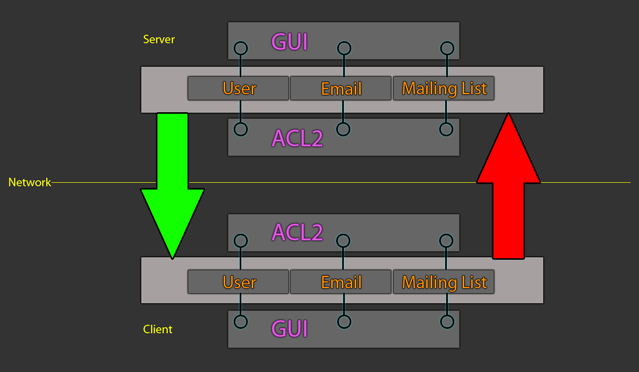
**Introduction**

The proposed application is that of an Email Client/Server transaction system that utilizes ACL2 in order to theoretically prove the correctness of the data transformations that occur on both the client and the server applications. While not all components of the system can be theoretically proven, we can provide the means to implement predicate based testing on the data acquired and formulate data structures in which we can inductively test, as opposed to the discrete methods currently implemented in the software industry. However, ACL2 is not without its shortcomings and as a result, we have had to implement some of the systems in a secondary language. For this application, we chose to use Java for interoperability. This also promotes a multilevel design that is easy to build upon and upgradable without detriment to the application.

**The Application Design**

  
There are two separate applications to this program, which must be running in a synchronized fashion. One is the client, which is the program that invokes the actions upon the program, and the other is the server, which is an automaton after the user starts it on a machine. This allows the client to send information to the server and expect a response based on the type of transaction it invokes. Each program consists of a modules, which in itself, is a standalone application. These modules may be called individually without the GUI intervention. They are, however, dependent upon ACL2 for data transformation. The structure of both applications is the same, with a GUI layer for client interaction, an ACL2 layer for data processing and a module layer for integrating the two and providing a means for invocation.

**ACL2**

The structure of the ACL2 that is implemented on the program consists of 3 separate parts:

1. The Logic
2. The Tests
3. The Actions

*The Logic*

The logic consists of the transformations that will occur upon the proposed data structures in the application. An example of this would be the *address-book.lisp* file.

; (addAddress addressBook address)  
; Appends an address onto the end of the address book structure. If the  
; user is already in the address book, then the return is the original   
; address book.  
; addressBook - the address book in which to add the address.  
; address - the address to add to the address book.

(defun addAddress (addressBook address)  
 (if (equal (isInAddressBook addressBook address) nil)  
 (append addressBook (list address))  
 addressBook))

In this code snippet, we examine the addressBook, which is a user defined structure – a list of addresses, address containing a domain, name and password. It uses a user defined function in order to determine whether the address is in the addressBook structure and if that is false, then we can append the address to the list of addresses. If not, then it is already in the addressBook, thus we can return the original addressBook.

There is not technically a type enforcement on this data, as it can actually hold numerical values. We can safely assume that we are not concerned with the type value until actionable information is passed from the invocation portion of the script – since we can explicitly cast this information, but safely assume our testing can include any such value this logic can be tested again – including integers, null values, strings, and even lists of strings. It suffices to say, at the logic level, we are not considerably concerned with the data the user has entered, rather whether the transformations are correct and true. This absolves the developer from considering this issue and allows them to focus explicitly on the logic.

The logic should be included in the modules root folder and a name given (such as *users* in the case of the example: *modules/users/address-book.lisp*). We will go into actions a bit later, but it is safe to say that anything that invokes this file should be a part of this module.

*The Tests*

Testing is done via Racket utilizing the Dracula environment. There are three alternative methods of testing – two of which are discrete, and one through induction. Proof through induction is the most solid means of verifying the transformations from the logic are true and correct. The other two include property based randomized testing (PBRT), and check-expects.

Proof through induction must be implemented through the ACL2 engine. You can use Racket in order to prove the formulated hypothesis, thus accepting it as a theorem into the logical world.

; Theorem:  
; If the address is not in the address book, then adding the address  
; to the address book will return an address book with the length of   
; the original address book + 1.

(defthm address-is-not-in-book-add-length-thm  
 (implies (and (listp addressBook)  
 (not (isInAddressBook addressBook address)))  
 (equal (+ (length addressBook) 1)   
 (length (addAddress addressBook address))))  
 :rule-classes (:rewrite :forward-chaining))

In this example, we are verifying that if we add an address to the addressBook structure that is currently not a part of the addressBook, we can conclude that the size of the addressBook is increased by one. Using the theorem prover yields us our result:

ACL2 Observation in ( DEFTHM ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM...): The :TRIGGER-TERMS for the :FORWARD-CHAINING rule   
ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM will consist of the list containing (LISTP ADDRESSBOOK).

ACL2 Warning [Non-rec] in ( DEFTHM ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM...): A :REWRITE rule generated from ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM will be triggered only by terms containing the non-recursive function symbol LENGTH. Unless this function is disabled, this rule is unlikely ever to be used.

ACL2 Warning [Free] in ( DEFTHM ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM...): A :REWRITE rule generated from ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM contains the free variable ADDRESS. This variable will be chosen by searching for an instance of (NOT (ISINADDRESSBOOK ADDRESSBOOK ADDRESS)) in the context of the term being rewritten. This is generally a severe restriction on the applicability of a :REWRITE rule. See :DOC free-variables.

ACL2 Warning [Subsume] in ( DEFTHM ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM...): The previously added rule COMMUTATIVITY-OF-+ subsumes a newly proposed :REWRITE rule generated from ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM, in the sense that the old rule rewrites a more general target. Because the new rule will be tried first, it may nonetheless find application.

ACL2 Warning [Free] in ( DEFTHM ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM...): When the :FORWARD-CHAINING rule generated from  
ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM is triggered by (LISTP ADDRESSBOOK) it contains the free variable ADDRESS. This variable will be chosen by searching for an instance of (NOT (ISINADDRESSBOOK ADDRESSBOOK ADDRESS)) among the hypotheses of the conjecture being rewritten. This is generally a severe restriction on the applicability of the forward chaining rule.

ACL2 Warning [Non-rec] in ( DEFTHM ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM...): The term (LISTP ADDRESSBOOK) contains the non-recursive function symbol LISTP. Unless this function is disabled, (LISTP ADDRESSBOOK) is unlikely ever to occur as a trigger for ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM.

<< Starting proof tree logging >>

Goal'  
([ A key checkpoint:

Goal'  
(IMPLIES (AND (CONSP ADDRESSBOOK)  
 (NOT (ISINADDRESSBOOK ADDRESSBOOK ADDRESS)))  
 (EQUAL (+ 1 (LEN ADDRESSBOOK))  
 (LEN (APPEND ADDRESSBOOK (LIST ADDRESS)))))

\*1 (Goal') is pushed for proof by induction.

])

Perhaps we can prove \*1 by induction. Three induction schemes are suggested by this conjecture. Subsumption reduces that number to two. These merge into one derived induction scheme. We will induct according to a scheme suggested by (LEN ADDRESSBOOK), but modified to accommodate (ISINADDRESSBOOK ADDRESSBOOK ADDRESS).

These suggestions were produced using the :induction rules BINARY-APPEND, ISINADDRESSBOOK and LEN. If we let (:P ADDRESS ADDRESSBOOK) denote \*1 above then the induction scheme we'll use is

(AND (IMPLIES (NOT (CONSP ADDRESSBOOK))  
 (:P ADDRESS ADDRESSBOOK))  
 (IMPLIES (AND (CONSP ADDRESSBOOK)  
 (:P ADDRESS (CDR ADDRESSBOOK)))  
 (:P ADDRESS ADDRESSBOOK))).

This induction is justified by the same argument used to admit LEN. When applied to the goal at hand the above induction scheme produces three nontautological subgoals.

Subgoal \*1/3  
Subgoal \*1/2  
Subgoal \*1/1  
Subgoal \*1/1.2  
Subgoal \*1/1.1

\*1 is COMPLETED!

Thus key checkpoint Goal' is COMPLETED!

Q.E.D.

Summary

Form: ( DEFTHM ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM ...)

Rules: ((:DEFINITION ADDADDRESS)  
 (:DEFINITION BINARY-APPEND)  
 (:DEFINITION ISINADDRESSBOOK)  
 (:DEFINITION LEN)  
 (:DEFINITION LENGTH)  
 (:DEFINITION LISTP)  
 (:DEFINITION NOT)  
 (:EXECUTABLE-COUNTERPART BINARY-+)  
 (:EXECUTABLE-COUNTERPART CONSP)  
 (:EXECUTABLE-COUNTERPART EQUAL)  
 (:EXECUTABLE-COUNTERPART LEN)  
 (:EXECUTABLE-COUNTERPART LENGTH)  
 (:FAKE-RUNE-FOR-TYPE-SET NIL)  
 (:INDUCTION BINARY-APPEND)  
 (:INDUCTION ISINADDRESSBOOK)  
 (:INDUCTION LEN)  
 (:REWRITE CDR-CONS)  
 (:REWRITE COMMUTATIVITY-OF-+)  
 (:TYPE-PRESCRIPTION BINARY-APPEND)  
 (:TYPE-PRESCRIPTION ISINADDRESSBOOK)  
 (:TYPE-PRESCRIPTION TRUE-LISTP-APPEND))

Warnings: Subsume, Free and Non-rec

Time: 0.00 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other: 0.00)

Prover steps counted: 1017

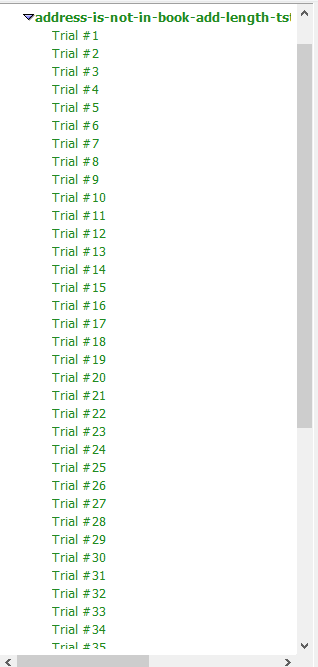
ADDRESS-IS-NOT-IN-BOOK-ADD-LENGTH-THM

As we can see, the theorem has been accepted and the inductions require are listed above. It also goes into a full textual description of the proving steps and lists the number of steps required to prove the function. Thus we can assume that the logic for adding an address to the addressBook is theoretically sound.

PBRT describes a hypothesis of the transformation of a function to be tested. Random values are assigned, and injected into this logic and tested for its trueness. The range of testing can be user specified, but for our sake, we have implemented a default value of 50 cycles. The formulation of the proof structure is similar to that of the theorem test, but we are required to define out input values.

(defproperty address-is-not-in-book-add-length-tst  
 (domain :where (stringp domain)  
 :value (random-string)  
 name :where (stringp name)  
 :value (random-string)  
 pass :where (stringp pass)  
 :value (random-string)  
 domain-not :where (stringp domain-not)  
 :value (random-string)  
 name-not :where (stringp name-not)  
 :value (random-string)  
 pass-not :where (stringp pass-not)  
 :value (random-string)  
 address :where (listp address)  
 :value (list domain name pass)  
 address-not :where (listp address-not)  
 :value (list domain-not name-not pass-not)  
 addressBook :where (listp addressBook)  
 :value (list address))  
 (implies (not (isInAddressBook addressBook address-not))  
 (equal (+ (length addressBook) 1)   
 (length (addAddress addressBook address-not))))  
 :rule-classes (:rewrite :forward-chaining))

We are using exactly the same hypothesis that we proved in ACL2. The only different is the discrete values that are populated. Though these values are considered to be random, they are limited, this is not a correct bearing for the correctness of the hypothesis, but rather standard foundation for augmenting out hypothesis if we are unable to get it to verify correctly in ACL2. The results of this test are:



We can view each individual trial, thus if there is an error, we can see the values as opposed to trying to change our hypothesis based on theoretical results – usually it is more intuitive for developers to see a value that is causing it to fail.

Sometimes developers may find this process frustrating as theory is generally more difficult to prove over that of expected results with known inputs. This is where the check-expect typically helps. With it, we can enter a value that we predetermine and verify it against the expected output (producing a true/false result).

(check-expect (length (addAddress '((1 2 3)) '(4 5 6))) 2)

This statement states that if we add the address *(4 5 6)* to the addressBook that contains *(1 2 3)* only, we will acquire the addressBook with the additional address, verified by the length of the total address book – in this case, addressBook is a list of lists or numbers (three values). Running this test will only result in an echo of “test passed” or “test failed”, with no reasoning.

Typically these logic files are included in the modular folder with the logic file (usually suffixed by *\_tests*). No naming conventions are enforced at this level other than knowing that this file is not to be invoked by the ACL2 subsystem when the module is running. It is explicitly used for testing purpose.

*The Actions*

Logic is for naught if we are not doing anything with it. This is where our actions come into play and the integration into or client/server environment. The actions are the standalone program portion of the module that that will perform the IO as well as any network connectivity. Portions of this are written in ACL2 and others are written in Java. There were some considerations previously of writing the whole module in ACL2, but network connectivity proved to be a challenge. This was later changed to shell scripting in the BASH shell, but synchronization through connectivity, yet again, proved to be an issue. We could send information across a network, but waiting for a response proved to be an issue. In the end, we decided that Java was the best alternative and many students would integrate into this environment seamlessly. This is also where the client and server start to deviate in code production.

**The Module**

There are typically two parts to the action of each module, but a user can define as many as they need to complete the task. You can implement GUI elements in this application or you may even perform some third party processing to get the information ready for ACL2 processing. For the sake of demonstration, we will be implementing this in a simplistic manner through the user registration process.

Since the action we defined is register, a subfolder in the *user* folder is created and will contain the files responsible for registration. We can define the files as needed, but we are going to invoke against the RegisterUser class. Since the execution of the server occurs from the root, we must include this Java class in the package: “*modules.user.register”* which makes the fully qualified name for invocation *modules.user.register.RegisterUser*. More will be compounded upon this when the GUI wrapper is introduced. We also need a minimum of one lisp file that will generate IO for the ACL2 environment and put the logic to work on data structures. This is also the point to which the user should be concerned with the data types being introduced into the ACL2 environment, since it will be responsible for writing persistent data.

Example of lisp required:

; (registerUser regXML abXML state)  
; Processes the information that is passed via XML string to add the user  
; to the global server address book.  
; regXML - The XML that is contained in the registration file - sent   
; dynamically via shell script.  
; abXML - The XML that is contained in the address book file - sent   
; dynamically via shell script.  
; state - The state of the streams in ACL2.

(defun registerUser (regXML abXML state)  
 (let\* ((tokens (tokenizeXML regXML))  
 (domain (getDomain tokens))  
 (name (getName tokens))  
 (pass (getPassword tokens))  
 (addressBook (getAddressBook (tokenizeXML abXML))))  
 (mv-let (error state)  
 (string-list->file   
 "store/address-book/temp\_address-book.xml"   
 (getAddressBookXML (addAddress addressBook (list domain name pass))) state)  
 (if error  
 (mv error state)  
 (mv "Wrote temp\_address-book.xml successfully" state)))))

This example provides an entry point that will attempt to add the user to the addressBook and store it for persistence. There are a couple of things to note – we are writing to ­*temp\_address-book.xml* which generally does not denote permanent storage, but because of one of the shortcomings of ACL2, we cannot overwrite a file that is currently open for read. This will be expanded upon when we discuss the GUI integration.

The XML is tokenized from the incoming request, parsed into the address and addressBook data structures. This information is fed directly from the Java file into this function and pushed into ACL2. The response is written and additional logic in the Java file ensures that the addressBook data structure has changed by verifying the sizes of the *temp\_address-book.xml* and *address-book.xml* files.

We can then move on to the java file construction. In this file, the networking (if needed) and typical IO functions that cannot be handled by ACL2 generally occur. Since this is a standalone application, it must have a main method associated in the file. From there we can call the *ProcessBuilder*, and invoke ACL2 by building “wrappers” for the ACL2 input and output. You are able to feed ACL2 commands and functions directly into the environment, after which you must call *(good-bye)* in order to exit the process. General cleanup of memory must occur and the process is repeated for every incoming request.

After the Java file is constructed, you must compile it and register the binary in the server module registration manager (for server side modules). Client side requires integration into the application at this point, which is direct manipulation of the source code and recompilation.

**The GUI**

Modules are invoked from a *ProcessBuilder* similar to the one that invokes ACL2. This requires that we register each module so the user can identify it. Since this is allowed on the server side, there is no overhead for integration into the Server UI, rather just standalone application specifics that are introduced when the module has been ran.

To register the module on the server, execute the Server class through a console (*java Server*). A GUI should show up with a menu at the top of the screen. Under the *“modules”* menu, you will find *“management”.* Clicking this will open a list of modules currently registered with the server and by clicking the “*add”* button, you can add the module. It is important to note that you cannot register modules that are listening on the same network port. They will create a confliction and fail to start. The registration will simply reject it completely. The fields are fairly straight forward with the name of the module, location and port. Location can be selected via the locator button, which will generate the expected package name for invocation.

\*\* Need to add:

* Timeline (Ghantt)
* Software Requirements (IE Interoperability, language)
* Defect database (Change Log, Google Bug Database)
* Inspiration for project
* Previous Ideas
* Requirement readjustments
* Software Development Process (Prototyping)
* Component delegation (Server Side / Client Side)
* Shell to Java conversion (specifically)
* Actual Email file format vs New Format
* External Email