Embedding ACL2 Models in End-User Applications

Jared Davis[1]

Abstract. Formal verification, based on mechanical theorem proving, can provide unique evidence that systems are correct. Unfortunately this promise of correctness is, for most projects, not enough to justify its high cost. Since formal models and proof scripts offer few other direct benefits to system developers and managers, the idea of formal verification is abandoned.

We have developed a way to embed functions from the ACL2 theorem prover into software that is written in mainstream programming languages. This lets us reuse formal ACL2 models to develop applications with features like graphics, networking, databases, etc. For example, we have written a web-based tool for hardware designers in Ruby on top of a 100,000+ line ACL2 codebase.

This is neat: we can reuse the supporting work needed for formal verification to create tools that are useful beyond the formal verification team. The value added by these tools will help to justify the investment in formal verification, and the project as a whole will benefit from the precision of formal modeling and analysis.

1 INTRODUCTION

ACL2 [16] is an interactive theorem prover. It combines a Lisp-based programming language for developing formal models of systems with a reasoning engine that can prove properties about these models. It has been used to formally verify hardware at companies like AMD [25], IBM [26], and Rockwell Collins [30], and software like compilers [23], virtual machines [20], and operating system kernels [24]. ACL2's authors were awarded the 2005 ACM Systems Award for "pioneering and engineering a most effective theorem prover... as a formal methods tool for verifying safety-critical hardware and software." An ACL2 team shared the gold medal with a KIV team in the 2012 VSTTE Software Verification Competition.

Normally ACL2, or any other interactive theorem prover, is used to formally verify an *artifact*—a hardware design, a C program, an algorithm, a protocol, etc. This work is usually done by a team of experts and largely consists of three interrelated activities:

1. *Modeling.* A model of how the artifact behaves is developed in the theorem prover's logic. This is often a large undertaking. For instance, to model a program we may need to develop translation tools like preprocessors, parsers, etc., and may also need to formalize how the programming language behaves.
2. *Specification.* A specification for the artifact is written down as logical formulas. This is easy for some artifacts, e.g., it may only take a few lines to say what a particular operation of an arithmetic hardware unit is to compute. Other cases are much more difficult, e.g., what does it mean for an operating system to be secure?
3. *Proof.* The theorem prover is guided to show that the model meets its specification. Usually this is quite hard. Just how hard depends on the scale and nature of the model and specification, on the team's skill in developing effective proof automation, etc.

[1] Centaur Technology Inc. 7600-C N. Capital of Texas Hwy, Suite 300. Austin TX, 78731, email: jared@centtech.com

Why would anyone go to all this trouble? Formal verification usually reveals subtle bugs in the artifact (which, once exposed, can be fixed). It leads to mathematical proofs, checked by machine, that serve as evidence that the artifact has been designed correctly. The strength of this evidence depends on the precision of the model, the correctness of the specification, and the soundness of the prover. These concerns can often be addressed very convincingly.

1.1 Reusing Formal Models and Specifications

Unfortunately, due to the time and expertise it requires, formal verification is expensive. From a simple economic viewpoint, it only makes sense to formally verify artifacts whose failures could be very costly or tragic. This is still the case despite a lot of good work to reduce the costs of theorem proving by improving proof automation, interfaces, and pedagogy.

A different way to improve the cost/benefit situation is to increase the benefit. One way to do this, and the focus of this paper, is to reuse the modeling and specification efforts from formal verification in useful ways. Here are some examples.

Example 1: Processor Simulators. Normally, long before a processor design is to be manufactured, a program called a *golden model* is written to explain how the hardware is supposed to behave. As the hardware design evolves, it is continually simulated on test cases and compared against the golden model. This is often the primary way that bugs are found in the design. Since writing a golden model is much like the *Specification* activity of formal verification, an idea is to reuse the formal specification as the golden model. [12, 14]

This is not without challenges. A golden model needs to be something that the hardware design team can understand and practically use. A basic requirement is that the model should run at high speeds; fortunately ACL2 models and specifications are typically executable as programs, and ACL2 has many features [13] that allow for efficient execution. Other challenges include, e.g., how to connect the model to simulation tools for hardware design languages so that the design can be tested against the model.

When these challenges can be overcome, what does reuse accomplish? It avoids the need to separately develop the golden model and formal specification, directly reducing costs. It improves confidence in the formal verification effort, since the designers will have exercised the specification in their simulations. It also allows for formal analysis of the golden model, itself.

Example 2: Push-Button Analyzers. The *Modeling* activities needed for formal verification may be even more amenable to reuse.

At Centaur Technology we design an X86 processor. As part of our formal verification effort [27], we wrote an ACL2-based Verilog parsing and translation tool that builds formal ACL2 models of our hardware modules. Since then, we have reused this code in other tools like a linter and an equivalence checker.

These tools can be used by hardware designers with no background in formal verification. They have been quite useful: the linter has found many bugs that testing missed, and circuit designers are frequently using the equivalence checker to check their work.

1.2 The Right Language for the Job

We want it to be very easy to reuse ACL2 models and specifications to develop useful, related applications for end-users outside of our formal verification team. A basic question toward this goal is: what programming language should we use to write these programs?

If the tool we want to write is, say, a simple command-line utility that only needs to read some files and produce some output, then we might just use ACL2 itself as the programming language. This makes it trivial to reuse functions in our formal model.

Unfortunately, ACL2 is a poor platform for developing almost any other kind of program. For instance, it has very little support for working with the file system, limited multi-threading, no networking support, and no graphical interface. It also has no libraries for generating parsers, connecting to databases, working with widely-used data formats like JSON, XML, or YAML, and so on.

Instead, we might write our program in Common Lisp. The ACL2 system itself is a Common Lisp program, and ACL2 models and specifications are compiled into Lisp functions, so it is easy to call ACL2 functions from Lisp. Using Lisp also makes up for some of the deficiencies of ACL2 as our development platform, e.g., Clozure Common Lisp (CCL) has nice threading and networking support.

But frankly, Lisp is a niche language. It lacks the depth of modern, actively developed, well-documented libraries and frameworks enjoyed by mainstream languages. When development time and cost are at a premium, this may limit the kinds of tools we can develop. Using Lisp can also be deterrent to working with developers from other groups since usually they don't know the language.

What we really want, then, is a good way to embed ACL2 models into programs written in other languages—say Ruby, Java, or Python—that are widely known and have plentiful libraries to support working with files, graphics, networks, threads, databases, and so forth. Ideally, we should be able to choose whatever language we think is the best fit for the kind of application we want to develop, and then incorporate our ACL2 models into this language.

This leaves us with a practical problem: how can we effectively integrate ACL2 models into programs written in other languages?

1.3 Contributions

This paper describes the *ACL2 Bridge*, which solves this problem in a general way.

We extend ACL2 with an ACL2 Bridge server that accepts connections from client programs. Clients may be local or remote, and may be written in any practical language. Each client interacts with ACL2 through a kind of read-eval-print loop. Multiple clients can simultaneously interact with the same ACL2 instance. (Section 2).

We describe a Ruby interface to the ACL2 Bridge. We show how a client program can abstract away the details of communicating with the server. Our Ruby interface can execute ACL2 commands in an atomic style. It turns Lisp errors into proper Ruby exceptions, and allows output from ACL2 commands to be streamed as it is produced or collected for analysis. We show how to translate between Ruby and ACL2 data structures. These approaches can be followed to develop clients in other programming languages. (Section 3).

We give a concrete example of a real, end-user program based on the ACL2 Bridge. VL-Mangle is a web-based Verilog refactoring tool. It makes use of a large (100,000+ line) ACL2 codebase for Verilog parsing and transformation. A hardware designer with no knowledge of ACL2 can use the tool, through an attractive GUI, to manipulate sets of Verilog hardware designs and ensure that his changes are correct. (Section 4).

2 THE ACL2 BRIDGE

The ACL2 Bridge works by extending an ACL2 with a server that can respond to client programs. The server code can be loaded with

```
(include-book "centaur/bridge/top" :dir :system)
```

Afterward, a server can be started with the `bridge::start` command. The server listens for connections on a socket. If clients will be run on the same machine, we can listen on a Unix domain socket, which provides some security. For this, we just give the file name for the socket, e.g.,

```
(bridge::start "./my-socket")
```

To support clients on different machines, TCP sockets can be used instead. For this, we just give the port number to use, e.g.,

```
(bridge::start 13721)
```

But TCP sockets have security risks. Any client that can connect to the Bridge can execute arbitrary Lisp commands, including, for instance, running arbitrary programs via system calls. The Bridge has no authentication or encryption mechanisms, so you should never run it on a TCP socket without appropriate firewalls.

2.1 Soundness Considerations

Normally, ACL2 *books* introduce some new logical definitions and prove some theorems, but they do not alter the actual code of the ACL2 system. Unless there is some kind of bug in ACL2 itself, loading a book is *sound*, i.e., it will not allow ACL2 to say it has proved formulas that are not theorems.

The ACL2 Bridge book, however, necessarily extends ACL2 with Common Lisp code for capturing output, starting threads, dealing with sockets, and so forth. Once a server has started, its clients will be allowed to run arbitrary Lisp commands. There are no protections to prevent clients from unsoundly tampering with ACL2's state, e.g., a client could add bad formulas as axioms.

Because of this, loading the Bridge book "infects" ACL2 with a *trust tag*. In short: any ACL2 proofs carried out after the Bridge book is loaded are marked as less trustworthy.

Fortunately, this Common Lisp code is only necessary when you want to start a server and allow clients to connect. To avoid any soundness concerns, our recommended approach (Figure 1) is to not even include the Bridge book during the formal verification effort; the Bridge should only be loaded in the derived application.

It is easy to imagine also using the Bridge to help formal verification engineers develop ACL2 proofs, e.g., by writing graphical tools that make it easier to understand what the prover is doing. It

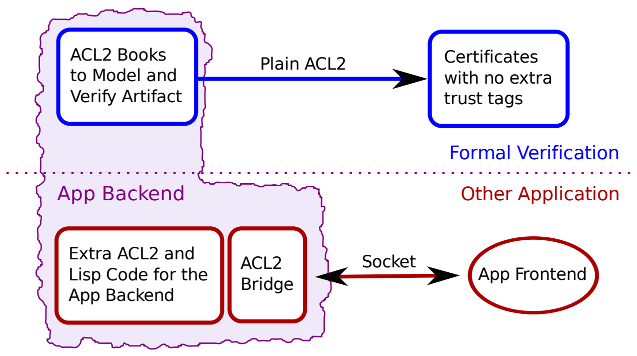

ACL2 Books to Model and Verify Artifact
Plain ACL2
Certificates with no extra trust tags
Formal Verification
App Backend
Other Application
Extra ACL2 and Lisp Code for the App Backend
ACL2 Bridge
Socket
App Frontend

Figure 1. Typical Soundness Approach

should be similarly easy to separate the ACL2 Bridge and such a tool from the formal verification effort. That is, the tool could be used while the proof is being developed, but not when the proof is *certified* (checked).

2.2 Communication

When a client program connects to the ACL2 Bridge, the server's listener thread creates a new worker thread to process its requests. The worker thread provides the client with a kind of read-eval-print loop for executing commands.

At the lowest level, all communication between a worker and its client is carried out using a simple message format which is meant to be easy to produce and parse in any language. In short:

$$\begin{array}{l} \textit{type}\ \ \textit{len}\backslash\texttt{n} \\ \textit{contents}\backslash\texttt{n} \end{array}$$

To be more precise:

- *type* is a label that matches `[A-Z][A-Z0-9_]*` and describes what kind of message this is,
- *len* matches `[0-9]+` and says how many bytes are in *contents* so that no escaping is necessary,
- *contents* are arbitrary bytes of length *len* which are the main part of the message,
- exactly one space separates *type* and *len*, and
- `\n` represents the newline character.

Upon startup, the worker sends the client a `HELLO` message with its own thread-name as the contents. (Some clients might want to remember the name of their worker to implement interrupts.) Then, the work loop begins. The loop has four steps:

Ready. The worker sends an empty `READY` message to indicate it is ready for a command. It then awaits input from the client.

Read. The client sends a command message to the worker. The type of this command can vary, and governs how the return value will be encoded (Section 2.3). The contents should always contain a single Lisp command (an S-expression) for the worker to evaluate.

Eval. If the command is well-formed, the worker runs it. During execution, the worker sends the client `STDOUT` messages with any printed output. These messages are sent as they are generated. A client might choose to display these messages to the end-user as they become available, or to collect them, or to ignore them.

Print. If the command completes successfully, the worker sends a `RETURN` message with the return value, encoded as requested by the client's command. In case of any run-time error, an `ERROR` message containing a description of the problem is sent, instead.

After sending the `RETURN` or `ERROR` message, the loop starts over again with a new `READY` message. The client continues to interact with the same worker until it disconnects.

2.3 Result Encoding

The ACL2 Bridge is intended to make it easy to embed ACL2 models into programs written in other languages. An important part of this is to allow the client to understand return values as proper objects in the client's programming language, not just as text.

In ACL2 and Lisp, objects are printed as S-expressions [21]. There is nothing especially bad about S-expressions, but they are not very popular and few programming languages have a standard library for parsing them. Because of this, if we want to write a client program that does something interesting with an ACL2 return value—say visualize some ACL2 structures, or compare our ACL2 model's answers against those from a hardware simulator—we would first need to write an S-expression parser.

While this would not be too bad, a much more convenient alternative is to have the Bridge encode return values in JSON [9] format. JSON libraries are readily available for any major programming language, and can be especially easy to use. To tie into these facilities, we added a JSON encoder for ACL2 objects to the server.

Clients can choose what kind of encoding should be used for the `RETURN` message on a per-command basis. The choice is just encoded into the command message type. We have four command types. The suffix `MV` here means "multiple values."

Command Type	**Result Format**
`LISP`	First return value as an S-expression
`LISP_MV`	List of all return values as an S-expressions
`JSON`	First return value as JSON text
`JSON_MV`	List of all return values as JSON text

Dealing with encoding on the ACL2 side, rather than on the client-side, makes each encoding available to clients from all programming languages. We might add other encoding options in the future.

3 A RUBY CLIENT

With the ACL2 Bridge server in place, how much work is it to connect ACL2 to another programming language? To see, we now walk through a Ruby client to the ACL2 Bridge. We first develop an `ACL2Bridge` class that deals with all aspects of messages and the work loop (Section 3.1). We then develop a mechanism for easily translating structured data between the two languages (Section 3.2).

3.1 Client-Side Communication

The `ACL2Bridge` class contains the actual socket and deals with the low-level aspects of communication. Its constructor establishes a connection to the ACL2 Bridge server. Its lowest-level routines implement our message scheme:

- `send_command(type, cmd)` sends the ACL2 server a message with the given `type` and with `cmd` as the contents.

- `read_message()` parses the next message from the server. It returns the type and content for a valid message, or throws an exception for a malformed message.

Higher-level routines bundle up the details of the read-eval-print loop to make it straightforward to just execute a command and get the result as a single step. A nice function is:

```
raw_command(type, cmd, stream=nil)
```

The syntax `stream=nil` means the `stream` argument is optional, and defaults to `nil` when omitted. What does this function do?

- It calls `send_command(type, cmd)` to send the command to the server.
- It reads messages from the server until a `READY` message is encountered. As each message is read, if an output `stream` (any object with a << method) has been provided, any `STDOUT` messages will be forwarded to this stream. This is useful for long-running ACL2 commands that print progress messages; you can show these messages to the user as they are generated, instead of having to wait for the command to complete.
- After `READY` has been read, it checks whether any `ERROR` message was encountered. If so, it throws the error as an exception. It then ensures that a `RETURN` message was encountered or throws an exception. Finally, it returns the contents of the `RETURN` message (i.e., the result of executing the ACL2 command, encoding according to `type`), and the concatenation of all `STDOUT` messages.

This sort of wrapper is very convenient when writing an end-user application. Instead of using `raw_command` directly, we usually use:

```
json_command(cmd, stream=nil),
```

a simple wrapper that calls `raw_command` with `JSON` as the command type, and then bundles up the result and all of the standard output into a single, JSON-encoded string.

Altogether the `ACL2Bridge` class comes to only 250 lines (of which 140 are blank or comments). It bundles up the details of messages and the work loop so that the client application simply submits commands and gets back the results and printed output. It converts any communication errors or ACL2-level errors into real Ruby exceptions, which can be caught and dealt with at the appropriate level of the client application. Porting this code from Ruby to other languages like Java, Python, C#, etc., would be quite easy.

3.2 Data Translation

The `ACL2Bridge` class abstracts away the message scheme and work loop, but still leaves us with an interface that is entirely string based. That is, the commands we send to the server need to be strings containing S-expressions. Likewise, the replies we get back are strings that contain printed S-expressions or JSON text.

Strings are fine in limited cases, but they are not a good representation for structured data. What we would ideally like, instead, is an easy way to translate between Ruby structures and ACL2 objects.

From Ruby to S-Expressions. Ruby has many kinds of objects. It makes sense to translate some of these (integers, symbols, arrays of strings, ...) into ACL2 objects. But for other kinds of Ruby objects (functions, sockets, its garbage collector, its Math package, ...) there is no sensible equivalent.

An especially nice translation from sensible Ruby objects into S-expressions can be implemented using Ruby's duck-typing and monkey-patching features.

- We start by extending (monkey patching) classes like `Integer` and `String` with a `to_lisp` method.
- We then similarly extend classes like `Array` and `Hash` with `to_lisp` methods that build suitable S-expressions using the `to_lisp` methods of their elements (duck typing). With just this, we can convert arrays/hashes whose elements are basic types, sub-arrays/hashes of basic types, etc., into S-expressions.
- As we write new Ruby classes for our application, we can add them to our translation scheme just by defining a `to_lisp` method. This immediately extends to arrays and hashes that contain these new objects.

This approach would not be directly portable to more strict languages like Java. In such a language, you might instead have a class with encoders for basic types, and use a `LispEncodable` interface with a `toLisp` function for new classes.

From ACL2 to Ruby Objects. In the other direction, we would like a way to convert ACL2 replies into Ruby objects. Using JSON encoding makes this particularly easy: in Ruby, we can convert some JSON-encoded string, `text`, into a native Ruby object, `obj`, like this:

```
obj = JSON.parse(text)
```

This approach works well for basic structures involving lists, alists, etc., but is not suitable for all ACL2 objects. For instance, if our data is represented as some special cons-tree, its JSON representation may end up being an bizarre nesting of two-element arrays. In these cases, it may be best to write a custom encoding function in ACL2, and explicitly run the encoder in your Lisp command.

4 EXAMPLE APPLICATION: VL-MANGLE

Connecting ACL2 to other languages lets us reuse a formal ACL2 codebase to develop modern applications for end-users beyond the formal verification group. As a concrete example, we have used the ACL2 Bridge to develop a web-based Verilog refactoring tool called VL-Mangle. VL-Mangle allows a hardware designer to mechanically transform sets of Verilog modules in certain ways. For instance, it can be used to:

- Inline away all uses of particular modules,
- Rewrite gate-level constructs into assignment statements,
- Remove excessive intermediate wiring,
- Eliminate unused wires and unnecessary expressions,
- Perform basic logical simplifications,
- Merge bidirectionally connected wires into a single wire, and
- Vectorize compatible sets of assignments.

These sorts of edits are tedious and error-prone to carry out on a large scale by hand. To support a particular design effort, we wanted a tool that hardware designers could use to automate these tasks.

A high-level picture of VL-Mangle's architecture is shown in Figure 2. The backend is written in ACL2. It reuses VL, our large (100,000+ line) ACL2 codebase for Verilog parsing and transformation. VL was originally developed to support our formal verification effort. To prove anything about our processor's modules (which are written in Verilog) we first needed a way to model these modules in ACL2, so we developed a parser and then translation code. Since then, we have significantly extended VL and used it to develop various command-line tools like a linter and equivalence checker.

The frontend of VL-Mangle is a web application written in the Ruby framework Sinatra [29] on the server side, and a typical mix

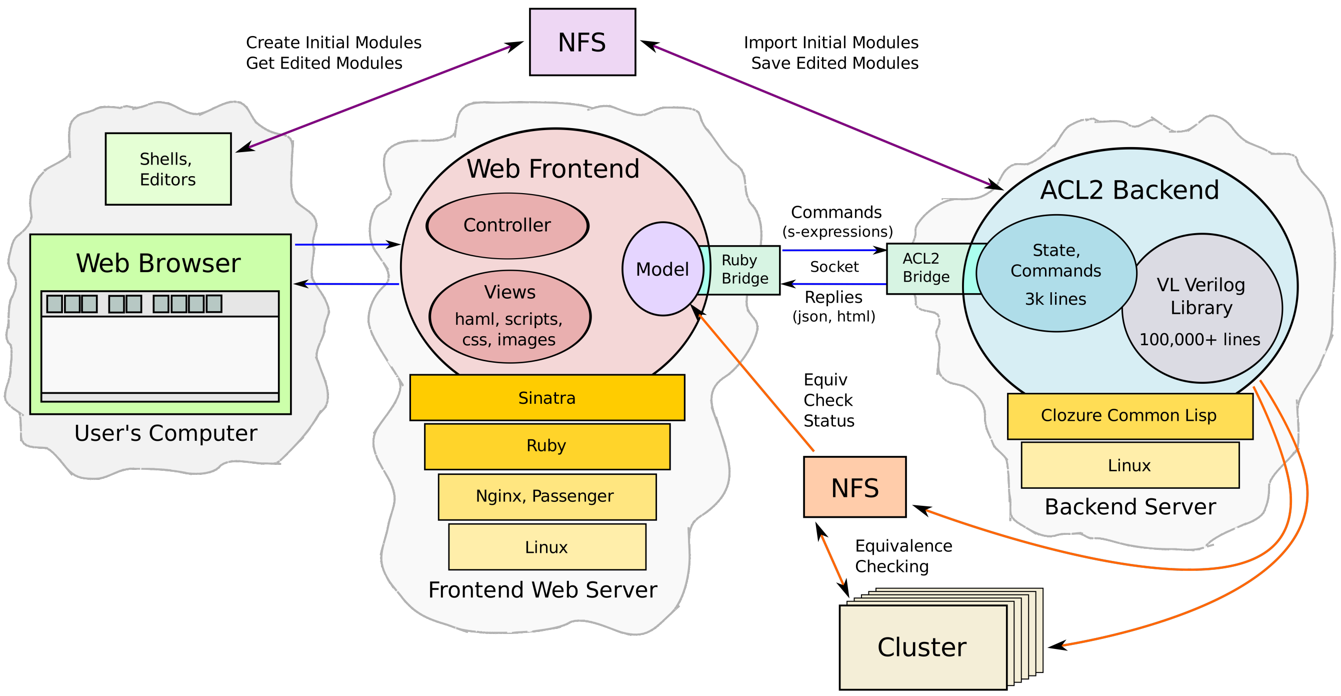


Figure 2. VL-Mangle Architecture

of HTML, CSS, Javascript/JQuery for the client. The end-user interacts with the tool using a graphical interface within his web browser. To give you a sense of the user experience, some screenshots of are shown in Figure 3.

4.1 VL-Mangle Architecture

The frontend follows the Model-View-Controller (MVC) [7] design pattern. In brief, this means it is divided into three parts. The *Model* includes the actual data and the ways of operating on this data. The *Views* are responsible for displaying data from the model to the user. The *Controller* is responsible for interpreting user input and translating it into operations on the model.

This architecture is very typical for web applications. In fact, there is almost nothing to say about our views and controller except that they are entirely conventional. Our controller is written using the Sinatra framework for routing and forms handling. Our views use libraries like HAML, Sass, and JQuery. All of this is separate from ACL2 except that we reuse some of the VL library's routines for pretty-printing Verilog modules.

The model is more interesting. Typical web applications might use an SQL database to hold their data. In VL-Mangle, most of the model is implemented within the ACL2 backend. The frontend, however, hides this behind a Ruby `Model` class that also regards certain files on a shared networked file system (NFS) as part of the model.

The ACL2 Backend. We think of the entire ACL2 backend as part of VL-Mangle's Model. The backend represents almost all of our application's data using ordinary ACL2 data structures. We reuse the Verilog representation from the VL library. The most interesting other data structures are *frames* and the global *state*.

A frame contains a list of Verilog modules and some other information. Each kind of automated edit (e.g., "inline modules") is implemented as a frame transformation. That is, given some starting frame, it produces a new frame that has updated modules. These are just ordinary ACL2 functions.

The global state has two stacks of frames: an undo stack and a redo stack. The top frame on the undo stack is the current frame. Basic undo and redo support is simple: to undo we move a frame from the undo stack to the redo stack; to redo we do the reverse.

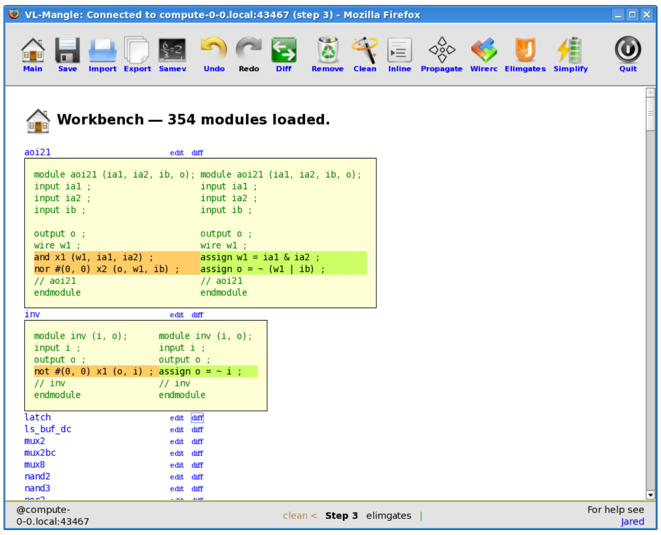
Since the state is an ordinary ACL2 object, it is easy to implement progress saving/reloading that preserves the full undo/redo history. Interestingly, none of this code needs to involve the ACL2 Bridge; we only need the Bridge when we want to connect our ACL2 model to the Ruby web application.

The Ruby `Model` Class. To connect the ACL2 model to our web application, we load the ACL2 Bridge book on the ACL2 side and load our Ruby `ACL2Bridge` class, S-expression encoding code, and the Ruby JSON library on the Ruby side.

Instead of exposing the `ACL2Bridge` instance to the rest of the Ruby application, we keep it within a `Model` class. In some ways this feels like overkill: there's not much to this class, and it might be simpler to just do without it. On the other hand, there are some nice features of this approach.

Having a `Model` class in Ruby allows us to treat data outside of ACL2 as part of the model. Our main use of this in VL-Mangle is for equivalence checking. To make equivalence checking faster, we set up a separate job to check each module, and run these jobs on a cluster. The VL-Mangle interface lets the user see the progress of these jobs and inspect failures. The data for these views are the log files from the equivalence checking tool. From an MVC perspective, then, it makes sense to regard these log files as part of the model. (Applications other than VL-Mangle might also want to consider various non-ACL2 resources like databases as part of their model.)

Another advantage of having a `Model` class is that it makes caching ACL2 queries quite easy. A particular view might be assembled out of independent parts. These different parts might each need to know, say, what the current module names are. We could just



Workbench — 354 modules loaded.

aoi21 edit diff

```
module aoi21 (ia1, ia2, ib, o); module aoi21 (ia1, ia2, ib, o);
input ia1 ;                     input ia1 ;
input ia2 ;                     input ia2 ;
input ib ;                      input ib ;

output o ;                      output o ;
wire w1 ;                       wire w1 ;
and x1 (w1, ia1, ia2) ;         assign w1 = ia1 & ia2 ;
nor #(0, 0) x2 (o, w1, ib) ;    assign o = ~ (w1 | ib) ;
// aoi21                        // aoi21
endmodule                       endmodule
```

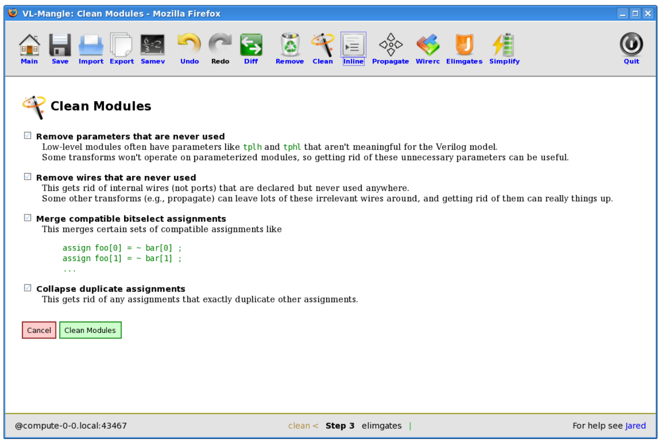
inv edit diff

```
module inv (i, o);       module inv (i, o);
input i ;                input i ;
output o ;               output o ;
not #(0, 0) x1 (o, i) ;  assign o = ~ i ;
// inv                   // inv
endmodule                endmodule
```

latch edit diff
ls_buf_dc edit diff
mux2 edit diff
mux2bc edit diff
mux8 edit diff
nand2 edit diff
nand3 edit diff

@compute-0-0.local:43467 clean < **Step 3** elimgates | For help see Jared

(a) The workbench shows the user the current versions of his modules, and lets him compare them with previous versions.



Clean Modules

☐ **Remove parameters that are never used**
Low-level modules often have parameters like tplh and tphl that aren't meaningful for the Verilog model.
Some transforms won't operate on parameterized modules, so getting rid of these unnecessary parameters can be useful.

☐ **Remove wires that are never used**
This gets rid of internal wires (not ports) that are declared but never used anywhere.
Some other transforms (e.g., propagate) can leave lots of these irrelevant wires around, and getting rid of them can really things up.

☐ **Merge compatible bitselect assignments**
This merges certain sets of compatible assignments like

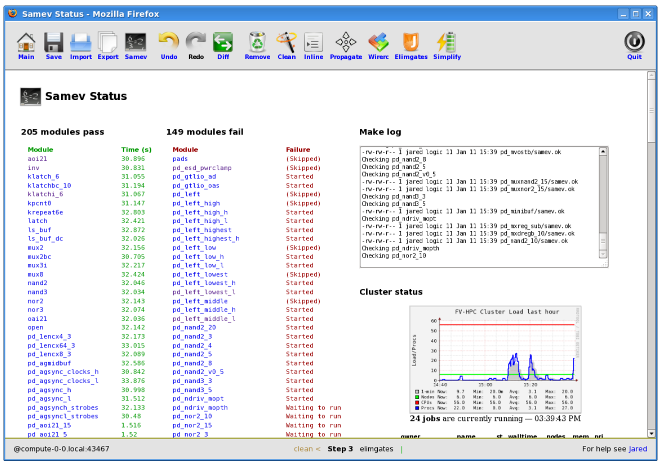
```
assign foo[0] = ~ bar[0] ;
assign foo[1] = ~ bar[1] ;
...
```

☐ **Collapse duplicate assignments**
This gets rid of any assignments that exactly duplicate other assignments.

Cancel | Clean Modules

@compute-0-0.local:43467 clean < **Step 3** elimgates | For help see Jared

(b) Tool pages let the user transform sets of modules in various ways.



Samev Status

205 modules pass / **149 modules fail**

Module	Time (s)	Module	Failure
aoi21	30.896	pads	(Skipped)
inv	30.831	pd_esd_pwrclamp	(Skipped)
klatch_6	31.055	pd_gtlio_ad	Started
klatchbc_10	31.194	pd_gtlio_oas	Started
klatchi_6	31.067	pd_left	(Skipped)
kpcnt0	31.147	pd_left_high	(Skipped)
krepeat6e	32.803	pd_left_high_h	Started
latch	32.421	pd_left_high_l	Started
ls_buf	32.872	pd_left_highest	Started
ls_buf_dc	32.026	pd_left_highest_h	Started
mux2	32.156	pd_left_low	(Skipped)
mux2bc	30.705	pd_left_low_h	Started
mux3i	32.217	pd_left_low_l	Started
mux8	32.424	pd_left_lowest	(Skipped)
nand2	32.046	pd_left_lowest_h	Started
nand3	32.034	pd_left_lowest_l	Started
nor2	32.143	pd_left_middle	(Skipped)
nor3	32.074	pd_left_middle_h	Started
oai21	32.036	pd_left_middle_l	Started
open	32.142	pd_nand2_20	Started
pd_lencx4_3	32.173	pd_nand2_3	Started
pd_lencx64_3	33.015	pd_nand2_4	Started
pd_lencx8_3	32.089	pd_nand2_5	Started
pd_agmidbuf	32.586	pd_nand2_8	Started
pd_agsync_clocks_h	30.842	pd_nand2_v0_5	Started
pd_agsync_clocks_l	33.876	pd_nand3_3	Started
pd_agsync_h	30.998	pd_nand3_5	Started
pd_agsync_l	31.512	pd_ndriv_mopt	Started
pd_agsynch_strobes	32.133	pd_ndriv_mopth	Waiting to run
pd_agsyncl_strobes	30.48	pd_nor2_10	Waiting to run
pd_aoi21_15	1.516	pd_nor2_15	Waiting to run
pd_aoi21_5	1.52	pd_nor2_3	Waiting to run

Make log

```
-rw-rw-r-- 1 jared logic 11 Jan 11 15:39 pd_mvostb/samev.ok
Checking pd_nand2_8
Checking pd_nand2_5
Checking pd_nand2_v0_5
-rw-rw-r-- 1 jared logic 11 Jan 11 15:39 pd_muxnand2_15/samev.ok
-rw-rw-r-- 1 jared logic 11 Jan 11 15:39 pd_muxnor2_15/samev.ok
Checking pd_nand3_3
Checking pd_nand3_5
-rw-rw-r-- 1 jared logic 11 Jan 11 15:39 pd_minibuf/samev.ok
Checking pd_ndriv_mopt
-rw-rw-r-- 1 jared logic 11 Jan 11 15:39 pd_mxreg_sub/samev.ok
-rw-rw-r-- 1 jared logic 11 Jan 11 15:39 pd_mxdregb_10/samev.ok
-rw-rw-r-- 1 jared logic 11 Jan 11 15:39 pd_nand2_10/samev.ok
Checking pd_ndriv_mopth
Checking pd_nor2_10
```

Cluster status

```
▨ 1-min  Now:  9.7  Min: 20.0m  Avg:  3.1  Max: 20.0
🟩 Nodes  Now:  6.0  Min:  6.0   Avg:  6.0  Max:  6.0
🟥 CPUs   Now: 56.0  Min: 56.0   Avg: 56.0  Max: 56.0
🟦 Procs  Now: 22.0  Min:  0.0   Avg:  3.1  Max: 27.0
```

24 jobs are currently running — 03:39:43 PM

owner name st walltime nodes mem pri

@compute-0-0.local:43467 clean < **Step 3** elimgates | For help see Jared

(c) The user can run equivalence checks on a cluster of machines to ensure his edits are behavior-preserving.

Figure 3. VL-Mangle Screenshots

separately query the Bridge each time we need to answer a question like this, but that is not very efficient since each query requires a round-trip to the ACL2 backend. Since the answer to this questions won't change during a single page load, a simple improvement is to cache the answers in the `Model` class so that we only need to consult ACL2 once, for the first query. In the particular case of VL-Mangle, this caching isn't necessary or important—we normally have a single user interacting with a single backend and performance is just not an issue—but for applications other than VL-Mangle, caching might be useful.

4.2 Connecting ACL2 to the Web

A very nice part of this whole system is just how easy it is to transfer input from HTML forms to the ACL2 backend and work with ACL2 replies. The user enters their input into ordinary HTML forms, with input names like this:

```
<input name="clean[parameters]" ... />
<input name="clean[wires]"      ... />
<input name="clean[assigns]"    ... />
```

Using the Sinatra framework, the corresponding handler in our controller can refer to `params[:clean]`, a Ruby hash that binds input names to their values. This makes it trivial to send these inputs directly to our Ruby `Model` instance:

```
reply = @model.clean(params[:clean], out)
```

The model just converts the arguments into an S-expression and runs the corresponding ACL2 command:

```
class Model
  ...
  def clean(args, out)
    @bridge.json_command(
      "(mpost-clean '#{args.to_lisp})", out)
  end
end
```

When we want to add new options and arguments to the cleaning transform, we can just extend the HTML form and its ACL2 implementation, without any changes to the Ruby model or controller.

Implementing an API for use in AJAX queries is also very simple. For instance, in the Ruby `Model` class we have a method to query ACL2 for the current module names.

```
class Model
  ...
  def get_modnames_json()
    @bridge.json_command("(mget-modnames)")
  end
  ...
end
```

Since `get_modnames_json` is already returning JSON-encoded data, we can just send this string directly to the web browser to respond to AJAX requests. All that is needed is an appropriate route in our controller. In Sinatra, this is just:

```
get "/get_modnames" do
  connect
  content_type :json
  @model.get_modnames_json
end
```

4.3 Threading Considerations

Most ACL2 functions can be thought of as pure, functional programs with no side-effects. These functions are especially well-behaved and no special care needs to be taken to make them thread-safe.

But not everything in ACL2 is pure. For greater execution efficiency, ACL2 models can also make use of certain non-pure idioms. For instance, they can use "single-threaded objects" that are updated destructively. At Centaur we actually use ACL2(h) [6], an extended version of ACL2 with hash-consing and memoization features. Unfortunately, its implementation of memoization is not thread-safe, and its implementation of hash-consing is most efficient when only a single thread is creating new hash-conses.

These sorts of features pose challenges when we are developing a multi-client applications where each client is served by a separate worker thread. Two clients might, for instance, simultaneously try to update the same single-threaded object, or both make use of memoized computations. The bridge does not do any automatic locking, so when we develop client programs, we must be aware of these issues and add the appropriate protections.

As a blunt solution, the Bridge does have a special "main thread" feature which is especially useful in the context of ACL2(h). In short: any computation can be wrapped in `in-main-thread` to ensure that it is run only by the main thread. This has the obvious disadvantage that a client may need to wait until the main thread becomes available. Another command, `try-in-main-thread`, is similar but just fails immediately if the main thread is not available. In VL-Mangle, we use this as our main locking mechanism.

5 OTHER APPROACHES

Getting separate programming languages to work together is a well-fought problem. Depending on the kinds of languages involved, we might combine the codebases into a single program by developing a foreign-function interface (FFI) or by sharing a multi-language platform like the Java Virtual Machine (JVM) or the Common Language Runtime (CLR). Alternately, we might keep the separate codebases as independent programs that simply process each others' files, or that communicate over pipes or sockets using anything from messages to elaborate protocols like COM and CORBA. Some of these approaches could perhaps be used, instead of the ACL2 Bridge, to connect ACL2 to other languages.

Common Lisp implementations like CCL and SBCL have foreign function interfaces that can call C functions from Common Lisp code, which could provide access to libraries for graphical interfaces, databases, etc. This can be particularly efficient. Calls through the ACL2 bridge have some communication overhead: the server converts return values into S-expressions or JSON representations, and the client generally has to parse this text into a sensible object. With an FFI, you may be able to directly construct or modify structures of interest in memory, without any parsing or printing. Unfortunately, while an FFI usually makes it reasonable to interface with C, connecting to higher-level languages like Java or Ruby is more difficult.

ACL2 does not run on any Common Lisp implementation that targets a platform like the JVM or CLR. However, there is at least one Common Lisp implementation on the JVM (ABCL) and languages like Clojure are similar to Lisp. Porting ACL2 to these platforms might open up interesting ways to connect it to the other languages that also run on the JVM, but would be a significant undertaking.

Instead of using separate programs that communicate in a cooperative way over a socket, we could perhaps use a pipe to run ACL2 as a sub-process and capture its standard input/output streams. This approach is used in the ACL2 Sedan [11], an Eclipse-based IDE for ACL2. This approach avoids the need to extend ACL2 with a server, which is nice since socket/threading code is not standard across Lisp implementations. Unfortunately, there are many practical difficulties for the client. The client must deal with invoking the process and capturing its streams, which is difficult in some programming languages. It must invent some way to tell when ACL2 is ready for more input, and to distinguish between output, return values, and error messages that are all printed to a single output stream. This approach is also limited to a single client, interacting with ACL2 via a single thread, on the same machine.

6 CONCLUSIONS

The ACL2 Bridge provides a straightforward way to embed formal ACL2 models and specifications into software written in any mainstream language. This allows us to reuse the work of formally modeling and specifying artifacts to develop full-featured applications that can be valuable beyond the verification team.

6.1 Related Work

In closely related work, Greve, Hardin, and Wilding [12, 14] explain how they have developed formal processor models that can be executed efficiently. This allows the formal model to be reused as a traditional processor simulator.

A unique reuse of executable formal hardware models is described by Albin, Brock, and Hunt [1]. They have reused a formal model of the FM9001 processor for post-fabrication testing. Test programs were run on the actual FM9001 chip while it was attached to a logic analyzer that recorded the values of its interfacing pins. These values were then compared against a gate-level formal model of the hardware design to show the physical device was behaving correctly.

To reuse formal models in other software, we need to be able to execute the formal model. ACL2 is unusual among theorem provers in that its logical definitions (i.e., for formal models and specifications) are directly executable as Common Lisp functions. In many other systems, logical definitions are often developed using, e.g., quantifiers, predicates, and relations that are not directly executable.

Even so, many provers have a mechanism for executing models. The Coq [5] system features a *program extraction* [19] capability that can translate subsets of Coq into OCaml programs. This capability has been used to develop some impressive standalone applications. For instance, Leroy [18] describes CompCert, a formally verified C compiler; Koprowski and Binsztok [17] present TRX, a verified parser generator. In each case, the programs extracted from these Coq developments could be useful to a wide audience.

A similar mechanism [3] for translating Isabelle/HOL specifications into ML has been used by Berghofer and Strecker [4] to create a verified compiler for a simplified Java. (It also has other uses within the theorem prover, e.g., Chaieb and Nipkow [8] have developed verified proof procedures for more efficient arithmetic reasoning.)

Similar to program extraction are schemes to *animate* [22] formal models in languages like Z and B, e.g., by translation into Prolog programs. Animation is ordinarily used to build confidence in the formal model by allowing it to be tested on examples. We are not aware of applications based on these animated models, but the ability to execute the model may serve as a useful step in this direction.

Less closely related, there are many cases where theorem provers have been connected to external programs like SAT solvers[28, 10], SMT solvers [2], symbolic algebra systems [15], and so on, which are usually written in languages like C or C++. These efforts allow formal verification engineers to automatically prove certain kinds of goals, but are not aimed at reusing formal models in applications.

6.2 Availability

The ACL2 Bridge, including for both the server-side ACL2 source code described in Section 2 and the Ruby client described in Section 3, is freely available under the GNU General Public License. It is included in the ACL2 Community Books for ACL2 6.0,

`http://acl2-books.googlecode.com/`,

under `books/centaur/bridge`. The Verilog library used in VL-Mangle, including parsing and many transformations, are also available under `books/centaur/vl`. However, the VL-Mangle web frontend is not publicly available.

ACKNOWLEDGEMENTS

I thank Scott Peterson and Patrick Roberts for very helpful conversations about web application development. I thank Sol Swords for his help to integrate equivalence checking into VL-Mangle. I thank Anna Slobadová, Sol Swords, and the anonymous reviewers for their helpful feedback on drafts of this paper.

REFERENCES

[1] Kenneth L. Albin, Bishop C. Brock, Warren A. Hunt, Jr., and Lawrence M. Smith, 'Testing the FM9001 microprocessor', Technical Report 90, Computational Logic, Inc., (January 1995).

[2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner, 'Verifying SAT and SMT in Coq for a fully automated decision procedure', in *PSATTT '11*, (2011).

[3] Stefan Berghofer and Tobias Nipkow, 'Executing higher order logic', in *Types for Proofs and Programs (TYPES)*, volume 2277 of *LNCS*, pp. 24–40. Springer, (2002).

[4] Stefan Berghofer and Martin Strecker, 'Extracting a formally verified, fully executable compiler from a proof assistant', in *Compiler Optimization Meets Compiler Verification (COCV)*, volume 82 of *Electronic Notes in Theoretical Computer Science*, (2003).

[5] Yves Bertot and Pierre Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer-Verlag, 2004.

[6] Robert S. Boyer and Warren A. Hunt, Jr., 'Function memoization and unique object representation for ACL2 functions', in *ACL2 '06*, pp. 81–89. ACM, (August 2006).

[7] Steve Burbeck. Application programming in Smalltalk-80: How to use Model-View-Controller (MVC). University of Illinois in Urbana-Champaign Smalltalk Archive. `http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html`, 1987. Accessed: January 2013.

[8] Amine Chaieb and Tobias Nipkow, 'Verifying and reflecting quantifier elimination for Presburger arithmetic', in *Logic Programming, Artificial Intelligence, and Reasoning (LPAR '05)*, volume 3835 of *LNCS*, pp. 367–380. Springer-Verlag, (2005).

[9] Douglas Crockford. JSON: The fat-free alternative to XML. `http://www.json.org/fatfree.html`, December 2006. Accessed: January 2013.

[10] Ashish Darbari, Bernd Fischer, and João Marques-Silva, 'Industrial-strength certified SAT solving through verified SAT proof checking', in *ICTAC '10*, volume 6255 of *LNCS*, pp. 260–274. Springer, (2010).

[11] Peter C. Dillinger, Panagiotis Manolios, J Moore, and Daron Vroon, 'ACL2s: The ACL2 Sedan', in *7th Workshop on User Interfaces for Theorem Proving (UITP)*, volume 172 of *Electronic Notes in Theoretical Computer Science*, pp. 3–18. Elsevier, (2006).

[12] David Greve, Matthew Wilding, and David Hardin, 'High-speed, analyzable simulators', in *Computer Aided Reasoning: ACL2 Case Studies*, eds., Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, 89–106, Kluwer, (2000).

[13] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J Strother Moore, Sandip Ray, José Ruiz-Reina, Rob Sumners, Daron Vroon, and Matthew Wilding, 'Efficient execution in an automated reasoning environment', *Journal of Functional Programming*, **18**(1), (January 2008).

[14] David Hardin, Matthew Wilding, and David Greve, 'Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle', in *Computer Aided Verification (CAV)*, volume 1427 of *LNCS*. Springer, (1998).

[15] John Harrison and Laurent Théry, 'A sceptic's approach to combining HOL and Maple', *Journal of Automated Reasoning*, **21**, 279–294, (1998).

[16] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June 2000.

[17] Adam Koprowski and Henri Binsztok, 'TRX: A formally verified parser interpreter', *Logical Methods in Computer Science*, **7**, 1–26, (June 2011).

[18] Xavier Leroy, 'Formal verification of a realistic compiler', *Communications of the ACM*, **52**(7), 107–115, (2009).

[19] Pierre Letouzey, 'Extraction in Coq: An overview', in *4th Conference on Computability in Europe (CiE)*, volume 5028 of *LNCS*, (2008).

[20] Hanbing Liu, *Formal Specification and Verification of a JVM and its Bytecode Verifier*, Ph.D. dissertation, University of Texas at Austin, 2006.

[21] John McCarthy, 'Recursive functions of symbolic expressions and their computation by machine, part 1', *Communications of the ACM*, **3**(4), 184–195, (April 1960).

[22] Tim Miller and Paul Strooper, 'Animation can show only the presence of errors, never their absence', in *13th Australian Software Engineering Conference (ASWEC)*, pp. 76–85, (2001).

[23] Lee Pike, Mark Shields, and John Matthews, 'A verifying core for a cryptographic language compiler', in *6th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2)*, pp. 1–10. ACM, (2006).

[24] Raymond J. Richards, 'Modeling and security analysis of a commercial real-time operating system kernel', in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, ed., David S. Hardin, Springer, (2010).

[25] David Russinoff, Matt Kaufmann, Eric Smith, and Robert Sumners, 'Formal verification of floating-point RTL at AMD using the ACL2 theorem prover', in *17th IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation*, (July 2005).

[26] Jun Sawada and Erik Reeber, 'ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool', in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 161–170. IEEE, (2006).

[27] Anna Slobadová, Jared Davis, Sol Swords, and Warren A Hunt, Jr., 'A flexible formal verification framework for industrial scale validation', in *Formal Methods and Models for Codesign (MemoCode)*, pp. 89–97. IEEE, (July 2011).

[28] Tjark Weber and Hasan Amjad, 'Efficiently checking propositional refutations in HOL theorem provers', *Journal of Applied Logic*, **7**(1), 26–40, (March 2009).

[29] Adam Wiggins and Blake Mizerany. Lightweight web services. RubyConf 2008. `http://rubyconf2008.confreaks.com/lightweight-web-services.html`, November 2008.

[30] Matthew M. Wilding, David A. Greve, Raymond J. Richards, and David S. Hardin, 'Formal verification of partition management for the AAMP7G microprocessor', in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, ed., David S. Hardin, Springer, (2010).