Integrating External Deduction Tools with ACL2 ⋆,⋆⋆

Matt Kaufmann *

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-0233, USA.

J Strother Moore

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-0233, USA.

Sandip Ray *

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-0233, USA.

Erik Reeber *

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-0233, USA.

Abstract

We present an interface connecting the ACL2 theorem prover with external deduction tools. The ACL2 logic contains several mechanisms for proof structuring, which are important to the construction of industrial-scale proofs. The complexity induced by these mechanisms makes the design of the interface challenging. We discuss some of the challenges, and develop a precise specification of the requirements on the external tools for a sound connection with ACL2. We also develop constructs within ACL2 to enable the developers of external tools to satisfy our specifications. The interface is available with the ACL2 theorem prover starting from Version 3.2, and we describe several applications of the interface.

Key words: automated reasoning, decision procedures, first-order logic, interfaces, theorem proving

1 Introduction

Recent years have seen rapid advancement in the capacity of automatic reasoning tools, in particular for decidable theories such as Boolean logic and Presburger arithmetic. For instance, modern BDD packages and satisfiability solvers can automatically solve problems with tens of thousands of variables and have been successfully used to reason about commercial hardware system implementations [2,3]. This advancement has sparked significant interest in the general-purpose mechanized theorem proving community, to improve the efficiency of theorem provers by developing connections with automatic reasoning tools. In this paper, we present a general interface for connecting the ACL2 theorem prover [4,5] with tools that are external to ACL2's built-in reasoning routines.

ACL2 consists of a functional programming interface based on Common Lisp [6], along with a first-order interactive theorem prover. The ACL2 theorem prover supports several deduction mechanisms such as congruence-based conditional rewriting, well-founded induction, several integrated decision procedures, and generalization. ACL2 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 [7–10], microcode for the Motorola CAP DSP [11], and separation properties of the Rockwell Collins AAMP7$^{\text{TM}}$ processor [12]. However, the applicability of ACL2 (as that of any theorem prover) is often limited by the amount of user expertise necessary to drive the theorem prover. Indeed, each of the above projects represents many man-years of effort. Yet, many of the necessary lemmas, for instance those establishing hardware invariants, can be expressed in a decidable theory and dispatched by a decision procedure.

On the other hand, it is non-trivial to establish a sound connection between ACL2 and other tools. ACL2 contains several logical constructs intended to

---

⋆ A preliminary version of this paper [1] appeared in the 6th International Workshop for Implementation of Logics (Phnom Penh, Cambodia, November 2006).

⋆⋆This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591, by the National Science Foundation under Grant No. ISS-0417413, and by DARPA under Contract No. NBCH30390004.

* Corresponding authors.

Email addresses: kaufmann@cs.utexas.edu (Matt Kaufmann), moore@cs.utexas.edu (J Strother Moore), sandip@cs.utexas.edu (Sandip Ray), reeber@cs.utexas.edu (Erik Reeber).

URLs: http://www.cs.utexas.edu/users/kaufmann (Matt Kaufmann), http://www.cs.utexas.edu/users/moore (J Strother Moore), http://www.cs.utexas.edu/users/sandip (Sandip Ray), http://www.cs.utexas.edu/users/reeber (Erik Reeber).

facilitate effective proof structuring [13]. These constructs are crucial to the applicability of ACL2 in large-scale verification projects; however, they complicate the logical foundations of the theorem prover. To facilitate connection between another tool and ACL2, it is therefore imperative (i) to determine the conditions under which a conjecture certified by a combination of the theorem prover and the tool is indeed a theorem, and (ii) to provide mechanisms that enable a tool implementor to meet these conditions.

The interface described in this paper enables the connection of ACL2 with other reasoning tools. In particular, it permits an ACL2 user to invoke an external tool to reduce a goal formula C to a list of formulas L_C during a proof attempt. Correctness of the tool involves showing that the provability of each formula in L_C (in the logic of ACL2) implies the provability of C. We present a sufficient condition (expressible in ACL2) that guarantees such provability claims, and discuss the logical requirements on the implementor of external tools for sound connection with ACL2. The interface design illustrates some of the subtleties and corner cases that need to be considered in augmenting an industrial-strength formal tool with a non-trivial feature.

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 via a system call interface. An unverified tool is introduced with a *trust tag* acknowledging that the validity of the formulas proven using the tool depends on the correctness of the tool.

The connection with unverified tools enables us to invoke external SAT solvers, BDD packages, and so on, for simplifying ACL2 subgoals. Why might one use verified tools? The formal language of ACL2 is a programming language, based on an applicative subset of Common Lisp. The close relation between ACL2 and Lisp makes it possible to write efficient programs in the ACL2 logic [6]. Indeed, most of the source code implementing the theorem prover is written in this language. It can therefore be handy for the ACL2 user to control proofs by (i) implementing customized reasoning code, (ii) verifying such code with ACL2, and (iii) invoking the code for proving theorems in a specific 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" [14]. However, such rules essentially augment ACL2's term simplifier without providing a way to manipulate directly the entirety of a subgoal generated during a proof. Furthermore, meta rules can only simplify a term to one that is provably equivalent; that is, they do not allow generalization. The connection with verified tools supports direct invocation of customized, provably correct, reasoning code for reducing a conjecture to a collection of (possibly

more general) subgoals.

The rest of the paper is organized as follows. In Section 2 we provide a brief review of ACL2; this section is intended to provide an overview of the facets of ACL2 that are relevant to the subsequent discussions, and can be skipped by readers familiar with the theorem prover without loss of continuity. In Sections 3 through 5 we present the interface for connecting external tools to ACL2, the logical requirements for the developer of such connections, and the necessary augmentations required to support the interface. In Section 6, we provide a few remarks on our implementation. We discuss related work in Section 7, and conclude in Section 8.

The interface described in this paper is available with ACL2 Version 3.2, and ACL2's hypertext documentation includes a topic, clause-processor, which provides further details for many of its features. In addition, the ACL2 distribution contains a directory `books/clause-processors/`, with proof scripts demonstrating many applications of the interface.

2 ACL2

The name "ACL2" stands for "A Computational Logic for Applicative Common Lisp". It is used to denote (i) a programming language based on an applicative subset of Common Lisp, (ii) a first-order logic of recursive functions, and (iii) a theorem prover for the logic. ACL2 is a complex theorem proving system, consisting of more than 8MB of source code (including comments and documentation), and has been used in both industry and academia to prove the correctness of complex computing systems. Readers interested in a thorough understanding of ACL2 are referred to the ACL2 Home Page [5], which contains extensive hypertext documentation and references to several books and papers about the theorem prover.

2.1 The logic

The kernel of the ACL2 logic consists of a formal syntax, some axioms, and some rules of inference. The kernel syntax describes terms composed of variables, constants, and function symbols applied to a fixed number of argument terms. The kernel logic introduces the notion of "formulas" as composed of equalities between terms and the usual propositional connectives. Kaufmann and Moore [15] provide a precise characterization of the kernel logic. The logic supported by the theorem prover is an extension of the kernel logic.

The syntax of ACL2 is the prefix-normal syntax of Lisp: 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 typically use the latter form, referring to the formal syntax only when it is relevant for the discussion. We also use more conventional notations for commonly used functions, thus writing $(x \times y)$ instead of `(* x y)` and (*if* x *then* y *else* z) instead of `(if x y z)`, dropping parentheses when it is unambiguous to do so.

ACL2 has axioms specifying properties of certain Common Lisp primitives. We show below the axioms about the primitives *equal* and *if*. Note that the kernel syntax is quantifier-free and each formula is implicitly universally quantified over all free variables in the formula.

Axioms.

$x = y \quad \Rightarrow \quad \mathit{equal}(x, y) = \texttt{T}$

$x \neq y \quad \Rightarrow \quad \mathit{equal}(x, y) = \texttt{NIL}$

$x = \texttt{NIL} \Rightarrow (\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) = z$

$x \neq \texttt{NIL} \Rightarrow (\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) = y$

The axiomatization of *equal* and *if* makes it possible to embed propositional calculus and equality into the term language. Indeed, an ACL2 user never writes formulas but only terms. Terms are interpreted as formulas by using the following convention. When we write a term τ where a formula is expected, it represents the formula $\tau \neq \texttt{NIL}$. Thus, in ACL2, the following term is an axiom relating the Lisp functions *cons*, *car*, and *equal*.

Axiom.

$\mathit{equal}(\mathit{car}(\mathit{cons}(x, y)), x)$

The axiom stands for the formula $\mathit{equal}(\mathit{car}(\mathit{cons}(x, y)), x) \neq \texttt{NIL}$, which is provably equal to $\mathit{car}(\mathit{cons}(x, y)) = x$. In this paper, we will feel free to interchange terms and formulas by the above convention. We will also apply the same logical connectives to a term or formula; thus when we write $\neg\tau$ for a term τ, we mean the term (or formula) $\mathit{not}(\tau)$, where *not* is axiomatized as:

Axiom.

$\mathit{not}(x) = \mathit{if}\ x\ \mathit{then}\ \texttt{NIL}\ \mathit{else}\ \texttt{T}$

The duality between terms and formulas enables us to interpret an ACL2 theorem as follows. If the term τ (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; `NIL` can thus be viewed as logical false. This informal "evaluation interpretation" of theorems provides motivation for deriving sufficient conditions for correctness of external tools

integrated with ACL2; see Theorem 1 in Section 3.

The kernel logic includes axioms that characterize the primitive Lisp functions over numbers, characters, strings, constant symbols such as `T` and `NIL`, and ordered pairs. These objects together make up the ACL2 standard universe; but the axioms do not preclude "non-standard" universes which may contain other objects. Lists are represented as ordered pairs, so that the list `(1 2 3)` is represented by the term $\mathit{cons}(1, \mathit{cons}(2, \mathit{cons}(3, \texttt{NIL})))$. For brevity, we will write $\mathit{list}(x, y, z)$ as an abbreviation for $\mathit{cons}(x, \mathit{cons}(y, \mathit{cons}(z, \texttt{NIL})))$. Another convenient data structure built out of ordered pairs is the *association list* (or *alist*) which is essentially a list of pairs, *e.g.*, $\mathit{list}(\mathit{cons}(\texttt{"a"}, 1), \mathit{cons}(\texttt{"b"}, 2))$. 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 and well-founded induction up to ε_0 (see below). [1] For instance, the formula $\mathit{car}(\mathit{cons}(2, x)) = 2$ 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), which contains the axioms of the kernel logic.

Note that the ACL2 logic allows well-founded induction up to ε_0 in spite of being a first order logic. The logical foundation for induction in ACL2 is provided as follows. GZ contains an embedding of ordinals up to ε_0, represented in Cantor Normal form [18], and GZ axiomatizes a binary relation $\prec$ to be an irreflexive total order on (the representation of) ordinals. The theory GZ is *inductively complete*: for any formula φ expressible in GZ, every first-order *induction axiom* of the following form belongs to GZ, where φ/σ denotes the formula obtained by applying the substitution σ to φ.

$$(\forall y \prec \varepsilon_0)[((\forall x \prec y)\varphi/\{y := x\}) \Rightarrow \varphi(y)] \Rightarrow (\forall y \prec \varepsilon_0)\varphi(y)$$

Finally, ACL2 only allows construction of theories that are extensions of GZ via the *extension principles* explained below, which allow axiomatization of new function symbols. When a new function symbol is introduced via the extension principles, the resulting theory $\mathcal{T}'$ is the extension of the original theory $\mathcal{T}$ with (i) the axiom explicitly introduced by the extension principle, and (ii) all the induction axioms in the language of the new theory.

[1] Here ε_0 is the least ordinal that is closed under exponentiation. For a comprehensive overview of ordinals, we refer the reader to Church and Kleene's treatment [16], which is recounted in Chapter 11 of Roger's book on computability [17].

2.1.1 Extension Principles

ACL2 provides *extension principles* allowing the user to introduce new function symbols. Below, we discuss two extension principles which are particularly relevant to us, namely (i) the *definitional principle* for introducing totally defined functions, and the (ii) *encapsulation principle* for introducing constrained functions. Other extension principles include a *defchoose principle* for introducing Skolem (choice) functions, and a *defaxiom principle* that enables the specification of a formula as an axiom. The latter is discouraged since the introduction of arbitrary axioms is potentially unsound. For this paper, unless explicitly mentioned otherwise, we ignore introduction of arbitrary axioms.

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.
$\mathit{fact}(n) = \mathit{if}\ \mathit{natp}(n) \wedge (n > 0)\ \mathit{then}\ n \times \mathit{fact}(n-1)\ \mathit{else}\ 1$

Here, $\mathit{natp}(n)$ 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 done by exhibiting a *measure* that maps the list of function arguments to the set of ordinals below ε_0, and showing that the measure decreases at every recursive call. For *fact* above, one possible measure is $\mathit{nfix}(n)$ (axiomatized in GZ) which returns n if n is a natural number, otherwise 0.

Encapsulation Principle and Functional Instantiation

The *encapsulation principle* allows the user to extend a theory by introducing functions with *constraints* rather than with full definitions. The constraints need not specify the return values of the functions for all inputs. For instance, consider introducing three functions, namely (i) a binary function *ac*, (ii) a constant function *ac-id*, and (iii) a unary function *ac-p*, axiomatized to satisfy the following five constraints. [2]

Encapsulation Axioms.
$\mathit{ac}(\mathit{ac}(x, y), z) = \mathit{ac}(x, \mathit{ac}(y, z))$

[2] The example is adapted from the topic functional-instantiation-example in the ACL2 documentation.

Definitional Axiom.
ac-list(l) = *if* ¬*consp*(l) *then* `NIL`
else if ¬*consp*(*cdr*(l)) *then* *car*(l)
else *ac*(*car*(l), *ac-list*(*cdr*(l)))

Fig. 1. Definition of *ac-list*.

ac(x, y) = *ac*(y, x)
ac-p(*ac*(x, y)) = `T`
ac-p(*ac-id*()) = `T`
ac-p(x) ⇒ *ac*(*ac-id*(), x) = x

The axioms stipulate that *ac* is an associative-commutative function that always returns an object in the domain recognized by the predicate *ac-p*, and *ac-id* is an identity function over that domain. The effect of an encapsulation is to extend a theory with the new function symbols constrained to satisfy only the encapsulation axioms. To ensure consistency, ACL2 requires the user to exhibit corresponding functions (called the *witnesses* to the encapsulation) that satisfy the alleged axioms. For instance, to introduce the encapsulation axioms above, one can use the following witnesses: (i) for *ac*, the constant function that always returns 42, (ii) for *ac-id*, the constant 42, and (iii) for *ac-p*, the function that returns `T` on input 42 and `NIL` on all other inputs.

For functions introduced using encapsulation, the only axioms known are the encapsulation axioms (*i.e.*, the constraints). Thus any theorem that can be proven about such functions is valid for any other set of functions that also satisfy the constraints. This observation is encoded in ACL2 via a derived rule of inference called *functional instantiation* [19]. Below, we illustrate the use of functional instantiation using the encapsulated functions *ac*, *ac-id*, and *ac-p* above. Consider the function *ac-list* shown in Fig. 1, which returns the result of applying *ac* to a list of objects. It is straightforward to prove the theorem named `ac-reverse-relation` below, which states that the repeated application of *ac* along the list l has the same effect as the repeated application of *ac* along the list obtained by reversing l. Here the list reversing function, *reverse*, is axiomatized in GZ.

Theorem `ac-reverse-relation`**:**
ac-list(*reverse*(l)) = *ac-list*(l)

Finally, note that the binary addition function "+" is associative and commutative and always returns a number, 0 is the left identity over numbers, and GZ has a predicate *acl2-numberp* that returns `T` if and only if its argument is a number. Thus, if we define the function *sum-list* that repeatedly adds all elements of a list l, then functional instantiation of the theorem `ac-reverse-relation` enables us to prove the following theorem under the functional substitution of

"+" for *ac*, 0 for *ac-id*, *numberp* for *ac-p*, and *sum-list* for *ac-list*.

Theorem `sum-reverse-relation`**:**
sum-list(*reverse*(l)) = *sum-list*(l)

As the example illustrates, encapsulation and functional instantiation enable the ACL2 user to prove properties of concrete definitions by reasoning about more abstract, generic functions. It is worth noting that the logic of ACL2 is limited compared to those of other general-purpose theorem provers such as HOL or PVS; while the latter provers support higher order logic (typically some form of typed λ-calculus), the logic of ACL2 is first order. Functional instantiation provides limited higher order reasoning capabilities in ACL2, albeit as a derived rule of inference. Indeed, many recent approaches implemented in ACL2 for automating program verification tasks have used proof strategies that involve functional instantiation of generic theories [20–22].

2.2 The Theorem Prover

ACL2 is an automated, interactive proof assistant. It is *automated* in the sense that no user input is expected once it has embarked on the search for the proof of a conjecture. It is *interactive* in the sense that the search is significantly affected 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. ACL2 also supports a goal-directed interactive loop (called the "*proof-checker*"), similar in nature to LCF-style provers like HOL [23] and Isabelle [24]; but it is much less frequently used and not relevant to the discussions in this paper.

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

Theorem `fact-is-natp`**:**
natp(*fact*(x)) = `T`

ACL2 attempts to prove a conjecture by applying a sequence of transformations to it, replacing each goal (initially, the conjecture) with a list of subgoals. Internally, ACL2 stores each goal as a *clause* represented as an object in the ACL2 universe. A goal of the form $\tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_n \Rightarrow \tau$ is represented as a list of terms, `(`$\neg\tau_1$ `...` $\neg\tau_n$ τ`)`, which is viewed as the disjunction of its elements (*literals*). ACL2 has a *hint* mechanism which the user can use to provide prag-

matic advice on proof search at any goal or subgoal; in this example, the user can advise ACL2 to begin the search by inducting on `x`.

Once a theorem is proven, it is stored in a database and used 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 if ACL2 subsequently encounters a term of the form $\mathit{natp}(\mathit{fact}(\tau))$ then the term is rewritten to `T`.

Users interact with the ACL2 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, called a *book*. Once the desired theorems have been proven, ACL2 can be instructed to *certify* a book to facilitate the use of the associated events in other projects: a book is certified once and then *included* during a subsequent ACL2 session without rerunning the proofs. ACL2 allows the user to mark some events in a book as *local events*. Typically, the user marks as local auxiliary function definitions and intermediate lemmas which are not deemed generally useful. When a book is included in a subsequent proof, the theorem prover's database is extended only with the non-local events in the book.

The ability to mark events as *local* is one of the key factors facilitating the applicability of ACL2 to large-scale verification tasks. In particular, it allows the user to develop and manage proofs in a disciplined manner without cluttering the database of the theorem prover or overwhelming its proof procedures and heuristics with a large number of unwanted definitions and lemmas. Also, once a proof has been completed, hiding auxiliary definitions and lemmas simplifies the resulting proof structure, which aids in proof presentation and proof maintenance. Kaufmann [25] provides an illustrative example of such use of local events in a proof of the Fundamental Theorem of Calculus.

However, the presence of local events complicates the logical foundations of ACL2, and the combination of local events with other features of the theorem prover can have surprising repercussions. In Sections 3 through 5 we will explain some of the subtleties involved in the interaction of local events with our interface for integrating external deduction tools, and the measures taken to account for such interactions. To understand why local events complicate the logical foundations, note that the feature is logically much more powerful than merely restricting the *applicability* of certain events to a specific scope. For instance, the local events in a book might include commands for introducing new functions (and hence new axioms) but when the book is included in a subsequent session, the logical theory is not extended by local events. Yet,

during the certification of the book, ACL2 might have used such local axioms to prove a *non-local* lemma. In order to justify the use of the non-local lemmas in a book while not extending the logical theory with the local axioms, one must clarify under what conditions it is legitimate to mark an event to be local. This question has been answered by Kaufmann and Moore [13] as follows: if a formula ϕ is proven as a theorem in an ACL2 session, then ϕ is in fact first-order derivable from the axioms of GZ together with only the axiomatic events in the session (that is, event commands that extend a theory with new axioms) that (hereditarily) involve function symbols in ϕ or in formulas introduced via the *defaxiom principle*. In particular, every extension principle other than the defaxiom principle produces an extension to the current theory which is *conservative*. Thus, any event that does not correspond to a *defaxiom* and does not involve the function symbols in the non-local events can be marked local. ACL2 implements this by making two passes on a book during certification. In the first pass it proves each theorem (and admits each axiomatic event) sequentially, and ensures that no *defaxiom* event is marked local. In the second pass it performs a *local incompatibility check*, skipping proofs and checking that each axiomatic event involved in any non-local event is also non-local.

2.3 The ACL2 Programming Environment

ACL2 is closely tied with Common Lisp. It employs Lisp syntax, and the axioms in GZ for the Lisp primitives are crafted so that the return values predicted by the axioms agree with those specified in the Common Lisp Manual [26] for arguments in the intended domains of application. Furthermore, events corresponding to the definitional principle are Lisp definitions. For instance, the formal event introducing *fact* above also serves as a Common Lisp definition:

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

The connection with Lisp enables users to execute formal definitions efficiently; ACL2 permits the execution of all functions axiomatized in GZ, and any function whose definition does not involve any constrained functions. The theorem prover makes use of this connection for simplifying ground terms. For instance, during a proof ACL2 will automatically simplify *fact*(3) to 6 by evaluation.

ACL2 also provides a logic-free programming environment to facilitate efficient code development. One can implement any applicative Lisp function and mark

it to be in *program mode*. No proof obligation is generated for such functions and no logical guarantee (including termination) is provided, but ACL2 can evaluate such functions via the Lisp evaluator. In addition, ACL2 supports a "system call" interface to the underlying operating system, which enables the user to invoke arbitrary executable code and operating system commands.

2.4 Evaluators

ACL2 provides a convenient notation for defining an *evaluator* for a fixed set of functions. Evaluators are integral to ACL2's "meta reasoning" capabilities. We provide an overview of evaluators here, since they are useful in characterizing the correctness of external tools. For more details about evaluators and their use in meta reasoning, we refer the reader to the documentation topics defevaluator and meta in the ACL2 documentation, along with previous papers [14,27] on the subject.

To understand the role of evaluators, consider the following simple challenge. Given a term that represents an equality between two sums, how do we arrange to cancel common addends from both sides of the equality? For instance, we want the term $(x+3y+z = a+y+b)$ to be simplified to $(x+2y+z = a+b)$. Note that ACL2 does not provide unification for associative-commutative functions. If one knows ahead of time the maximum number of addends that could potentially appear in a sum then one can write a (potentially large) number of rewrite rules to handle all permutations in which the common addends could appear. But this does not work in general, and is laborious, both for the user in developing the rules, and for ACL2 in sorting through a large database of rules any one of which is unlikely to be useful in most situations.

One solution to the challenge is to write a customized reasoning function (called a *metafunction*) for manipulation of terms. Terms are represented naturally as objects in the ACL2 universe; the term $\mathit{foo}(x)$ is represented as the object `(foo x)` which is a list of two elements. Metafunctions manipulate this *internal representation* of a term, producing (the internal representation of) a provably equivalent term. It is an easy exercise to write a metafunction that cancels common addends from a term representing equality between two sums.

The notion of an evaluator makes explicit the connection between a term and its internal representation. 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 specify a suitable evaluation semantics for the internal representation of terms composed of $f_1, \ldots, f_n$; such axioms are referred to as *evaluator axioms*. A precise characterization of all the evaluator axioms is described in the ACL2 Manual [5]

under the documentation topic defevaluator; below we show one of the axioms for illustration, which defines the evaluation semantics for an m-ary function symbol f. Here, $'f$ is assumed to be the internal representation of f and $'\tau_i$ is the internal representation of τ_i, for $1 \leq i \leq m$.

An Evaluator Axiom.
$ev(list('f, '\tau_1, \ldots, '\tau_m), a) = f(ev('\tau_1, a), \ldots, ev('\tau_m, a))$

In the formula above, it is convenient to think of a as an alist that maps the (internal representation of the) variables in $\tau_1, \ldots, \tau_m$ to ACL2 objects. Then the axiom specifies that the evaluation of the list $('f\ '\tau_1\ \ldots\ '\tau_m)$ (which corresponds to the internal representation of $f(\tau_1, \ldots, \tau_m)$) under some mapping of free variables is the same as the function f applied to the evaluation of each τ_i under the same mapping.

Evaluators allow us to state (and prove) correctness of metafunctions as formulas in the ACL2 logic. Let *cancel-addends* be the metafunction that cancels the common addends in an equality, and let *ev* be an evaluator. Then the following theorem is sufficient to justify the use of *cancel-addends* during proofs.

Theorem `cancel-addends-correct`
$ev(cancel\text{-}addends(\tau), a) = ev(\tau, a)$

The notion of evaluators is related to *reflection*, which is essentially the computation of ground terms in the metatheory. Harrison [28] provides a comprehensive survey of reflection in theorem proving. Reflection mechanisms are available in most theorem provers, for instance HOL, Isabelle, and PVS. The difference between these mechanisms and the evaluator constructs of ACL2 arises from the fact that the logic of ACL2 is first order. Note that in first order logic one cannot define a single closed-form function axiomatized to be an evaluator for an arbitrary set of functions. Thus, in ACL2, one must define separate evaluators for different (fixed) sets of functions. However, ACL2 provides a macro, called `defevaluator`, that takes a collection of function names $f_1, \ldots, f_k$, and a new function symbol *ev*, and introduces (via encapsulation) the constraints axiomatizing *ev* to be an evaluator for $f_1, \ldots, f_k$.

3 Verified External Tools

In this section, we start with a description of our interface to connect *verified* external tools with ACL2. The ideas and infrastructure we develop here will be extended in the next two sections to support 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 `(`$\neg\tau_0$ $\neg\tau_1$ `...` $\neg\tau_n$ τ`)`. Our interface enables the user to transform the current clause with custom code. More generally, a *clause processor* is a function that takes a clause C (together with possibly other arguments) and returns a list L_C of clauses.[3] 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 consists of 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 motivate the theorem alluded to above for installing *tool0* as a clause processor. Recall that theorems in ACL2 can be thought of in terms of evaluation: a formula Φ is a theorem if, for every substitution σ mapping each free variable of Φ to some object, the instance Φ/σ does not evaluate to `NIL`. Let C be a clause whose disjunction is the term τ, and let *tool0*, with C as its argument, produce the list `(`C_1 `...` C_n`)` that represents the conjunction of corresponding terms $\tau_1, \ldots, \tau_n$. (That is, each τ_i is the disjunction of clause C_i.) Informally, we want to ensure that if τ/σ evaluates to `NIL` for some substitution σ then there is some σ' and i such that τ_i/σ' also evaluates to `NIL`.

The condition is made precise by extending the notion of evaluators (cf. Section 2.4) from terms to clauses. To do so, assume that the ACL2 Ground Zero theory GZ contains two functions *disjoin* and *conjoin** axiomatized as shown in Fig. 2. Informally, the axioms specify how to interpret objects representing clauses and clause lists; for instance, *disjoin* specifies that the interpretation of a clause `(`τ_0 τ_1 τ_2`)` is the same as that of `(if` τ_0 `T (if` τ_1 `T (if` τ_2 `T NIL)))`, which represents the disjunction of τ_0, τ_1, and τ_2. Let *ev* be an evaluator for a set of function symbols that includes the function *if*; thus *ev*(*list*(`if`, τ_0, τ_1, τ_2), `a`) stipulates how the term "*if* τ_0 *then* τ_1 *else* τ_2"

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

$$
\begin{array}{l}
\mathit{disjoin}(C) \quad = \mathit{if}\ \neg\mathit{consp}(C)\ \mathit{then}\ *\texttt{NIL}* \\
\qquad\qquad\quad \mathit{else}\ \mathit{list}(\texttt{if},\ \ \mathit{car}(C),\ \ *\texttt{T}*,\ \ \mathit{disjoin}(\mathit{cdr}(C))) \\
\\
\mathit{conjoin}\text{*}(L_C) = \mathit{if}\ \neg\mathit{consp}(L_C)\ \mathit{then}\ *\texttt{T}* \\
\qquad\qquad\quad \mathit{else}\ \mathit{list}(\texttt{if},\ \ \mathit{disjoin}(\mathit{car}(L_C)),\ \ \mathit{conjoin}\text{*}(\mathit{cdr}(L_C)),\ \ *\texttt{NIL}*)
\end{array}
$$

Fig. 2. Axioms in GZ for supporting clause processors. `*T*` and `*NIL*` are assumed to be the internal representation of `T` and `NIL` respectively. The predicate *consp* is defined in GZ such that *consp*(x) returns `T` if x is an ordered pair, and `NIL` otherwise.

$$
\begin{array}{l}
\quad \mathit{pseudo\text{-}term\text{-}listp}(C) \wedge \\
\quad \mathit{alistp}(a) \wedge \\
\quad \mathit{ev}(\mathit{conjoin}\text{*}(\mathit{tool0}(C, \texttt{args})), \mathit{tool0\text{-}env}(C, a, \texttt{args})) \\
\Longrightarrow \mathit{ev}(\mathit{disjoin}(C), a)
\end{array}
$$

Fig. 3. Correctness Theorem for clause processors. Function *ev* is an evaluator for a set of functions that includes *if*; `args` represents the remaining arguments of *tool0* (in addition to clause C); *pseudo-term-listp* is a predicate axiomatized in GZ that returns `T` if its argument is an object in the ACL2 universe representing a list of terms (therefore a clause); and *alistp*(a) returns `T` if a is an association list, otherwise `NIL`.

is evaluated. For any theory $\mathcal{T}$, a clause processor function *tool0*(C, `args`) 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 Fig. 3 is a theorem.[4] 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 performed by *tool0*. We note that a weaker theorem logically suffices, replacing *tool0-env*(c, a, `args`) with an existentially quantified variable.

The obligation shown in Fig. 3 (once proven) can be associated with the new rule class for recognizing clause processors. This instructs ACL2 to use the function *tool0* as a verified external tool. Theorem 1 below guarantees that the obligation is sufficient. An analogous argument justifies ACL2's current meta reasoning facilities, which appears as a comment titled "Essay on Correctness of Meta Reasoning" in the ACL2 source code.

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 C is also provable in $\mathcal{T}$.*

[4] For the curious reader, the the ACL2 function *pseudo-term-listp* in Fig. 3 is so named since it actually only checks if its argument has the *syntactic structure* of a list of terms, *e.g.*, it does not check that the functions in the "terms" are defined and called with the correct number of arguments. The reasons for using this predicate are technical and not germane to this paper.

Proof: The theorem is a simple consequence of Lemma 1 below, taking τ_i to be v_i and given the obligation shown in Fig. 3 along with the definitions of *conjoin** and *disjoin*. □

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* `(`$\langle$`'`v_0`, '`$\tau_0\rangle$ $\ldots$ $\langle$`'`v_n`, '`$\tau_n\rangle$`)`*, where `'`v_i and `'`τ_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 `'`τ be the internal representation of the term τ. Then the following formula is provable:* $ev('\tau, e) = \tau/\sigma$.

Proof: The lemma follows by induction on the structure of term τ. □

In spite of the simplicity of the proof, the statement of Theorem 1 is perhaps more subtle than it appears. Note that the theorem restricts the use of a clause processor to *a theory in which the clause processor is legal*. This restriction might appear too strong by the following flawed reasoning. We know that each extension principle in ACL2 produces a conservative extension. Also, the function symbols occurring in formulas simplified by the application of a clause processor might not necessarily occur in the definition of the clause processor itself. For instance, consider a clause processor called *generalize* that replaces each occurrence of the object `'(f x)` (which is the internal representation of the term $f(x)$) in a clause with a new free variable. Note that although *generalize* manipulates the internal representation of *f*, the function *f* itself never appears in the formal definition of *generalize*. Thus by conservativity, a theorem proven by the application of *generalize* is valid in a theory in which *generalize* is not defined (*e.g.*, *generalize* does not add any new axiom about *f*), and hence one should be able to mark the definition of *generalize* (and any definition or theorem involving the definition of *generalize*) local in spite of being used in a hint as a clause processor. But this is precluded by Theorem 1.

To see the flaw in the reasoning above, consider defining a clause processor *simplify-foo* that replaces terms of the form *foo*(τ) by τ. Suppose a book, `book1`, contains the following event:

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

We can add *simplify-foo* as a legal clause processor in `book1`. Using an evaluator *ev-foo* for the function symbols *foo* and *if*, it is trivial to prove the obligation in Fig. 3 and thus install *simplify-foo* as a clause processor. Note that the definition of *foo* is local, and also that the definition of *simplify-foo* only manipulates the internal representation `'(`*foo* τ`)` of *foo*(τ); thus the definition of *simplify-foo* does not contain any occurrence of the function symbol *foo*. However, suppose we permit the installation of *simplify-foo* (non-locally) as a clause processor but mark the corresponding evaluator axioms for *ev-foo*

to be local. Then we can create a book, `book2`, with the following events.

Local Definitional Axiom.
$\mathit{foo}(x) = \mathit{cons}(x, x)$

Include `book1`.

We now are in a theory in which the application of *tool0* is inconsistent with the current definition of *foo*, resulting in unsoundness.

The example illustrates some of the subtleties arising in ACL2 because of its support for local events. The problem here is that the evaluator *ev-foo* which justifies the use of *simplify-foo* as a verified clause manipulation function is an evaluator for the locally defined function *foo*; for instance, one of the evaluator axioms for *ev-foo* is the following:

Evaluator Axiom for *ev-foo*
$\mathit{ev\text{-}foo}(\mathit{list}(\mathit{foo}, x), a) = \mathit{ev\text{-}foo}(\mathit{foo}(x), a)$

This axiom and the local definition of *foo* are necessary in proving the obligation in Fig. 3. The legality requirement ensures that the current theory contains the above axiom, which in turn requires that the functions "interpreted" by the evaluator are non-local. We remark that the example above, though simple, illustrates a soundness bug related to meta rules that existed in ACL2 for many years. [5] This bug resulted from the failure to track the fact that the theory in which the rules are applied is not properly supported by evaluators.

3.1 Applications of Verified External Tools

Verified clause processors enable an ACL2 user to bypass ACL2's heuristics and use customized, verified code on large-scale proofs in a specific domain. We now discuss some applications of these capabilities. The examples shown below illustrate the flexibility provided by the interface, and serve as a guide for the reader interested in customizing ACL2 for efficiently reasoning about a specific domain. Scripts supporting the examples below are distributed with ACL2 in the directory `books/clause-processors/`. Our description is consistent with these scripts, but we take some liberties when appropriate, for instance using different function names.

[5] We have checked that the bug exists as far back as in the December, 1997 release, and probably it was never absent before its fix in the February, 2006 release.

$$
\begin{array}{l}
\quad \mathit{pseudo\text{-}term\text{-}listp}(C) \wedge \\
\quad \mathit{alistp}(a) \wedge \\
\quad \mathit{ev\text{-}ifnot}(\mathit{conjoin}^*(\mathit{addHypothesis}(C, \tau)), a) \\
\Longrightarrow \mathit{ev\text{-}ifnot}(\mathit{disjoin}(C), a)
\end{array}
$$

Fig. 4. Correctness Theorem for the *addHypothesis* clause processor. Here *ev-ifnot* is an evaluator for the function symbols *if* and *not*. Note that we have used $\mathit{tool0\text{-}env}(C, a, \tau) = a$.

Adding a Hypothesis

Our first example is the trivial clause processor *addHypothesis* below, which weakens a formula with a user-supplied hypothesis:

$$\mathit{addHypothesis}(C, \tau) = \mathit{list}(\mathit{cons}(\mathit{list}(\mathit{not}, \tau), C), \tau)$$

Thus, if Φ_C is the formula represented by clause C, then the application of *addHypothesis* causes it to be replaced by the implication $\tau \Rightarrow \Phi_C$, together with a new proof obligation to show that τ is a theorem. The installation of *addHypothesis* as a clause processor, based on the proof obligations shown in Fig. 3, can be achieved by proving the theorem shown in Fig. 4, which is proven automatically by ACL2.

One application of the *addHypothesis* tool is to strengthen a formula before applying induction. For example, assume that we wish to prove $a \Rightarrow b$, and b can be proven by induction. One way to prove $a \Rightarrow b$ is to use the *addHypothesis* tool with $\tau := b$. This returns the trivially true subgoal $b \Rightarrow (a \Rightarrow b)$, together with the subgoal b which can now be dispatched by induction.

We should note that the *addHypothesis* clause processor, though useful, is merely pedagogical. Indeed, it is possible to mimic the action of *addHypothesis* with a certain hint available with ACL2. Nevertheless, its use illustrates how verified reasoning code enables the user to control proof search.

Equality Reasoning

Our next example is a clause processor that enables the user to control reasoning about a set of equalities. To motivate the use of a clause processor for equality reasoning, consider the following scenario. Let f and g be unary functions, and p be a unary predicate such that the formula $p(g(x), g(y))$ is a theorem. Consider proving the trivial theorem:

Theorem.
$(\mathit{f}(x) = \mathit{g}(x)) \wedge (\mathit{f}(y) = \mathit{g}(y)) \Rightarrow \mathit{p}(\mathit{f}(x), \mathit{f}(y))$

ACL2 cannot prove the above theorem automatically. To understand why, we explain a bit the heuristics involved in the term simplification performed by ACL2. In summary, ACL2 internally has a notion of the syntactic "complexity" of a term, and while simplifying terms that represent equality it rewrites complex terms to simpler ones. In the above scenario, ACL2 determines the terms involving g to be more complex than the corresponding terms involving f, and thus simplification will attempt to rewrite in the "wrong" order, failing to prove the theorem. [6]

We can, however, bypass these limitations by implementing a clause processor *equalitySubstitution*. [7] Given a clause C and a list of n pairs of terms $(\langle \tau_1, \tau'_1 \rangle \ \ldots \ \langle \tau_n, \tau'_n \rangle)$, the clause processor produces $n+1$ clauses as follows. The first clause is obtained by replacing each occurrence of τ_i in C with τ'_i. The remaining n clauses are formed by adding the term $(\tau_j = \tau'_j)$ to C, for $j = 1 \ldots n$; hence, if C represents the formula Φ_C then the j-th clause can be viewed as the subgoal $(\tau_j \neq \tau'_j) \Rightarrow \Phi_C$. In our scenario, the use of *equalitySubstitution* with the list $(\langle f(x), g(x) \rangle \ \langle f(y), g(y) \rangle)$ produces the following three trivial subgoals each of which can be proven automatically by ACL2.

Subgoals.

(1) $(g(x) = g(x)) \wedge (g(y) = g(y)) \Rightarrow p(g(x), g(y))$
(2) $(f(x) \neq g(x)) \Rightarrow ((f(x) = g(x)) \wedge (f(y) = g(y)) \Rightarrow p(f(x), f(y)))$
(3) $(f(y) \neq g(y)) \Rightarrow ((f(x) = g(x)) \wedge (f(y) = g(y)) \Rightarrow p(f(x), f(y)))$

The use of *equalitySubstitution* is an interesting example of the use of customized reasoning code to alleviate limitations in the simplification heuristics of the theorem prover. The correctness of *equalitySubstitution* merely depends on the correctness of substitution of equals, and is straightforward to verify. On the other hand, it is tricky to craft general-purpose heuristics that automatically perform effective equality-based simplification on arbitrary terms given large databases of applicable theorems, in the presence of induction and other proof techniques outside the domain of decision procedures. Indeed, many ACL2 users over the years have been "stumped" by limitations in the equality reasoning heuristics; the clause processor frees the user from dependence on, and complicated work-arounds for, such limitations.

[6] ACL2 also has so-called "cross-fertilization" heuristics that temporarily allow substitution of more complex terms for simpler ones, but the heuristics fail to prove the theorem in the scenario described.

[7] The *equalitySubstitution* clause processor has been implemented and verified by Jared Davis.

Sorting Bit Vector Addition

Our third example illustrates the use of verified clause processors to implement efficient simplification routines for domain-specific applications. The domain here is bit vector arithmetic, and we consider the problem of normalizing terms composed of bit vector additions. Such normalization is necessary in proofs involving arithmetic circuits, for instance to verify multiplier designs. Furthermore, although our focus is on bit vector addition, the general principle can be used in proofs involving any associative-commutative binary function.

Let $+_{32}$ be the binary function representing 32-bit bit vector addition, and consider the problem of proving the following theorem:

$$(a_1 +_{32} (a_2 +_{32} (\ldots +_{32} a_n))) = (a_n +_{32} (\ldots +_{32} (a_2 +_{32} a_1)))$$

The standard approach to proving a theorem as above is to normalize both sides of the equality using a syntactic condition on the terms involved in the summands, and finally show that the resulting normal forms are syntactically identical. To do this via rewriting (in the absence of built-in associative-commutative unification), one proves the associativity and commutativity of $+_{32}$ as oriented rewrite rules which are then repeatedly applied to reduce each side of the equality to normal form. A slight reflection would, however, indicate that such an approach is inefficient. In particular, since the rewriting in ACL2 is inside-out, the normalization requires $O(n^2)$ applications of the associativity and commutativity rules.

Our solution to the efficiency problem is to implement a clause processor *SortBVAdds* that sorts the summands in a term involving bit vector additions using an $O(n \log n)$ mergesort algorithm. The correctness of the clause processor is verified by showing that (i) mergesort returns a permutation of its input and (ii) the application of a sequence of $+_{32}$ functions produces the same result on any permutation of the summands. With this clause processor, we can obtain significant efficiency in simplifying terms involving $+_{32}$. As empirical evidence of the efficiency, the equality theorem above can be proven in 0.01 seconds for 500 summands and in 0.02 seconds for 1000 summands on a 2.6GHz Pentium IV desktop computer with 2.0GB of RAM; the naive rewriting strategy outlined above takes 11.24 seconds and 64.41 seconds respectively.

We close the discussion on *SortBVAdds* with two observations. First note that *SortBVAdds* sorts bit vector addition occurring within subexpressions of ACL2 formulas, not merely at the top level. For example, given any unary function f, it can be used to simplify $f(a_2 +_{32} a_1 +_{32} a_0)$ into $f(a_0 +_{32} a_1 +_{32} a_2)$; furthermore, such a function f can be introduced *after* the installation clause processor. Secondly, it might be possible to mimic the efficiency reported above

using ACL2's meta reasoning abilities (cf. Section 2.4). However, writing such a general-purpose efficient metafunction is awkward since meta-rules are closely tied to ACL2's inside-out rewriter.

4 Basic Unverified External Tools

Verified clause processors enable the user to employ verified reasoning code for customized clause manipulation. However, the main motivation behind the development of our interface is to connect ACL2 with (unverified) tools that are external to the theorem prover, such as Boolean satisfiability solvers and model checkers. In this section and the next, we present mechanisms to enable such connections.

Our interface for unverified tools involves a new event that enables ACL2 to recognize a clause manipulation function *tool1* as an *unverified* reasoning tool. Here *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 is the same as if *tool1* were introduced as a verified clause processor: hints can be used to invoke the function during subsequent proof search to replace a goal with a list of subgoals.

Suppose an unverified tool *tool1* is invoked to simplify a goal conjecture in the course of a proof search. What guarantees must the implementor of *tool1* provide (and the user of *tool1* trust) in order to claim that the conjecture is indeed a theorem? A sufficient guarantee is that *tool1* could have been formally defined in ACL2 together with appropriate evaluators such that the obligation shown in Fig. 3 is a theorem about *tool1*. 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 *trust tag*[8] for the user of the tool to acknowledge this burden, and (ii) a concept of *supporters* enabling the tool developer to guarantee that the logical restrictions are met. We now explain these two constructs.

The trust tag associated with the installation of an unverified tool *tool1* is a symbol (the name of the tool itself by default), which must be used to acknowledge that the applicability of *tool1* as a proof rule depends on *tool1* satisfying the logical guarantees above. The certification of a book that contains an

[8] ACL2's trust tag mechanism, introduced in Version 3.1, is quite general, with applications other than to unverified clause processors. See the topic defttag in the ACL2 documentation [5].

unverified tool, or includes (hereditarily, even locally) a book containing an unverified tool, requires the user to tag the certification command with that tool's trust tag. 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, we insist on tagging the certification of any book containing an *installation* of an unverified tool (whether subsequently used or not) for implementation convenience. Recall that the local incompatibility check (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 application of an unverified tool, we disallow the possibility of certifying a book that *locally* introduces and uses an unverified tool without acknowledging the application of the tool.

Finally we turn to *supporters*. To understand this construct, recall the problem outlined in the preceding section, demonstrating inconsistency with a verified clause processor if the evaluator axioms could be local. The problem was that the tool *simplify-foo* had built-in knowledge about some function definitions and we wanted to ensure that when it is applied for clause manipulation the axioms in the current theory match this knowledge. In the verified case, this was guaranteed by permitting the use of the tool only in a theory in which the proof obligations for its legality are met. However, suppose we want to use *simplify-foo* as an *unverified* tool in the scenario described. Then there is no obligation proven (or stated) in the logic, other than the (perhaps informal) guarantee from the implementor that it *could* be proven in principle given an appropriate formal definition of *foo*.

The *supporters* construct enables a tool implementor to insist that the axiomatic events for all functions "understood" by the tool are present in any theory in which it is used. The implementor can list the names of such axiomatic events (typically function symbols that name their definitions, for example *foo* in the suggested example for *simplify-foo*) in the *supporters* field of the event installing unverified clause processors. We will refer to these events as *supporting events*. 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.

4.1 Applications of Unverified External Tools

The connection of ACL2 with external tools has been a topic of extensive interest, and there has been significant work connecting different decision procedures with the theorem prover. These include integration with the Cadence SMV model checker [29], Zchaff and Minisat SAT solvers [30], and UCLID [31]. These connections have been successfully used to automate large-scale verifi-

cation tasks with ACL2. For instance, the UCLID connection has been used to automatically verify deep pipelined machines [32]. However, in the absence of the disciplined mechanisms that we present here, these connections have necessarily required substantial "hacking" of the ACL2 source code. The interface we present below has been implemented to enable the ACL2 user to connect with tools such as those above without imposing demands on understanding the inner workings of the theorem prover implementation. In particular, the interface is sufficient for connecting all the tools cited above. [9]

In addition to the above connections, the unverified clause processor interface has been used to implement a decision procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA) [30]. SULFA is a decidable subclass of ACL2 formulas that is composed of the primitives *if*, *consp*, *cons*, *car*, and *cdr*; a SULFA formula can also involve restricted applications of other primitives, as well as user-defined functions. The subclass is powerful enough to express finite step hardware properties specified by first order definitions.

The decision procedure for SULFA works as follows. A SULFA formula is translated into Boolean conjunctive normal form (CNF), using ACL2's internal functions to look up bodies of functions that must be unrolled; the CNF formula is given to an external SAT solver, such as zChaff, to prove or disprove. If a counterexample is found by the SAT solver, then it is converted into a counterexample to the original formula and presented to the user.

The SULFA clause processor has been applied in the verification of components of the TRIPS processor [33], a prototype multi-core processor designed and built by the University of Texas at Austin and IBM. In particular, the Verilog implementation of the protocol used to communicate between the four load store queues has been formally verified. The use of the SULFA decision procedure makes it possible to combine fully automated SAT-based decision procedures with traditional theorem proving, achieving a higher level of automation than was previously possible with the ACL2 theorem prover [34,30].

The SULFA clause processor has also been used to develop a general-purpose solver for the standard SMT theory of bit vectors; the solver performs preliminary simplification with ACL2 before invoking the SULFA clause processor to dispatch the resulting simplified problem. This tool has been used to verify fully automatically all the problems in the SMT benchmark suite [35]. We note that the close connection with ACL2 makes this SMT solver more flexible than any other solver we are aware of, albeit perhaps with some loss in efficiency. In particular, one can augment the solver with definitions of additional bit

[9] The sufficiency has been ensured by significant discussions with the authors of these previous connections in the design of the interface. Furthermore, Sudarshan Srinivasan [private communication] is working on connecting an SMT solver with ACL2 and has expressed a desire to use our interface.

vector primitives and add theorems about such primitives; these theorems are proven correct with ACL2 and then applied to efficiently simplify terms that involve the relevant primitives.

5 Unverified External Tools for Implicit Theories

Our view so far of an unverified tool is that if it replaces some clause with a list of clauses, then the provability of the resulting clauses implies the provability of the original clause. Such a tool 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. However, some ACL2 users have found the necessity to use more sophisticated tools that implement their own theories. In this section we will discuss such tools and the facilities necessary to connect them with ACL2.

To motivate the need for such tools, assume that we wish to prove a theorem about some hardware design. Hardware designs are typically implemented 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 [36,37]; however the complexity of a modern HDL often makes this approach impractical in practice [38]. On the other hand, most commercial model checkers can parse designs written in VHDL or Verilog. A practical alternative is thus 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 explicitly constrained properties of the interpreter to establish the correctness of the design.

This approach has shown promise in scaling formal verification to industrial designs. For instance, in recent work Sawada and Reeber [39] verify a VHDL implementation of an industrial floating-point multiplier using a combination of ACL2 and an IBM internal verification tool called SixthSense [40]. For this project, 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 functions are constrained only to return a bit and bit-vector respectively. The key properties of the different multiplier stages are proven using SixthSense: 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. Such properties are then composed using ACL2 to show 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 easily 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 augment the extension principles of ACL2 with a new principle called *encapsulation templates* (or simply *templates*). Function symbols introduced via templates are constrained functions analogous to those introduced via the encapsulation principle, and the conservativity of the resulting extension is analogously guaranteed by exhibiting local witnesses satisfying the constraints. However, there is one significant distinction between the encapsulation principle and templates: the constraints introduced are marked *incomplete* acknowledging that they might not encompass all the constraints on the functions. ACL2 therefore must disallow any functional instantiation that requires replacement of a function symbol introduced via templates.

With templates, we can integrate ACL2 with tools like SixthSense above. Suppose that we wish to connect ACL2 with an unverified tool *tool1* that implements a theory not defined explicitly in ACL2. We then declare *tool1* to be an unverified clause processor by providing (i) a template to introduce the function symbols (say f and g) regarding which the theory of the clause processor contains additional axioms, and (ii) stipulating that the supporters of the clause processor include f and g.

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 one can 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 conditions 1-4 below hold. Note that the guarantee is obviously outside the purview of the mechanized

reasoning system of ACL2; the obligations merely codify a set of sufficient conditions under which it is logically sound to use *tool1*.

(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) *Tool1* is correct when E is interpreted as E_P, in the following sense. Suppose *tool1* is applied to a clause C to yield a list L_C of clauses, and at any such point, let $\mathcal{T}$ be the current ACL2 theory augmented with the axioms introduced by E_P. Then: if each clause in L_C is a theorem of $\mathcal{T}$, then C is a theorem of $\mathcal{T}$.

Condition 2 is necessary to ensure that E_P does not implicitly introduce additional function symbols with constraints that might conflict with the later axiomatization of these functions. The conditions above enable us to view any template event as representing its promised encapsulation.

We note that the interface does not permit the introduction of a template that is separated from the declaration of an unverified clause processor. One might think that this is too restrictive and there is no harm in introducing a template as a separate event, with the view that every theorem proven with ACL2 (without a tool) for the introduced functions is valid for any admissible promised encapsulation of the template. That is, if a tool is later introduced supporting the template, such theorems would seem to be valid for the promised encapsulation of the tool as well. However, we disallow this possibility since it is possible to exploit such "dangling" templates to prove a contradiction.

As a simple example, consider the following scenario which would be possible without such a restriction. Suppose a tool developer develops a book `book1` consisting of the following sequence:

Template.
Introduce f with no explicit constraint.

Template.
Introduce g with no explicit constraint.

Unverified Clause Processor.
Define a tool tool1 and add g as supporter, with promised encapsulation for g providing the following constraint:
$g(x) = f(x)$

Now suppose a (possibly different) tool developer develops a book `book2` consisting of the following sequence:

Template.
Introduce g with no explicit constraint.

Template.
Introduce f with no explicit constraint.

Unverified Clause Processor.
Define a tool tool2 *and add f as supporter, with promised encapsulation for f providing the following constraint:*
$f(x) = \neg g(x)$

Thus, both `book1` and `book2` would be (independently) certifiable. Now imagine an ACL2 session that first includes `book1` and then `book2`. The inclusion of `book2` re-introduces the two template events, but since they are identical to the corresponding template events in `book1` we would not expect any complaint from ACL2. However, one could presumably prove a contradiction in this session using the conflicting promised encapsulations implemented by the two tools.

We can view the above example from a logical perspective, as follows. The logical foundation of ACL2 [13] associates each ACL2 session with a sequence of formal events (a so-called *chronology*) that captures its logical content. When a template is introduced together with a tool, that template can be viewed logically as its promised encapsulation; then no further change to the above association needs to be made. If however a template is separated from its tool, the logical view is complicated significantly. Indeed, the example above demonstrates that it is problematic to work out a logical view for templates whose promised encapsulations are delayed; in essence, there is no chronology associated with the above session.

We conclude our discussion of unverified external clause processors by saying a bit more about their logical view. As suggested above, the correspondence between ACL2 sessions and logical chronologies, as laid out in the treatment of ACL2's foundations, remains unchanged even for unverified external tools with their own theories, by viewing such tools as their promised encapsulations. An important part of the logical view is the justification of local events in a book, since these are ignored when the book is included. Since the logical view remains unchanged with unverified external tools, it would seem to be sound for the ACL2 user to mark as local either the definition or use of such a tool, and to mark as local the inclusion of a book that has such definitions or uses. This soundness claim indeed holds if those external tools are correct, that is, if each deduction they make is sound with respect to the ACL2 theory present in which the deduction is made (see Condition 4 above). So, it is important to have a way of tracking which tools we are trusting to be correct. In Section 4 we touched on how the *trust tag* mechanism can track the introduction, and

hence the use, of unverified clause processors. This tracking must be done (and is done) even through local events. But if all tracked external tools are correct, then our logical view tells us that it is indeed sound to remove local events. The use of trust tags when certifying and including books that contain unverified external tools is a small price for the ACL2 user to pay, since the use of local events is typically a key activity of ACL2 users.

6 Remarks on Implementation

Our presentation in the preceding three sections have primarily focused on the different flavors of clause processors from the point of view of a *user* interested in connecting ACL2 with other reasoning tools. In this section, we discuss some aspects of our implementation of these connections. The details of the various implementation considerations touched upon in this section are specific to the ACL2 system. However, we believe that they provide a sense of the issues involved in extending an industrial-strength theorem prover with a non-trivial feature. Note that an understanding and appreciation of some of the considerations discussed here will benefit from some exposure to the ACL2 theorem prover and perhaps some familiarity with ACL2's design architecture. The ACL2 user interested in further details is also encouraged to browse the book `books/clause-processors/basic-example.lisp` in the ACL2 distribution, which exercises several interesting corner cases of the implementation, in particular demonstrating approximately 80 helpful error messages.

6.1 Basic Design Decisions

ACL2 is a complex system with many primitive types of event commands (besides just definitions and theorems) and many implicit invariants. Augmenting such a system with a feature like an interface connecting external tools requires strict adherence to good software engineering practice, and significant care must be taken to ensure that the new feature does not interfere with the existing features in a surprising way or break any of the design invariants. To minimize complexity, both for the implementation and the user experience, the design choices involved in the implementation of any new feature to the theorem prover are based on the following two guiding principles:

- Implement the new feature by reusing as much of the existing features and infrastructures as possible.
- Disallow complex corner cases that are unlikely to be experienced by the user in a typical scenario, adding support for such cases in a demand-driven fashion as a user encounters them in practice.

We now discuss how the architecture and implementation of our interface are affected by these two design principles.

First recall that the interface for invoking clause processors is implemented as a new *hint*. Thus, one can invoke an external tool (verified or not) to simplify any subgoal that arises during a proof search, possibly a goal generated after several transformations to the user-supplied conjecture by the internal reasoning strategies of ACL2. This is in contrast with the implementation of similar features in HOL and Isabelle [41,42], where the connection is provided by a special-purpose construct that "ships off" a user-supplied conjecture to an external tool. Our choice is partially motivated to support the application of external tools to simplify subgoals arising in proofs, but it also allows the use of ACL2's fairly mature hint mechanism to be reused to support the connection. The hint mechanism enables the user to provide the theorem prover with pragmatic advice on proof search. Hints can be used to force case splits, add known facts as hypotheses, suggest the use of an appropriate induction, and so on. In addition, ACL2 supports notions of *default* and *computed* hints, which permit the user to write code that generate hints dynamically during proof search, based on the shape of the current subgoal. The reuse of the hint mechanism provides these sophisticated capabilities to the interface for free.

Now we turn to the implementation issues involved in *designating* a tool as a clause processor. For verified clause processors, we reuse ACL2's existing *rule class* mechanism, which has traditionally been used to classify theorems into categories such as rewrite rules, forward chaining rules, type prescription rules, and so on, that enable the theorem prover to apply such theorems in subsequent proofs. The new rule class designating clause processors thus fits well into this framework. When a theorem is designated as a clause processor, ACL2 associates a special property with the corresponding clause processor function, to be checked when a *clause processor* hint is supplied later.

More interesting is the designation of a function as an *unverified* clause processor, for which there is no associated theorem (or rule class) explicitly stated or stored in the theorem prover's database. In addition, recall that unverified clause processors might be associated with their own implicit theories specified by templates. To support unverified clause processors we use another general mechanism available in ACL2, called *table events*. A *table* is simply a named finite function mapping keys to values, and *table events* support table update, *i.e.*, making a new key/value association. To support unverified clause processors, we use a new built-in table (initially empty) that associates the name of each unverified clause processor with a Boolean value indicating whether it is associated with a template. A clause processor associated with a template is referred to as a *dependent* clause processor. Support for dependent clause processors, however, is tricky, principally because of the requirement of *atomicity*. Recall from Section 5 that a template event must not be separated

from its dependent clause processor. We now discuss the subtleties involved in implementing this requirement.

To provide user-level support for atomicity, we design a macro that introduces the template event together with the table event designating the dependent clause processor. The subtlety, however, arises from the fact that the logical requirement demands that we *enforce* the atomicity for these two events, not merely support it. For instance, a naive implementation of the macro might be one that expands into a sequence of events that successively introduces the template and the table event. However, in ACL2 macros do not have any semantic content, and the user can introduce these two components of the sequence separately without using the macro.

To resolve the issue above, we note that a template is an encapsulation, and thus contains a sequence of events. Our macro therefore inserts the table event *inside* the corresponding template. However, this macro is merely syntactic sugar. At a deeper level, ACL2 *defines* the notion of template (that is, defines when the encapsulation introduces incomplete constraints) as follows: an encapsulation is a template if and only if it constrains at least one new function and also introduces an unverified clause processor. Thus, while finishing the admission of an encapsulation, the implementation checks (using the new table) whether the encapsulation introduces at least one dependent clause processor (that was not present before the encapsulation). If so, then the newly-constrained functions are all marked as having incomplete constraints.

Finally, we consider one subtlety that arises as a result of the above "definition" of templates from the implementation perspective, namely the disambiguation of dependent clause processors in the context of *nested encapsulations*. Note that the sequence of events introduced by an encapsulation can include another encapsulation. Suppose that a clause processor, *tool*, is introduced in the scope of two nested encapsulations, each of which introduces at least one constrained function. Which of these two encapsulations should be considered to be the promised encapsulation associated with that dependent clause processor? Note that the implementor of *tool* needs an answer to this question in order to meet the logical burden explained in the preceding section.

Our "solution" to the above dilemma is to simply disallow the scenario! We have not found an application which provides any compelling ground to support such a situation, so the implementation simply causes an error in this case. We postpone consideration of this complexity until a user provides a specific application where such functionality makes sense, thus providing guidance for the design of the resulting extension.

Our description above focuses on basic design decisions for the clause processor interface in ACL2. We now mention briefly *engineering considerations* involved in the implementation. These considerations are important both to provide a pleasant user experience and to enforce critical logical requirements. A comprehensive treatment of the numerous choices made in the implementation of the interface is beyond the scope of this paper; we merely list a few illustrative issues to provide a flavor of the implementation complexity and choices.

Support for user flexibility: In our description of the interface, we confined ourselves to the simplistic view that a clause processor takes a clause and returns a list of clauses. The implementation, however, supports a more general notion of a *clause processor*. In particular, the implementation permits a clause processor to return multiple values, where the first value is an error flag and the second is the clause list of interest. If the application of a clause processor during a proof search causes the error flag to be set then the proof aborts. The implementation allows the error flag to be a string or a formatted string (a list with arguments for formatted output directives), which is printed when a proof aborts. The designer of a connection of ACL2 with a specific external tool can therefore use the flag to output a helpful error message. Note that the form of the theorem installing a verified clause processor is suitably adjusted for tools returning multiple values so that the proof obligations are trivial in the case that the tool returns an error. Finally, we note that when a clause-processor hint returns, without error, the one-element list containing the given clause, then the hint is considered a no-op and the proof continues.

Implementing execution support: Note that a tool introduced as a clause processor is a function introduced in ACL2. But a clause processor function requires further restrictions on how it can be introduced. The reason is that a clause processor function is *executed* on the ACL2 subgoal on which it is invoked; however, some functions introduced in ACL2 via the extension principles cannot be executed, in particular those introduced as constrained functions via encapsulation. We therefore check that a function designated to be a clause processor is executable. Furthermore, to enable efficient execution, we check that (i) the *guard* (input constraint) on a verified clause processor is trivially implied by the assumption that it is given a clause,[10] and (ii) the result of invoking a clause processor is a well-formed list of clauses. The latter is dynamically checked during the invocation of the clause processor.

[10] We also make appropriate guard checks in the presence of single-threaded objects.

Syntactic Checks and Error Messages from the Interface: One of the acclaimed strengths of the ACL2 system is the detailed feedback provided by the theorem prover in an aborted proof attempt. Such feedback is essential in the applicability of a general-purpose theorem prover in large-scale verification projects. We incorporate several syntactic checks with the interface, whose violation produces helpful feedback in the form of error messages. These include checking that (i) the *supporters* field contains a list of function symbols which have been introduced in the current theory, (ii) a template event introduces at least one new function, and (iii) the theorem installing a verified clause processor uses distinct variables for the clause and the alist arguments.

Checks in Functional Instantiation: Recall from Section 5 that due to soundness concerns, functions introduced via templates cannot be functionally instantiated. The implementation enforces such restrictions, as we now explain. Functional instantiation applies a given functional substitution to a previously proved theorem; but it requires that upon applying the substitution to the constraints on functions in its domain, the resulting formulas are all theorems. So, we need to be able to collect those constraints. During the use of functional instantiation we therefore cause an error if (i) there is a partially constrained function in the domain of the given functional substitution, or (ii) a partially constrained function makes it impossible to collect all the requisite constraints. How might the latter be possible? Consider for example introducing a formula φ as an axiom via the *defaxiom* principle. Suppose φ is allowed to mention a partially constrained function f. Since the set of constraints explicitly specified for f is incomplete, we cannot determine if the *unspecified* constraints involve a function g in the domain of the given functional substitution, whose constraints must be considered as mentioned above. Hence we do not allow a partially constrained function to support a *defaxiom* event.

Signature Checking in Templates: A template with an unverified tool is implemented with an encapsulation together with a table event designating the requisite clause processor. ACL2's encapsulation syntax includes a *signature* that specifies the set of functions being introduced by encapsulation, but as a convenience to the user, ACL2 also allows additional functions to be defined non-locally in an encapsulation. When this is done, ACL2 attempts to "move" such definitions outside the encapsulation, introducing them before or after the introduction of the functions designated by the signature in the encapsulation itself. If ACL2 fails to move such a definition then the function is introduced in the encapsulation with its definitional equation stipulated as a constraint. However, for encapsulations designated as templates, it is unclear whether the promised encapsulation intended by the tool implementor involves constraining such non-locally defined functions

or not; so we require that each function introduced by a template must be explicitly documented in the signature of the template event.

Concerns with flattening encapsulations: ACL2 has a command called `:puff` that replaces an encapsulation event with the events introduced in the encapsulation. However, recall that template events must atomically introduce both the partially constrained functions and the dependent clause processor. To enforce this, the implementation of `:puff` is modified to have no effect on encapsulations introducing clause processors. Note, however, that it is possible to "fool" `:puff` into destroying this atomicity when the template is introduced manually rather than via the user-level macro for dependent clause processors mentioned above. Fortunately, the use of `:puff` leaves a mark on the resulting ACL2 session to indicate that the proofs done in the session cannot be trusted; `:puff` is used only for "hacking" during interactive proof development. This restriction on `:puff` is thus merely to support a reasonable hacking experience.

We conclude the discussion of the implementation with a brief remark on its impact. As the different concerns above illustrate, developing a sound interface for connecting a mature theorem prover like ACL2 with other tools requires significant attention to the interplay of the interface with features already existing in the system. It can therefore be dangerous (and potentially unsound) to embark on such a connection by modifying the source code of the theorem prover without adequate understanding of the relevant subtleties. This in turn underlines the significance of providing a disciplined mechanism that enables the user to build such connections without the overhead of acquiring a deep understanding of the internals of the theorem prover's source code and subtle logical issues. However, prior to the development of the interface described here, an ACL2 user endeavoring to use external tools in proofs was left with no choice but to hack the internals of the system. With the new feature, a user now has the ability to integrate ACL2 smoothly with other tools by employing the user-level mechanisms provided by the interface.

7 Related Work

The importance of providing means for connecting with 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" [43], 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 provers. The PVS theorem

prover provides connections with several decision procedures such as model checkers and SAT solvers [44,45]. The Isabelle theorem prover [24] 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 [42,46]. Oracles are also used in the HOL family of higher order logic theorem provers [23]; for instance, the PROSPER project [47] 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 [48], and also uses that oracle interface to connect HOL4 with ACL2 as mentioned in the next section. Meng and Paulson [49] interface Isabelle with a resolution theorem prover.

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 [41]. 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 issue is not significant 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 the *uses* of such an event, especially since hints are ignored when including previously certified books.

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 [50] describes such an interface connecting HOL with first-order logic. McCune and Shumsky [51] 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.

Finally, as mentioned in Section 4, several ACL2 users have integrated external tools with ACL2, albeit requiring implementation hacks on the ACL2 source code. Ray, Matthews, and Tuttle integrate ACL2 with SMV [29]. Reeber and Hunt connect ACL2 with the Zchaff satisfiability solver [30], and Sawada and Reeber provide a connection with SixthSense [39]. Manolios and Srinivasan connect ACL2 with UCLID [31,32].

8 Conclusion and Future Work

Different deduction tools bring 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. But tools based on decision procedures, when applicable, typically provide more automated proof procedures than general purpose provers. Several ACL2 users have requested ways to connect ACL2 with automated decision procedures. The mechanisms described in this paper support a disciplined approach for using ACL2 with other tools, with a clear specification of the expectations from the tool in order to maintain soundness. Furthermore, the design of verified clause processors allows the user to control a proof through means other than ACL2's heuristics.

We have presented an interface for 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 [52]. Preliminary experiments with our interface have been promising, and we expect that ACL2 users will find many ways to decompose problems that take advantage of the new interface, in effect creating new tools that are stronger than their components.

Our interface may perhaps be criticized on the grounds that developing a connection with an external tool requires knowledge of ACL2. But a connection between different formal tools must involve a connection between two logics, and the developer of such a connection must have a thorough understanding of both the logics, including the legal syntax of terms, the axioms, and the rules of inference. Note that the logic of ACL2 is 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; the interface requires constructs such as *supporters* essentially to enable tool developers to provide logical guarantees in the presence of local events. However, we believe that these constructs make it possible to implement connections with ACL2 without unreasonable demands on understanding the theorem prover implementation or esoteric aspects of the ACL2 logic.

Finally, the restrictions for the tool developers that we have outlined in this paper preclude certain external deduction tools. For instance, there has been recent work connecting HOL with ACL2 [53], enabling 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. However, HOL cannot be connected using our interface in a way in which the obligations outlined for the developer of the connection can be met. To understand why, note that the obligations ensure that the theory used

by the external tool could *in principle* be formalized in ACL2. For instance, if template events are used to connect ACL2 with a tool whose theory is incompletely formalized, the burden is on the developer of the connection to guarantee that every theorem proven by the tool is a theorem of the theory obtained by replacing the template with its promised encapsulation. Since the logic of ACL2 is first order, this requirement rules out connections with logics stronger than first order logic. 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 viable 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

We thank Peter Dillinger, Robert Krug, Pete Manolios, John Matthews, Sudarshan Srinivasan, and Rob Sumners for numerous comments and feedback. In particular, Dillinger made crucial contributions in the design of ACL2's trust tag mechanism, which serves as a foundation for our interface connecting unverified tools with ACL2, and Jared Davis implemented and verified the *equalitySubstitution* clause processor described in the paper. The anonymous referees also made several important expository suggestions.

References

[1] M. Kaufmann, J. S. Moore, S. Ray, E. Reeber, Integrating External Deduction Tools with ACL2, in: C. Benzmüller, B. Fischer, G. Sutcliffe (Eds.), Proceedings of the 6th International Workshop on Implementation of Logics (IWIL 2006), Vol. 212 of CEUR Workshop Proceedings, 2006, pp. 7–26.

[2] K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.

[3] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, in: Proceedings of the 38th Design Automation Conference (DAC 2001), ACM Press, 2001, pp. 530–535.

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

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

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

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

[8] 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 (1998) 148–200.

[9] D. Russinoff, A. Flatau, RTL Verification: A Floating Point Multiplier, in: M. Kaufmann, P. Manolios, J. S. Moore (Eds.), Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers, Boston, MA, 2000, pp. 201–232.

[10] A. Flatau, M. Kaufmann, D. F. Reed, D. Russinoff, E. W. Smith, R. Sumners, Formal Verification of Microprocessors at AMD, in: M. Sheeran, T. F. Melham (Eds.), 4th International Workshop on Designing Correct Circuits (DCC 2002), Grenoble, France, 2002.

[11] B. Brock, 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), IEEE Computer Society Press, Austin, TX, 1997, pp. 31–36.

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

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

[14] R. S. Boyer, J. S. Moore, Metafunctions: Proving them Correct and Using Them Efficiently as New Proof Procedure, in: R. S. Boyer, J. S. Moore (Eds.), The Correctness Problem in Computer Science, Academic Press, London, UK, 1981, pp. 103–184.

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

[16] A. Church, S. C. Kleene, Formal Definitions in the Theory of Ordinal Numbers, Fundamenta Mathematicae 28 (1937) 11–21.

[17] H. Rogers, Jr., Theory of Recursive Functions and Effective Computability, MIT Press, 1987.

[18] P. Manolios, D. Vroon, Algorithms for Ordinal Arithmetic, in: F. Baader (Ed.), Proceedings of the 19th International Conference on Automated Deduction (CADE 2003), Vol. 2741 of LNAI, Springer-Verlag, Miami, FL, 2003, pp. 243–257.

[19] R. S. Boyer, D. Goldshlag, M. Kaufmann, J. S. Moore, Functional Instantiation in First Order Logic, in: V. Lifschitz (Ed.), Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, Academic Press, 1991, pp. 7–26.

[20] W. J. Legato, Generic Theories as Proof Strategies: A Case Study for Weakest Precondition Style Proofs, in: M. Kaufmann, J. S. Moore (Eds.), 5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004), Austin, TX, 2004.

[21] S. Ray, J. S. Moore, Proof Styles in Operational Semantics, in: A. J. Hu, A. K. Martin (Eds.), Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004), Vol. 3312 of LNCS, Springer-Verlag, Austin, TX, 2004, pp. 67–81.

[22] J. Matthews, J. S. Moore, S. Ray, D. Vroon, Verification Condition Generation Via Theorem Proving, in: M. Hermann, A. Voronkov (Eds.), Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006), Vol. 4246 of LNCS, 2006, pp. 362–376.

[23] M. J. C. Gordon, T. F. Melham (Eds.), Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic, Cambridge University Press, 1993.

[24] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher Order Logics, Vol. 2283 of LNCS, Springer-Verlag, 2002.

[25] M. Kaufmann, Modular Proof: The Fundamental Theorem of Calculus, in: M. Kaufmann, P. Manolios, J. S. Moore (Eds.), Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers, Boston, MA, 2000, pp. 59–72.

[26] G. L. Steele, Jr., Common Lisp the Language, 2nd Edition, Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.

[27] W. A. Hunt, Jr., M. Kaufmann, R. Krug, J. S. Moore, E. Smith, Meta reasoning in ACL2, in: J. Hurd, T. Melham (Eds.), Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005), Vol. 3603 of LNCS, Springer-Verlag, Oxford, England, 2005, pp. 163–178.

[28] J. Harrison, Metatheory and Reflection in Theorem Proving: A Survey and Critique, Tech. Rep. CRC-053, SRI International Cambridge Computer Science Research Center (1995).

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

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

[31] P. Manolios, 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), IEEE Computer Society Press, Paris, France, 2004, pp. 168–175.

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

[33] D. Burger, S. W. Keckler, K. S. M. M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, Scaling to the End of Silicon with EDGE Architectures, IEEE Computer 37 (7) (2004) 44–55.

[34] W. A. Hunt, Jr., E. Reeber, Formalization of the DE2 Language, in: W. Paul (Ed.), Proceedings of the 13th Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005), LNCS, Springer-Verlag, Saarbrücken, Germany, 2005, pp. 20–34.

[35] S. Ranise, C. Tinelli, The SMT-LIB Standard: Version 1.2, Tech. rep., Department of Computer Science, The University of Iowa, available at `www.SMT-LIB.org` (2006).

[36] B. Brock, W. A. Hunt, Jr., The Dual-Eval Hardware Description Language, Formal Methods in Systems Design 11 (1) (1997) 71–104.

[37] D. Russinoff, A Formalization of a Subset of VHDL in the Boyer-Moore Logic, Formal Methods in Systems Design 7 (1/2) (1995) 7–25.

[38] M. J. C. Gordon, The Semantic Challenges of Verilog HDL, in: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS 1995), IEEE Computer Society Press, 1995, pp. 136–145.

[39] J. Sawada, E. Reeber, ACL2SIX: A Hint used to Integrate a Theorem Prover and an Automated Verification Tool, in: A. Gupta, P. Manolios (Eds.), Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006), Springer-Verlag, San Jose, CA, 2006, pp. 161–168.

[40] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, A. Kuehlmann, Scalable Automated Verification via Exper-System Guided Transformations, in: A. J. Hu, A. K. Martin (Eds.), Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004), Vol. 3312 of LNCS, Springer-Verlag, 2004, pp. 217–233.

[41] E. L. Gunter, Adding External Decision Procedures to HOL90 Securely, in: J. Grundy, M. C. Newey (Eds.), Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1998), Vol. 1479 of LNCS, Springer-Verlag, 1998, pp. 143–152.

[42] O. Müller, T. Nipkow, Combining Model Checking and Deduction of I/O-Automata, in: E. Brinksma (Ed.), Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of

Systems (TACAS 1995), Vol. 1019 of LNCS, Springer-Verlag, Aarhus, Denmark, 1995, pp. 1–16.

[43] J. D. Guttman, A Proposed Interface Logic for Verification Environments, Tech. Rep. M-91-19, The Mitre Corporation (Mar. 1991).

[44] S. Rajan, N. Shankar, M. K. Srivas, An Integration of Model-Checking with Automated Proof Checking, in: P. Wolper (Ed.), Proceedings of the 8th International Conference on Computer-Aided Verification (CAV '95), Vol. 939 of LNCS, Springer-Verlag, 1995, pp. 84–97.

[45] N. Shankar, Using Decision Procedures with Higher Order Logics, in: R. J. Boulton, P. B. Jackson (Eds.), Proceedings of the 14th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2001), Vol. 2152 of LNCS, Springer-Verlag, 2001, pp. 5–26.

[46] D. Basin, S. Friedrich, Combining WS1S and HOL, in: D. M. Gabbay, M. de Rijke (Eds.), Frontiers of Combining Systems 2, Research Studies Press/Wiley, Baldock, Herts, UK, 2000, pp. 39–56.

[47] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, T. F. Melham, The PROSPER toolkit, in: S. Graf, M. Schwartbach (Eds.), Proceedings of the 6th International Conference on Tools and Algorithms for Constructing Systems (TACAS 2000), Vol. 1785 of LNCS, Springer-Verlag, Berlin, Germany, 2000, pp. 78–92.

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

[49] J. Meng, L. C. Paulson, Experiments on Supporting Interactive Proof Using Resolution, in: D. A. Basin, M. Rusinowitch (Eds.), Proceedings of the 2nd International Joint Conference on Computer-Aided Reasoning (IJCAR 2004), Vol. 3097 of LNCS, 2004, pp. 372–384.

[50] J. Hurd, An LCF-Style Interface between HOL and First-Order Logic, in: A. Voronkov (Ed.), Proceedings of the 18th International Conference on Automated Deduction (CADE 2002), Vol. 2392 of LNCS, Springer-Verlag, 2002, pp. 134–138.

[51] W. McCune, O. Shumsky, Ivy: A Preprocessor and Proof Checker for First-Order Logic, in: P. Manlolios, M. Kaufmann, J. S. Moore (Eds.), Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers, Boston, MA, 2000, pp. 217–230.

[52] M. Kaufmann, J. S. Moore, Should We Begin a Standardization Process for Interface Logics?, Tech. Rep. 72, Computational Logic Inc. (CLI) (Jan. 1992).

[53] M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, J. Reynolds, An Integration of HOL and ACL2, in: A. Gupta, P. Manolios (Eds.), Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006), IEEE Computer Society Press, 2006, pp. 153–160.