Toward a Practical Module System for ACL2

Carl Eastlund and Matthias Felleisen

Northeastern University
Boston, Massachusetts, U.S.A.
{cce,matthias}@ccs.neu.edu

Abstract. Boyer and Moore's ACL2 theorem prover combines first-order applicative Common Lisp with a computational, first-order logic. While ACL2 has become popular and is being used for large programs, ACL2 forces programmers to rely on manually maintained protocols for managing modularity. In this paper, we present a prototype of Modular ACL2. The system extends ACL2 with a simple, but pragmatic functional module system. We provide an informal introduction, sketch a formal semantics, and report on our first experiences.

1 A Logic for Common Lisp, Modules for ACL2

In the early 1980s, the Boyer and Moore team decided to re-build their Nqthm theorem prover [1] for a first-order, functional sub-language of a standardized, industrial programming language: Common Lisp [2]. It was an attempt to piggyback theorem proving on the expected success of Lisp and functional programming. Although Common Lisp didn't succeed, the ACL2 system became the most widely used theorem prover in industry. Over the past 20 years, numerous hardware companies and some software companies turned to ACL2 to verify critical pieces of their products [3]; by 2006, their contributions to the ACL2 regression test suite amounted to over one million lines of code. The ACL2 team received the 2005 ACM Systems Award for their achievement.[1]

During the same 20 years, programming language theory and practice have evolved, too. In particular, programming language designers have designed, implemented, and experimented with numerous module systems for managing large functional programs [4]. One major goal of these design efforts has been to help programmers reason locally about their code. That is, a module should express its expectations about imports, and all verification efforts for definitions in a module should be conducted with respect to these expectations. Common Lisp and thus ACL2, however, lack a proper module system. Instead, ACL2 programmers emulate modular programming with Common Lisp's namespace management mechanisms, or by hiding certain program fragments from the theorem prover. Naturally, the manual maintenance of abstraction boundaries is difficult and error prone. Worse, it forces the programmer to choose between local reasoning and end-to-end execution, as functions hidden from the theorem prover cannot be run.

[1] campus.acm.org/public/pressroom/press_releases/3_2006/software.cfm

A. Gill and T. Swift (Eds.): PADL 2009, LNCS 5418, pp. 46–60, 2009.

Over the past year, we have investigated the design of a module system for ACL2. Specifically, we have extended ACL2's language with modules and produced two translations for modular programs: a compiler to ACL2 executables and a logic translator to ACL2 proof obligations. With the latter, programmers can now reason locally about individual modules. One goal is to empower ACL2 programmers with large code bases to gradually migrate their monolithic program into a modular world. Another goal is to expand Rex Page's [5] use of this industrial-strength theorem prover in software engineering courses to teach theorem proving in a modular setting. Without modules, such a software engineering course simply isn't convincing enough.

This paper is our first report on bringing this module technology to ACL2. In section 2, we demonstrate our module system and its prototype implementation. In section 3, we present our formal model of the module system. We have also implemented several projects as modules; in section 4 we describe the positive and negative outcomes of these experiments. Section 5 presents related work, and the last section sketches our future challenges.

2 Reasoning with Modules

ACL2. The ACL2 theorem prover is similar to a LISP read-eval-print loop; it accepts *events* such as function definitions or logical conjectures from the user, verifies each in turn, and updates the logical state for the next event. Its interface is purely text-based; the system comes with an Emacs mode as the preferred interface for professional ACL2 users.

Four years ago, Rex Page (Oklahoma University) started the ambitious effort of teaching a senior-level course sequence on software engineering in ACL2 [5]. Students reported difficulty with the text-based interface to ACL2; in response, Felleisen and Vaillancourt produced Dracula [6] as a graphical user interface for ACL2. Dracula has since been used in courses on software engineering and symbolic logic [7].

Dracula. Dracula is a language level in the DrScheme integrated development environment. It provides a simulation of Applicative Common Lisp (ACL), the executable component of ACL2. Dracula incorporates DrScheme's usual programming tools, including a syntax checker, stack traces, unit testing, and a functional, graphical toolkit geared toward novice programmers. It provides an interface to the ACL2 theorem prover for the logical component.

Figure 1 shows a screenshot of Dracula in action. The left-hand side of the Dracula interface provides two windows for formulating and executing programs: the definitions window, where users edit their programs, and the interactions window, where users may try out their functions.

The right-hand side of the display is Dracula's interface to the ACL2 theorem prover. It provides buttons to invoke ACL2 and to send each term from the definitions window to the theorem prover. Dracula paints the terms green when ACL2 proves them sound and red when it fails. Green terms are locked from further editing to faithfully represent ACL2's logical state; users may edit red

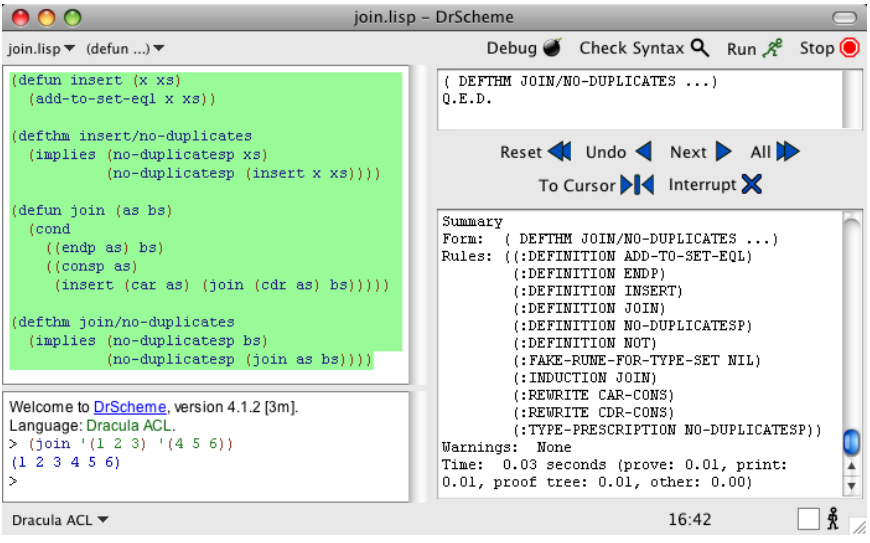

Fig. 1. The Dracula graphical user interface

terms or undo the admission of green terms to edit those. Below the control buttons, Dracula shows the theorem prover's output; above them, it shows a proof tree, naming key checkpoints for quick diagnosis of a failed attempt.

Figure 1 shows a program with two functions and two theorems. The functions are *insert*, which adds a single element to a set, and *join*, which adds multiple elements. The theorems *insert/no-duplicates* and *join/no-duplicates* state that the functions preserve the the uniqueness of set elements.

Dracula's simulation of ACL ignores the theorems, as they are logical rather than executable, and compiles the rest. As we can see in the interactions window, *join* produces the expected output when given '(1 2 3) and '(4 5 6) as input.

In contrast, the ACL2 theorem prover attempts to verify the logical soundness of each term successively. First it checks *insert*, which it must prove terminating for all inputs—a requirement of all functions in ACL2's logic. Next ACL2 checks *insert/no-duplicates*, for which it must prove that the conjecture expression produces a true value (non-**nil**). Free variables in **defthm** conjectures (such as x and xs) are implicitly universally quantified over all ACL2 values. ACL2 repeats the verification process for *join* and *join/no-duplicates*.

ACL2 successfully admits all these terms. The ACL2 output window displays a list of rules used in the proof of *join/no-duplicates*. The list includes the definitions of *insert* and *add-to-set-eql*, but not *insert/no-duplicates*. Rather than using the lemma proved above to reason about *join*, ACL2 re-examined the definition of *insert* to prove the uniqueness of its elements. The theorem prover's search strategies often prefer to delve into a function definition rather than use an existing lemma, resulting in duplicated proofs that span several functions.

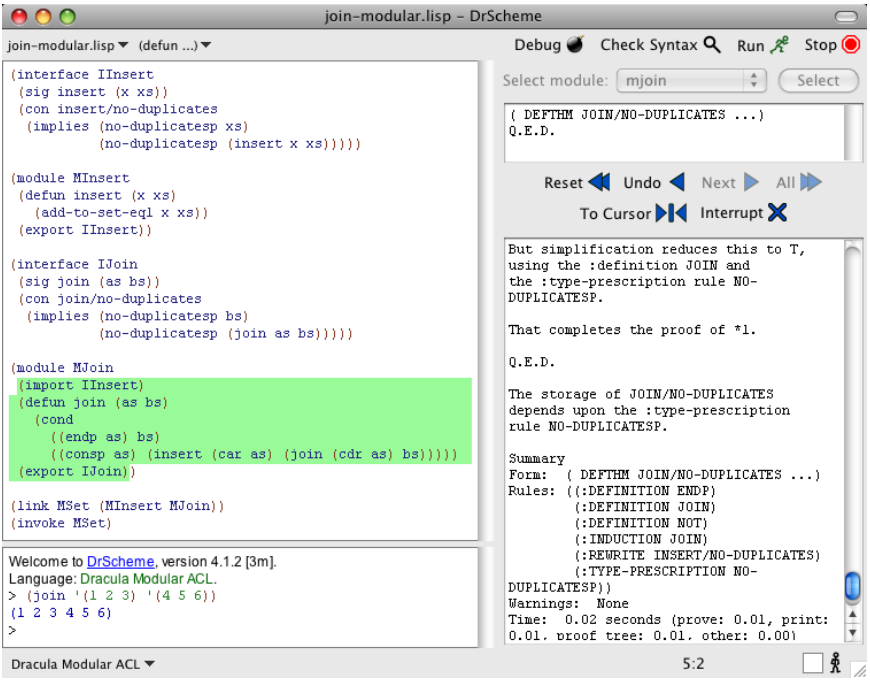

Fig. 2. A modular program in Dracula

Modular ACL2. Figure 2 shows a version of the *join* program in our new language, Modular ACL2. The definitions window contains two **interface**s, two **module**s, a **link** clause, and an **invoke** clause.

Interfaces contain **sig**natures and **con**tracts. A signature declares a function, providing its name and argument list. A contract declares a logical property that may refer to the signatures. Interfaces may also **include** other interfaces. This allows them to refer to other signatures in their contracts, extending them with new properties or stating relationships between multiple interfaces. The *IInsert* interface contains a signature *insert* and a contract *insert/no-duplicates*. They have the same arity and state the same property as the previous *insert* and *insert/no-duplicates*, but the interface does not provide a definition for *insert*. The *IJoin* interface similarly contains a signature and the *join/no-duplicates* contract for *join*.

Modules contain definitions, **import** clauses, and **export** clauses. The **import** and **export** clauses each name an interface. Definitions form the body of the module; they may refer to functions from imported interfaces, and rely on the properties declared by imported contracts. Conversely, the body of the module must define all functions declared in exported modules in a way that satisfies the associated contracts. A **link** clause constructs a new module from two existing

modules. The exports of all the modules are combined, and the imports of each module are connected to the matching exports of any prior module.[2]

The *MInsert* module contains the same definition of *insert* we saw before and exports *IInsert*. This obligates *insert* to satisfy *insert/no-duplicates*. The *MJoin* module imports *IInsert*. This allows it to call the binary function *insert* and assume *insert/no-duplicates* holds. It then defines *join* as before, and exports *IJoin*. Once again, *join* must satisfy *join/no-duplicates*. This time, however, its soundness is not with respect to a concrete definition of *insert*, but rather with respect to the imported signature and its associated contract.

The *MSet* module in our example provides *IInsert* from *MInsert* and *IJoin* from *MJoin*; the reference to *insert* in *MJoin* is resolved to the definition in *MInsert*. Linking is applicative; the original *MJoin* is unchanged and may later be linked to a different implementation of *IInsert*. Finally, our example program **invoke**s *MSet*, making its exported functions available outside the module.

As with standard ACL, Dracula compiles the modular program to an executable form and disregards the logical aspects. It compiles *insert* and *join*, links them together, and provides them for use in the interactions window.

Reasoning locally. The ACL2 GUI allows the user to verify each module separately using the theorem prover. Once the user selects a module, Dracula provides ACL2 with *stubs* (abstract functions) representing its imported signatures and *axioms* (unproven logical rules) asserting its imported contracts. Dracula then passes the body of the module to ACL2. Once that is admitted, it sends ACL2 a theorem corresponding to each exported contract. If ACL2 admits all three stages—stubs and axioms for imports, body definitions, and theorems for exports—the module is guaranteed to satisfy its export interface for any sound implementation of its import interface.

The presence of stubs and axioms may seem troubling; these are unverified assumptions added to ACL2's logical state. Using them is sound with respect to a fully linked program, however. The interface imported by one module must be linked to an export from another, so contracts assumed as axioms in one module must be proved as theorems in another before the whole program is verified.

Dracula only admits *primitive* modules, such as *MInsert* and *MJoin*, via ACL2. It safely disregards *linked* modules, such as *MSet*; once *MInsert* and *MJoin* have been verified separately, they can be linked to any module with a matching interface without need for re-verification.

In figure 2, we see that ACL2 has admitted *MJoin*. This time the proof of *join/no-duplicates* does not refer to the definitions of *insert* or *add-to-set-eql*; instead, it uses the imported contract *insert/no-duplicates*.

Manual modularization in ACL2. ACL2 has mechanisms for abstract reasoning and proof reuse. Certain definitions in a book (separate file) or **encapsulate** block (lexical scope) may be declared **local**, which hides some or all of their definition from the remaining proof, but renders them unexecutable as well.

[2] As ACL2 does not allow forward references, neither do linked modules; this prevents cyclic definitions and preserves each module's termination proofs.

These abstract proofs may later be applied to concrete functions, but the rules must be applied on a theorem-by-theorem basis, and no executable content is reused. Logical rules may be selectively disabled in the global theory, but they may be re-enabled later, defeating abstraction boundaries.

Worse, these mechanisms require the programmer to maintain the invariants of an abstraction boundary *manually*, setting up a "negative interface" by declaring which logical entities are not available for reasoning rather than which are. ACL2 can simulate a normal, "positive interface" by layering these mechanisms, but not a reusable, externally stated one.

3 The Dual Semantics of Modules

The purpose of our module system is to enable programmers to develop units of code in isolation and to reason about them independently. This informal specification implies the need for two additions to core ACL2: modules and interfaces. For an untyped language such as ACL2, a module consists of definitions and manages the scope of names. An interface describes the functions that a module provides in terms of signatures and contracts, which play the role of both obligations on the exporting module and promises for the importing one.

Naturally a module can use the services of another module, i.e., it can import functions and rely on the contracts that hold for them. Using just those contracts and the definitions in the module, a programmer must be able to verify the module's export interface. That is, it is the task of the module system to reformulate the imported contracts and the module body so that the ACL2 theorem prover can verify the exported contracts from these premises.

Another design choice concerns the connection between modules. One alternative is to used fixed links between modules, specified via interfaces. The other one is to think of modules as relations from interfaces to interfaces and to link modules separately. Based on our experience with Scheme units [8,9] and ML functors [4], we have chosen the second alternative. Finally, we also decided to separate module invocation from module linking. The rest of the section presents a model of Modular ACL2, its syntax and two semantic mappings.

Syntax. Figure 3 shows the core syntax of ACL2 and Modular ACL2. ACL2 has two variable namespaces: function parameters and local variables (v), and functions and theorems (n). Modular ACL2 introduces a third namespace for modules and interfaces (N).

An ACL2 program consists of of a sequence of *def*initions and *expr*essions. Definitions give names to functions, stubs, theorems, or axioms, or may in turn be a sequence of other definitions. Expressions include variables, literal constants, function application, conditionals, and variable bindings.

Modular programs consist of a sequence of top-level forms including interface definitions, primitive module definitions, linking specifications, module invocations, and expressions from the core language. An **interface** contains *Spec*ifications, including **sig**natures, **con**tracts, and other **include**d interfaces, as described in section 2. A primitive **module** contains a sequence of *Def*initions,

$$
\begin{array}{rcl}
\mathit{prog} & = & \mathit{top} \ldots \\
\mathit{top} & = & \mathit{def} \mid \mathit{expr} \\
\mathit{def} & = & (\textbf{defun}\ n\ (v \ldots)\ \mathit{expr}) \\
 & \mid & (\textbf{defstub}\ n\ (v \ldots)\ \textbf{t}) \\
 & \mid & (\textbf{defthm}\ n\ \mathit{expr}) \\
 & \mid & (\textbf{defaxiom}\ n\ \mathit{expr}) \\
 & \mid & (\textbf{progn}\ \mathit{def} \ldots) \\
\mathit{expr} & = & v \mid \mathit{const} \\
 & \mid & (n\ \mathit{expr} \ldots) \\
 & \mid & (\textbf{cond}\ (\mathit{expr}\ \mathit{expr}) \ldots) \\
 & \mid & (\textbf{let}\ ((v\ \mathit{expr}) \ldots)\ \mathit{expr}) \\
\mathit{const} & = & \textbf{t} \mid \textbf{nil} \mid \mathit{number} \mid \mathit{string}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Prog} & = & \mathit{Top} \ldots \\
\mathit{Top} & = & \mathit{Ifc} \mid \mathit{Mod} \mid \mathit{Link} \mid \mathit{Inv} \mid \mathit{expr} \\
\mathit{Ifc} & = & (\textbf{interface}\ N\ \mathit{Spec} \ldots) \\
\mathit{Mod} & = & (\textbf{module}\ N\ \mathit{Def} \ldots) \\
\mathit{Link} & = & (\textbf{link}\ N\ (N\ N)) \\
\mathit{Inv} & = & (\textbf{invoke}\ N) \\
\mathit{Spec} & = & (\textbf{sig}\ n\ (v \ldots)) \\
 & \mid & (\textbf{con}\ n\ \mathit{expr}) \\
 & \mid & (\textbf{include}\ N) \\
\mathit{Def} & = & \mathit{Imp} \mid \mathit{Exp} \mid \mathit{def} \\
\mathit{Imp} & = & (\textbf{import}\ N) \\
\mathit{Exp} & = & (\textbf{export}\ N\ (n\ n) \ldots)
\end{array}
$$

Fig. 3. The core grammars of ACL2 (left) and Modular ACL2 (right)

extended from ACL2 to allow **import**s and **export**s via interfaces. Exported interfaces allow renaming, in case the internal and external names of a function differ. A compound module **link**s together two other modules.[3] Fully-linked modules—those whose imports have all been resolved—may be **invoke**d, making their declared exports available to top level expressions.

Dual Semantics. Modular ACL2 programs can be verified logically, and they can be executed. For this reason, modules in a program are either translated into ACL2 proof obligations, or linked together and run as an ACL2 program.

The two semantics are closely related, so that verification has meaning with respect to execution. Specifically, once a module is verified, its exports are guaranteed to satisfy their contracts whenever the implementations of their imports satisfy theirs as well. Put another way, once every module in a program has been verified, every contract must hold true at run-time.

We do not present the straightforward description of a static semantics for determining the syntactic well-formedness of programs. In order for a Modular ACL2 program to translate to well-formed ACL2, it must avoid forward references, name clashes within interfaces and modules, modules that import one interface without importing another that it includes, and a few other errors.

Logical Semantics. A Modular ACL2 program is verified by tranforming each primitive module into an ACL2 proof obligation stating that its definitions satisfy its exported contracts, predicated on the correctness of its imports. We represent this transformation as the function L (for "Logical") that consumes a Modular ACL2 program and produces a sequence of ACL2 programs, one for each module. Figure 4 shows the definition of L and its auxiliary functions.

The L function transforms a program by invoking LT with two accumulators: a list of interfaces and a list of obligations. This function traverses the top-level definitions of a modular program. Each interface LT encounters is added to Γ. Each primitive module is transformed into a proof obligation; the proof

[3] In our full implementation, imported interfaces allow renaming as well, and compound modules may link any number of modules.

$$
\begin{array}{l}
\hline
L : \mathit{Prog} \to \mathit{prog}\ldots \\
L(\mathit{Prog}) = LT(\epsilon, \mathit{Prog}, \epsilon) \\
\hline
LT : (\mathit{Ifc}\ldots, \mathit{Top}\ldots, \mathit{prog}\ldots) \to \mathit{prog}\ldots \\
LT(\Gamma, \epsilon, \Phi) = \Phi \\
LT(\Gamma, \mathit{Ifc}\ \mathit{Top}\ldots, \Phi) = LT(\Gamma\ \mathit{Ifc}, \mathit{Top}\ldots, \Phi) \\
LT(\Gamma, \mathit{Mod}\ \mathit{Top}\ldots, \Phi) = LT(\Gamma, \mathit{Top}\ldots, \Phi\ LM(\Gamma, \mathit{Mod})) \\
LT(\Gamma, \mathit{Top}_0\ \mathit{Top}\ldots, \Phi) = LT(\Gamma, \mathit{Top}\ldots, \Phi) \text{ where } \mathit{Top}_0 = \mathit{Link} \mid \mathit{Inv} \mid \mathit{expr} \\
\hline
LM : (\mathit{Ifc}\ldots, \mathit{Mod}) \to \mathit{prog} \\
LM(\Gamma, (\textbf{module}\ N\ \mathit{Def}\ \ldots)) = LD(\Gamma, \epsilon, \mathit{Def}\ldots, \epsilon) \\
\hline
LD : (\mathit{Ifc}\ldots, n \to n, \mathit{Def}\ldots, \mathit{def}\ldots) \to \mathit{prog} \\
LD(\Gamma, \rho, \epsilon, \Delta) = \Delta \\
LD(\Gamma, \rho, \mathit{def}\ \mathit{Def}\ldots, \Delta) = LD(\Gamma, \rho, \mathit{Def}\ldots, \Delta\ \mathit{def}) \\
LD(\Gamma, \rho, (\textbf{import}\ N)\ \mathit{Def}\ldots, \Delta) = LD(\Gamma, \rho, \mathit{Def}\ldots, \Delta\ LI(\mathit{Spec}\ldots, \epsilon)) \\
\qquad \text{where } \Gamma(N) = (\textbf{interface}\ N\ \mathit{Spec}\ \ldots) \\
LD(\Gamma, \rho, (\textbf{export}\ N\ (n_1\ n_2)\ \ldots)\ \mathit{Def}\ldots, \Delta) = LD(\Gamma, \rho[n_2/n_1\ldots], \mathit{Def}\ldots, \Delta\ \Delta') \\
\qquad \text{where } \Gamma(N) = (\textbf{interface}\ N\ \mathit{Spec}\ \ldots) \\
\qquad \text{and } LE(\rho', \mathit{Spec}\ldots, \epsilon) = \Delta' \\
\hline
LI : (\mathit{Spec}\ldots, \mathit{def}\ldots) \to \mathit{def}\ldots \\
LI(\epsilon, \Delta) = \Delta \\
LI((\textbf{include}\ N)\ \mathit{Spec}\ldots, \Delta) = LI(\mathit{Spec}\ldots, \Delta) \\
LI((\textbf{sig}\ n\ (v\ \ldots))\ \mathit{Spec}\ldots, \Delta) = LI(\mathit{Spec}\ldots, \Delta\ (\textbf{defstub}\ n\ (v\ \ldots)\ \textbf{t})) \\
LI((\textbf{con}\ n\ e)\ \mathit{Spec}\ldots, \Delta) = LI(\mathit{Spec}\ldots, \Delta\ (\textbf{defaxiom}\ n\ e)) \\
\hline
LE : (n \to n, \mathit{Spec}\ldots, \mathit{def}\ldots) \to \mathit{def}\ldots \\
LE(\rho, \epsilon, \Delta) = \Delta \\
LE((\textbf{include}\ N)\ \mathit{Spec}\ldots, \Delta) = LE(\rho, \mathit{Spec}\ldots, \Delta) \\
LE(\rho, (\textbf{sig}\ n\ (v\ \ldots))\ \mathit{Spec}\ldots, \Delta) = LE(\rho, \mathit{Spec}\ldots, \Delta) \\
LE(\rho, (\textbf{con}\ n\ e)\ \mathit{Spec}\ldots, \Delta) = LE(\rho, \mathit{Spec}\ldots, \Delta\ (\textbf{defthm}\ \rho[\![n]\!]\ \rho[\![e]\!])) \\
\hline
\end{array}
$$

Fig. 4. Translation from Modular ACL2 to one proof obligation per module

obligation is added to Φ. Link clauses, invocations, and expressions are ignored, as they carry no additional logical obligations.

Within the definition of L and its helpers, Γ is treated as an environment. The notation $\Gamma(N)$ represents looking up an interface by name.

The LM function converts a module to a proof obligation by calling LD on its internal definitions. The LD function traverses the module's definitions, accruing a substitution that associates external names with internal names as declared by **export** clauses, and a list Δ of resulting definitions. The function converts imported signatures and contracts to stubs and axioms with LI and exported contracts to conjectures (**defthm**) with LE respectively. Regular ACL2 definitions are left as-is.

Executable Semantics. In addition to a logical meaning, we also need a regular run-time semantics for modular programs. Modular ACL2 programs are translated to executable form by two main processes. One is the successive

$E : \mathit{Prog} \to \mathit{prog}$

$E(\mathit{Prog}) = ET(\epsilon, \epsilon, \mathit{Prog}, \epsilon)$

$ET : (\mathit{Top} \ldots, n \to n, \mathit{Top} \ldots, \mathit{def} \ldots) \to \mathit{prog}$

$$
\begin{aligned}
ET(\Gamma, \rho, \epsilon, \Delta) &= \Delta \\
ET(\Gamma, \rho, \mathit{Ifc}\ \mathit{Top} \ldots, \Delta) &= ET(\Gamma\ \mathit{Ifc}, \rho, \mathit{Top} \ldots, \Delta) \\
ET(\Gamma, \rho, \mathit{Mod}\ \mathit{Top} \ldots, \Delta) &= ET(\Gamma\ EM(\Gamma, \mathit{Mod}), \rho, \mathit{Top} \ldots, \Delta) \\
ET(\Gamma, \rho, \mathit{Link}\ \mathit{Top} \ldots, \Delta) &= ET(\Gamma\ EL(\Gamma, \mathit{Link}), \rho, \mathit{Top} \ldots, \Delta) \\
ET(\Gamma, \rho, \mathit{Inv}\ \mathit{Top} \ldots, \Delta) &= ET(\Gamma, \rho\ \rho', \mathit{Top} \ldots, \Delta\ \Delta') \text{ where } EI(\Gamma, \mathit{Inv}) = (\rho', \Delta') \\
ET(\Gamma, \rho, \mathit{expr}\ \mathit{Top} \ldots, \Delta) &= ET(\Gamma, \rho, \mathit{Top} \ldots, \Delta\ \rho[\![\mathit{expr}]\!])
\end{aligned}
$$

$EM : (\mathit{Top} \ldots, \mathit{Mod}) \to \mathit{Mod}$

$EM(\Gamma, (\textbf{module}\ N\ \mathit{Def}\ \ldots)) = (\textbf{module}\ N\ \mathit{Imp} \ldots\ \mathit{def}\ \mathit{Exp}\ \ldots)$
where $\mathit{sort}(\mathit{Def} \ldots) = \mathit{Imp} \ldots\ \mathit{def} \ldots\ (\textbf{export}\ N_1\ (n_1\ n_2) \ldots) \ldots$
and $(n_3 \ldots) \ldots = \mathit{names}(\Gamma(N_1)) \ldots$
and $\mathit{Exp} \ldots = (\textbf{export}\ N_1\ (n_3\ [n_2/n_1 \ldots][\![n_3]\!]) \ldots) \ldots$

$EL : (\mathit{Top} \ldots, \mathit{Link}) \to \mathit{Mod}$

$EL(\Gamma, (\textbf{link}\ N\ (N_1\ N_2))) = (\textbf{module}\ N\ \mathit{Imp} \ldots\ \mathit{def} \ldots\ \mathit{Exp} \ldots)$
where $\Gamma(N_1) =$
$(\textbf{module}\ N_1\ (\textbf{import}\ A_1) \ldots\ \mathit{def}_1 \ldots\ (\textbf{export}\ B_1\ (b_1\ a_1) \ldots) \ldots)$
and $\Gamma(N_2) =$
$(\textbf{module}\ N_2\ (\textbf{import}\ A_2) \ldots\ \mathit{def}_2 \ldots\ (\textbf{export}\ B_2\ (b_2\ a_2) \ldots) \ldots)$
and $A_3 \ldots = (A_2 \ldots) - (A_1 \ldots B_1 \ldots)$
and $b_3 \ldots = \mathit{names}(\Gamma((A_2 \ldots) \cap (B_1 \ldots)) \ldots)$
and $\rho_1 = \mathit{freshen}(\mathit{def}_1 \ldots)$
and $\rho_2 = \mathit{freshen}(\mathit{def}_2 \ldots)$
and $\rho_3 = [\rho_1[\![[a_1/b_1 \ldots][\![b_3]\!]]\!]/b_3 \ldots]$
and $\mathit{Imp} \ldots = (\textbf{import}\ A_1) \ldots\ (\textbf{import}\ A_3) \ldots$
and $\mathit{def} \ldots = \rho_1[\![\mathit{def}_1]\!] \ldots\ \rho_2[\![\rho_3[\![\mathit{def}_2]\!]]\!] \ldots$
and $\mathit{Exp} \ldots = (\textbf{export}\ B_1\ (b_1\ \rho_1[\![a_1]\!])\ \ldots) \ldots$
$(\textbf{export}\ B_2\ (b_2\ \rho_2[\![\rho_3[\![a_2]\!]]\!])\ \ldots) \ldots$

$EI : (\mathit{Top} \ldots, \mathit{Inv}) \to (n \to n, \mathit{def} \ldots)$

$EI(\Gamma, (\textbf{invoke}\ N)) = ([\rho[\![n_2]\!]/n_1 \ldots], \rho[\![\mathit{def}]\!] \ldots)$
where $\Gamma(N) = (\textbf{module}\ N\ \mathit{def} \ldots\ (\textbf{export}\ N'\ (n_1\ n_2) \ldots) \ldots)$
and $\rho = \mathit{freshen}(\mathit{def} \ldots)$

$\mathit{sort}(\mathit{Def} \ldots)$: Sort module body into imports, definitions, and exports.
$\mathit{names}(\mathit{Ifc} \ldots)$: Extract the names of signatures from interfaces.
$\mathit{freshen}(\mathit{def} \ldots)$: Produce a substitution giving fresh names to definitions.

Fig. 5. Translation from Modular ACL2 to an executable program

linking of each compound module into a primitive one. The other is the extraction of definitions from each invoked primitive module; these are concatenated with top-level expressions. We perform this transformation with the function E (for "Executable"), shown in figure 5 along with some auxiliary translation functions. To simplify the presentation, we introduce a and b for function and theorem names, and A and B for interface and module names. We use Γ as an environment again, this time for both interfaces and modules.

This E function invokes ET with an empty environment, substitution, and sequence of result terms. The ET function adds interfaces to the environment, as well as primitive modules reduced to canonical form by EM. All modules in the environment contain imports first, then internal definitions, and finally exports with fully explicit external/internal name associations. Compound modules are converted to primitive modules by EL and stored in the environment. The EI function extracts definitions and a substitution from a module in the environment, which ET uses to splice the module's body into the top level and link top-level expressions to it.

The EL function combines two primitive modules into one. It looks up their definitions in the environment, then extracts their imports, exports, and internal definitions. The definitions are given fresh names and linked together by substituting names exported (from N_1) and imported (to N_2) across a shared interface. Finally, EL concatenates both sets of source imports (except any resolved by linking), definitions, and exports.

4 Experience with Modules

Designing a new language is insufficient; one must program in it to determine its merit. We have therefore added a prototype of Modular ACL2 to Dracula and have used it to convert a number of ACL2 programs into modular shape. In this section, we present our experience writing, verifying, and executing three illustrative examples. We then demonstrate the advantages of modules for ACL2 and explain the most serious problem encountered.

Illustrative Experiments. The *Worm game* is illustrative of the projects assigned to freshmen at Northeastern University and the courses at Oklahoma.

The top-left box in figure 6 displays a concise description of the game. The implementation consists of three main modules implementing the food, the worm, and the game itself. These are supported by three other modules, defining a pseudorandom number generator, basic point geometry, and the game grid. We implemented the game and verified two nontrivial properties: the worm's tail stays within the grid during the game, and it never crosses itself. Figure 6 shows portions of the game and point interfaces.

Graph traversal is the first canonical ACL2 case study [3]. The task is to represent directed graphs, implement an algorithm to find a path from one node to another, and prove the algorithm always produces a valid path.[4]

We designed our graph traversal program around two interfaces: one for representing a graph, the other for the search algorithm. See figure 7 for details. A successful *find-path* is guaranteed to produce a path, specified by *pathp* in *IGraph* as a list of adjacent nodes. We produced four modules in total: neighbors list and edge list representations of graphs, and depth-first and breadth-first search. The modules are interchangeable; either graph representation may be linked with either search algorithm.

[4] The original case study also proves that it finds a path so long as one exists.

Game Description:
The player directs a constantly-moving worm on a grid. The grid has walls and, somewhere, a piece of food. If the worm eats the food, the worm grows in length and a new piece of food appears. If the worm runs into a wall or its own tail, the game ends.

```
(interface IPoint
 (sig point-uniquep (pt pts))
 (sig points-uniquep (pts))
 (con points-uniquep/nil
  (points-uniquep nil))
 (con points-uniquep/cons
  (implies
   (and (pointp pt)
        (point-listp pts)
        (points-uniquep pt pts)
        (points-uniquep pts))
   (points-uniquep (cons pt pts)))))
```

```
(interface IGame
 (include IPoint)
 (sig live-gamep (v))
 (sig uncrossedp (v))
 (sig worm-tail (g))
 (sig game-tick (g))
 (con uncrossedp/worm-tail
  (implies (uncrossedp g)
           (points-uniquep (worm-tail g))))
 (con initial-game/uncrossedp
  (uncrossedp (initial-game)))
 (con game-tick/uncrossedp
  (implies (and (uncrossedp g)
                (live-gamep g))
           (uncrossedp (game-tick g))))
 (con game-key/uncrossedp
  (implies (and (uncrossedp g)
                (live-gamep g)
                (stringp k))
           (uncrossedp (game-key g k)))))
```

Fig. 6. Interface excerpts from the Worm game

```
(interface IGraph
 (sig graphp (v))
 (sig nodep (g n))
 (sig edgep (g a b))
 (sig pathp (g x y p))
 (con pathp/one
  (iff (pathp g x y (list a))
       (and (equal x a) (equal y a) (nodep g a))))
 (con pathp/append
  (implies
   (and (edgep g b c) (pathp g a b p) (pathp g c d q))
   (pathp g a d (append p q)))))
```

```
(interface IFindPath
 (include IGraph)
 (sig find-path (g x y))
 (con find-path/pathp
  (implies
   (and (graphp g)
        (nodep g x)
        (nodep g y)
        (find-path g x y))
   (pathp
    g x y
    (find-path g x y)))))
```

Fig. 7. Interface excerpts from the graph search program

Different strategies for implementing *language interpreters* suggest natural exercises in proving equivalence of two recursive algorithms. In our interpreters, an expression is either an integer or a binary operator $+$, $-$, or $*$ applied to two expressions. Our small-step interpreter reduces the leftmost redex in an expression, producing a new expression until no reductions remain. Our alternative interpreter uses a big-step strategy. We specified the program in four interfaces: expressions, big-step evaluation, small-step reductions, and equivalence between the two. Figure 8 shows some representative excerpts.

```
(interface ILanguage ;; datatypes
 (sig exprp (v))
 (sig calcp (v))
 (sig calc (f a b)))
```

```
(interface IBigStep
 (include ILanguage)
 (sig eval (e))
 (con eval/plus
  (equal (eval (calc '+ a b))
         (+ (eval a) (eval b)))))
```

```
(interface IEquivalence
 (include ILanguage)
 (include IBigStep)
 (include ISmallStep)
 (con eval=last-reduction
  (implies (exprp e)
           (equal (last (reduce-all e))
                  (list (eval e))))))
```

```
(interface ISmallStep
 (include ILanguage)
 (sig reduce (e))
 (sig reduce-all (e))
 (con reduce/plus
  (implies
   (and (integerp a) (integerp b))
   (equal (reduce (calc '+ a b)) (+ a b))))
 (con reduce/left
  (implies (calcp a)
           (equal (reduce (calc f a b))
                  (calc f (reduce a) b))))
 (con reduce-all/calcp
  (implies
   (and (exprp e) (calcp e))
   (equal (reduce-all e)
          (cons e (reduce-all (reduce e)))))))
```

Fig. 8. Interface excerpts from the interpreter program

Theorem	Mono.	Mod.	Theorem	Mono.	Mod.
game-tick/uncrossedp	845.95	0.06	*game-mouse/uncrossedp*	8.82	0.01
game-tick/gamep	387.12	0.03	*connected-wormp/wormp*	3.00	0.06
game-tick/in-bounds	362.97	0.03	*worm-turn/uncrossed-wormp*	0.48	0.10
connected-gamep/gamep	173.55	0.03	*worm-move/uncrossed-wormp*	0.38	0.05
game-key/uncrossedp	148.65	0.05	*worm-grow/uncrossed-wormp*	0.25	0.04
game-key/in-bounds	64.58	0.03	*worm-turn/in-bounds-wormp*	0.23	0.06
game-key/gamep	64.24	0.02	*random-nat/range*	0.10	0.10
uncrossedp/gamep	10.75	0.01	*modulo/range*	0.08	0.08

Fig. 9. Time (in seconds) to verify theorems from two versions of the Worm game

Performance Improvements. Programming in a modular style naturally reduces the scope of ACL2's proof search space and improves the engine's efficiency. Plain ACL2 typically requires "hints"—e.g., restrictions of the global theory—to complete or speed up a proof. Modules restrict theories by design and in a disciplined manner; it is often unnecessary to guide the search.

Our modular verification of the Worm game required no hints at all; the verification takes just a few seconds per module. We compared this to a naïve translation into a monolithic ACL2 proof. We concatenated the contents of the modules and inserted the contracts of each module's exports as theorems. ACL2 was able to verify the monolithic version as well, but took several orders of magnitude longer.

Figure 9 shows the CPU time (in seconds) used to prove the slowest nine theorems from each version of the Worm game. The modular version never takes over a tenth of a second, while the monolithic proof peaks at several minutes. Near the end of the monolithic program, proof attempts had the entirety of the game to inspect, while the modular proof started with a clean slate per module. The slow performance measured here does not reflect the professional ACL2 user's experience; rather, such ACL2 users refine their proofs by manually maintaining abstraction boundaries that occur naturally with modules.

Conciseness and Reuse. Modular design also promotes abstraction and code reuse. Standard ACL2 programs cannot, in general, change their implementation strategy without adjusting the accompanying theorems. Put differently, separating implementations and specifications imposes a serious cost of manual coding and, because of that, prevents common patterns of code reuse and refactoring.

In contrast, Modular ACL2 encourages and simplifies reuse. Clients of our graph modules may swap representations or search algorithms freely in a **link** clause without changing a single **defthm**. Even undergraduates can now program for reuse in ACL2.

Limitations. Unfortunately, our gains in terms of local reasoning come with a serious loss, best illustrated with our interpreter example. In this example, our final equivalence proof imports *ILanguage*, *IBigStep*, and *ISmallStep* (fig. 8), representing respectively the grammar and two interpreters. Sadly, while a natural modularization calls for this organization, doing so prevents ACL2's search engine from finding an inductive proof.

The key problem is that, on one hand, ACL2 associates induction schemes with function definitions, and that, on the other hand, Modular ACL2's interfaces hide function definitions. For the specific case of our interpreters, the main theorem must perform induction on the structure of expressions and of the two interpreter algorithms. Because these definitions are hidden behind module barriers, ACL2's proof engine can't possibly find a proof. The only way to expose the induction schemes to ACL2 is to provide a concrete function definition, but exposing *eval* and *reduce-all* defeats the abstraction boundaries of *ISmallStep* and *IBigStep*.

From a high level perspective, we have traded improved local reasoning for a loss in global reasoning. Naturally we consider this a major limitation of our current approach. Induction is a critical aspect of ACL2, and inductive proofs should not be limited to individual modules. Hence, our next step in designing Modular ACL2 is to add a linguistic mechanism for specifying induction principles across interfaces and verifying their correct implementation as exports.

5 Related Work

The design of the module system derives from PLT Scheme's unit system [8,9], with linking semantics based on mixins [10,11]. More precisely, Modular ACL2 contributes contracts to the unit model, but inherits the idea of linking primitive

and compound modules in hierarchical shape. It subtracts recursive linking as this would complicate ACL2's termination proofs.

Coq [12,13], Twelf [14], and similar proof assistants adopt an ML-like module system for encapsulating proofs about metatheory. Our modules and interfaces correspond closely to ML's functors and signatures. Modular ACL2 can express type specifications via contracts and sharing constraints via interface inclusion; it cannot currently express nested modules. However, we face different challenges, having chosen to work with a first-order functional language and an automated theorem prover with idiosyncratic limitations. We must deal with the lack of abstract induction schemes, the inexpressibility of higher-order logical statements such as a module's proof obligation, and the lack of execution-preserving proof abstraction mechanisms.

Extended ML (EML) [15] equips SML [16] with logical properties and a verification semantics. The language is designed around the methodology of beginning with an abstract specification and refining it step-by-step to a concrete implementation. EML offers signatures, structures, and functors, any of which may contain axioms, analogous to our modules and interfaces with contracts. EML also offers the abstract term "**?**" for specified but unimplemented types, values, or structures; Modular ACL2's stubs and axioms serve a similar purpose. EML has the benefit of SML's powerful type system, but lacks a theorem prover. In contrast, Modular ACL2 is based on the industry's leading, general-purpose, automated theorem prover.

Some theorem proving languages also provide named scopes, such as Isabelle's locales [17], Coq's sections [18], and the "little theories" of IMPS [19]. These scopes allow local and global definitions, and export the global ones by translating or abstracting over the local ones. They provide a lightweight approach to abstraction and namespace management, but do not support explicit interfaces or introduce abstraction beyond that of the underlying language.

6 Conclusion

While Boyer and Moore took an existing functional language and constructed a theorem prover for it, we have chosen to take an existing theorem prover and to equip it with a pragmatic module system. Thus far, we have designed a series of models and prototypes. In this paper, we present the first version that makes modular programming truly practical. Our examples in this paper illustrate how Modular ACL2 introduces and encourages information hiding and code reuse. As a result, Modular ACL2 naturally improves the performance of the proof search engine. Novices to the system now easily succeed with complex proofs where before professionals would have had to manually encode search strategies.

Unsurprisingly, the development of Modular ACL2 pinpoints the major problem with modularization of ACL2 programs: the hiding of inductive structures. We intend to tackle this challenging problem over the next year and expect to report progress on Modular ACL2 then. In the meantime, we will deploy and maintain our implementation to get feedback through classroom experience.

References

1. Boyer, R.S., Moore, J.S.: Mechanized reasoning about programs and computing machines. In: Veroff, R. (ed.) Automated Reasoning and Its Applications: Essays in Honor of Larry Wos, pp. 146–176. MIT Press, Cambridge (1996)
2. Steele Jr., G.L.: Common Lisp—The Language. Digital Press (1984)
3. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, Dordrecht (2000)
4. Harper, R., Pierce, B.C.: Design considerations for ML-style module systems. In: Advanced Topics in Types and Prog. Languages, pp. 293–345. MIT Press, Cambridge (2004)
5. Page, R.: Engineering software correctness. J. of Func. Prog. 17(6), 675–686 (2007)
6. Vaillancourt, D., Page, R., Felleisen, M.: ACL2 in DrScheme. In: Proc. 6th Intern. Works. ACL2 Theorem Prover and its Applications, pp. 107–116. ACM Press, New York (2006)
7. Eastlund, C., Vaillancourt, D., Felleisen, M.: ACL2 for freshmen: First experiences. In: Proc. 7th Intern. ACL2 Workshop, pp. 200–211. ACM Press, New York (2007)
8. Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. In: ACM SIGPLAN Conference on Prog. Language Design and Implementation, pp. 236–248 (June 1998)
9. Owens, S., Flatt, M.: From structures and functors to modules and units. In: ACM SIGPLAN Intern. Conference on Func. Prog., pp. 87–98. ACM Press, New York (2006)
10. Bracha, G., Lindstrom, G.: Modularity meets inheritance. In: Proceedings of the International Conference on Computer Languages, pp. 282–290. IEEE, Los Alamitos (1992)
11. Dreyer, D., Rossberg, A.: Mixin' up the ML module system. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 307–320. ACM, New York (2008)
12. Chrzaszcz, J.: Implementing modules in the Coq system. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 270–286. Springer, Heidelberg (2003)
13. Courant, J.: $\mathcal{MC}_2$: A Module Calculus for Pure Type Systems. J. of Func. Prog. 17, 287–352 (2006)
14. Harper, R., Pfenning, F.: A module system for a programming language based on the LF logical framework. Journal of Logic and Computation 8(1), 5–31 (1998)
15. Sannella, D.: Formal program development in Extended ML for the working programmer. In: Proc. 3rd BCS/FACS Workshop on Refinement, pp. 99–130 (1991)
16. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (2e). MIT Press, Cambridge (1990)
17. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales: A sectioning concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 149–166. Springer, Heidelberg (1999)
18. The Coq Development Team: The Coq Proof Assistant Reference Manual (2006), `http://coq.inria.fr/V8.1pl3/refman/index.html`
19. Farmer, W.M., Guttman, J.D., Thayer, F.J.: IMPS: An interactive mathematical proof system. Journal of Automated Reasoning 11, 213–248 (1993)