**Introduction**

The client program of the email transaction system utilizes ACL2 to verify and prove the correctness of transmissions that occur on the client program. Each part of the client has components that cannot be theoretically proven due to invariant data. To provide a means of testing these functions, we need to develop predicates based on the expected input and output of these modules.

Not all modules in the client could be written in ACL2. In order to implement the system monitoring tools, user interface, and network connections, we used Java programs to handle each of these aspects. This allows for the multilevel design and independence of operating system environment that can be used with this program.

**Client Requirements**

The client will be the user end interface to the email system. Since the client will work directly with the users, we need to ensure that we account for user freedoms and input mistakes. This means that we will need to use theorems to ensure data consistency between functions.

Basic requirements for the client include:

* The client will need to be able to register users on the server. There should be functionality to accept a username, domain name, and password.
* The need to accept input for an email message. These inputs need to be the address(es) of the recipients (comma ',' separated for multiple recipients), the sender's address, the subject, and the message content.
* The ability to call ACL2 functions to obtain messages from the server.
* The ability to route files from the incoming directly, process the contents, and place the results in appropriate store files.
* The ability to split emails addressed to multiple recipients into separate messages.

Details of the implementation process of these features are discussed in the following sections.

In the implementation structure of the program, the following tools will be used in developing features for the client.

**ACL2 version 6.1** (<http://www.cs.utexas.edu/~moore/acl2/>) will be used for all the logic functionality of the program as well as generating IO and data processing of email messages and requests.

**Java SDK 1.7** (<http://www.oracle.com/technetwork/java/index.html>) will be used for all the invocation of ACL2 in the program as well as socket connections to the server.

**Dr. Racket with Dracula** (<http://www.ccs.neu.edu/home/cce/acl2/>) will be used for all Property-based test, check-expects, and theorems associated with the logic functions of the program.

**Proof Pad** (<http://proofpad.org>) an alternative IDE for ACL2.

**Client Design**

The client is developed into modules with three major components: the logic, test, and actions. Each module runs independently from each other, thus making each module a stand alone application itself. The modules are tied together into the GUI in order to make reading the results and executing the application easier. However, the GUI is not needed in order to make the application run. The layout of each layer is described below.

**ACL2**

The ACL2 components are structured into 3 parts:

1. The Logic
2. The Test
3. The Actions

**Logic**

The logic consists of transformations that need to be handled by the proposed data structures in the application. An example of this would be the client-email.lisp file.

;(email to from sub msg)

;Generates the email XML for email messages based on user string input.

(defun email (to from sub msg)

(if (consp (cdr (splitToField to)))

(multRecip (splitToField to) from sub msg)

(getEmailXML (generateEmailFromStrings (car (splitToField to)) from sub msg))

)

)

In this code, we take in the contents of an email message that will need to be generated into an XML format. The function takes in four parameters with each being a corresponding component to an email message. Once the function has completed, we are left with the XML contents for an email message that will then need to be stored for output, which will be handled in the actions component. An example of the raw XML output would be:

<?xml version='1.0'?><!DOCTYPE user SYSTEM '../../../dtd/messages.dtd'>

<email>

<to>howell@localHost</to>

<from>crist@localhost</from>

<subject>SUB001</subject>

<content>MSG001</content>

</email>

As you can see, each of the four fields is generated into an email message format. This XML format is strictly defined in the document type definition located in the directory specified. This DTD must be included on BOTH the client and server in order for the XML definitions to pass.

Logic files will need to be stored in the root directory of the module and given a name such that it represents the logic it is performing (For the above example, the email module on the client is stored in the following directory: modules/email/client-email.lisp). Any operation that needs to include this main logic file should be part of this module.

**Testing**

Testing is done through the Dr. Racket and Dracula interface. There are three methods of testing which include Theorems, Properties, and Checks. Theorems are proofs through induction of your methods. Properties and Checks utilized specified and random data to test your code for expected outputs.

Proofs through inductions that are accomplished through the theorems are the soundest way of testing your logic code. It is safe to say that if your theorem passes your inductive hypothesis, your function produces correct output according to your hypothesis. You can use Dr. Racket for theorem proving. An example theorem would be:

;(genEmailProducesMessage)

;Theorm to test that the email structure

;returned from generateEmailFromStrings is the correct

;email structure ((list) str str str)

(defthm genEmailProducesMessage

(implies (AND (stringp msg)

(AND (stringp sub)

(AND (stringp to)

(stringp from))))

(AND (listp

(generateEmailFromStrings to from sub msg))

(listp

(car (generateEmailFromStrings to from sub msg))))

))

This theorem proves that an email message structure is returned from the generateEmailFromStrings function. It test to see if the format of the output follows this format. ((list) str str str) If it does, then the correct structure of an email message has been processed and returns a true result. Since this theorem passes ALC2 logic, it is safe to say that our logic produces the desired output.

Property based testing and Check Expects are needed to test boundary cases of your logic functions. These forms of testing are useful if you are experiencing trouble with specified conditions and if random data is needed for your project. Since the email client relies heavily on invariant data, these forms of testing are not idea for this application.

It is also worth noting that through the Racket interface, a property-based test can be passed to the ACL2 logic mechanism. The same logical induction is performed on the properties and its results are the same as if the test were a theorem. So you can use a property as if it were a theorem in this case.

**Actions**

The actions are where the logic portions of the module are executed. The actions file should contain all the external dependencies of the module. Also the actions should also perform all the IO operations for your logic. This is done to help the module become its own stand alone application. In the client, the actions layer is developed into two programs. A Java program handles the invocation and network connections, and the ACL2 program which handles the ALC2 logic and IO for the module.

We need to expand on the Java program. Each module will need a Java program to be included with the actions of a module. This will take care the network connections, ACL2 invocation, and interfacing with the GUI. The Java program will need to be developed as a library, which deviates from the server. This will allow the module to stand on its own, as well as be expanded on in the future. An example of a header for the Java program follows:

package modules.email.action;

import java.io.\*;

import java.util.\*;

import java.net.\*;

public class GetEmail {

public final static String OUTPATH = "store/email/outbox/";

public final static String INPATH = "incoming/email";

public static void getEmail (String name, String domain, String password){

As you can see, this Java program is registered in a package. This package will need to match your file path the location of your Java class. This will allow other applications in the client program to import your Java class and use its functions.

Also this program will contain statically defined functions, which the calling program will be able to use. These functions will use Java’s ProcessBuilder to build out a process to ACL2 and run the ACL2 environment. An example of the ACL2 call for this action would be:

String script = "(in-package \"ACL2\")(include-book \"modules/email/action/rw-email\"" +" :uncertified-okp t) (readEmail \"incoming/email/"+f.getName()+"\" \""+unique+"\" state)";

try{

//Run on ACL2

// Initialize ACL2 and dump its output to the log

System.out.println("Executing ACL2 runtime for Email Generation...");

ProcessBuilder processBuilder = new ProcessBuilder("acl2");

File log = new File("logs/acl2\_log.txt");

processBuilder.redirectErrorStream(true);

processBuilder.redirectOutput(log);

Process process;

process = processBuilder.start();

PrintWriter procIn = new PrintWriter(process.getOutputStream());

// Write the ACL2 to the process, close ACL2

procIn.println(script);

procIn.println("(good-bye)");

procIn.flush();

procIn.close();

} catch(IOException e) {

e.printStackTrace();

}

This code will create a new process; call the ACL2 environment, and the build a string that is an ACL2 call to your ACL2 action file. The output of the ACL2 environment is then stored in a log folder in your client’s programs root directory. This is a way of seeing where the client program failed during debugging.

**The Module**

There are usually two parts to the to the action of a module. But it is possible to expand a module to more than two actions if it is needed to get a task completed. In order to explain the structure of a module, we will show the example of sending and Email message to the server.

The action to be performed is email; a subfolder in the modules folder is created. This root folder, email, will contain the logic and test for the client-email platform. Within the email directory, another subfolder called actions is created. This folder will contain both the ACL2 actions and the Java actions for the email module.

Since the execution of this module comes from the main root of the client application, we need to register the Java classes associated with sending emails with the following path: “modules.email.action” which makes the fully qualified name for invocation modules.email.action.sendEmail. We also need at least one lisp file in this directory which will contain the ACL2 IO functions needed to generate an email message to be sent.

An example of one of these lisp files that will need to be included in the actions directory:

(in-package "ACL2")

(include-book "../../../include/xml-scanner" :uncertified-okp t)

(include-book "../../../include/io-utilities" :uncertified-okp t)

(include-book "../client-email" :uncertified-okp t)

(set-state-ok t)

(defun writeEmailToFile (xmlStr to ts state)

(mv-let (error state)

(string-list->file (concatenate 'string

"store/email/outbox/"

to ts ".xml")

xmlStr

state)

(if error

(mv error state)

(mv "XML File Written Successfully" state))))

(defun writeMessage (to from sub msg state)

(let\* ((msgs (email to from sub msg)))

(writeEmailToFile msgs to from state))

)

This example lisp file will attempt to generate the XML file for the email message the user has inputted into the function *writeMessage* and store it into the store/email/outbox directory on the client.

The input though, is first sent to the *email* function in the client logic to generate the XML, if the user had addressed the email to multiple people, then the logic will handle the delegation of generating a list of multiple XML outputs. Once the logic has completed the function then saves the list of XML files in the outbox directory.

Once ALC2 has finished and saved the XML file in the Outbox, the Java process continues. It will open the file in the outbox and check the file to see if one or more XML messages are contained in the file. If more than one message is detected by using a regular expression, Java will split the file into multiple files such that only one message is included with each file and holding with the DTD definition.

Once the files are split, Java will send the messages to the server using a Java socket command. This is outlined in the Java program and is simple to implement. Most of the communications will occur on the following ports: 20002 for user verification, 20003 for user registration and 20005 for sending emails. Since we are sending an email, we will use port 20005 on the server to send the message.

This gives a basic overview of how the actions are performed in the client program. The other modules follow a similar pattern as sendEmail except each action has its own port as noted earlier. Once you complete a Java file, make sure you compile the *.class* file in the same directory as the Java source code file. This will allow the action to be reachable from the root of the application and be used in other applications such as a GUI interface.

**The GUI**

Modules are invoked by a static method inside the modules Java library. This requires you to import the library in the Java program you are using. To import your Java library, you should use the import command using the path to your source code. This path should match the package name of your Java class. An example import would be:

import modules.email.action.\*;

This will allow you to call the static action functions in the library and use the ACL2 modules from a Java program. For the *sendEmail* example, you would call SendEmail.sendEmail(“To”, “From”, “Subject”, “Message”). This will call the library and invoke the send email process in ACL2.

By producing the Java programs in this fashion, we allow modularity of the program to remain independent of the GUI and other invocations. This has also allowed us to develop a stand-alone client GUI interface for the client. Once this is compiled on your machine. You will need to access the client directory from your command terminal. Once here you can issue the command:

java ACL2Email

This will launch the Email client GUI interface. The current modules already imported into the interface are:

* Send Email
* Verify User and Get Email
* User Registration

Once more modules are developed; they can be easily implemented into this interface. With this structure, you are also able to make changes to the action files without having to worry about changing the GUI. The only action the GUI should make is to the static function in the actions library file.

**Implementation Issues and Solutions**

One issue that arose was to issue of multiple file IO with ACL2. ACL2 provides the mechanisms for a single read and write operation. You cannot recursively call an IO operation in ACL2 as it won't admit to ACL2's built in logic mechanism. The solution to this issue was to have ACL2 output a single file that contained multiple email messages in XML format. Then we used a regular expression in the invocation script to split the file at each XML header. This was a simple but effective solution that allowed ACL2 to handle messages sent to multiple recipients with a single invocation of the *generateEmail* function.

For ACL2 invocation and network monitoring, the original idea we developed was to use a program monitor written in C, which could then be compiled, on a local machine. This would allow an operating system independent environment as long as C could be compiled.

However, when we decided to switch to shell scripting, we had to force the requirement of using a Unix machine to run our program. As we continued to develop the in the Unix environment, the issue of process synchronization was troublesome. This allowed for only one transmission at a time to occur over the network. Then we would have to wait for the receiver to finish handling the transmission before sending the next one. This caused a loss of data when multiple transfers were occurring.

To solve this issue, we decided to switch the shell scripts to Java programs. Since Java has built in network capabilities that make it easier for the synchronization of transmissions to occur. This required a re-write of all the shell scripts to Java programs. This caused a slight delay as the scripts were converted to Java programs that were introduced earlier. This process took about a week to complete.

**Remaining Implementation Issues for the Client**

Currently the client side of the program is unable to change a password. This causes a new user to be generated each time we attempt to change the password. This causes a security issue to where the new user can access the older user's files and messages. A password manager or module will need to be created for password and username verification. On registration, if a username exist, the system should reject the request.

The other issue is the static definitions of the invocation scripts in the client. In order to make a nicer interface to the ACL2 modules, a dynamic way of accessing users will need to be implemented.

**Defect Prevention**

In order to have a successful implementation, we needed to take several step to eliminate defects. The first step of defect prevention was the use of the modular structure outlined in this document. This allowed each module to be developed independently from each other. This allowed only one program to be written to execute each action as needed. This also allowed us to keep IO and logic files separate. By doing this, generating theorems and test for the logic was less complicated since we enforced the requirements of having no dependencies in the ACL2 logic portions of the modules.

The next step in defect prevention was the use of ACL2 theorems to verify that the output of our logic functions, in fact, were correct. The theorem structure outlined in the *Testing* section allowed us to provide a logically sound proof that the functions we defined generated the guaranteed output. This guaranteed output along with strict XML DTD’s provided assurance that no stray data is going to be sent to or from the server.

The final step we took was careful planning and design of the modules. Since we designed this program modularly, we knew that once a module is completed, it would continue to work regardless of the progress made on other modules. This careful design and constant planning for of the program allowed all developers to be on the same page as the development process began. Constant updates to the design were inevitable but rarely deviated from the initial modular concept of the application.