June, 98

JavaCADD: A Java-based Server and GUI for Providing Distributed ECAD Services

Daniel H. Linder , Robert B. Reese(reese@erc.msstate.edu), Jon Robinson, Sam Russ

Abstract—**JavaCADD is a Java-based server and client used for distributed ECAD services. The server/client combination allows user access to batch-oriented ECAD services (synthesis, place/route, etc) without having to provide full login-access for the users. The JavaCADD GUI uses templates stored at the server for GUI definition, so new services can be added without modifying the Java source code of the GUI. Java Remote Method Invocation (RMI) is used for client/server communication. Both client and server are Java applications, and share the operating system independence unique to Java. JavaCADD has been used by undergraduate classes at MSU to access ECAD applications and computing resources located at the MSU/NSF Engineering Research Center. JavaCADD allows access to these resources without having to grant full login priviledges to this large, transient user population.**

Index terms—**Java, Distributed, ECAD**

I. Introduction

Distributed computing has been a moving target over the years. It has evolved in meaning from simple remote disk access to distributed task execution over heterogeneous computing platforms and operating systems. The introduction of the WWW browser and the Java programming language has accelerated the evolution of distributed computing significantly. The Microsystems Prototyping Laboratory (MPL) associated with the Mississippi State University/Nation Science Foundation Engineering Research Center (MSU/NSF ERC) has developed a Java-based server and client used for distributed ECAD services. This paper will discuss both the client and server architectures.

II. Distributed ECAD Services

Our goal in this project was to make ECAD services available outside users without having to grant the users full login privileges. The basic reason for doing so is to support user populations whose members either change rapidly are whose members are not known ahead of time. One example of such a user group are students in VLSI or digital design classes which change on a semester basis.

There are two approaches to providing distributed-services:

A. *Browser based Services*

In this approach, a browser is used to access the service interface and a WWW-server is used to execute the service. Typically, a user will fill out a form, and the form parameters will be sent to the WWW server where a CGI script can be used to execute the service. Results can be returned via email, or a custom HTML page can be built to display the results. The service interface can be built using a HTML, or a Java applet can be used for a more sophisticated capability. One problem with both of these approaches is that the browser is limited in its ability to access the local file system for security reasons. User files either must either stored on the file system where the server resides or the user must manually transfer these files between the client file system and the server file system. Email services can be setup to make this approach somewhat transparent, but it is difficult to achieve a seamless capability.

Important advantages of browser-based services include ease of implementation, and portability of the client-side interface.

B. *Custom Client/Server Architectures*

We tried the browser-based approaches discussed above ([1]) in our first generation ECAD services implementation. However, the file passing limitations caused us to look at other approaches for our second generation ECAD services implementation; specifically, a custom client/server architecture. Custom client/server architectures offer the utmost in flexibility; however, the number of different client/server protocols and data packaging options to choose from are many. A recent addition to client/server architectures has been Java Remote Method Invocation (RMI). This protocol offers portability of both the server and client because of Java's

The authors are faculty members with the Electrical & Computer Engineering Department, Mississippi State University. They are also researchers within the MSU/NSF Engineering Research Center. This report is distributed as MSSU-COE-ERC-98-07.

operating system independence, and provides a very sophisticated data packing mechanism that allows complex objects to be passed between client/server with minimum programmer effort. These two features of Java RMI operating systems independence and powerful client/server object passing caused us to choose this approach for implementation of our ECAD services.

III. JavaCADD Server

Two Java applications make up the JavaCADD architecture – the client and the server. The function of the server is to accept task requests from client, perform some action to resolve the task request, and then return results to the client. Usually, the task request is a request to spawn an external tool under the control of a user specified shell script but can also include executing a dynamically loaded Java class method or other internal server functions. While the Java portion of the server is OS-independent, the method for spawning external tasks is OS-dependent, and a configuration file is used to specify how this is accomplished for a particular operating system.

A. *Internal Server Architecture*

Part of the task request is to pass a set of parameters to the server. These parameters are passed within a Java hashtable as key/value pairs; the communication mechanism is Java Remote Method Invocation (RMI). For security purposes, the server checks the key/value pairs to see if they match the types defined within a property file associated with that task request.. If a key/value pair is unknown or does not match the type defined in the property file, then the task request is rejected. Each distinct task or service to be performed by the server requires a property file. By convention, the property files are stored under *task_server/config/forms* ;one of the parameters passed within the hashtable is a pathname that tells the server which property file to use.

The property file is divided into property file groups, which each group defining attribute values for the property. A property group has the format:

```
propertyName.attribute1Name =  value
propertyName.attribute2Name =  value
etc......
```

The attributes specify the type of the property, whether the property is required or not, the default value of the property, etc. The attributes which each property MUST have and which are checked by the server are:

.required : the value is either *true* or *false*. This attribute defines whether the property is required or not. The task will not be executed if the property is required, but not found by the server in the hashtable passed by the client.
.userName : the value can be any string; this string is used in any error messages returned by the server concerning this property.
.type : the value specifies the type of the property; the value must be one of the types supported by the server. An example of a type is *IdentifierField* , which only allows alphanumeric characters and the '_' character. A complete description of allowed types is contained within the server software distribution documentation [2]. The purpose of type checking is to provide a security mechanism on the data that is passed as part of task requests.

Any other attributes can be added by the user and are simply ignored by the server. There is a method by which the client can request the contents of a property file from the server. The server returns the property file as a hashtable of hashtables. The keys used in the first-level hashtable are the propertyNames, the keys used in the 2nd level hashtable are the attribute names. This feature allows the client to 'discover' new tool features when the client is started. This is how the JavaCADD client queries the server for the current tool list stored on the server.

The server expects the hashtable passed to it by the client to contain a key whose name is *propertyName*, and whose value is the string value to be passed to the tool that will execute the task. All values of keys in the hashtable passed by the client to the server should be Java *bytestrings.*

B. *Properties Recognized by the Server*

There are some properties that are recognized by the server. These are:

action : The value of this property tells the server what type of action is to be done on behalf of the client. Two documented values are *ExternalAction* and *echoProperties*. *ExternalAction* is used when you want the server to execute an external command such as a script on behalf on the client. *echoProperties* is used to return the contents of a specified property file to the client. If *ExternalAction* is used, two additional lines must be in the property file; these lines specify the pathname of the external command and a string to be used in error messages concerning the execution of this command. The format of these lines are shown below:

```
   externalCmd = pathname_to_script
   cmdUserName = anystring
```

properties : The value of this property is the pathname of the properties file to be used when checking the key/value pairs of the hashtable from the client. The pathname is relative to the config directory of the task server ; an example value would be *forms/clientTest/clientTest1.props*.

password : This is an optional property. It can be used to password protect a particular service (at this time one password is assigned to the entire service, it is not possible to give passwords for different users of the service). The file *task_server/config/forms/passwords* contains key/value pairs that associate passwords with property files. The value of the password property passed by the client would need to match the password key in the password file, the property file name would need to match as well. Password protection of a service is optional. All other properties in a property file are user defined.

C. Server Execution of an External Command

As mentioned previously, if the value of the *action* property is *ExternalAction*, then the *externalCmd* line found in the property file will be executed as the 'task' by the server. It should be noted that the value of *externalCmd* is always read from the property file; it is not read from the hashtable passed by the client. The server creates a temporary directory called
task_server/task_server_tmp/task_dir_NNNN

where NNNN is the directory id for use by the task. The command line used by the TaskServer to spawn the task is:

```
taskServer.execCmd  TmpDir ExternalAction
```

where the *taskServer.execCmd* is specifed in the server configuration file. For Solaris OS, the default configuration file causes the command to be:

```
perl execProcess.pl  TmpDir ExternalAction
```

The 'execProcess.pl' script first calls the 'setpgrp()' function so that all processes spawned by this script have the same group process ID, then changes to 'TmpDir' before executing the *ExternalAction* command. The 'execProcess.pl' script also prints out its process ID to STDERR so that the TaskServer reads this value and use it later for killing the job if necessary. The 'setpgrp()' function is VERY important since it allows all tasks spawned by this script to belong to the same process group, and thus all tasks can be killed by killing the process group. You can change the property specifed by *taskServer.execCmd* to be whatever you wish, but the ProcessID must be written to STDERR for the TaskServer to read. Another action which should be performed by the 'taskServer.execCmd' executable is to set up a common environment for all tasks that are to be executed by this TaskServer.

The task server will pass the key/value pairs from the client hashtable inside a file named 'request.props'; this file contains lines of the form:

```
key = value
```

The 'request.props' file will be in the *task_dir_NNNN* directory if the hashtable contains no properties of the type 'fileField'. If the hashtable contains properties of the type 'fileField', then the 'requests.props' file will be in a directory called *request*. A file will be created in the *request/* directory for each property of type 'fileField'; the filename will match the property name and the file contents will be the value passed in the hashtable for that property.
To pass information back to the client after the task has been completed, the task script should create a directory named "/response". Any files in this directory with extension of ".props" will be assumed to contain key/value pairs which will be placed in the hashtable which is returned by the server to the client. Any files in the "/response" directory without a ".props" extension will be placed in the returned hashtable as key/value pairs where the key is the filename, and the value is the file contents.

The temporary directory space is reclaimed by the server upon task completion unless the task creates a file named 'state' in the *task_dir_NNNN/* directory (contents of the 'state' file are unimportant). The task server configuration file contains a variable named *dir.maxDirCnt* which controls the maximum number of temporary directories; if this count is exceeded the server begins reclaiming temporary directories starting with the oldest directory. The purpose of the 'state' file is to be able to examine temporary directory space usage during script debugging; it can also be used to pass information between different tasks.

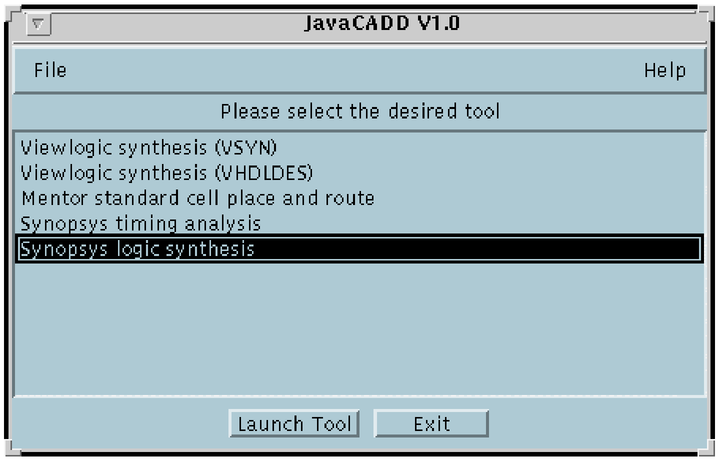
The object returned by the server to the client task is a Java object of class *FormResult*. The instance variables of this object are:

- **public int dirId** : This contains the NNNN field of the temporary directory *task_dir_NNNN* used for executing the task.
- **public int errorCode** : If nonzero, then indicates that an error was encountered during execution of the task.
- **public byte[] response** : This will contain the error message if an error occurred. The error message is formatted in HTML.
- **public Hashtable responseHash** : This contains the key/value pair as returned by the task script execution as discussed above.

IV. The JavaCADD Client

The JavaCADD client is a Java GUI for remote task invocation built upon the capabilities of the JavaCADD server. At startup, the JavaCADD GUI contacts the server for the contents of the property file "config/forms/release/metatool.props". The contents of this property file tells the client which tool services are available on the server. This information is then displayed in a local window as shown in Figure 1.

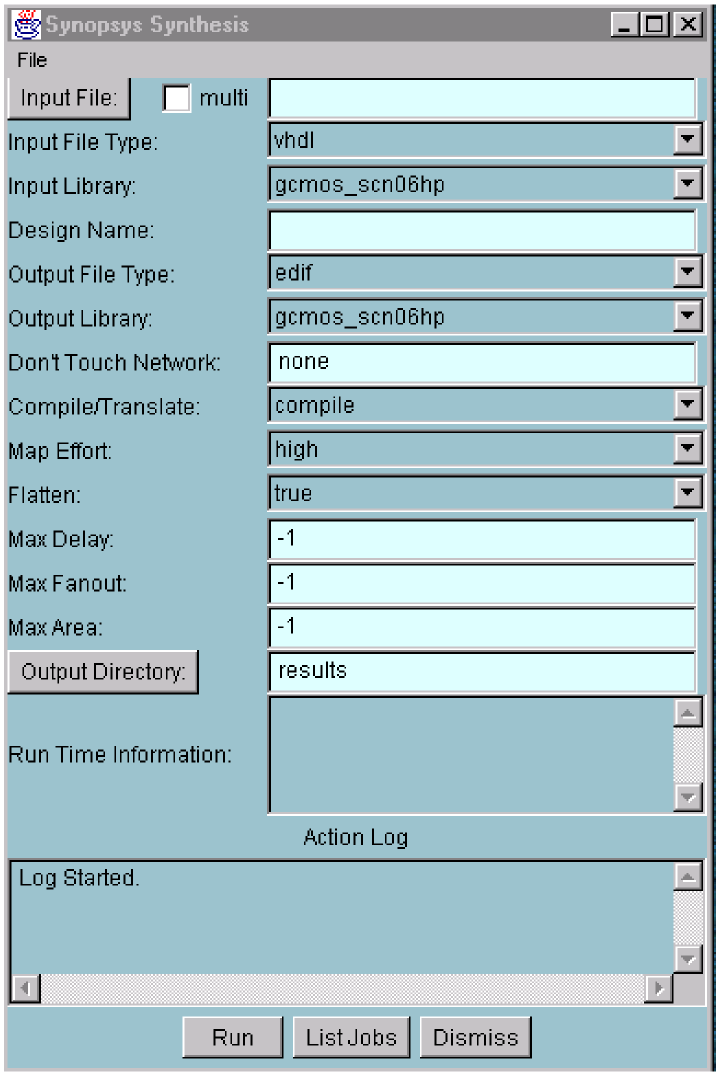
Left clicking on a service will select the service; then left clicking on **Launch Tool** will cause the JavaCADD application to query the server for the GUI-specifics for that tool. Once the tool personalization information has been returned, the JavaCADD application window customized for that tool will appear.



Viewlogic synthesis (VSYN)
Viewlogic synthesis (VHDLDES)
Mentor standard cell place and route
Synopsys timing analysis
Synopsys logic synthesis

Figure 1: JavaCADD Client Tool List

The JavaCADD application window for the "Synopsys Synthesis" service appears below:



Input File:
Input File Type: vhdl
Input Library: gcmos_scn06hp
Design Name:
Output File Type: edif
Output Library: gcmos_scn06hp
Don't Touch Network: none
Compile/Translate: compile
Map Effort: high
Flatten: true
Max Delay: -1
Max Fanout: -1
Max Area: -1
Output Directory: results
Run Time Information:

Log Started.

Figure 2: Synopsys Task Window

During execution of the server, a tool application window will be opened on the client machine that allows the user to monitor the progress of the task. The user also has the ability to kill the task at any time.

There is no limit to the number of active tool application windows; you can also have multiple JavaCADD launch windows present, each connected to a different server if desired

The tool property file for a JavaCADD GUI tool service not only describes what parameters will be passed to the tool script during execution, but also determines how the JavaCADD GUI for the tool service will look.

Property Attributes used by the JavaCADD GUI are:

- **.level** : this is an integer value which determines the order in which fields appear on the GUI. The higher the number, the higher 'up' on the form the field will appear. The 'level' values do not have to be in sequential order.
- **.choices**: This will cause a list menu to be presented to the user, the contents of list menu will be the white-space seperated strings used for choices attribute value.
- **.choicedelimiters**: An optional attribute, this specifies the delimiters to be used for the items in the choices attribute. If not specified, the delimiters default to whitespace. Leading/trailing whitespace is always trimmed during parsing of the string in the *choices* attribute. Note that if the *choicedelimiters* attribute is specified (non-whitespace is being used for delimiters), you will need to specify the *type* attribute of this property as either MultiIdentifierField or StringField.
- **.default**: Default value to be presented to the user as a value for this property.
- **.multi**: This can be used in conjunction with a 'FileField' type property. It places a 'multi' pushbutton on the GUI next to type-in field for this property. If the button is pushed, then type-in field of this property on the GUI specifies a file which contains a LIST of files, these files will be sent to the server as property key names "propertyName0, propertyName1,... ". See the property files "config/release_msu/Syn_synth.props" for an example use of this attribute.

The JavaCADD GUI requires two properties for supporting redirection of the tool output to the tool application window. These properties and associated attributes are given below, and should be in every tool property file used by the JavaCADD GUI:

```
redirectHost.required = false
redirectHost.userName = redirectHost
redirectHost.type = StringField
redirectHost.default = none

redirectPort.required = false
redirectPort.userName = redirectPort
redirectPort.type = IntField
redirectPort.default = 0
```

The JavaCADD GUI uses the general server/client parameter passing methods as discussed previously. However, there are some specifics that need to be mentioned:

- Any properties of type 'Filefield' are passed as compressed Java GZIP streams. As discussed previously, 'FileField' properties are created as files in the 'request/' directory where the the file name is the propertyname, and the contents is the byte stream. It is the responsibility of the task script to uncompress these streams.
- The JavaCADD GUI looks for a special key called 'results.zip' in the returned hashtable from the server. The value of this key is treated as a ZIP stream and unzipped into the results directory specified by the user. Any other keys found in the returned hashtable are echoed by the JavaCADD GUI to the 'log' window.

V. Use of JavaCADD

JavaCADD has become the standard method for student access to synthesis tools used in our upper level Digital System design class. The synthesis tools run on computing resources within the MSU/NSF Engineering Research Center and full login privileges for this transient user population is neither desired nor needed. Student response to the JavaCADD interface has been very positive to date. One result in making JavaCADD our standard interface has been a decoupling of tool implementation and service interface. Tutorial notes now refer to the JavaCADD GUI interface and not to the particular tool implementing the service. This means that students are more protected from tool changes via vendor upgrades. It also makes CAD tool system administration easier since vendor upgrades only effect the server side scripts.

VI. References

1. R. Reese and D. Linder. “ A Generator-Based Standard Cell Library using Mentor ICGEN”, Mentor User Group Meeting, October 21-23rd, 1996.
2. R. Reese, http://www.erc.msstate.edu/mpl/vela, JavaCADD Server and Client Distribution page.