Hacking and Extending ACL2*

Peter C. Dillinger
College of Computer and Information Science
Northeastern University
pcd@ccs.neu.edu

Matt Kaufmann
Department of Computer Sciences
University of Texas at Austin
kaufmann@cs.utexas.edu

Panagiotis Manolios
College of Computer and Information Science
Northeastern University
pete@ccs.neu.edu

ABSTRACT

The ACL2 theorem prover provides the user a wide range of mechanisms for customization and extension while preserving soundness. ACL2 researchers and power users, however, sometimes work outside this realm in order to add new core functionality, to attach new user interfaces, or to connect ACL2 with other reasoning engines. We first describe new features of ACL2 that enable users to add to the set of "trusted code" in a trackable way. The advantage is that users can dynamically install system extensions they choose to trust in reaching their verification results, and ACL2 will track what was trusted in the process. We then describe features, idioms, and abstractions that leverage the freedoms given to trusted code and the dynamic, reflective nature of Common Lisp to modify ACL2 in deep ways at runtime. Our abstractions are designed to make it *easier* for system hackers to preserve sound reasoning when writing metatheoretic code.

1. INTRODUCTION

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." ACL2 consists of a first-order functional programming language, a logic for that language, and an "industrial-strength" automatic theorem prover [6, 4, 3]. ACL2 has been used to prove some of the most complex theorems ever proved about commercial systems.

As ACL2 has become more widely used, the need and/or desire to extend ACL2 in unsupported ways has risen. Within the bounds of a command-line automatic theorem prover based largely on term rewriting, ACL2 is quite extensible, not only with user-defined theorems and theories, but also with computed hints, meta rules, powerful metaprogramming, and more. Furthermore, when the user base was smaller, ACL2 authors were more able to adapt and extend the system itself to suit user needs. A new reality has set in, in which a growing number of sophisticated users are seeking to adapt ACL2 to new environments [1], modify or extend core functionality [5], or combine it with other reasoning engines [8, 7]. The ACL2 authors cannot possibly satisfy all these requests for new functionality from ACL2, nor, arguably, should they/we. The needs and desires of such "power users" can extend far beyond what the current ACL2 can be practically made to accommodate soundly by the ACL2 authors. It may seem ironic that as ACL2 has grown more powerful, the demand for more functionality has grown; but it is certainly being applied to a larger variety of problems.

To address this problem, ACL2 Version 3.1 introduced a feature, *trust tags* ("ttags" for short), which has "opened the flood gates" to potentially unsound extensions of ACL2 by end-users. The basic idea behind trust tags is not particularly novel (see for example [2]), though ACL2's approach may be unique in that it allows for rather arbitrary system extensions written in the system itself. But trust tags presented unique challenges for implementation in ACL2. The basic idea is that constructs that can render ACL2 unsound are now legal in the command loop and in books, provided they are used in a context with an active trust tag. Defining (activating) a trust tag is subject to authorization and after doing so, an ACL2 session is thence forth "tainted," at least in the sense that sound reasoning depends on code that is not a part of ACL2 proper. Trust tags enable authors to make such code small and concise and enable other users to track down easily which code must be trusted.

This paper introduces, documents, and demonstrates some applications of trust tags. Section 2 gives some background information and terminology on ACL2's inner workings. Section 3 introduces trust tags in some detail, giving some rules and conventions on their use. Section 4 enumerates the potentially unsafe constructs ttags make available. We also discuss some of our own low-level tools that can be added to provide more powerful and robust extensions of ACL2. Section 5 describes some higher-level idioms for modifying and extending ACL2 behavior, which have proved quite useful. Section 6 concludes.

Keep in mind that this paper does not offer any insight into proving theorems with ACL2, but a basic knowledge of the ideas in Section 3 is important for users who might use non-

*This research was funded in part by National Science Foundation grants CCF-0429924, IIS-0417413, CCF-0438871, and EIA-0303609, and also based upon work supported by DARPA and NSF under Grant No. CNS-0429591

standard extensions. Later sections are intended for ACL2 system hackers, who seek to alter or extend ACL2 in fundamental ways. Beware that such nonstandard extensions are error-prone and can run into compatibility issues with the various implementations of Common Lisp. To the contrary, standard use of ACL2 is designed to be quite robust and implementation-agnostic.

Throughout this paper, we will "offer," "introduce," or "define" some functionality that is not in ACL2—at least at the time of writing. This code is available in the supporting materials for this paper.

The examples we present will be drawn from the development of ACL2s, the "ACL2 Sedan." ACL2s is an Eclipse-based development environment for ACL2 that is designed to make formal reasoning in ACL2 easier to learn [1]. Key features of ACL2s are based on extending ACL2 to enhance invisible communication with the development environment, and on limiting or extending the theorem prover in non-standard but visible ways. While previously ACL2s required a specially-built ACL2 image to implement these extensions, trust tags enable them to be installed into a standard ACL2 image at runtime, in an "approved," trackable manner.

2. BACKGROUND INFORMATION

2.1 Terminology and Basics

The standard, safe ways of extending ACL2 involve extending the *world*. The ACL2 world is a property list consisting of everything about the current interpretation of the ACL2 logic, all the theorem prover rules and settings, and definitions that are not part of the logic. When, for example, a new function is defined, an extension of the current world is computed and then *installed* to make it the *current world*. Undoing that definition reinstalls a *retraction* of the current world, one that was previously the current world. Even though the world is an applicative object, "behind the scenes" optimizations make extension, retraction, and look-ups fast for the current world.

Roughly speaking, *events* are inputs to ACL2 that extend the world, and *books* are files of events. *Certifying* a book means verifying all of its events in some *certification world* and writing out a *certificate* that enables fast *inclusion* of the book.

The ACL2 *state* is a *single-threaded object*, which means it is modifiable by virtue of there being only one "live" instance, enforced through static checks at definition time. The state includes the current world, assignments to *global variables*, and many other things that do not show up in this paper.

"The loop" is the ACL2 read-eval-print loop. Unlike books, the loop accepts types of input other than events. Some of these inputs can modify the ACL2 state, including the world, and some even interact with the external environment, such as reading from or writing to files.

We will refer to a critical subset of the full ACL2 state as the "logical state". We can formally define the *logical state* as that which is protected by `MAKE-EVENT` (discussed below), but it basically consists of the world and many global variables that ACL2 uses internally. ACL2 must maintain certain invariants on the logical state to maintain sound reasoning.

The `ACL2-DEFAULTS-TABLE` is a special part of the world that keeps track of settings that govern how the theorem prover behaves and how some events are interpreted. The `ACL2-DEFAULTS-TABLE` is the only ACL2 *table* that is reset at the beginning of every book.

2.2 Intricacies (for Section 4+)

ACL2 function definitions are stored as elements (properties) of a list (the world). Installing a world makes the functions executable by defining them in the underlying Common Lisp. In fact, to support ACL2 functions being fully defined (unlike Common Lisp functions) all functions actually have two raw definitions: a Common Lisp definition and an executable counterpart. ACL2 uses one or both of the raw definitions whenever it executes the function on a particular input; the executable counterpart often defers to the Common Lisp definition. In this paper, we will only consider cases that use the Common Lisp definition, so we will not need to investigate manipulation of executable counterparts. Outside of this paragraph, the "raw" definition is the Common Lisp definition.

Macros have raw definitions also, but only one per macro, a "Common Lisp" definition. When we use a macro in ACL2, including those as simple and common as `DEFUN`, it is expanded by looking up its definition in the world, binding its lambda variables, and evaluating the body in that environment. Thus, the raw macro definition is not involved in macro expansion of input to the ACL2 loop. To the contrary, executing a function (on a particular input) from the ACL2 loop always uses one or both of the raw function definitions.

The raw definitions of macros are only relevant for code interpreted by Common Lisp. Most importantly, this includes raw function bodies, because these are interpreted (and/or compiled) by Common Lisp. But this also includes the bodies of books. Books bundle together definitions of functions, macros, and theorems to be used by ACL2, but they can also be compiled by raw Lisp. In fact, this is the default case. For example, a `DEFUN` in a book when processed by the ACL2 loop is macro-expanded to a function that installs a world updated with the new definition. When processed by raw Lisp, either during compilation or inclusion of the compiled file, the Common Lisp meaning of `DEFUN` is used, which installs the compiled version of the raw functions and macros. As another example, `DEFTHM` proves theorems and potentially adds them as rules in the world, but when interpreted by raw Lisp, `DEFTHM` is a no-op. This different behavior is due to the difference between the Common Lisp and ACL2 definitions of the `DEFTHM` macro.

3. TRUST TAGS IN ACL2

Here we introduce the ACL2 notion of *trust tags*. For more details, start with ACL2 documentation topic `DEFTTAG`.

3.1 History

`MAKE-EVENT`, introduced in ACL2 Version 3.0, made it possible to embed arbitrary ACL2 code to be executed on the certification or inclusion of a book. By contrast, books were

previously restricted primarily to contain definitions, theorems, and evaluation of stateless functions. Since all that is needed from the code in a MAKE-EVENT is the return value, MAKE-EVENT is able to protect the logical state from corruption by saving it, executing the code, and then restoring the logical state.

However, some ACL2 functions interact with the external environment or can have other dangerous effects that aren't tracked as part of the logical state. Bob Boyer demonstrated the incompleteness of the protection offered by MAKE-EVENT, by creating a book that derives a contradiction in ACL2 by embedding in a MAKE-EVENT a SYS-CALL that invokes an external debugger to overwrite memory in the ACL2 process. (See the ACL2 documentation for SYS-CALL for details.)

An approach to patching this problem that was rejected after some consideration was to add a parameter to CERTIFY-BOOK and INCLUDE-BOOK to allow or disallow use of SYS-CALL inside the corresponding book. This was analogous to existing :DEFAXIOMS-OKP and :SKIP-PROOFS-OKP parameters, but there was an important difference. DEFAXIOM and, in a sense, SKIP-PROOFS do not affect the soundness of ACL2's reasoning. If the formulae introduce inconsistencies, we can prove NIL (derive an apparent contradiction), but if we interpret the constructs as simply adding new formulas to the set of axioms/postulates, the reasoning is still sound [1]. Also, any DEFAXIOM events, and any events under SKIP-PROOFS, can be readily undone with no lingering impact.

Constructs such as SYS-CALL and SET-RAW-MODE, on the other hand, can irreversibly render an ACL2 session unsound. That is, these constructs can break ACL2's implementation of its underlying logic. Since each of these constructs has the potential to affect ACL2 in the worst possible way, it makes little sense to distinguish between them for the purposes of allowing or disallowing use. More important than which construct is used is where, by whom, and for what purpose. This is what we intend to capture with trust tags, in a concise, understandable, and usable way.

3.2 Defining Trust Tags

A trust tag (or *ttag*) instance has a name and a location. The name is any (non-NIL) ACL2 symbol, and the location is either the top-level command loop (indicated by NIL) or a book location (indicated by a path string). A trust tag is activated by passing the name to DEFTTAG. The location associated with the trust tag is always the current location. The name of the current active trust tag is stored in the :TTAG entry of the ACL2-DEFAULTS-TABLE, alongside settings such as :DEFUN-MODE and :WELL-FOUNDED-RELATION. There can be only one active ttag at a time, or a NIL setting indicates there is no active ttag, as produced by (defttag nil). Like other ACL2-DEFAULTS-TABLE settings, the current ttag setting is local to the book in which it occurs.

The :TTAG setting becomes relevant when command loop code or book code attempts to use a *dangerous* function, a function such as SYS-CALL that is known to enable unsound reasoning. Use of such functions is permitted if and only if there is an active trust tag. Intuitively, defining a trust tag for a block of code declares that the author (or at least that block of code) must be trusted to preserve soundness. The trust tag indicates who or what is to blame if unsoundness results.

Each time a trust tag is declared, the ACL2 session should be considered "tainted" by another extension. There is no tracking of which subsequent lemmas depend on the potentially unsound extension, not only because ACL2 has trouble determining which theorems were required for a proof, but also because the extensions are often metatheoretic, incapable of being described succinctly by a set of rules or postulates. In a sense, ACL2 takes a conservative approach, in which *every* subsequent lemma is considered to depend on all ttags seen. Along these lines, a book is tagged with any ttags seen in the logical world that certified the book, in addition to any ttags seen in the resulting logical world after certification. This tracking includes ttags used in a local context, but does not include ttags used in worlds that have been undone or ttags that try to hide their existence. This is a limitation to the conservativity: "taintedness" tracking is subject to cooperation of the ttagged code. Thus, printed ttag notes and manual inspection must be considered the authority.

3.3 Authorizing Trust Tags

An important aspect of trust tags is that an included book can generally only define trust tags that have been authorized by the user (but see the discussion of "ttag notes" in Section 3.4). The ACL2 authors intend for this to provide a degree of protection for the user's session and entire system; that is, ACL2 is designed so that an included book cannot "taint" the user's session (or system!) without specific authorization to use ttags. Thus, it should be safe to certify and include books that do not require ttags without inspecting them first. Attempting to define a ttag without authorization, of course, fails.

In the current implementation, defining a trust tag inside the command loop is always authorized. Recall that just because a ttag is active in the command loop doesn't mean an included book can use dangerous functions without defining its own ttag; the :TTAG setting does not propagate to included books in that way.

Defining a ttag in book certification or inclusion requires authorization, and the user specifies what is authorized by the :TTAGS keyword parameter to CERTIFY-BOOK or INCLUDE-BOOK. This parameter defaults to NIL, indicating no ttags are authorized. Specifying :ALL allows all ttags. One can also be more precise by specifying names and/or locations, and we refer the reader to the ACL2 documentation on DEFTTAG for that syntax. The reason we would want to be able to specify locations other than the one we are including is that the specification for authorized ttags also restricts what that book can authorize to books it includes, and so on recursively. Thus, books can only reduce the set of authorized ttags when including books themselves (unless they use an active ttag to get around the restriction, as in Section 3.6).

3.4 The Certifier

[1] If the postulates are inconsistent, no models satisfy the postulates; thus any formula is trivially satisfied by all models satisfying the postulates.

We now explore what is involved in the job of the certifier, a person who takes the role of being skeptical that ACL2 has reached its conclusions through sound reasoning. If the certifier trusts an "untainted" ACL2, then he must at least confirm that the result was reached without tainting ACL2.

The main caveat is that although we can ask ACL2 which ttags have tainted it (see the ACL2 documentation for `TTAGS-SEEN`), that answer is only trustworthy if the session has *not* been tainted. Thus, as a skeptic, the certifier learns nothing from that query. This observation demonstrates the need for *ttag notes*. Whenever `DEFTTAG` is called with a non-nil value, ACL2 prints out the following (but for the appropriate name and location), on a single line:

```
TTAG NOTE: Adding ttag T from the top level loop.
```

The first `DEFTTAG` is guaranteed to print out such a note, and the certifier should determine the effects of that ttagged code first because it could modify ACL2 behavior in arbitrary ways, including modifications to the printing of later ttag notes. Once convinced the code "behaves nicely", including preserving the printing of later ttag notes, the certifier can move on to the next ttag note and repeat until all potentially dangerous code has been approved.

Unfortunately, we can't really define "behaves nicely" much more precisely than "extends ACL2 only as intended, and the intended extensions, as used, preserve soundness." The job of certifying an extension against possible uses is much harder than against actual uses, but in either case, the certifier's job will require some knowledge of ttags, the extension mechanisms enabled by them, and ACL2 internals.

3.5 Etiquette

To ease certification and interoperability, a developer who uses trust tags should try to follow some rules that we propose here.

Naming Conventions. The canonical ttag name for "I just want to use some dangerous functions, and this is not going to affect anyone else" is `T`, as in `(defttag t)`. Otherwise, the name should be something short that describes the capability that is added to ACL2 under that trust tag. It could, for example, correspond to the name of an external tool, as in `:MINISAT` or `:UCLID`. We use the ttag `:ACL2S` to add the capability for a session to talk to the ACL2s development environment. An informative name helps to document which potentially unsound extensions were used in reaching some result.

Uniqueness of names is not critically important, since use of a ttag always traces back to its location, but use of a documentation string with `DEFTTAG` makes unique naming more important. We encourage use of a documentation string with a ttag so that a user can find out the capability provided by and authorship of a ttag with the `:DOC` command. So to improve our likelihood of unique naming without being too verbose, one might use the ttag `:UCLID-GT` rather than `:UCLID` for code written by Georgia Tech students to interface with an external tool called UCLID. A name like `|Bob Smith's ACL2 interface to UCLID Version 0.73a|`, though informative, is hard to type.

Regarding version numbers in ttag names, on the one hand, updating version numbers in what one authorizes can be a hassle, but on the other hand, that hassle reminds users that changes have been made and they should proceed with caution. When or whether such reminders are needed is a judgement call.

Since ttag names are symbols, they have a package. With the exception of `T`, we have assumed use of keyword symbols (package `"KEYWORD"`) for ttag names so far. This is great for avoiding package import issues, and, as mentioned, name clashes are not a big problem. It's hard to foresee a case that strongly motivates using symbols other than keywords.

Extent. Let us now consider the issue of how much functionality should be encompassed by one ttag, or by the same ttag name in different locations, etc. First, observe that it is possible to use `DEFTTAG` more than once within a single book. One case of this is using `(defttag nil)` to mark the end of a block of code that uses dangerous constructs; this is highly encouraged, except when it would make sense to relegate that block of code to a separate book. (It's not necessary to put `(defttag nil)` at the end of a book, but can serve as documentation.) In fact, the certifier's job is easiest when ttagged code is restricted to small, stand-alone books.

It is also possible to activate and deactivate the same ttag multiple times in the same book. Using the same name in the same file refers to the same ttag. This is preferable to leaving a ttag active where not needed, but when it's convenient, we suggest grouping together related dangerous code.

Finally, although it is possible to declare ttags of different names in the same book, we recommend instead the splitting of more than one capability into separate books.

3.6 Subsumption

There is no built-in notion of inheritance or subsumption with trust tags, so under that scheme, if a capability ttagged as `A` depends on or includes a stand-alone capability `B`, both `A` and `B` would have to be authorized and both would show up as having "tainted" the session.

One can easily imagine, however, wanting a mechanism for one ttag to subsume others. A common idiom here is wanting to encapsulate a piece of functionality into a book that requires authorization of one ttag, even if that book uses other reusable books that have their own ttags associated with them. A trusting user of that book is concerned with the function interface to that book, not which other books or ttags the book uses.

Recall that ttags enable us to do virtually anything, including the modification of some settings and recorded history that allow ttag subsumption. We first save the state global variable `TTAGS-ALLOWED`, governing which ttags are authorized, and add to it which ttags we want to subsume. We next save the world global value `TTAGS-SEEN`, which records

which ttags have affected the current world. Then we execute the events that use the ttags we want to subsume, and after we restore TTAGS-ALLOWED and TTAGS-SEEN to the old values, ACL2 will not complain that we have used trust tags we were not authorized to use.

This could be considered subversive behavior, but we consider it okay as long as it is done in a way that would be clear to a certifier. We provide a macro PROGN+SUBSUME-TTAGS that handles the mechanics of such subsumption. See its documentation for examples.

4. BASIC EXTENSION MECHANISMS

Here we describe the ACL2 constructs that can only be used when a trust tag is in effect. When appropriate, we also describe more elegant abstractions that we have implemented and make available in the supporting materials.

4.1 Arbitrary Code

PROGN! is a basic building block for potentially dangerous ACL2 extensions because it allows unprotected embedding of arbitrary code into certifiable books. Just as PROGN collects many inputs into a single event, PROGN! does so without the restriction that the inputs be *embedded event forms*. And PROGN! does not protect logical state as MAKE-EVENT does.

SET-RAW-MODE also requires an active ttag and can be used in PROGN! to embed raw Lisp code in a book. For example, we could embed some code that turns off garbage collection messages in GNU Common Lisp (GCL):

```
(progn! (set-raw-mode t)
        (when (member-eq :gcl *features*)
          (set (intern "*NOTIFY-GBC*" "SI")
               nil)))
```

Note that changing the raw mode setting inside a PROGN! does not propagate beyond its body, so a matching (set-raw-mode nil) is not necessary. For some raw inputs, ACL2 will complain about not knowing how many values are returned; this can be avoided by wrapping forms in (progn ... nil). Considering these things, we define a convenient abstraction (in-raw-mode ...) to be (progn! (set-raw-mode t) (progn ... nil)).

As a small limitation, we were not allowed to write (setq si::*notify-gbc* nil) because the code must be an ACL2 object, and ACL2 does not recognize GCL's "SI" package. As shown, we can usually work around this problem by generating symbols from their constituent strings with INTERN.

But PROGN! in general has bigger caveats. First, the code can be executed several times, because of the dual passes of certify-book and/or encapsulate, because the code is part of what is compiled and executed in raw Lisp if the containing book is compiled, and because the containing book could be "undone" and reloaded. Thus, code in PROGN! should be idempotent, though ACL2 provides no check for that.

Second, we are limited in how we can safely modify the world inside a PROGN!. The reason is that code inside a PROGN! is interpreted by both the ACL2 loop and the Common Lisp compiler. Problems arise if compiled code attempts to change the world. Embedded event forms like DEFUN and DEFTHM (see the ACL2 documentation for EMBEDDED-EVENT-FORM) are fine inside PROGN! because they have raw Lisp definitions that do not affect the world, but directly calling INSTALL-EVENT, for example, is discouraged (if not illegal) in a PROGN!, because it always modifies the world. Observe that DEFUN calls INSTALL-EVENT, but only when executed in the ACL2 loop. Outside the loop, DEFUN corresponds to the underlying Lisp's DEFUN, which knows nothing of the ACL2 world, and, thus, DEFUN is fine inside PROGN!.

Third, PROGN! allows us to specify changes to ACL2 that are not captured by the logical world, but does not support "undoing" those changes in any way. For example, if we include a book that uses PROGN! to change the raw Lisp definition of a function, that change is not reverted if we undo the inclusion of the book! And if that raw Lisp code calls user-defined ACL2 functions, the system breaks if those ACL2 functions are undone. The only built-in, robust workaround to this problem is to ensure the book is never undone by executing (reset-prehistory t) within it or following its inclusion. As its documentation describes, this use of RESET-PREHISTORY sets a new starting point for the world, before which nothing can be undone.

In response to these issues with PROGN!, we have developed a complementary construct that we call DEFCODE. DEFCODE is able to "embed" code at some point in a book in four different ways; thus, a call to DEFCODE is optionally given, for each of those different ways, a code block to embed in that way:

:LOOP code is executed each time the DEFCODE form is executed by the ACL2 loop, such as in an INCLUDE-BOOK or each ACL2 pass of CERTIFY-BOOK or ENCAPSULATE. Any code that modifies the world or checks that the current world is compatible with what we want to do should go under :LOOP. Like PROGN!, a soft error returned by :LOOP code aborts further processing.

:LOOP code is typically used for checking that the current world is compatible with how we intend to extend it. It can also be used for embedded event forms and any other world-modifying code.

:COMPILE code is compiled and loaded by the underlying Lisp on CERTIFY-BOOK and INCLUDE-BOOK (if book compilation is chosen). This is the only code in a DEFCODE seen by the Lisp compiler, so authors need not worry about how other code (in particular, code specified by :LOOP) would be compiled. Further, this code is only seen by the Lisp compiler, so authors need not worry about how the ACL2 loop would execute it (though the code itself must be constructed of ACL2 objects).

:COMPILE code will typically only contain DEFUN and DEFMACRO forms to be compiled in the underlying Lisp.

:EXTEND code is executed each time the DEFCODE event is installed into the current ACL2 world. Except in error cases, this happens after the ACL2 loop successfully processes the :LOOP code. The :EXTEND code is also

executed when a world is "resurrected", as with the :OOPS command or a *REDO* in ACL2s.

:EXTEND code will typically use IN-RAW-MODE to make some modifications in raw Lisp or modify ACL2 state global variables. It is an error for :EXTEND code to modify the world; ACL2 seems to catch the error and recover, but more importantly, it doesn't make sense to extend the world in the middle of extending the world.

:RETRACT code is executed each time the DEFCODE event is retracted from the current ACL2 world, such as with :UBT or the multiple passes of CERTIFY-BOOK or ENCAPSULATE. :RETRACT code should undo the effect of the corresponding :EXTEND code, and it is an error for :RETRACT code to modify the world.

To clarify the relationship between DEFCODE, PROGN!, and ordinary book events, consider these inputs, which are equivalent if a trust tag is in effect:

```
1: (defun x (y) (z y))
2: (progn! (defun x (y) (z y)))
3: (defcode
     :loop    ((defun x (y) (z y)))
     :compile ((defun x (y) (z y))))
```

As illustrated, DEFCODE separates the two interpretations applied to PROGN! code and ordinary book code, allowing more customization. We do not need :EXTEND and :RETRACT in this case because the DEFUN in the :LOOP code extends the world in a way that already causes ACL2 to update the raw definition of X on world extension/retraction. :EXTEND and :RETRACT cause the DEFCODE itself to update the world with custom code that is executed on world extension and retraction.

Here is an example that defines a function *only* in raw Lisp, using :EXTEND and :RETRACT code to associate the definition with the world created by the DEFCODE:

```
(defcode
  :loop ((in-raw-mode
          (when (or (fboundp 'x)
                    (macro-function 'x))
            (hard-error 'defcode
                 "X already defined."))))
  :compile ((defun x (y) (z y)))
  :extend ((in-raw-mode (defun x (y) (z y))))
  :retract ((in-raw-mode (fmakunbound 'x))))
```

In that example, the :EXTEND block defines a function X in raw Lisp and the :RETRACT block removes that definition. We use the :LOOP block to check that X does not yet have a raw function or macro definition. We choose this place for such checks rather than :EXTEND because (1) if all raw definitions are managed properly with DEFCODE, we can safely assume the check is satisfied in the case of world "resurrection", and (2) throwing an error within :EXTEND or :RETRACT is typically difficult for ACL2 to recover from; though it seems to recover, it is probably best to assume such an error corrupts the session.

Note that using (in-raw-mode ...) in the :COMPILE block is unnecessary, since that part is only used by the raw Lisp compiler. Eliminating the :COMPILE block would prevent the definition from being compiled during book compilation, but it would still be possible to compile the function later with (in-raw-mode (compile 'x)) or similar.

MAKE-EVENT is not supported inside DEFCODE, which is, perhaps, the only reason DEFCODE does not make PROGN! obsolete. In fact, PROGN! has recently been endowed with a capability that makes it well-suited for setting up modified environments in which to process embedded event forms, including MAKE-EVENT. Specifically, PROGN! can optionally take a list of state global variable bindings to pass to a STATE-GLOBAL-LET* surrounding the body of the PROGN! when it is processed in the ACL2 loop. The key difference between using that option, (progn! :state-global-bindings <bindings> <forms>), and writing the STATE-GLOBAL-LET* in the body oneself, (progn! (state-global-let* <bindings> (progn! <forms>))), is that MAKE-EVENT can be used in <forms> in the former but not in the latter. See documentation for PROGN! and our implementation of PROGN+TOUCHABLE and PROGN+REDEF (described below) for more information and examples.

It is very easy to make an ACL2 session inconsistent or unsound by using DEFCODE, PROGN!, or SET-RAW-MODE. But in later sections, we describe abstractions that make it easier to avoid soundness pitfalls.

4.2 Untouchables

ACL2 *untouchables*, which include untouchable functions/macros[2] and untouchable state-global variables[3], enable much of ACL2 to be written in its own language while maintaining metatheoretic invariants on the logical state in the presence of user code. As part of the ACL2 build process, the ACL2 world is populated with a preset list of functions/macros and variables that should be inaccessible to direct call or assignment by the user. If the user attempts to call an untouchable function or macro or assign an untouchable state-global variable, either in code for immediate evaluation, in the body of a function definition, or in the macro expansion of either of those, an error results.

For example, INSTALL-EVENT is an untouchable function because the user could use that function to install an arbitrary extension of the current world, including those that are logically inconsistent. ACL2-RAW-MODE-P is an untouchable variable because assigning T to that variable would enter raw mode without authorization.

But untouchable functions and variables are used *on behalf of* the user all the time. DEFTHM and DEFUN, for example, use INSTALL-EVENT to extend the world with the new formulas, but only if they are found to be admissible. Thus, the key for "touchable" functions that use untouchables is that they guarantee the untouchables are only used in ways

[2] (global-val 'untouchable-fns (w state))
[3] (global-val 'untouchable-vars (w state))

that maintain the required invariants.

PUSH-UNTOUCHABLE allows the user to add to the list of untouchable functions or variables. Though no trust tag is required for this operation, it will typically only be needed by hackers who create functions or use variables that need to be protected. For example, ACL2s defines a function PUT-GLOBALS that is able to assign values to any state-global variables, regardless of whether they are untouchable. Obviously, this function needs to be untouchable to the untrusted user.

Also important is the ability for a trusted user to use untouchables in building potentially unsound extensions to ACL2. REMOVE-UNTOUCHABLE, which is the inverse of PUSH-UNTOUCHABLE and requires an active ttag, was the first way of getting access to untouchables from user ACL2 code, but we decided to discourage its use in favor of a new construct that better captured the notion of making some functions or variables touchable with the intention of making them untouchable again later.

Thus, we introduced a notion of temporary touchability, captured in two (untouchable) state-global variables: TEMP-TOUCHABLE-FNS and TEMP-TOUCHABLE-VARS. These have touchable "updater" functions SET-TEMP-TOUCHABLE-FNS and SET-TEMP-TOUCHABLE-VARS respectively, which require an active trust tag to access. Each of these variables is bound to either a (potentially empty) list of symbols that are temporarily touchable or T, indicating all fns or vars are touchable.

One might ask why we did not choose a simpler solution of disabling untouchable checking when a ttag is active. The answer is to make the job of the certifier easier. Rather than needing to check through all ttagged code for use of untouchables, the certifier can check just those places in which there are temporary touchables. We make it easy for the author to identify to the certifier those places where untouchables are made temporarily touchable, by way of abstractions based on the ACL2 event aggregator PROGN. Our PROGN+TOUCHABLE and PROGN=TOUCHABLE add to and override (respectively) the set of temporarily touchable fns and vars for a sequence of events.

An example application of using an untouchable function in developing a system-level extension is in the "super history" code for ACL2s, whose *script management* interface calls for a more powerful notion of undoing and redoing commands than standard ACL2 has. Among other things, our code saves "the world" after each command is executed, and to revert to a previous logical state, we must reinstall an old world with SET-W!, which is untouchable. So we wrap the definition of the appropriate function with (progn+touchable :fns set-w! ...). To maintain soundness, the ACL2s code (if correct) only installs legitimate old worlds. This is part of the trust involved in using the ACL2s script management interface.

4.3 Redefinition

The ACL2 user can turn on a setting that allows redefinition of user functions from the command loop without using a ttag, but the user cannot certify a book in a world in which functions have been redefined, because one can easily use redefinition to reach a contradiction. With a trust tag, however, it is possible to redefine any function, even in a certifiable book. Our PROGN+REDEF implements this capability, enabling redefinition for the scope of its constituent events.

Because most of the ACL2 system is written in the ACL2 language, we can override its behavior with redefinition in the ACL2 loop. For example, the ACL2s "Recursion & Induction" mode, designed by J Moore, disables automatic, heuristically-chosen induction by the theorem prover, and this is implemented by redefining a few system functions.

The main caveat of redefining these functions is that we should avoid changes that could lead to a logical contradiction. This first means we should not redefine a function in the logic (:LOGIC mode) unless the new version computes the same function (presumably in a different way). Fortunately, most of the functions that we would want to behave differently are not defined in the logic but are in :PROGRAM mode. But even :PROGRAM mode functions can affect the logic in two ways: macro expansion and promotion to :LOGIC mode.

:PROGRAM mode functions, including those that take single-threaded objects (stobjs) such as STATE[4] can be used to define macros, and logical consistency of ACL2 depends on macros expanding in the same way in any extension of a world. :PROGRAM mode functions can also be promoted to :LOGIC mode with VERIFY-TERMINATION if they meet definitional requirements for the logic.

As of Version 3.2, ACL2 has a way of disabling these loopholes on a function-by-function basis. Specifically, if a :PROGRAM mode function is (re)defined after its name has been added to the state global variable BUILT-IN-PROGRAM-MODE-FNS, it can no longer be used in macro expansion or promoted to :LOGIC mode. These are insignificant concessions that take care of the loopholes. Note that BUILT-IN-PROGRAM-MODE-FNS is untouchable, so a ttag is required to manipulate it.

We offer an event ENSURE-PROGRAM-ONLY that first asserts that a function is defined only in :PROGRAM mode, and then adds it to BUILT-IN-PROGRAM-MODE-FNS if not already there. Note that this change is not tied to the world and so is not undoable in the traditional sense, but one could fix this minor issue with DEFCODE.

Another subtle caveat of redefinition is that if a function has a raw Lisp definition that does not match its ACL2 definition, redefining it based on modifying its ACL2 definition can cause problems. In fact, ACL2 utilizes many such functions, because they bridge the gap between the functionality available in the ACL2 programming world and that available in raw Lisp. Unfortunately, ACL2 does not currently keep track of which built-in functions are in this category, so we introduce a state global variable HAS-SPECIAL-RAW-DEFINITION, which should contain a list of all function names with a raw definition that is not observationally equivalent to its ACL2 definition.

GOOD-BYE-FN, for example, has a function body of NIL in ACL2, but it has a raw Lisp definition that exits the un-

[4]Stobjs are not available in the macro language, but logical counterparts can be used in their place.

derlying Lisp process. Redefining this function in the ACL2 loop would cause the function to lose its special behavior, so it should be in HAS-SPECIAL-RAW-DEFINITION and should only have its behavior changed by modifying its raw definition.

We define ENSURE-SPECIAL-RAW-DEFINITION-FLAG to add to the list HAS-SPECIAL-RAW-DEFINITION, for example when changing the raw definition of an ACL2 function, and ASSERT-NO-SPECIAL-RAW-DEFINITION to assert something is not on the list, for example when redefining a function in the ACL2 loop. In the future, the list might automatically be initialized with all the appropriate entries, but presently, we only include some such built-in functions that have been manually listed.

4.4 External Interaction

Trust tags also enable connecting the proof engine with arbitrary ACL2 code, via clause processors [7], and allow ACL2 code to write to arbitrary files, via OPEN-OUTPUT-CHANNEL!, and run arbitrary programs, via SYS-CALL. Each of these can alone render ACL2 unsound, as described in their respective documentation, which is why they require a trust tag. These are important to the overall picture of hacking and extending ACL2, but are not the focus of this paper. See [7] for more information.

4.5 Summary

Here are the abstractions we recommend for basic extension. Some are built-in and the rest are defined in the accompanying supporting materials:

- (defcode [:loop <loop-code>] [:compile <raw-compile-code>] [:extend <extend-code>] [:retract <retract-code>])
- (progn! [:state-global-bindings <bindings>] <form>*)
- (in-raw-mode <raw_form>*)
- (progn+touchable [:fns <fns-or-:all>] [:vars <vars-or-:all>] <event-form>*)
- (progn=touchable [:fns <fns-or-:all>] [:vars <vars-or-:all>] <event-form>*)
- (progn+redef <event-form>*)
- (ensure-program-only <fn>*)
- (ensure-special-raw-definition-flag <fn>*)
- (assert-no-special-raw-definition <fn>*)

5. HIGH-LEVEL EXTENSION IDIOMS

Whereas the previous section focused on low-level constructs and how they support hacking, we now focus on coherent, high-level idioms for hacking ACL2, abstractions we provide that take care of low-level details, and how these idioms can be used and combined to customize ACL2 in deep ways.

5.1 Raw-only Definitions

We define functions that safely add raw Lisp definitions from within the ACL2 loop. They are safe because definitions are made only if the names to be defined are not in use by ACL2 or raw Lisp. These use DEFCODE to associate the definitions with the ACL2 world, enabling them to be undone and even resurrected.

The names are DEFUN-RAW, DEFMACRO-RAW, DEFSTRUCT-RAW, DEFPARAMETER-RAW, DEFVAR-RAW, and DEFCONSTANT-RAW, which of course correspond to their Common Lisp counterparts without "-RAW" in the name. (See the summary below for basic syntax of each.)

DEFSTRUCT-RAW is the most complex because it figures out what functions would be defined by the underlying raw Lisp DEFSTRUCT. This one is special also because there is no portable way to undo a DEFSTRUCT in Common Lisp. We can, however, unbind all the functions defined by the event.

These functions also "stub out" the ACL2 names (with DEFLABEL) for which they give a definition in raw Lisp. This ensures that later ACL2 definitions will not interfere with the raw definitions made.

Also, these functions respect a non-nil LD-REDEFINITION-ACTION by removing the requirement that the names not be in use. If in use in ACL2, though, they must only be ACL2 labels to proceed, as would be the case if the names were introduced using one of our "-RAW" functions. This should not be viewed as a shortcoming, though, since the next section describes a meaningful way of changing the raw definition of ACL2 functions.

5.2 Bridging Raw Lisp and ACL2

The ACL2 loop rightly prohibits the definition of functions that depend on functions or constants defined only in raw Lisp. The way to bridge this gap is to let an ACL2 function have a special raw definition, as we described with HAS-SPECIAL-RAW-DEFINITION. Our DEFUN-BRIDGE creates new ACL2 functions that are implemented as special raw definitions, and our MODIFY-RAW-DEFUN adds to or replaces the functionality of an existing raw function definition.

DEFUN-BRIDGE appears much like DEFUN except that it takes declarations and a body for ACL2 and declarations and a body for raw Lisp. It first asserts no previous raw or ACL2 definitions (unless LD-REDEFINITION-ACTION is set) and then calls ENSURE-PROGRAM-ONLY and ENSURE-SPECIAL-RAW-DEFINITION-FLAG to avoid caveats described above. Then the :PROGRAM-mode ACL2 definition is made, and the custom raw Lisp definition is installed. We do not support :LOGIC mode for DEFUN-BRIDGE because it is intended for new metatheoretic ("system") functions.

MODIFY-RAW-DEFUN, on the other hand, can be applied to any raw function definition, whether it has a :LOGIC-mode ACL2 definition, a :PROGRAM-mode ACL2 definition, or no function definition at all in ACL2. MODIFY-RAW-DEFUN allows building a new raw definition of a function using the old definition as a "black box" function in the new definition. This code from ACL2s, for example, disables GOOD-BYE for the user:

```
(modify-raw-defun
  good-bye-fn ()
  :name-for-old-raw original-good-bye-fn
  :raw (progn
         (when (acl2s-protected-modep state)
           (hard-error
            'good-bye
            "Please use the user interface ~
             to exit.~%"
            ()))
         (original-good-bye-fn)))
```

The definition that was attached to GOOD-BYE-FN is now globally attached to ORIGINAL-GOOD-BYE-FN, and GOOD-BYE-FN now throws an error in some cases rather than invoking the old functionality (exiting).

5.3 Redefining System Functions

If a :PROGRAM-mode system function does not have a special raw definition, however, we can change its behavior with more control by using the ACL2 loop to redefine the function. In this case, we can look up the old body in the world, modify it according to our changes, and then use redefinition to store the new version.

Our REDEFUN looks just like a DEFUN but it takes care of all the details that promote soundness-preservation: checking that the existing definition is one we can overwrite, checking that the new definition has the same input and output signature, and calling ENSURE-PROGRAM-ONLY to keep it out of the logic. No LD-REDEFINITION-ACTION setting is required to use REDEFUN, but it does, of course, require an active trust tag.

A more flexible version of REDEFUN is our REDEFUN+REWRITE, which computes the new function body based on applying specified transformations to the old body. Here we take full advantage of the fact that, like Lisp, ACL2 code is composed of simple ACL2 objects.

REDEFUN+REWRITE's first parameter is the name of the function to modify, and any subsequent parameters comprise a *code rewrite specification* for transforming the body. Note that these "rewrite specs" have no connection with ACL2 rewrite rules, which solve a significantly different problem. Specifically, ACL2 rewrite rules are for simplifying new expressions we might encounter; our code rewrite specs are for changing the meaning of known code, modulo any non-interfering changes. Each uses "rules" for specifying pieces of translation, but ACL2 rewrite rules are "semantic" in the sense that they refer to properties of functions applied to values, while our code rewrite rules are "syntactic" in that they refer to code independent of its meaning or interpretation and can universally quantify over raw syntax. Our code rewrite rules can be recursive, but in a way that is well-founded, so that we never get stuck looping in translation. Finally, the order of rule application is easily defined, giving our translations only one interpretation.

Let us consider an example from ACL2s, in which we modify some functionality of the top-level ACL2 read-eval-print loop. Specifically, in LD-READ-EVAL-PRINT, we replace the one call of REVERT-WORLD-ON-ERROR with ACL2s's REVERT-SUPERHIST-ON-ERROR if we are in the top-most command loop. (The purpose of this change is to revert other important ACL2 settings on error, in addition to the world. For example, in pure ACL2 (er-progn (set-guard-checking nil) (defun foo (x) (foo y))) would have a side effect of disabling guard checking even though the (defun foo ...) failed, but ACL2s's "super history" reverts the setting.)

```
(redefun+rewrite
 ld-read-eval-print
 (:pat (revert-world-on-error %form%)
  :recvars %form%
  :mult 1
  :repl (if (= 1 (@ ld-level))
            (revert-superhist-on-error %form%)
            (revert-world-on-error %form%))))
```

The :PAT specifies a pattern to match in the body of the function; everything matches only itself, except for symbols among the :VARS and :RECVARS (both optional), each of which match any one thing. By convention, we put percent signs around our variables to make them stand out. :MULT asserts that the rule is applied a certain number of times, or within some range; an error is returned if there are too many or too few matches. :REPL is what the pattern is replaced by after substituting the values of the variables.

In the above example, the pattern and replacement are rather simple, matching any call to REVERT-WORLD-ON-ERROR, since variables such as %FORM% can match anything. More interesting is that we used :RECVARS instead of :VARS and added :MULT 1. Specifying :MULT 1 causes the translation to fail if we rewrite more than one instance of REVERT-WORLD-ON-ERROR. Using :RECVARS instead of :VARS in this case means that we also search for uses of REVERT-WORLD-ON-ERROR nested within other uses, because if a variable is in :RECVARS, then the current "simultaneous" set of rules is applied to the binding of that variable before using it in the :REPL [5].

We have used our code rewrite specification language to specify how to change the call to REVERT-WORLD-ON-ERROR in LD-READ-EVAL-PRINT, and to force reexamination of the situation if and only if there is not exactly one REVERT-WORLD-ON-ERROR in that function. There are more features to the code rewriting, and the above description is far from a specification. See the supporting materials for more information.

5.4 Copying Definitions

On many occasions we have wanted to create a new function based on the definition of an existing function. For example, before redefining a function whose behavior we want to change, we might define another function in the same way to preserve the old functionality under a different name. We provide some constructs to accommodate this and similar situations.

From the ACL2 loop, our COPY-RAW-DEFUN is a safe way of defining a raw Lisp function to be exactly the same as

[5]To make the recursion well-founded, we prohibit the case in which the pattern is a stand-alone RECVAR.

another. The copy is direct, in the sense that if the function associated with the source symbol changes, the function associated with the destination is unchanged, except as it refers back to the source function. Using `COPY-RAW-DEFUN` on recursive functions, therefore, can give undesirable results. `MODIFY-RAW-DEFUN` actually uses `COPY-RAW-DEFUN` in its implementation. `COPY-RAW-DEFMACRO` is analogous for raw Lisp macros.

`COPY-DEFUN` uses a `MAKE-EVENT` to generate a `DEFUN` event that defines a destination function from a source ACL2 function definition. Later redefinition of the source does not modify the destination, but `COPY-DEFUN`, like `REDEFUN+REWRITE`, asserts that the source definition at inclusion time be the same as it was at certification time. `COPY-DEFUN+REWRITE` is like `COPY-DEFUN` except it uses a `REDEFUN+REWRITE`-like specification to transform the copied function body.

5.5 Summary

- (`defun-raw` <name> <ll> <decl>* <body>)
- (`defmacro-raw` <name> <ll> <decl>* <body>)
- (`defstruct-raw` <name-and-opts> <slot desc>+)
- (`defvar-raw` <name> [<initial-value>])
- (`defparameter-raw` <name> <initial-value>)
- (`defconstant-raw` <name> <initial-value>)
- (`defun-bridge` <name> <ll> [`:doc` <doc-string>] [`:loop-declare` <decl-lst>] `:loop` <acl2-body> [`:raw-declare` <decl-lst>] `:raw` <raw-body>)
- (`modify-raw-defun` <name> <name-for-old> <ll> <decl>* <body>)
- (`redefun` <name> <ll> <decl>* <body>)
- (`redefun+rewrite` <name> <rewrite-spec>*)
- (`copy-defun` <src-name> <dst-name>)
- (`copy-defun+rewrite` <src-name> <dst-name> <rewrite-spec>*)

6. CONCLUSION

The first contribution of this paper is a discussion of trust tags in ACL2. We describe their motivation, when a trust tag is appropriate, and how to use them when appropriate. Trust tags open up ACL2 constructs that are dangerous in the sense that they can render ACL2 unsound or even effect malice on a user's computer, but they also facilitate unprecedented runtime customization of ACL2.

The other contribution of this paper is a set of ACL2 extensions for modifying and extending ACL2. In the process of hacking ACL2 in various ways, we have produced a nice set of dynamic extension idioms that promote soundness preservation, safe mixing of extensions, and checking of incompatibilities. These idioms bear some resemblance to aspect-oriented programming; we can specify where and how to insert or change code. We also attach changes to the ACL2 world, so that, if we so choose, the normal undo mechanisms can revert our code changes.

We have used these dynamic extension mechanisms quite successfully for the ACL2s development environment. Whereas we used to require building a custom ACL2 image with our communication hooks, we can now load them into a standard ACL2 image as we would a book of theorems. In addition, it is easier to adapt our changes to new versions of ACL2, because our method of specifying changes is based on structured syntax, and can be made as sensitive to other changes as is appropriate for each of our changes. Dynamic extension has also enabled us to package an automatic termination analysis based on calling context graphs (CCGs) [5] as a book that can be used orthogonally from the rest of ACL2s.

We hope these features encourage the development of new, innovative extensions of ACL2. That said, we recommend the use of ACL2's built-in mechanisms for customization and extension (from rewrite rules to `MAKE-EVENT`) before resorting to the types of extension we have described.

7. ACKNOWLEDGEMENTS

We thank Sandip Ray, Erik Reeber, and J Moore for useful contributions to conversations on topics of this paper. We also thank Daron Vroon for being a gracious customer and tester of this work.

8. REFERENCES

[1] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore. ACL2s: The ACL2 Sedan. In *Proceedings of the User Interfaces for Theorem Provers workshop (UITP), 2006*. ENTCS, 2006. Part of FLOC '06.

[2] E. L. Gunter. Adding External Decision Procedures to HOL90 Securely. In *Proceedings of the* 11*th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1998)*, volume 1479 of *LNCS*, pages 143–152. Springer-Verlag, 1998.

[3] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000. See URL `http://www.cs.utexas.edu/users/moore/-publications/acl2-books/acs/`.

[4] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000. See URL `http://www.cs.utexas.edu/users/moore/-publications/acl2-books/car/`.

[5] M. Kaufmann, P. Manolios, J. S. Moore, and D. Vroon. Integrating CCG analysis into ACL2. In *Eighth International Workshop on Termination*, August 2006. Part of FLOC '06.

[6] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

[7] M. Kaufmann, J. S. Moore, S. Ray, and E. Reeber. Integrating External Deduction Tools with ACL2. In *Proc. of the* 6*th International Workshop on Implementation of Logics (IWIL 2006)*, volume 212 of *CEUR Workshop Proceedings*, pages 7–26, 2006. Expanded version to appear, *Journal of Applied Logic*.

[8] P. Manolios and S. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *ICCAD-2005, International Conference on Computer-Aided Design*, 2005.