Integrating External Deduction Tools with ACL2 *

Matt Kaufmann, J Strother Moore, Sandip Ray, Erik Reeber
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
{kaufmann,moore,sandip,reeber}@cs.utexas.edu
http://www.cs.utexas.edu/users/{kaufmann,moore,sandip,reeber}

Abstract

We present an interface connecting the ACL2 theorem prover with external deduction tools. The logic of ACL2 contains several constructs intended to facilitate structuring of interactive proof development, which complicates the design of such an interface. We discuss some of these complexities and develop a precise specification of the requirements from external tools for sound connection with ACL2. We also develop constructs within ACL2 to enable the developers of external tools to satisfy our specifications.

1 Introduction

Recent years have seen rapid advancement in the capacity of automatic reasoning tools, most notably for decidable theories such as propositional calculus and Presburger arithmetic. For instance, modern BDD packages and Boolean satisfiability solvers can automatically solve problems with tens of thousands of variables and have been successfully used to reason about commercial hardware system implementations. This advancement has sparked significant interest in the general-purpose, interactive theorem proving community to improve the efficiency and automation in theorem provers by providing a connection with state-of-the-art automatic reasoning tools. In this paper, we present a general mechanism we are building to connect deduction tools, external to the ACL2 theorem prover, with that prover.

ACL2 [KMM00, KM06] is an industrial-strength interactive theorem proving system. It consists of an efficient programming interface based on an applicative subset of Common Lisp [KM94], and a first-order, inductive theorem prover for a logic of recursive functions. The ACL2 theorem prover supports several deduction mechanisms such as congruence-based conditional rewriting, well-founded induction, several decision procedures, and generalization. The theorem prover has been particularly successful in the verification of microprocessors and hardware designs such as

---

the floating point multiplication, division, and square root algorithms of AMD processors [MLK98, Rus98, RF00, FKR$^+$02], microcode for the Motorola CAP DSP [BH97], separation properties for the Rockwell Collins AAMP7$^{\text{TM}}$ processor [GRW04], and a non-trivial pipelined machine with interrupts, exceptions, and speculative instruction execution [SH97]. However, the applicability of ACL2 (as in fact that of any theorem prover) is often limited by the amount of user expertise required to drive the theorem prover; indeed, the verification projects referenced above represent many man-years of effort. Yet, a significant number of lemmas proven in the process, in particular many proofs exhibiting invariance of predicates over executions of hardware design implementations, can be expressed in a decidable theory and automatically dispatched by an automatic decision procedure for the theory.

On the other hand, it is not trivial to connect ACL2 with an external deduction tool. The logic of ACL2 is complicated by the presence of several constructs intended to facilitate effective proof structuring [KM01]. It is therefore imperative (i) to determine under what logical constraints a conjecture certified by a combination of the theorem prover and other tools can be claimed to be a valid theorem, and (ii) to provide mechanisms so that a tool implementor might be able to meet the logical constraints so determined.

In this paper, we propose a general interface for connecting external tools with ACL2. The user can instruct ACL2 to use external deduction tools for reducing a goal formula C to a list of formulas L_C during a proof attempt. The claim is that provability of each formula in L_C implies the provability of C. We present a sufficient condition expressible in ACL2 guaranteeing this claim, and discuss the soundness requirements on the tool implementor. We also propose a modest augmentation of the logical guarantees provided by ACL2, in order to facilitate connection with certain types of tools (cf. Section 5).

We distinguish between two classes of external tools, namely (i) tools verified by the ACL2 theorem prover, and (ii) unverified but trusted tools. A verified tool must be formalized in the logic of ACL2 and the sufficient condition alluded to above must be formally established by the theorem prover. An unverified tool can be defined using the ACL2 programming interface, and can invoke arbitrary executable programs using calls to the underlying operating system via a system call interface. An unverified tool is introduced with a "tag" acknowledging that the validity of the formulas proven using the tool depends on the correctness of the tool.

The interface for unverified tools enables us to invoke Boolean Satisfiability solvers, BDD packages, etc., for simplifying ACL2 subgoals. Why might *verified* tools be of interest? The formal language of ACL2 is a programming language, based on an applicative subset of Common Lisp. The close connection with Lisp makes it possible to write efficiently executable programs in the ACL2 logic [KM94]. In fact, most of the ACL2 source code is implemented in this language. We believe it will be handy to provide facilities to the ACL2 user to control proofs by (i) implementing customized domain-specific reasoning code, (ii) verifying with ACL2 that the code is sound, and (iii) invoking the code for proving theorems in the target domain. In fact, ACL2 currently provides a way for users to augment its built-in term simplifier with their own customized reasoning code, via the so-called "meta rules" [BM81]. However, such rules essentially augment the reasoning engine of ACL2 without providing the user control to manipulate a specific subgoal arising during a proof. Furthermore, meta rules only allow reducing a term to one that is provably equivalent—they do not allow generalization.

With our interface, an ACL2 user can invoke directly a customized, verified reasoning tool to replace a subgoal by a collection of possibly more general subgoals.

The remainder of the paper is organized as follows. In Section 2 we provide a brief overview of the ACL2 system. In Sections 3 through 5 we present our interface for connecting verified and unverified external tools with ACL2, touching upon the logical underpinnings involved. We discuss related work in Section 6 and conclude in Section 7. No previous familiarity with ACL2 is assumed in this presentation; the relevant features of the logic and the theorem prover are discussed in Section 2.

2 ACL2

The name "ACL2" stands for "A Computational Logic for Applicative Common Lisp". The name is used to denote (i) a programming language based on an applicative subset of Common Lisp, (ii) a first-order logic of recursive functions with induction, and (iii) a theorem prover for the logic. In this section, we provide a brief overview of ACL2. The review is not complete, but only intended to lay the foundation for our work. Readers interested in learning ACL2 are referred to the ACL2 Home Page [KM06] which contains extensive hypertext documentation together with references to several books and papers.

2.1 The logic

The kernel of the ACL2 logic consists of a formal syntax, some rules of inference, and some axioms. Kaufmann and Moore [KM97] provide a precise description of the kernel logic. The logic supported by the theorem prover is an extension of the kernel logic.

The kernel syntax describes terms composed of variables, constants, and function symbols applied to a fixed number of argument terms. The kernel logic introduces "formulas" as composed of equalities between terms and the usual propositional connectives.

The syntax of ACL2 is the prefix-normal syntax of Lisp; thus, the application of a binary function f on arguments a and b is represented by (f a b) rather than the more traditional $f(a, b)$. However, in this paper we will use the formal syntax only when it is relevant for the associated discussion. In particular we will write $(x \times y)$ instead of (* x y) and (*if* x *then* y *else* z) instead of (if x y z).

The axioms of ACL2 describe the properties of certain Common Lisp primitives. For example, the following are axioms about the primitives *equal* and *if*:

Axioms.
$$x = y \quad \Rightarrow \textit{equal}(x, y) = \ \texttt{T}$$
$$x \neq y \quad \Rightarrow \textit{equal}(x, y) = \ \texttt{NIL}$$
$$x = \texttt{NIL} \Rightarrow (\textit{if } x \textit{ then } y \textit{ else } z) \ = \ z$$
$$x \neq \texttt{NIL} \Rightarrow (\textit{if } x \textit{ then } y \textit{ else } z) \ = \ y$$

Notice that the kernel syntax is quantifier-free and each formula is implicitly universally quantified over all the free variables in the formula. Furthermore, the use of function symbols *equal* and *if* make it possible to embed propositional calculus and equality into the term language. When we write a term τ in place of a formula, it stands for the formula $\tau \neq \texttt{NIL}$. Thus, in ACL2, the following term is an axiom relating the Lisp functions *cons*, *car*, and *equal*.

Axiom.
$equal(cons(car(x, y)), x)$

This axiom stands for the formula $equal(car(cons(x, y)), x) \neq$ NIL, which is provably equivalent to $car(cons(x, y)) = x$. With this convention, we will feel free to interchange terms and formulas. We will similarly feel free to apply logical connectives to a term or formula. Thus when we write $\neg\tau$, where τ is a term, we mean the term (or formula by the above convention) obtained by applying the function symbol *not* to τ, where *not* is axiomatized as:

Axiom.
$not(x) = $ *if* x *then* NIL *else* T

The convention above enables us to interpret an ACL2 theorem as follows. If the term τ (when interpreted as a formula) is a theorem then for all substitutions σ of free variables in τ to objects in the ACL2 universe the (ground) term τ/σ evaluates to a non-NIL value. This alternative view will be critical in deriving sufficient conditions for correctness of external tools integrated with ACL2.

The kernel logic includes axioms that characterize the primitive Lisp functions over numbers, characters, strings, symbols, and ordered pairs. These objects together make up the ACL2 standard universe; but "non-standard" ACL2 universes may contain other objects. Lists are represented as ordered pairs, so that the list (1 2 3) is represented as $cons(1, cons(2, cons(3, $ NIL$)))$. For brevity, we will write $list(x, y, z)$ as an abbreviation for $cons(x, cons(y, cons(z, $ NIL$)))$. Another convenient data structure built out of ordered pairs is the *association list* (or *alist*) which is essentially a list of pairs, *e.g.*, $list(cons($"a"$, 1), cons($"b"$, 2))$. We often use alists for describing finite mappings; the above alist can be thought as a mapping that associates the strings "a" and "b" with 1 and 2, respectively.

In addition to propositional calculus and equality the rules of inference of ACL2 include instantiation, together with first-order induction over ϵ_0 (see below). For instance, the formula $car(cons(2, x)) = 2$ is provable by instantiation from the above axiom relating *car*, *cons*, and *equal*.

The ACL2 theorem prover initializes with a boot-strapping (first-order) theory called the *Ground Zero* theory (GZ for short). In the sequel, whenever we mention an ACL2 theory, we mean a theory obtained by extending GZ via the *extension principles* explained below. The theory GZ contains the axioms of the kernel logic. In addition, it also contains a well-founded first-order induction principle, by way of an embedding of ordinals below ϵ_0. In particular, GZ is assumed to be *inductively complete*, that is, it is assumed implicitly to contain all the first-order well-founded induction axioms expressible using formulas ϕ in the language of GZ:

$$(\forall y < \epsilon_0)[((\forall x < y)\phi/\{y := x\}) \Rightarrow \phi(y)] \Rightarrow (\forall y < \epsilon_0)\phi(y)$$

2.1.1 Extension Principles

ACL2 also provides several *extension principles* that allow the user to extend a theory by introducing new function symbols and axioms about them. Two extension principles that are particularly relevant to us are (i) the *definitional principle* to introduce total

functions, and (ii) the *encapsulation principle* to introduce constrained functions,[1] and we discuss them in some detail. Note that whenever we say (below) that a theory is extended by axiomatizing new function symbols we implicitly assume that the resulting theory is also inductively complete, that is, all the induction axioms in the language of the extended theory are also introduced together with the axioms explicitly specified.

Definitional Principle:

The *definitional principle* allows the user to extend a theory by axiomatizing new total (recursive) functions. For example, one can use this principle to introduce the unary function symbol *fact* axiomatized as follows, which returns the factorial of its argument.

Definitional Axiom.
$fact(n)$ = if $natp(n) \wedge (n > 0)$ then $n \times fact(n-1)$ else 1

Here, $natp(n)$ is axiomatized in GZ to return T if n is a natural number, and NIL otherwise. To ensure that the extended theory is consistent, ACL2 first proves that the recursion terminates. This is achieved by exhibiting some *measure m* that maps the set of function arguments to some well-founded structure derived from the embedding of ordinals below ϵ_0. For the axiom above, an appropriate measure is $nfix(n)$ which is axiomatized in GZ to return n if n is a natural number, otherwise 0.

Encapsulation Principle:

The *encapsulation principle* allows the extension of the ACL2 logic with functions introduced with constraints rather than full definitions. This principle, for instance, allows us to extend a theory by introducing a new unary function *foo* with only the following axiom that merely posits that *foo* always returns a natural number:

Encapsulation Axiom.
$natp(foo(\texttt{x}))$

The encapsulation axioms are also referred to as *constraints*, and the functions introduced via this principle are called *constrained functions*. To ensure the consistency of the resulting theory, one must show that there exist (total) functions satisfying the alleged constraints; such functions are called *witnesses* to the constraints. For *foo* above, an appropriate witness is the constant function that always returns 1.

For a constrained function f the only axioms known are the constraints. Therefore, any theorem proved about f is also valid for a function f' that also satisfies the constraints. More precisely, call the conjunction of the constraints on f the formula ϕ. For any formula θ, let θ' be the formula obtained by replacing the function symbol f by the function symbol f'. Then a derived rule of inference, *functional instantiation*, specifies that if ϕ' and ψ are theorems then ψ' is also a theorem. Consider, for example, the constant function of one argument that returns 10. This function satisfies the constraint for *foo*; thus if $bar(foo(x))$ is provable for some function *bar* then functional instantiation can be used to prove $bar(10)$. .

---

[1] Other extension principles include the introduction of Skolem (choice) functions and specification of a formula as an axiom. The latter is discouraged since one can introduce unsoundness by adding arbitrary axioms. For this paper, we will ignore the possibility of introducing arbitrary axioms.

2.2 The Theorem Prover

As a theorem prover, ACL2 is an automated, interactive proof assistant. It is *automated* in the sense that no user input is expected once the theorem prover has embarked on the search for the proof of a conjecture. It is *interactive* in the sense that the proof search is largely determined by the previously proven lemmas in its database at the beginning of a proof attempt; the user essentially programs the theorem prover by stating lemmas for it to prove, to use automatically in subsequent proofs. There is also a goal-directed interactive loop (called the "*proof-checker*"), similar in nature to what is offered by LCF-style provers; but it is much less frequently used and not relevant to the discussion below.

Interacting with the ACL2 theorem prover principally proceeds as follows. The user creates a relevant theory (extending GZ) using the extension principles to model some artifact of interest. Then she poses some conjecture about the functions in the theory and instructs the theorem prover to prove the conjecture, possibly providing hints on how to proceed in the proof search. For instance, if the artifact is the factorial function above, an appropriate conjecture might be the following formula, which says that *fact* always returns a natural number.

 Theorem `fact-is-natp`:
  $natp(fact(\text{x})) = \text{T}$

The theorem prover attempts to prove such a conjecture by applying a sequence of transformations to it, replacing each goal (initially, the conjecture) with a list of subgoals. ACL2 provides a *hint* mechanism that enables the user to instruct the theorem prover on how to proceed with its proof search at any goal or subgoal. For instance, the user can instruct the theorem prover to begin its proof search by inducting on `x`.

Once a theorem is proven, the theorem prover stores it in a database, for use in subsequent derivations. This database groups theorems into various *rule classes*, which affects how the theorem prover will automatically apply them. The default rule class is `rewrite`, which causes the theorem prover to replace instances of the left-hand-side of an equality with its corresponding right-hand-side. If the conjecture `fact-is-natp` above is a rewrite rule, then subsequently whenever ACL2 encounters a term of the form $natp(fact(\tau))$ in the course of a proof attempt, it rewrites the term to `T`.

ACL2 users interact with the theorem prover primarily by issuing a sequence of *event* commands for introducing new functions and proving theorems with appropriate rule classes. For example, `fact-is-natp` is the name of the above theorem event. During proof development the user typically records events in a file, often referred to as a *book*. Once the desired theorems have been proven, the user instructs ACL2 to *certify* such a book in order to facilitate the use of the events in other projects. A book can be certified once and then *included* during a subsequent ACL2 session without rerunning the associated proofs. To facilitate structured proof development, the user is permitted to mark some of the events in a book as *local events*. For instance, to prove some relevant theorem the user might introduce several auxiliary functions and intermediate lemmas that are not generally useful; such events are typically marked to be local. When a book is included in a subsequent proof project, only the non-local events in the book are accessible, thus preventing unwanted clutter in the database of the theorem prover.

The presence of local events complicates the soundness claims for ACL2. Note from above that local events in a book might include commands for introducing new functions (thus extending an ACL2 theory with new axioms), which are not available in the subsequent sessions where the book is loaded. Yet, in order to prove some non-local theorem in the book ACL2 might have used some of these local axioms. One must therefore answer under what condition it is legitimate to mark an axiomatic event in a book as local, and what formal soundness claims can be provided for an ACL2 session in which such a pre-certified book is loaded. Such questions have been answered by Kaufmann and Moore [KM01]: if a formula ϕ is proven as a theorem in an ACL2 session, then ϕ is in fact first-order derivable (with induction) from the axioms of GZ together with (hereditarily) only the axiomatic events in the session that involve the function symbols in ϕ. (In particular, every ACL2 session corresponds to a theory that is a *conservative* extension of GZ.) Thus, any definition or theorem that does not involve the function symbols in the non-local events of a book can be marked local. To implement this requirement, book certification involves two passes. In the first pass, ACL2 proves each theorem (and admits each axiomatic event) sequentially. In the second pass, it skips proofs, and makes a so-called *local incompatibility* check, checking primarily that each axiomatic event involved in any non-local theorem in the book is also non-local.

2.3 The ACL2 Programming Environment

ACL2 is closely tied with Common Lisp. The formal syntax of the logic is essentially the syntax of Lisp, and the axioms in GZ for the primitive Lisp functions are carefully crafted so that the return value of a function as predicted by the axioms matches with the value specified in the Common Lisp Manual on arguments in the intended domain of its application. Furthermore, events corresponding to functions introduced using the definitional principle are essentially Lisp definitions. For instance, consider the factorial function *fact* described above. The formal event introducing the definitional axiom of *fact* is written in ACL2 as follows.

```
(defun fact (n) (if (and (natp n) (> n 0)) (* n (fact (- n 1))) 1))
```

This is essentially a Lisp definition of the function! The connection with Common Lisp enables the users of ACL2 to execute formal definitions by using the underlying Lisp evaluator. Since Lisp is an ANSI-standard, efficient functional programming language, ACL2 users often make use of the connection to implement formally defined yet efficient code. Indeed, the theorem prover itself makes use of this connection for simplifying ground terms during proof search; for instance, ACL2 will simplify *fact*(3) to 6 by evaluation in the underlying Lisp.

In order to facilitate efficient code development, ACL2 also provides a logic-free programming environment. A user can implement any applicative Lisp function and mark it to be in *program mode*. No proof obligation is generated for such functions. ACL2 can evaluate such functions using the Lisp evaluator, although no logical guarantee (including termination) is provided. Furthermore, ACL2 provides an interface to the underlying operating system, which enables the user to invoke arbitrary executable code (and operating system commands) from inside an ACL2 session.

2.4 Evaluators

ACL2 provides a convenient notation for defining an evaluator for a fixed set of functions. Evaluators are used to support *meta reasoning* [BM81]. We will not consider meta reasoning in this paper, but we briefly mention evaluators since they will be useful in characterizing the correctness of external tools.

A proof search involves applying transformations to reduce a goal to a collection of subgoals. Internally, ACL2 stores each goal as a *clause* represented as an object in the ACL2 universe. For instance, when ACL2 attempts to prove a theorem of the form $\tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_n \Rightarrow \tau$, it represents the proof goal internally as a list of terms, $(\neg\tau_1 \ \ldots \ \neg\tau_n \ \tau)$, which can be thought of as the disjunction of its elements (*literals*). When ACL2 works on any subgoal, the transformation procedures work on the internal representation of the subgoal, called the *current clause*. Since this representation is an ACL2 object, we can define functions over such objects.

An *evaluator* makes explicit the connection between terms and their internal representations. Assume that $f_1, \ldots, f_n$ are functions axiomatized in some ACL2 theory $\mathcal{T}$. A function ev, also axiomatized in $\mathcal{T}$ is called an *evaluator* for $f_1, \ldots, f_n$, if the axioms associated with ev can be viewed as specifying an evaluation semantics for the internal representation of terms composed of $f_1, \ldots, f_n$ that is consistent with the definitions of these functions; such axioms are then referred to as *evaluator axioms*. A precise characterization of all the evaluator axioms is described in the ACL2 Manual [KM06] under the documentation topic <u>defevaluator</u>; here we only mention one for illustration, which corresponds to the evaluation of the m-ary function symbol f_i:

An Evaluator Axiom.
$$ev(list('\mathtt{f_i},'\tau_1,...,'\tau_m),a) = f_i(ev('\tau_1,a),\ldots,ev('\tau_m,a))$$

Here 'f_i is assumed to be the internal representation of f_i and 'τ_j is the internal representation of τ_j, for $1 \leq j \leq m$. It is convenient to think of a as an association list that maps the (internal representation of the) free variables in $\tau_1, \ldots, \tau_m$ to ACL2 objects. Then the axiom specifies that the evaluation of the list $('f_i \ '\tau_1 \ \ldots \ '\tau_m)$ (which corresponds to the internal representation of $f_i(\tau_1, \ldots, \tau_m)$) under some mapping of free variables to objects is the same as the function f_i applied to the evaluation of each τ_j under the same mapping.

3 Verified External Tools

In this section, we discuss *verified* external tools. We consider verified tools first since they are amenable to perhaps a simpler understanding than unverified ones. The ideas and infrastructure we develop in this section will be extended successively in the next two sections to support connections with unverified tools.

We will refer to external deduction tools as *clause processors*. Recall that ACL2 internally represents terms as clauses, so that a subgoal of the form $\tau_0 \wedge \tau_1 \wedge \ldots \wedge \tau_n \Rightarrow \tau$ is represented as a disjunction by the list $(\neg\tau_0 \ \neg\tau_1 \ \ldots \ \neg\tau_n \ \tau)$. Our interface enables the user to transform the current clause with custom code. More precisely, a *clause processor* is a function that takes a clause C (together with possibly other arguments)

$disjoin(C)$	$=$	$if\ \neg consp(C)\ then\ *NIL*\ else\ list(\texttt{if}, car(C), *T*, disjoin(cdr(C)))$
$conjoin(L_C)$	$=$	$if\ \neg consp(L_C)\ then\ *T*\ else\ list(\texttt{if}, disjoin(car(L_C)), conjoin(cdr(L_C)), *NIL*)$

Figure 1: Axioms to support clause processors in GZ. Here $\texttt{*T*}$ and $\texttt{*NIL*}$ are assumed to be the internal representation of $\texttt{T}$ and $\texttt{NIL}$ respectively. The predicate *consp* is defined in GZ such that *consp*(x) returns $\texttt{T}$ if x is an ordered pair, and $\texttt{NIL}$ otherwise.

and returns a list of clauses L_C.[2] The intention is that if each clause in L_C is a theorem of the current ACL2 theory then so is C. In the remainder of the paper, when we talk about clause processors, we will mean such clause manipulation functions.

Our interface for verified external tools constitutes the following components.

- *A new rule class for installing clause processors.* Suppose the user has defined a function *tool0* that she desires to use as a clause processor. She can then prove a specific theorem about *tool0* (described below) and attach this rule class to the theorem. The effect is to install *tool0* in the ACL2 database as a clause processor for use in subsequent proof attempts.

- *A new hint for using clause processors.* Once *tool0* has been installed as a clause processor it can be invoked via this hint to transform a conjecture during a subsequent proof attempt. If the user instructs ACL2 to use *tool0* to help prove some goal G, then ACL2 transforms G into the collection of subgoals generated by executing *tool0* on (the clause representation of) G.

We now explain the theorem alluded to above for installing a function *tool0* as a clause processor. Recall that one way to interpret a formula proven by ACL2 is via an evaluation semantics; that is, a formula Φ is a theorem if, for every substitution σ mapping each free variable of Φ to some object, Φ/σ does not evaluate to $\texttt{NIL}$. Our formal proof obligation for installing functions as clause processors is based on this evaluation semantics. Let C be a clause whose disjunction is the term τ, and let *tool0*, with C as its argument, produce the list $(C_1\ \ldots\ C_n)$ whose respective disjunctions are the terms $\tau_1, \ldots, \tau_n$. Informally, we want to ensure that if τ/σ evaluates to $\texttt{NIL}$ for some substitution σ then there is some σ' and i such that τ_i/σ' also evaluates to $\texttt{NIL}$. This condition can be made precise in the logic of ACL2 by extending the notion of evaluators discussed in Section 2.4 from terms to clauses. Before describing the extension, we will assume that the ACL2 ground zero theory GZ contains two functions *disjoin* and *conjoin* axiomatized as shown in Figure 1. Informally, the axioms specify how to interpret objects representing clauses and clause lists. For instance, the function *disjoin* specifies that the interpretation of a clause $(\tau_0\ \tau_1\ \tau_2)$ is the same as the interpretation of $(\texttt{if}\ \tau_0\ \texttt{T}\ (\texttt{if}\ \tau_1\ \texttt{T}\ (\texttt{if}\ \tau_2\ \texttt{T}\ \texttt{NIL}))))$, which represents the disjunctions of the terms τ_0, τ_1, and τ_2.

Based on these axioms, we can formalize the correctness of clause processors by defining an evaluation semantics for clauses. In particular, assume that *ev* is an evaluator for the single function *if*. Thus *ev*$(list(\texttt{if}, \tau_0, \tau_1, \tau_2), \texttt{a})$ stipulates how the term "*if* τ_0 *then* τ_1 *else* τ_2" can be evaluated. For any theory $\mathcal{T}$ (obtained by extending GZ

---

[2]The formal definition of a clause processor is somewhat more complex. In particular, it can optionally take as argument the current ACL2 state among others, and return, in addition to the list of clauses, an error message and possibly a new ACL2 state. We will ignore such details in this paper.

$$\Rightarrow \quad \begin{array}{l} \textit{term-listp}(C) \wedge \textit{alistp}(a) \wedge (\textit{ev}(\textit{disjoin}(C), a) = *\texttt{NIL}*) \\[6pt] \textit{ev}(\textit{conjoin}(\textit{tool0}(\texttt{args}, C)), \textit{tool0-env}(\texttt{args}, C, a)) = *\texttt{NIL}* \end{array}$$

Figure 2: Correctness condition for clause processors. Here *ev* is assumed to be an evaluator for *if*, and `args` represents the remaining arguments of *tool0* (in addition to clause C). The predicates *term-listp* and *alistp* are axiomatized in GZ such that (i) *term-listp*(x) returns a Boolean, which is T if and only if x is an object in the ACL2 universe representing a well-formed list of terms (and hence a clause), and (ii) *alistp*(a) returns a Boolean, which is T if and only if a is a well-formed association list.

via the extension principles), a clause processor function *tool0*$(\texttt{args}, C)$ will be said to be *legal* in $\mathcal{T}$ if there exists a function *tool0-env* in $\mathcal{T}$ such that the formula shown in Figure 2 is a theorem. The function *tool0-env* returns an association list like σ' in our informal example above: it potentially modifies the original association list to respect any generalization being performed by *tool0*. Note that a weaker theorem would logically suffice, replacing the use of the association list *tool0-env*$(\texttt{args}, c, a)$ by an existentially quantified variable.

A theorem of the form shown in Figure 2 can be tagged with the new rule class for clause processors, instructing ACL2 to use the function *tool0* as a new verified external tool. Theorem 1 below, based on the "Essay on Correctness of Meta Reasoning" comment in the ACL2 sources, guarantees that the above condition is sufficient for the soundness of using *tool0* to transform goal conjectures.

Theorem 1 *Let $\mathcal{T}$ be an ACL2 theory for which *tool0* is a legal clause processor, and let *tool0* return a list L_C of clauses given an input clause C. If each clause in L_C is provable in $\mathcal{T}$, then $\mathcal{C}$ is also provable in $\mathcal{T}$.*

Proof: The theorem is a simple consequence of the following lemma, given the correctness condition shown in Figure 2. ∎

Lemma 1 *Let τ be a term with free variables $v_0, \ldots, v_n$, *ev* an evaluator for the function symbols in τ, and e a list of *cons* pairs of the form $(\langle \texttt{'v0}, \texttt{'}\tau_0 \rangle \ldots \langle \texttt{'vn}, \texttt{'}\tau_n \rangle)$, where $\texttt{'}v_i$ and $\texttt{'}\tau_i$ are internal representation of v_i and τ_i respectively. Let σ be a substitution mapping each v_i to τ_i, and let $\texttt{'}\tau$ be the internal representation of the term τ. Then the following formula is a theorem: $\textit{ev}(\texttt{'}\tau, e) = \tau/\sigma$.*

Proof: An easy induction on the structure of term τ. ∎

The simplicity of the above proof might belie some of the subtleties involved. For instance, recall that each ACL2 theory $\mathcal{T}$ is a conservative extension of GZ. Furthermore, note that theorems whose proofs use an invocation of *tool0* often do not involve the function symbols occurring in the definition of the function *tool0* itself. For instance, assume that *tool0* is a simple clause generalizer that replaces each occurrence of a specific subterm in a clause by a free variable not present in the original clause. Such a function can be invoked for generalization in the proof of a formula Φ although Φ might not

contain any occurrence of *tool0*. On the completion of a successful proof of Φ, can we then mark *tool0* as local? The answer is in general "no", since Theorem 1 only guarantees provability of the clause input to the clause processor from those returned *in the theory in which the clause processor is legal*. In particular such a theory must contain the definitions of the function symbols being manipulated by *tool0*, and for this it suffices that *tool0* not be marked local. In fact a soundness bug in a previous but very recent release of ACL2 occurred in an analogous context for meta rules, due to ACL2's previous inability to track the fact that the theory in which such rules are applied indeed included the definitions supporting the corresponding evaluators.

4 Basic Unverified External Tools

Verified clause processors are useful when the user intends to augment the reasoning engine of ACL2 with mechanically checked code for customized clause manipulation. However, more often, we want to manipulate goal conjectures using a tool that is external to the theorem prover, for instance a state-of-the-art Boolean satisfiability solver or model checker. In this section, we will consider an extension of the mechanisms to incorporate such tools. In the next section we will present additional constructs to facilitate integration with more general tools.

Our interface for unverified tools involves extending the theorem prover with a new event that enables ACL2 to recognize some function *tool1* defined by the user as an *unverified* clause processor. Here the function *tool1* might be implemented using *program mode* and might also invoke arbitrary executable code using ACL2's system call interface (cf. Section 2.3). The effect of the event in subsequent proof search with ACL2 is the same as if *tool1* were introduced as a verified clause processor: hints can be used to invoke the function for manipulating terms arising during proofs.

Suppose an unverified tool *tool1* simplifies a clause in the course of proving some goal conjecture. What guarantees should an implementor of *tool1* provide (and must the user trust) in order to claim that the goal conjecture is indeed a theorem? In this simple case, a sufficient guarantee is that there is a theory $\mathcal{T}$ containing the definition of *tool1* and appropriate evaluators such that the formula analogous to the one shown in Figure 2 in the previous section for *tool1* is a theorem of $\mathcal{T}$. The soundness of the use of *tool1* then follows from Theorem 1.

Since the invocation of an unverified tool for simplifying ACL2 conjectures carries a logical burden, the event introducing such tools provides two constructs, namely (i) a *tag* for the user of the tool to acknowledge this burden, and (ii) a concept of *supporters* for the tool developer to implement the tool in a way as to be able to guarantee that the logical restrictions are met. We now explain these two constructs.

The tag associated with an event installing an unverified tool *tool1* is a symbol (the default value being the name of the tool itself), which must be used to acknowledge that the soundness of any theorem proven by an application of *tool1* depends on the implementor of *tool1* satisfying the logical guarantees above. The certification of any book that contains an event installing an unverified clause processor (or hereditarily includes such a book, even locally) requires the user to tag the certification command with the name of the tags introduced with the event. Note that technically the mere act

of installing an unverified tool does not introduce any unsoundness; the logical burden expressed above pertains to the *use* of the tool. Nevertheless, our decision to insist that the certification of any book with an *installation* of an unverified tool (whether subsequently used or not) to be tagged is governed by implementation convenience. Recall that the local incompatibility check (that is, the second pass of a book certification) skips proofs, and thereby ignores the hints provided during the proof process. By "tracking" the installation rather than the application of an unverified clause processor, we disallow the possibility of a user certifying a book that *locally* introduces an unverified tool and uses it for simplifying some formulas, without acknowledging the application of the tool.

Finally we turn to *supporters*. This construct enables a tool developer to provide the guarantee outlined above in the presence of local events. To understand why this construct is necessary, consider the following scenario. Suppose a developer creates a book (say, `book1`) in which the function f is introduced *locally* with the following definitional axiom:

Local Definitional Axiom.
   $f(x) = x$

Suppose further that `book1` also installs an unverified clause processor *tool1*. Assume that the definition of *tool1* does not involve invocation of f, but it replaces terms of the form $f(\tau)$ with τ; thus the correctness of *tool1* depends on the intended definition of f. However, if an ACL2 session is extended by including `book1`, then the extended session contains the definition of *tool0* tagged as an unverified clause processor, but does not contain the (local) definition of f. Thus we can write another book (say, `book2`) that includes `book1` and then provides a new definition of f, for instance the following:

Definitional Axiom.
   $f(x) = cons(x, x)$

We now are working in a theory in which *tool1* may be used to perform term manipulations that are completely unjustified by the current definition of f, thus invalidating any guarantee provided by the implementor of *tool1*.

In general, then, suppose that a tool has built-in knowledge about some function symbols. The tool implementor cannot meet the logical burden expressed above unless the user of the tool is required to include the axioms that have been introduced for those function symbols. The *supporters* construct of the event installing unverified clause processors provides a way for the implementor to insist that such axioms are present, by listing the names of axiomatic events (typically function symbols that name their definitions, e.g., f in the example above). We will refer to these events as the *supporting events* for the clause processor. Whenever ACL2 encounters an event installing a function *tool1* as an unverified clause processor with a non-empty list of supporters, it will check that *tool1* and all of the supporting event names are already defined.

5   Templates and Generalized External Tools

The view above of unverified tools is that if a clause processor replaces some clause with a list of clauses then the provability of the resulting clauses implies the provability of the original clause. A clause processor is thus an efficient procedure for assisting in proofs

of theorems that could, in principle, have been proven from the axiomatic events of the current theory. This simple view is sufficient in most situations; for instance, one can use it to connect ACL2 with a Boolean satisfiability solver that checks if a propositional formula is a tautology. However, some ACL2 users have found the necessity to use more sophisticated tools that implement their own theory. We will now discuss an extension to the ACL2 logic that facilitates connection with such tools.

To motivate the need for such tools, assume that we wish to prove a theorem about some hardware design. Most such designs are written in a Hardware Description Language (HDL) such as VHDL or Verilog. One way of formalizing such designs is to define a semantics of the HDL in ACL2, possibly by defining a formal interpreter for the language. However, defining such an interpreter is typically extremely complex and labor-intensive. On the other hand, there are several model checkers available which can parse designs written in VHDL or Verilog. An alternative is merely to *constrain* some properties of the interpreter and use a combination of theorem proving and model checking in the following manner:

- Establish low-level properties of parts of a design using model checkers or other decision procedures.

- Use the theorem prover to compose the properties proven by the model checker together with the constrained properties of the interpreter to establish the correctness of the design.

The above approach has shown promise in scaling formal verification to industrial designs. For instance, Sawada and Reeber [SR06] have recently verified an industrial VHDL floating-point multiplier using a combination of ACL2 and an IBM internal verification tool called SixthSense [MBP+04]. They introduce two functions, *sigbit* and *sigvec*, with the following assumed semantics:

- *sigbit*(e, s, n, p) returns a bit corresponding to the value of bit signal s of a VHDL design e at cycle n and phase p.

- *sigvec*(e, s, l, h, n, p) returns a bit vector corresponding to the bit-range between l and h of s for design e at cycle n and phase p.

In ACL2 these two functions are constrained only to return a bit and bit-vector respectively. The key properties of the different multiplier stages are proven using SixthSense. For instance, one of the properties proven is that *sigvec* when applied to (i) a constant C representing the multiplier design, (ii) a specific signal s of the design, (iii) two specific values lb and hb corresponding to the bit-width of s, and (iv) a specific cycle and phase, returns the sum of two other bit vectors at the previous cycle; this corresponds to one stage of the Wallace-tree decomposition implemented by the multiplier. All such theorems are then composed by ACL2 to verify that the multiplier, when provided two vectors of the right size, produces their product after 5 cycles.

How do we support this verification approach? Note that the property above is *not* provable from the constraints on the associated functions alone (namely *sigvec* returns a bit vector). Thus if we use encapsulation to constrain *sigvec* and posit the property as a theorem then functional instantiation can derive an inconsistency. The problem

is that the property is provable from the constraints *together with* axioms about *sigvec* that are unknown to ACL2 but assumed to be accessible to SixthSense.

Our solution to the above is to extend the extension principles of ACL2 with a new principle called *encapsulation templates* (or simply *templates*). Function symbols introduced via templates are constrained functions just like those introduced via the encapsulation principle, and the soundness of extending an ACL2 theory is analogously guaranteed by exhibiting a local witness satisfying the constraints. However, there is one significant distinction between encapsulation principle and templates: the constraints introduced are marked *incomplete*, acknowledging that they might not encompass all the constraints on the functions. ACL2 therefore disallows functional instantiation of theorems by substituting for functions introduced via templates.

The use of template events facilitates integration of ACL2 with tools like SixthSense above. Suppose that we wish to connect ACL2 with an unverified tool *tool1* that implements a theory that we do not wish to define explicitly in ACL2. We then use a template event to introduce the function symbols (say f and g) regarding which the theory of the clause processor contains additional axioms. Finally we introduce *tool1* as an unverified clause processor, marking f and g as supporting events.

We now explain the logical burden for the developer of such a connection. Assume that an ACL2 theory $\mathcal{T}$ is extended by a template event E, and suppose that the supporting events for *tool1* mention some function introduced by E. Then the developer of *tool1* must guarantee that it is possible, in principle, to introduce f and g via the encapsulation principle (which we will refer to as the "promised" encapsulation E_P of the functions) such that the following conditions hold:

1. The constraints in E_P include the constraints in E.

2. E_P does not introduce any additional function symbols other than those introduced by E.

3. E_P is admissible in theory $\mathcal{T}$.

4. For any extension $\mathcal{T}_0$ of $\mathcal{T}$ together with the constraints in E_P, if one can invoke *tool1* to reduce some clause C to a list of clauses L_C then if each clause of L_C is first-order provable (with induction) in $\mathcal{T}_0$ then C must be provable in $\mathcal{T}_0$.

Furthermore, in order to make logical guarantees regarding ACL2 sessions that contain events corresponding to *several* unverified external tools, ACL2 enforces the following "disjointness condition": a template event may not be extended to a promised encapsulate by two different clause processors. Thus, when an unverified clause processor installation event has a supporting event name, f, such that f is a function symbol that had been introduced by a template, it is required that no unverified clause processor has been previously installed in the current ACL2 session that has a supporting event name that is a function symbol introduced in the same template. This makes it possible to view the event as essentially the (unique) promised encapsulation whose existence is guaranteed by the implementor of the tool. Note that condition 2 above is necessary for this purpose to preclude the possibility that the theory implemented by different external tools might have conflicting implicit axioms in their promised encapsulations for function symbols not introduced by the template.

With these conditions, we can make the following informal claim for an ACL2 session which includes templates together with the use of unverified clause processors:

> Perform the following transformation in sequence to each template event E in the session. If there is a tool **tool1** whose supporting events mention a function symbol introduced by E then replace E with the encapsulation E_P promised by the developer of **tool1**. Otherwise extend E to an arbitrary admissible encapsulate. (Note that at least one such extension exists, namely one in which no additional constraint is introduced.) Then every alleged theorem in the session is in fact derivable in first-order logic (with induction) from the axiomatic events in the session produced after this transformation.

The informal claim above can be made precise by formalizing the notion of an ACL2 session. Kaufmann and Moore [KM01] describe such a formalization where a valid session is modeled as a *chronology*, inductively defined as a sequence of events that is either (i) the empty sequence, or (ii) constructed from a sequence by introducing one of the legal ACL2 events such as commands for introducing new functions, and proving theorems. We omit that description here and refer the reader to their paper [KM01] for details. For this paper, we point out that given a careful inductive characterization of a session as a chronology, it is easy to see that an ACL2 session transformed as above really corresponds to a chronology. The basic observation is that for any chronology in which no function introduced by an encapsulation is functionally instantiated, the encapsulation may be strengthened and the result is still a chronology. The proof is by induction on the formation of chronologies, and each proof obligation encountered in the inductive step is discharged against the possibly stronger theory.

6   Related Work

The importance of allowing the hooking up of external tools has been widely recognized in the theorem proving community. Some early ideas for connecting different theorem provers are discussed in a proposal for so-called "interface logics" [Gut91], with the goal to connect automated reasoning tools by defining a single logic L such that the logics of the individual tools can be viewed as sub-logics of L. More recently, with the success of model checkers and Boolean satisfiability solvers, there has been significant work connecting such tools with interactive theorem proving. The PVS theorem prover provides connections with several decision procedures such as model checkers and SAT solvers [RSS95, Sha01]. The Isabelle theorem prover [Pau] uses unverified external tools as *oracles* for checking formulas as theorems during a proof search; this mechanism has been used to integrate model checkers and arithmetic decision procedures with Isabelle [MN95, BF00]. Oracles are also used in the HOL family of higher order logic theorem provers [GM93]; for instance, the PROSPER project [DCN+00] uses the HOL98 theorem prover as a uniform and logically-based coordination mechanism between several verification tools. The most recent incarnation of this family of theorem provers, HOL4, uses an external oracle interface to decide large Boolean formulas through connections to state-of-the-art BDD and SAT-solving libraries [Gor02], and also uses that oracle interface to connect HOL4 with ACL2 as discussed in the next section.

The primary basis for interfacing external tools with theorem provers for higher-order logic (specifically HOL and Isabelle) involves the concept of "theorem tagging", introduced by Gunter for HOL90 [Gun98]. The idea is to introduce a tag *in the logic* for each oracle and view a theorem certified by the oracle as an implication with the tag corresponding to the certifying oracle as a hypothesis. This approach enables tracking of dependencies on unverified tools at the level of individual theorems. In contrast, our approach is designed to track such dependencies at the level of files, that is, ACL2 books. Our coarser level of tracking is at first glance unfortunate: if a book contains some events that depend on such tools and others that do not, then the entire book is "tainted" in the sense that its certification requires an appropriate acknowledgement for the tools. We believe that this will not prove to be an issue in practice, as ACL2 users typically find it easy to move events between books. On the positive side, it is simpler to track a single event introducing an external tool rather than uses of such an event, especially since hints are ignored when including previously certified books. As an aside, we note that a very general tagging mechanism is under development for ACL2, serving as a foundation in particular for tagging of unverified clause processors.

There has also been work on using an external tool to search for a proof that can then be checked by the theorem prover without assistance from the tool. Hurd [Hur02] describes such an interface connecting HOL with first-order logic. McCune and Shumsky [MS00] present a system called Ivy which uses Otter to search for first-order proofs of equational theories and then invokes ACL2 to check such proof objects. Meng and Paulson [MP04] interface Isabelle with a resolution theorem prover.

Several ACL2 users have integrated external tools with ACL2; but without the disciplined mechanisms of this paper, such integration has essentially involved implementation hacks on the ACL2 source code. Ray, Matthews, and Tuttle integrate ACL2 with SMV [RMT03]. Reeber and Hunt connect ACL2 with the Zchaff satisfiability solver [RH06], and Sawada and Reeber provide a connection with SixthSense [SR06]. Manolios and Srinivasan connect ACL2 with UCLID [MS04, MS05].

7   Conclusion and Future Work

Different deduction tools bring in different capabilities to formal verification. A strength of general purpose theorem provers compared to many tools based on decision procedures is in the expressive power of the logic, which enables succinct definitions. Automatic decision procedures provide more automated proof procedures for decidable theories. Several ACL2 users have requested ways to connect ACL2 with automated decision procedures. We believe that the mechanisms described in this paper will provide a disciplined way of using ACL2 with other tools with a clear specification of the expectations from the tool in order to guarantee soundness of the ACL2 session. Furthermore, we believe that verified clause processors will provide a way for the user to control a proof more effectively without relying on ACL2's heuristics.

We have presented an approach to connecting ACL2 with external deduction tools, but we have merely scratched the surface. It is well-known that developing an effective interface between two or more deduction tools is a complicated exercise [KM92]. It remains to be seen how to effectively decompose theorem proving problems so as to

make effective use of clause processors to provide the requisite automation.

Some researchers have criticized our interface on the grounds that developing a connection with an external tool requires significant knowledge of the ACL2 logic. While we acknowledge that our interface requires understanding of that logic, including the term representation, we believe that such requirement is necessary for any developer interested in developing connections between formal tools. A connection between different formal tools must involve a connection between two logics, and the builder of such connection must understand both the logics, including the legal syntax of terms, and the axioms and rules of inferences. It should be noted that the logic of ACL2 is perhaps more complex than many others, principally because it offers proof structuring mechanisms by enabling the user to mark events as *local*. This complexity manifests itself in the interface; constructs such as *supporters* are provided essentially to enable the tool implementor to provide logical guarantees in the presence of local events. However, we believe that with these constructs it will be possible for the tool developers to implement connections with ACL2 with reasonable understanding of the theorem prover.

Finally, note that the restrictions for the tool developers that we have outlined preclude certain external deduction tools. For instance, there has been recent work connecting HOL with ACL2 [GHKR06a, GHKR06b]; the approach there has been for a HOL user to make use of ACL2's proof automation and fast execution capabilities. It might be of interest to the ACL2 user to take advantage of HOL's expressive power as well. We are working on extending the logical foundations of ACL2 to facilitate such a connection. The key idea is that the ACL2 theorem prover might be viewed as a theorem prover for the HOL logic. If the view is accurate then it will be possible for the user of ACL2 to prove some formulas in HOL and use them in an ACL2 session, claiming that the session essentially reflects a HOL session mirrored in ACL2.

Acknowledgements

References

[BF00]   D. Basin and S. Friedrich. Combining WS1S and HOL. In D. M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[BH97]   B. Brock and W. A. Hunt, Jr. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In *Proceedings of the* 1997 *International Conference on Computer Design: VLSI in Computers & Processors (ICCD 1997)*, pages 31–36, Austin, TX, 1997. IEEE Computer Society Press.

[BM81]   R. S. Boyer and J S. Moore. Metafunctions: Proving them Correct and Using Them Efficiently as New Proof Procedure. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, UK, 1981.

[DCN+00]   L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. F. Melham. The PROSPER toolkit. In S. Graf and M. Schwartbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for Constructing Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 78–92, Berlin, Germany, 2000. Springer-Verlag.

[FKR+02]   A. Flatau, M. Kaufmann, D. F. Reed, D. Russinoff, E. W. Smith, and R. Sumners. Formal Verification of Microprocessors at AMD. In M. Sheeran and T. F. Melham, editors, *4th International Workshop on Designing Correct Circuits (DCC 2002)*, Grenoble, France, April 2002.

[GHKR06a]   M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An embedding of the ACL2 logic in HOL. In P. Manolios and M. Wilding, editors, *6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, Seattle, WA, August 2006.

[GHKR06b]   M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An integration of HOL and ACL2. In A. Gupta and P. Manolios, editors, *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006)*, San Jose, CA, November 2006. Springer-Verlag.

[GM93]   M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[Gor02]   M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, 2002.

[GRW04]   D. A. Greve, R. Richards, and M. Wilding. A Summary of Intrinsic Partitioning Verification. In M. Kaufmann and J S. Moore, editors, *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.

[Gun98]   E. L. Gunter. Adding External Decision Procedures to HOL90 Securely. *Lecture Notes in Computer Science*, 1479:143–152, 1998.

[Gut91]   J. D. Guttman. A Proposed Interface Logic for Verification Environments. Technical Report M-91-19, The Mitre Corporation, March 1991.

[Hur02]   J. Hurd. An LCF-Style Interface between HOL and First-Order Logic. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE 2002)*, volume 2392 of *LNCS*, pages 134–138. Springer-Verlag, 2002.

[KM92]   M. Kaufmann and J S. Moore. Should We Begin a Stanrdization Process for Interface Logics? Technical Report 72, Computational Logic Inc. (CLI), January 1992.

[KM94]   M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic Incorporated (CLI), 1717 West Sixth Street, Suite 290, Austin, TX 78703, 1994.

[KM97]   M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. See URL `http://www.cs.utexas.edu/users/moore/publications/km97.ps.gz`, 1997.

[KM01]   M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[KM06]   M Kaufmann and J S. Moore. ACL2 home page, 2006. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

[KMM00]   M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[MBP$^+$04] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable Automated Verification via Exper-System Guided Transformations. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 217–233. Springer-Verlag, 2004.

[MLK98]   J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Kernel of the AMD5K86 Floating-point Division Algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[MN95]   O. Müller and T. Nipkow. Combining Model Checking and Deduction of I/O-Automata. In E. Brinksma, editor, *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1995)*, volume 1019 of *LNCS*, Aarhus, Denmark, May 1995. Springer-Verlag.

[MP04]   J. Meng and L. C. Paulson. Experiments on Supporting Interactive Proof Using Resolution. In D. A. Basin and M. Rusinowitch, editors, *Proceedings of the 2nd International Joint Conference on Computer-Aided Reasoning (IJCAR 2004)*, volume 3097 of *LNCS*, pages 372–384, 2004.

[MS00]   W. McCune and O. Shumsky. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In P. Manlolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 217–230. Kluwer Academic Publishers, Boston, MA, June 2000.

[MS04]   P. Manolios and S. Srinivasan. Automatic Verification of Safety and Liveness of XScale-Like Processor Models Using WEB Refinements. In *Design, Automation and Test in Europe (DATE 2004)*, pages 168–175, Paris, France, 2004. IEEE Computer Society Press.

[MS05]   P. Manolios and S. Srinivasan. Refinement Maps for Efficient Verification of Processor Models. In *Design, Automation and Test in Europe (DATE 2005)*, pages 1304–1309, Munich, Germany, 2005. IEEE Computer Society Press.

[Pau]   L. Paulson. The Isabelle Reference Manual. See URL `http://www.cl.-cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2003/doc/ref.pdf`.

[RF00]   D. Russinoff and A. Flatau. RTL Verification: A Floating Point Multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA, June 2000. Kluwer Academic Publishers.

[RH06]   E. Reeber and W. A. Hunt, Jr. A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA). In U. Furbach and N. Shankar, editors, *Proceedings of the* 3*rd International Joint Conference on Computer-Aided Reasoning (IJCAR 2006)*, volume 4130 of *LNAI*, pages 453–467, 2006.

[RMT03]   S. Ray, J. Matthews, and M. Tuttle. Certifying Compositional Model Checking Algorithms in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J S. Moore, editors, 4*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, Boulder, CO, July 2003.

[RSS95]   S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model-Checking with Automated Proof Checking. In P. Wolper, editor, *Proceedings of the* 8*th International Conference on Computer-Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.

[Rus98]   D. Russinoff. 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. *LMS Journal of Computation and Mathematics*, 1:148–200, December 1998.

[SH97]   J. Sawada and W. A. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. In O. Grumberg, editor, *Proceedings of the* 9*th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[Sha01]   N. Shankar. Using Decision Procedures with Higher Order Logics. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the* 14*th International Conference on Theorem Proving in Higher-Order Logics (TPHOLS 2001)*, volume 2152 of *LNCS*, pages 5–26. Springer-Verlag, 2001.

[SR06]   J. Sawada and E. Reeber. ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool. In A. Gupta and P. Manolios, editors, *Proceedings of the* 6*th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006)*, San Jose, CA, November 2006. Springer-Verlag.