on proof strategies as coded in rules and hints. They then design books to implement those strategies and expect those books to succeed in proving a wide variety of theorems in the given problem domain.

[16]Centaur's SVEX, AIG, and BDD packages and the loading of the entire VIA Eden design would be impractical were it not for hons and memoization.

[17]Aside from guards, another commonly used feature at Centaur and other companies is `mbe` (see :DOC `mbe`), a way to attach faster executable code to a logical expression provided they are proved equivalent.

This is key to what we call *proof maintenance*, which is of particular concern to industry. Designs are not static. A design that is verified today might be changed tomorrow and need to be "re-verified" (although technically it is not being *re*-verified because the design is different from the one that was verified). One therefore wants a fully automatic prover for the problem domain, i.e., a prover configured with a general enough strategy to succeed repeatedly despite "minor" modifications to the definitions and conjectures. That ACL2 permits the development of such strategies is evidenced by its routine nightly use at Centaur.

We should also mention that the 3-clause BSD license carried by the ACL2 sources is also regarded as industrially important. For many years ACL2 carried a GNU GPL license but some companies were reluctant to use the prover in-house. This changed when the BSD license was adopted for ACL2 in 2012. According to that license, "Redistribution and use in source and binary forms, with or without modification, are permitted" provided three rather liberal conditions are met. See the license link on the ACL2 home page [2].

The weaknesses of ACL2 for industrial can be highlighted by Centaur's tool chains.

Most of Centaur's FV code is written in ACL2. There are a few places where non-ACL2 ("raw") Common Lisp offers considerably better execution than guard verified ACL2. In some cases the FV team has "cheated" by defining unverified executable counterparts to ACL2 functions to improve performance of, e.g., string manipulation and printing, and to provide finer-grained control over garbage collection. Some of these "cheats" have been adopted by the ACL2 implementors and others probably will be. See Section 9.

ACL2 is not a particularly efficient or convenient scripting language. Centaur uses Perl, Ruby, shell scripts and Makefiles for system build support. Those languages are used because of the faster loading, better file system support, and support of parallelism. Because of its nature, this support code does not need to be verified. On the other hand, the majority of the code that processes SVEX expressions is written in pure ACL2, and has been subjected to verification.

Centaur's connection to external SAT solvers is written in raw Common Lisp; this provides the ability to interrupt the SAT solver if it does not provide an answer in an acceptable time. Raw Common Lisp is also used for OS level routines like listing directories, copying files, etc., which are not a part of ACL2. Many of these capabilities are made available (via trust tags) in Centaur's Operating System Utilities books (see :DOC oslib).

Connection to the ABC (`www.eecs.berkeley.edu/~alanmi/abc/abc.htm`) equivalence and property checker is written outside ACL2 as well. This tool is used (via a trust tag) for manipulation of AIGs whenever a direct call to SAT is unsatisfactory.

Some Centaur tools, in particular those used by designers or that require visualization, have web interfaces. The web application components are written in HTML, Javascript, and CSS instead of ACL2. A Common Lisp web server is used for the RTL design browser. Similarly, the web-based interface to the linter and the management of automatically-generated error-report emails is written in Javascript. The analysis part of the linter code is entirely written in ACL2.

The tool Centaur uses to validate its ACL2 ISA level specification against various x86 hardware platforms contains C and assembler, because this allows a convenient way to interact with a host operating system; e.g., to keep track of floating-point exceptions. That connection between ACL2 and C uses low-level Common Lisp.

We note that the main industrial "complaints" about ACL2 have less to do with its specification and proof power than the ability to integrate it smoothly with the wide variety of programming languages and tools in the conventional industrial workflow.

9. Prover Development and Maintenance

Another reason ACL2 is used in industry is that the ACL2 developers are responsive to requests from industrial users. This fundamentally goes back to the coherence, shared vision and team spirit mentioned earlier: the entire community wants to see practical formal methods succeed and cares less about traditional academic publications than practical success.

A new version of ACL2 is released roughly twice a year. With each release, there is a documentation topic briefly discussing the changes. These release notes are routinely divided into five main sections: Changes to Existing Features, New Features, Heuristic and Efficiency Improvements, Bug Fixes, and Changes at the System Level. Individual notes generally acknowledge the users who identified the problem and/or suggested solutions. See, for example, :DOC note-7-2.

There are too many changes to summarize them here. For example, there have been over 1000 individual changes since Centaur started using ACL2. We therefore focus here on change requests from Centaur.

The release note records indicate that Centaur has been engaged with ACL2 since 2009. ACL2 Version 3.5 was released in May, 2009, and so we start our statistics gathering there and go through Version 7.2, released in January, 2016. We count the changes requested by Centaur, by category.

Changes to Existing Features	95
New Features	44
Heuristic and Efficiency Improvements	22
Bug Fixes	72
Changes at the System Level	18

These changes range from trivial to profound. For example, many changes relate to improved facilities for documentation, warning, and error messages. Others relate to new features reflecting the underlying machine state or "raw" Lisp such as wall-clock time, passing additional directives to the Common Lisp compiler, inlining, etc. Others relate to faster input/output facilities, new read macros permitting Verilog syntax to be parsed more easily, and deep performance improvements in book certification and inclusion. Finally, some fundamental new concepts were added to ACL2 such as *congruent*, *abstract*, and *nested* single-threaded objects [58] (see :DOC stobj), as well as *patterned congruences* [49] (see :DOC patterned-congruence). At Centaur these features are used especially in SVEX and the AIG libraries. It should be noted that some of these changes were also requested or suggested by other ACL2 users so they are not all uniquely driven by Centaur.

10. Lessons

Why has ACL2 succeeded to the extent that it has? Among the reasons are

- Everything is done in a single logic and that logic is an efficient, executable, ANSI standard programming language; models have dual use as emulators and formal semantics, and properties can be composed.
- ACL2 encourages modelers to focus on problems that are appropriate for modeling with a programming language and to be bit- and cycle-accurate; conventional mathematical techniques are used to hide or expose the resulting complexity and to glue results together.
- While being careful to obey the Common Lisp standard, ACL2 supports facilities for system programming. It furthermore allows the user to verify utilities thus built, making ACL2 useful for building trusted design and analysis tools. It supports the use of trusted but unverified external tools for which the user takes responsibility via trust tags.
- ACL2 puts a "human in the loop" doing things humans do well – inventing concepts and lemmas – which can facilitate proof maintenance.
- ACL2 has been engineered to be rugged and to handle large models.
- ACL2 is well documented, available for free in open source form, includes many useful books oriented towards industrial-scale projects, is well supported, and carries a fairly unrestrictive license.
- The ACL2 community shares the vision of making mechanized verification practical and is very supportive of industrial projects.

- Industry needs help: modern machines are too complicated to be designed accurately without mechanized reasoning.

Competing Interests. The author(s) declare that they have no competing interests.

Funding. Preparation of this paper was partially supported by DARPA under Contract No. FA8750-15-C-0007. We are grateful for support of ACL2 development and deployment for over 20 years from many sources; see :DOC acknowledgments.

Acknowledgements. This paper would not be possible without the dedication and enthusiasm of the ACL2 user community and the companies that use or have used ACL2, including AMD, Centaur, IBM, Intel, Kestrel Institute, Oracle, and Rockwell Collins. Thanking the individuals who led important industrial proof projects and/or contributed books or improvements used by the community would require a recapitulation of the paper, so we just thank all the people mentioned in the text! We also wish to thank Sol Swords for helpful feedback on a draft of this paper, and to thank the many sponsors of the ACL2 group at the University of Texas at Austin, including NSF and DARPA. Finally, we would especially like to thank ForrestHunt, Inc., for its continuing support of the ACL2 project.

References

1. Steele GL Jr.
   Common Lisp The Language, Second Edition.
   30 North Avenue, Burlington, MA. 01803: Digital Press; 1990.
2. Kaufmann M, Moore JS.
   The ACL2 Home Page.
   In: http://www.cs.utexas.edu/users/moore/acl2/. Dept. of Computer Sciences, University of Texas at Austin; 2014. .
3. Boyer RS, Moore JS.
   A Computational Logic Handbook, Second Edition.
   New York: Academic Press; 1997.
4. Kaufman M, Moore JS.
   Design Goals for ACL2.
   Computational Logic, Inc.; 1994. CLI Tech Report 101.
   See http://www.cs.utexas.edu/users/moore/publications/km94.pdf.
5. Hunt WA Jr, Brock B.
   A Formal HDL and its use in the FM9001 Verification.
   Proceedings of the Royal Society. 1992 April;.
6. Moore JS.
   Piton: A Mechanically Verified Assembly-Level Language.
   Automated Reasoning Series, Kluwer Academic Publishers; 1996.
7. Boyer RS, Yu Y.
   Automated Proofs of Object Code for a Widely Used Microprocessor.
   Journal of the ACM. 1996 January;43(1):166–192.
8. Bevier WR, Hunt WA Jr, Moore JS, Young WD.
   Special Issue on System Verification.
   Journal of Automated Reasoning. 1989;5(4):409–530.
9. Brock B, Kaufmann M, Moore JS.
   ACL2 Theorems about Commercial Microprocessors.
   In: Srivas M, Camilleri A, editors. Formal Methods in Computer-Aided Design (FMCAD'96). Heidelberg: Springer-Verlag, LNCS 1166; 1996. p. 275–293.
   http://www.cs.utexas.edu/users/moore/publications/bkm96.ps.Z.
10. Brock B, Hunt WA Jr.
    Formal Analysis of the Motorola CAP DSP.
    In: Industrial-Strength Formal Methods. Springer-Verlag; 1999. .
11. Brock B, Moore JS.
    A Mechanically Checked Proof of a Comparator Sort Algorithm.

In: Engineering Theories of Software Intensive Systems. vol. 195. Springer NATO Science Series II; 2005. .
12. Standards Committee of the IEEE Computer Society.
IEEE Standard for Binary Floating-Point Arithmetic.
IEEE, 345 East 47th Street, New York, NY 10017; 1985. IEEE Std. 754-1985.
13. Moore JS, Lynch T, Kaufmann M.
A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating Point Division Algorithm.
IEEE Transactions on Computers. 1998 September;47(9):913–926.
14. Russinoff D.
A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions.
London Mathematical Society Journal of Computation and Mathematics. 1998 December;1:148–200.
Http://www.onr.com/user/russ/david/k7-div-sqrt.html.
15. Greve DA.
Symbolic Simulation of the JEM1 Microprocessor.
In: Gopalakrishnan G, Windley P, editors. Formal Methods in Computer-Aided Design – FMCAD. LNCS 1522. Heidelberg: Springer-Verlag; 1998. .
16. Sawada J, Hunt WA Jr.
Verification of FM9801: An Out-of-order Microprocessor Model with Speculative Execution, Exceptions, and Program Modification Capability.
Formal Methods in Systems Design. 2002;20(2):187–222.
17. Kaufmann M, Manolios P, Moore JS.
Computer-Aided Reasoning: An Approach.
Boston, MA.: Kluwer Academic Press; 2000.
18. Kaufmann M, Manolios P, Moore JS, editors.
Computer-Aided Reasoning: ACL2 Case Studies.
Boston, MA.: Kluwer Academic Press; 2000.
19. Sumners R.
Correctness Proof of a BDD Manager in the Context of Satisfiability Checking.
In: Proceedings of ACL2 Workshop 2000. Department of Computer Sciences, Technical Report TR-00-29; 2000. http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/final/sumners2/paper.
20. Sawada J.
Formal Verification of Divide and Square Root Algorithms Using Series Calculation.
In: Proceedings of the ACL2 Workshop, 2002. Grenoble: http://www.cs.utexas.edu/users/moore/acl2/workshop-2002; 2002. .
21. Greve D, Wilding M.
Evaluatable, High-Assurance Microprocessors.
In: NSA High-Confidence Systems and Software Conference (HCSS). Linthicum, MD: http://hokiepokie.org/docs/hcss02/proceedings.pdf; 2002. .
22. Greve D, Wilding M. A Separation Kernel Formal Security Policy.
citeseer.ist.psu.edu/greve03separation.html; 2003.
23. Liu H, Moore JS.
Executable JVM Model for Analytical Reasoning: A Study.
In: Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME '03). San Diego, CA: ACM SIGPLAN; 2003. .
24. Liu H.
Formal Specification and Verification of a JVM and its Bytecode Verifier.
University of Texas at Austin; 2006.
25. Manolios P.
Mechanical Verification of Reactive Systems.
The University of Texas at Austin, Department of Computer Sciences; 2001.
26. Ershov AP.
On Programming of Arithmetic Operations.
Communications of the ACM. 1958 August;118(3):427–430.
27. Goto E.
Monocopy and Associative Algorithms in Extended Lisp.

University of Toyko; 1974. Technical Report TR-74-03.
28. Michie D.
Memo functions: a Language Feature with Rote Learning Properties.
Department of Artificial Intelligence, University of Edinburgh, Scotland; 1967. Technical Report MIP-R-29.
29. Michie D.
Memo Functions and Machine Learning.
Nature. 1968;218:19–22.
30. Hunt WA Jr.
The DE Language.
In: Kaufmann M, Manolios P, Moore JS, editors. Computer-Aided Reasoning: ACL2 Case Studies. Boston, MA.: Kluwer Academic Press; 2000. p. 151–166.
31. Swords S. A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover; 2010.
http://hdl.handle.net/2152/ETD-UT-2010-12-2210.
32. Slobodova A, Davis J, Swords S, Warren Hunt J.
A Flexible Formal Verification Framework for Industrial Scale Validation.
In: Singh S, editor. MEMOCODE: 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign. IEEE/ACM; 2011. p. 89–97.
33. Slobodova A.
Pragmatic Approach to Formal Verification.
In: SAT '15: Proceedings of Theory And Applications Of Satisfiabilioty Testing. Springer, LNCS 9340; 2015. p. IX–XI.
34. Davis J, Kaufmann M.
Industrial Strength Documentation for ACL2.
In: ACL2 '14. EPTCS; 2014. p. 9–25.
35. Intel®64 and IA32 Architecture Software Developer Manuals;.
Available from: http://www.intel.com/products/processor/manuals.
36. Davis J, Slobodova A, Swords S.
Microcode-Verification – Another Piece of the Microprocessor Verification Puzzle.
In: ITP '14: Proceediungs of Interactive Theorem Proving. Springer, LNCS 8558; 2014. p. 1–16.
37. Hunt WA Jr, Swords S.
Centaur Technology Media Unit Verification.
In: CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification. Berlin, Heidelberg: Springer, LNCS 5643; 2009. p. 353–367.
38. Hunt Jr W, Swords S, Davis J, Slobodova A.
Use of Formal Verification at Centaur Technology.
In: Hardin D, editor. Design and Verification of Microprocessor Systems for High-Assurance Applications. Springer; 2010. p. 65–88.
39. Hunt Jr W, Swords S.
A Mechanically Verified AIG-to-BDD Conversion Algorithm.
In: ITP '14: Proceedings of Interactive Theorem Proving. Springer, LNCS 6172; 2010. p. 435–449.
40. Hunt, Jr W.
Verifying VIA Nano Microprocessor Components.
In: Bloem R, Sharygina N, editors. FMCAD '10: Proceedings of the Formal Methods in Computer-Aided Design. ACM/IEEE; 2010. p. 3–10.
41. Moore JS.
Proof Pearl: Proving a Simple Von Neumann Machine Turing Complete.
In: Klein G, Gamboa R, editors. Proceedings of ITP 2014: 5th International Conference on Interactive Theorem Proving. vol. LNAI 8558. Springer-Verlag; 2014. p. 406–420.
42. Swords S, Davis J.
Bit-Blasting ACL2 Theorems.
In: ACL2 '11. vol. 70 of Electronic Proceedings in Theoretical Computer Science; 2011. p. 84–102.
43. Davis J, Swords S.
Verified AIG Algorithms in ACL2.
In: ACL2 '13. EPTCS; 2013. p. 95–110.
44. Wetzler ND.

Efficient, Mechanically-Verified Validation of Satisfiability Solvers.
The University of Texas at Austin, Department of Computer Sciences; 2015.
45. Davis J.
Embedding ACL2 Models in End-User Applications.
In: Proceedings of Do-Form '13. AISB; 2013. p. 49–56.
46. Kaufmann M, Rager D, editors.
Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications. vol. 192.
EPTCS; 2015.
47. Myreen MO.
Formal verification of machine-code programs.
University of Cambridge; 2009.
48. Brock B, Kaufmann M, Moore JS.
Rewriting with Equivalence Relations in ACL2.
Journal of Automated Reasoning. 2008;40(4):293–306.
49. Kaufmann M, Moore JS.
Rough Diamond: An Extension of Equivalence-Based Rewriting.
In: Klein G, Gamboa R, editors. Proceedings of ITP 2014: 5th International Conference on Interactive Theorem Proving. vol. LNAI 8558. Springer-Verlag; 2014. p. 537–542.
50. Hunt WA Jr, Krug R, Moore JS.
Linear and Non-Linear Arithmetic in ACL2.
In: Geist D, editor. Proceedings of CHARME 2003. vol. 2860 of Lecture Notes in Computer Science. Springer Verlag; 2003. p. 319–333.
51. Hunt WA Jr, Kaufmann M, Krug RB, Moore JS, Smith EW.
Meta Reasoning in ACL2.
In: Hurd J, Melham T, editors. 18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005. vol. 3603 of Lecture Notes in Computer Science. Springer; 2005. p. 163–178.
52. Kaufmann M, Moore JS, Ray S, Reeber E.
Integrating External Deduction Tools with ACL2.
Journal of Applied Logic. 2009 March;7(1):3–25.
53. Kaufmann M, Moore JS.
Proof Search Debugging Tools in ACL2.
In: A Festschrift in honour of Prof. Michael J. C. Gordon FRS. Royal Society, London; 2008. .
54. Boyer RS, Warren A Hunt J.
Function Memoization and Unique Object Representation for ACL2 Functions.
In: ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications. New York, NY, USA: ACM; 2006. p. 81–89.
55. Kaufmann M.
ACL2 Support for Verification Projects.
In: Kirchner C, Kirchner H, editors. Proceedings 15th International Conference on Automated Deduction. Heidelberg: LNAI 1421, Springer-Verlag; 1998. p. 220–238.
56. Goel S, Hunt WA, Kaufmann M.
Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls.
In: Claessen K, Kuncak V, editors. FMCAD'14: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design. EPFL, Switzerland; 2014. p. 91–98.
57. Kaufmann M.
Abbreviated Output for Input in ACL2: An Implementation Case Study.
In: Proceedings of ACL2 Workshop 2009; 2009. http://www.cs.utexas.edu/users/sandip/acl2-09.
58. Boyer RS, Moore JS.
Single-Threaded Objects in ACL2.
In: PADL 2002. Heidelberg: Springer-Verlag LNCS 2257; 2002. p. 9–27.
http://www.cs.utexas.edu/users/moore/publications/stobj/main.ps.gz.