JSPICE: A Component-based Distributed Java Front-End For SPICE

Dan Souder, Morgan Herrington, Rajat P. Garg and Dennis DeRyke
{Dan.Souder, Morgan.Herrington, Rajat.Garg, Dennis.DeRyke}@Sun.COM
Market Development Engineering
Sun Microsystems, Inc.
901, San Antonio Road, Palo Alto, CA 94303
www.sun.com

Abstract

The design and implementation of a component-based distributed Java front-end for the Berkeley SPICE circuit-simulation program is described. The front-end is architected as a client-server application implemented *via* the Remote Method Invocation (RMI) API. A graphical user-interface is implemented on the client-side which has multiple components for command and control, graphing, text-editing and communication with the server. JavaBeans API is used as the underlying component model. On the server side, the native SPICE code (in C language) is converted into a dynamic library and is interfaced with the Java Server code using Java Native Interface (JNI) API. Implementation issues related to modifications in the SPICE code enabling multiple invocations as a shared object, efficient input/output of simulation data from the Java code, waveform compression and extraction for display on client side, real-time transfer of SPICE generated runtime messages from the server to client *etc.* are discussed. Benchmark results comparing performance of file input using various Java I/O classes are presented.

1 Introduction

In a very short time Java ([1, 2]) has proven itself to be more than a mechanism for building fancy web pages. The multi-platform acceptance and network-oriented design has led to emergence of new computing paradigms. Capable of running on diverse devices such as retail point-of-sale scanners and consumer electronics products, Java promises to impact the way we all do business. Further, the development of a component-based architecture, namely, Java Beans API (see [3]) additionally promises to decrease development costs by accelerating software design based on reusable modules. In one sense, Java Beans is a convention for software developers to follow in the naming and packaging of their functions. The incentive to adhere to this convention is that the resulting software components can be used as building blocks through visual application builder tools such as JavaStudio (see [4]) and the Bean Developers' Kit [5].

However, Java still faces many challenges. Performance is a purported short-coming and there is ample skepticism in the technical computing community regarding Java's applicability to large, complex applications, especially integration with legacy systems. Consequently, at present there are few Java based scientific computing applications. A noteworthy Java application in Electronic Design Automation (EDA) area is the MAJIC VLSI layout editing tool described in [6], designed to support distributed VLSI design and collaboration.

The present work is motivated by the desire to explore the applicability of Java's distributed computing framework to an existing engineering code. We consider the publically available Berkeley SPICE (Simulation Program with Integrated Circuit Emphasis) program (see [7]), one of the most widely used and perhaps the *de facto* standard circuit-simulation tool. Arising from research at the Universtity of California, Berkeley, in the early Seventies, SPICE represents a large legacy system that grew in popularity with the advent of the personal computer. Today SPICE-based applications are available from several vendors, and on multiple platforms.

In addition to the above-mentioned objective of implementing a functional client-server simulator as a "proof of concept" of Java technology, we were also guided by some other high-level goals: minimizing the reengineering and rewriting of the SPICE code, leveraging of existing Java components (*i.e.* JavaBeans) on the client-side, effectively exploiting the processing power of a "thin-client" and documenting the development and performance issues arising in an investigation, such as the present one. The paper is organized as follows: a brief overview of SPICE code is given in section 2. This is followed by a description of JSPICE architecture in section 3 and a detailed discussion of the implementation issues in section 4. Finally, some conclusions are presented in section 5.

2 Overview of SPICE

SPICE solves the nodal equations in a circuit describing Kirchhoff's voltage and current laws. It is capable of performing a variety of analyses, such as DC analysis, small-signal AC analysis, transient analysis, Pole-zero analysis, small-signal distortion analysis, sensitivity analysis, and noise analysis. The original SPICE program was written in FORTRAN but in the mid-Eighties it was rewritten in the C language. In the present work, the C based version 3f3 is used (see [8]).

The SPICE input files are ASCII text (also called "netlist" files), which describe the circuit to be analyzed. Most components in the circuit can be described with single-line entries in the input file, which include the type of component, its parameter value, and the two circuit nodes spanned by the component. Depending on the voltage-current characteristics of the circuit components, SPICE uses either a direct or an iterative solution method. Since most practical circuits consist of devices with nonlinear voltage-current characteristics, the most common solution procedure is based on an iterative process, which converges when the branch currents and node voltages fall within a certain tolerance range. Parameters such as iteration limits and error tolerance can be specified in the input file, as well as flags that control the output format.

SPICE may be run in batch mode or interactively. When run in batch mode, it generates a large output file (called "raw-file") which contains the time (or frequency) series of the nodal variables. The command-line driven NUTMEG post-processing program may be used to plot and visualize the results.

3 JSPICE Architecture

The architecture of JSPICE consists of a single process on the client machine and a single process on the server machine, as illustrated in figure 1. The client accepts user input, displays simulator output and provides post-processing support including data buffering and graphing. The components used in the client software include a text editor which enables the user to modify the SPICE input files, and a plotter component which displays the simulation output as line plots. The modular properties of the component paradigm make it possible to substitute other components in place of these components. For example, a spreadsheet component could be used in place of the plotter Bean for performing post-processing manipulation on the resulting data. Likewise, if a graphical data file manipulator became available which allowed the user to build circuits by dragging and dropping electronic component icons, this could be implemented into JSPICE in place of the text editor.

While the JSPICE client software can take advantage of a heavily-configured workstation's resources, it was designed to be able to run effectively on a light-weight desktop. This approach potentially allows a design group to focus more of its limited hardware budget towards a state-of-the-art server machine, which can be shared by the team members for running their SPICE simulations.

The JSPICE server includes the actual simulation engine. To minimize network traffic and to put an upper bound on the requirements of the client machine, some amount of data filtering has been built into the server. Thus, only those portions of the resulting solution that are of interest to the user are delivered to the client. The server is designed to allow easy addition of more extensive filtering capabilities.

4 Implementation and Performance Issues

A description of the implementation and performance issues for the architecture outlined in the previous section will be presented in this section.

An important implementation issue was the selection of the interprocess communication between the client and the server. Possible choices included sockets, Common Object Request Broker Architecture (CORBA) (see [9]), Remote Procedure Call (RPC), shared memory and Java RMI ([10, 11]). A comparative evaluation of the usefullness of the various approaches is a major project it its own right and was not pursued here. Instead, as an experiment and a desire to use 100% Pure Java API's to the extent possible, we decided to use RMI as the framework for implementing the distributed architecture.

Somewhat like RPC, RMI provides the illusion of a simple function call which actually crosses process, and typically machine, boundaries. It is synchronous by nature; the client blocks until the server completes the request and returns the result. It also imposes a direction to the protocol; without extra threads and a second RMI connection, requests can only come from the client and responses can only come from the server. Contrast this with shared memory which can quite easily allow shared control of an IPC protocol.

Our experience with RMI is that it is quite easy to use and provides a distributed programming model almost unchanged from a non-distributed mode; this was also observed recently by Atwood *et. al.* ([12]) who describe a RMI based client-server implementation of a Computational Fluid Dynamics application.

Implementation issues pertaining to the design of the server-side and the client-side code are discussed next.

4.1 Server Side Implementation

As described in the previous section, the server side of JSPICE is responsible for carrying out the simulation and sending appropriate waveform information back to the client. Issues that arose in the server design are described below.

1. *Modification of SPICE for invocation as a shared object* – The execution of SPICE could have been accomplished by simply remotely executing the native SPICE executable from a Java RMI based service to allow client access. But to investigate JNI and the effect of executing native applications as part of a Java Virtual Machine (JVM) process and have more direct control of the simulation, a shared library approach was chosen.

   A large part of the data in a SPICE simulation is moved by declaring a number of global variables, evidence of its Fortran origin. In order to minimize the effort required to fix the data initialization issues, the SPICE shared library is dynamically loaded and closed[1] at the end of the simulation run. When the library is loaded for the next call the globals are reinitialized correctly.

   Java interfaces to SPICE are provided by using JNI to define native functions which are available to the Java VM as methods. These methods are made available by dynamically loading a native shared library and defining the native functions as methods within a Java class. This provided a very simple and effective way to leverage the existing code and include the simulator as part of the Java VM.

[1] Accomplished using Solaris *dlopen()* and *dlclose()* system calls.

2. *Simulation data input/output and waveform extraction* – The SPICE shared object (discussed above) is executed in batch mode and at the end of the run it generates a binary or ASCII "rawfile". The JSPICE server then reads in this file and stores the variables for transmission back to client and "on-Server data-processing". Depending on the size of the input circuit and the type of analysis performed, the size of "raw-file" could be rather large (many ten's to hundred's of megabytes). Efficient reading of this file is therefore crucial to Server performance. Further, compression and filtering of the data is needed to decrease the memory overhead on the server and the network traffic between the client and server. Presently, by way of waveform compression, we truncate the 64-bit SPICE output values to 32-bit data. To reduce network overhead, a simple piecewise signal-clipping is done on the waveforms before they are sent to client for display. Simple utilities to extract waveforms in a variety of ways from the simulation output are also implemented on the server side.

   The default setting in SPICE is to generate the "raw-file" in binary format; there is an option to write it in ASCII format also. Additionally, since C generated binary files are non-portable between little-endian and big-endian machines, the SPICE distribution includes a utility (called `sconvert`) to convert binary data to ASCII and *vice-versa*. In this regard, we note that Java provides a happy-marriage between file portability and storage-efficiency by requiring that the storage format of primitive data-types is to be always in big-endian format regardless of the underlying microprocessor architecture. We have added a "raw-file" output object on the server to convert SPICE binary "raw-files" into Java binary files; this provides the user with the option to generate a portable simulation database.

   The `java.io` package in JDK 1.1 API (see [13]) provides a variety of classes for input/output from files. As shown in [14], efficient file I/O in java is tricky and is often the bottleneck to high performance. We performed benchmarks comparing different `java.io` classes for both ASCII and binary SPICE "raw-files". The file sizes generated using both ASCII and binary options for some sample SPICE circuits[2] are listed in table 1 and performance numbers are presented in tables 2–3. The Ultra1 170E used in this study contained a 167MHz UltraSPARC module with 0.5MB external cache ([15]) and running Solaris 2.5.1 operating system. It is clear from the table that the use of buffered I/O classes (a trivial modification to the code) leads to an order of magnitude performance improvements. Also, not surprisingly, reading binary data is three to four times faster than reading ASCII data. This coupled with the fact that Java binary files are platform-independent provides an incentive for Java application developers to use it's binary file generation capability.

4.2 Client Side Implementation

A snapshot of the JSPICE client-side GUI is shown in figure 2. The GUI consists of user input portion for selecting the SPICE input circuit file, simulation messages display window and various control buttons. Some design considerations in the client-side implementation are now described.

1. *Waveform Graphing* – One of our goals was to explore leveraging Java's component technology by reusing off-the-shelf JavaBeans ([3]). To enable this, we segregated the graphing functionality into multiple parts: (i) a SPICE dependent "control" portion which is knowledgeable about SPICE data formats, (ii) a generic graphing component driven by the graphing control and (iii) a spreadsheet component to carry out simple computations (such as addition/subtraction of waveforms). This way we could incorporate an off-the-shelf spreadsheet bean for the purposes of performing client-side processing of simulation output sent by the server (*eg.* the Integer spreadsheet bean from Athena design, Inc. [16]). We are currently

[2] Note, these are fairly small circuits and have been used here, primarily, for illustration purposes.

exploring the possibility of using tool-generated linkage to these component beans. Figure 3 shows a snapshot of the graph control component. Both the plotting (see figure 4) and spreadsheet beans gets invoked when the user selects the "Plot" button.

2. *Simulation Error Messages and Run Statistics* – Initially, we treated the SPICE run as a batch process and displayed all error messages and run-statistics upon completion of the run. This was soon found to be unacceptable and instead, it was felt that it is necessary to provide for feedback in real time (particularly for errors/warnings so users can appropriately control the simulation).

   These messages don't quite fit the simple model previously described where the client requests information and blocks until it is returned by the server. Since the messages are generated during the simulation (and are not synchronized with the existing RMI calls), a plausible solution is to handle them with a separate thread of control in the client.

   The most straight-forward implementation in our thinking is to create a socket on the server side and bind it to the `stdout` and `stderr` descriptors of the SPICE code. Then in the client, one could create a simple thread which would read from the pipe and copy the result to a text area.

   We also considered the possibility of opening another RMI channel rather than a socket, but this turned out to be more complicated with no overwhelming advantage. Based on these considerations, the former approach was chosen in the present implementation.

3. *Text-Editing Component* – In a practical design scenario, the user iteratively modifies the SPICE input netlist file and performs repeated simulations until the output results match the expected circuit design parameters. To facilitate this process, we integrated a simple text-editing component (figure 5) in the JSPICE GUI which provides the functionality of editing and saving the SPICE input netlist files. Since, this component has limited editing capabilities, we are currently exploring the possibility of integrating a more sophisticated "off-the-shelf" text-editing JavaBean on the client.

4. *Interrupting the Simulation* – This capability hasn't been implemented yet, but could have a significant impact on the implementation. Since Java doesn't support the notion of "signals", we will possibly have to set up separate threads in the client and the server which can communicate out-of-band with respect to the normal flow of simulation data over RMI. This is part of an ongoing effort and the results will be reported in a later manuscript.

5. *Printing Capability* – A realistic GUI based application needs to provide the ability to generate hardcopy output of the information (waveform plots) generated in the simulation. In the present implementation, the JDK1.1 API based printing functionality was incorporated in a straightforward fashion in the plotting bean (see figure 4); this provides the ability to print (save) the output to a postscript printer (file). The upcoming JDK1.2 API release will provide significantly enhanced printing support for applications.

5 Conclusions

One of the main objectives of this work was to devise a methodology that shows how EDA vendors can exploit a client-server architecture without significantly reimplementing their production code. The alternative is simply too expensive to consider. Our results indicate that this appears to be possible, at least, in the context of SPICE based circuit-simulation tools.

The bulk of the SPICE simulation engine has been left untouched in this study and mostly the simulation driver was modified in converting SPICE into a shared library. The changes needed were primarily internalizing global variables and stripping NUTMEG front-end (and other unnecessary portions) from SPICE code. The difficulty arising due to the possibility of inconsistent internal state of the simulator between multiple invocations (on the same or different circuit files) was circumvented by using Solaris dynamic library opening and closing API's on the SPICE shared object *via* the JNI interface, at every invocation. This proved to be a simple workaround; the alternative involved going through the entire SPICE code and modifying the data-structures to maintain consistency between calls. We think that this simple approach is applicable to other similar codes as well. Additionally, the overhead involved in this process is quite small.

Our results also suggest that Java can be useful in portable and storage-efficient file I/O as compared to I/O based on using ASCII format. This is important as simulation codes (in C/F77/C++) typically generate large binary format, non-portable data-files. These databases (due to their storage format) necessarily become tied to the server on which they are generated and additional programs are usually needed to extract ASCII format information out of them. The portability of Java binary representation can be useful in a heterogeneous design environment as designers can upload/download simulation data between machines without worrying about internal storage representation (*i.e.* little-endian *vs.* big-endian issues).

Another goal of this project was to show how vendors can minimize their development costs by taking advantage of the component technology supported by Java. JSPICE client-side demonstrates that Java beans with different functionalities (*eg.* graphing) can be integrated in the client in a relatively straight-forward manner. The increasing availability of "off-the-shelf" components can be a critical factor in driving the development costs down.

Finally, the present architecture is also designed to make effective use of the client machine's processing capability. While we wanted to take advantage of the superior resources of a heavily configured server as the simulation engine, we also wanted to make effective use of the client to buffer, reprocess and display the resultant data. Note the distinction between this client-server design compared with simply remotely running the simulation and displaying the results back via X-windows protocol. Our model makes best use of the server for the CPU/memory intensive simulation and offloads the rest to the client. On the other hand, the "X" based model burdens the server with all computations and loads the network with every pan/zoom of the display.

References

[1] Gosling, J., Joy, W. & Steele, G.; *Java Programming Language.* Addison-Wesley, Reading, Massachusetts, (1996).

[2] Cornell, G. & Horstmann, C.S.; *Core Java.* The SunSoft Press, Mountain Views, California, (1997).

[3] *JavaBeans API definitions;* `www.javasoft.com/beans/docs`.

[4] *JavaStudio*; `www.sun.com/studio`.

[5] *Beans Development Kit 1.0*; `www.javasoft.com/beans/software`.

[6] Cherry, Y.; *MAJIC: A distributed VLSI Layout tool.* Presented at the ACM 1997 Conference on Java for Science and Engineering, Las Vegas, Nevada, June 1997; www.npac.syr.edu/projects/javaforcse.

[7] Nagel, L.W.; *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Technical Report No. ERL-M520, University of California, Berkeley, 1975.

[8] *SPICE Version 3F3 User's Guide.* University of California, Berkeley, 1993.

[9] Orfali, R. and Harkey, D.; *Client/Server Programming with Java and CORBA.* John Wiley and Sons, 1997.

[10] Wollrath, A., Waldo, J. & Riggs, R.; *Java Remote Methods Invocation System.* Submitted to *IEEE Micro.* (1997).

[11] Sutherland, D.; *Distributing Java across your Enterprise.* In *Internet and Java Advisor*, vol. 1, no. 10, (1996).

[12] Atwood, C.A., Garg, R.P. & DeRyke, D.; *A Prototype Computational Fluid Dynamics Case Study in Java.* In *Concurrency: Practice and Experience*, vol. 9(11), (1997).

[13] *Java Platform 1.1.5 Core API Spiecification;* www.javasoft.com/products/jdk/1.1/docs/api/.

[14] *Java on Solaris 2.6: A Whitepaper.* www.sun.com/solaris/java/wp-java/.

[15] *The Ultra1 Architecture – Technical White Paper*, www.sun.com/desktop/products/Ultra1.

[16] *Integer: the Real Time Distributed Collaborative Spreadsheet Component*, Athena Design, Inc., www.athena.com.

Circuit-name	ASCII file		Binary file
	Size (MB)	No. of lines	Size (MB)
p2	0.3	16043	0.1
bst	19.0	827515	6.3
grey	14.5	1090567	8.7
jmnx	14.5	633957	4.9

Table 1: ASCII and binary "raw-file" sizes for benchmarking of file input time using various `java.io` classes.

`java.io` class	**p2** Time (sec.)	**bst** Time (sec.)	**grey** Time (sec.)	**jmnx** Time (sec.)
DataInputStream	18	1081	1517	835
DataInputStream(BufferedInputStream)	2	142	201	110
BufferedReader	2	143	194	108

Table 2: Comparison of *java.io* classes for input of ASCII "raw-files". Runs were done on an Ultra 1/170E running Solaris 2.5.1 using SunSoft JDK 1.1.4.

`java.io` class	**p2** Time (sec.)	**bst** Time (sec.)	**grey** Time (sec.)	**jmnx** Time (sec.)
DataInputStream	5	335	464	274
DataInputStream(BufferedInputStream)	0.5	34	46	26

Table 3: Comparison of *java.io* classes for input of binary "raw-files". Runs were done on an Ultra 1/170E running Solaris 2.5.1 using SunSoft JDK 1.1.4.

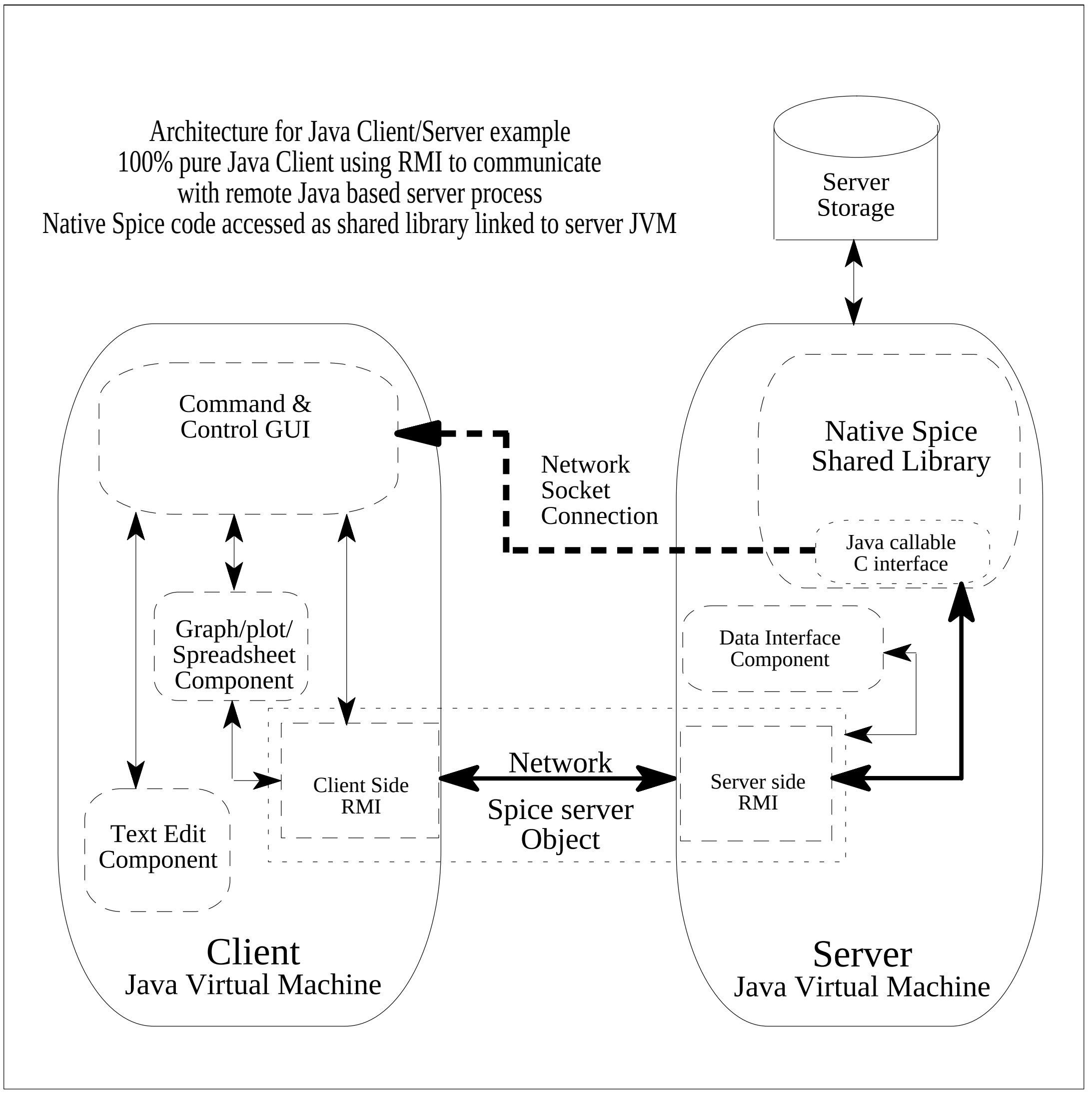

Figure 1: Architecture of JSPICE.

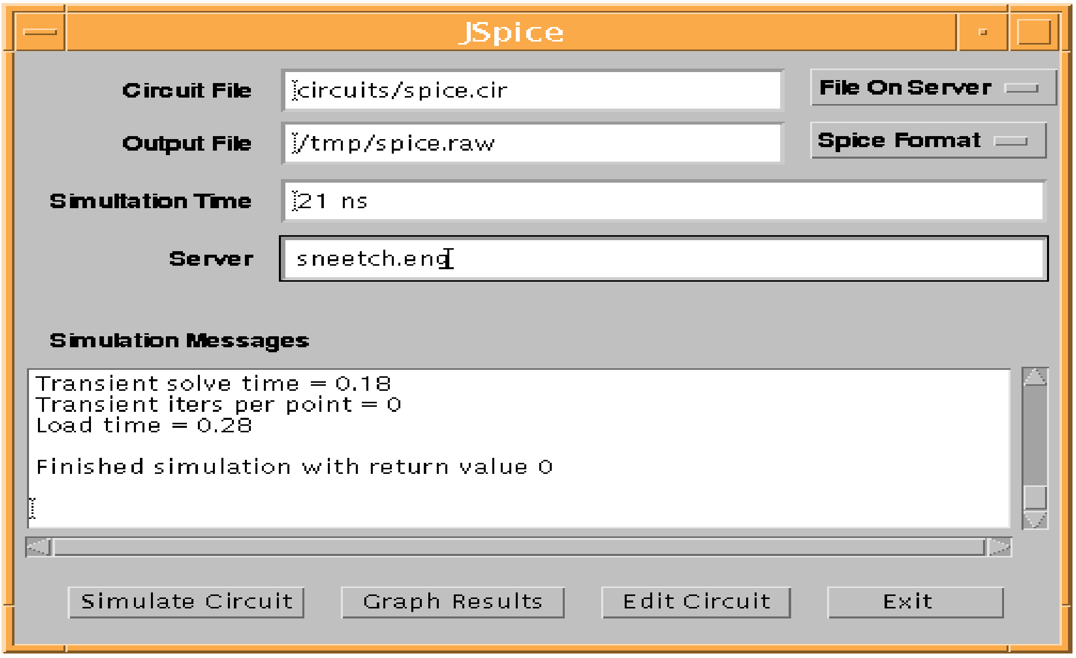

Figure 2: A snapshot of the JSPICE client-side command and control GUI.

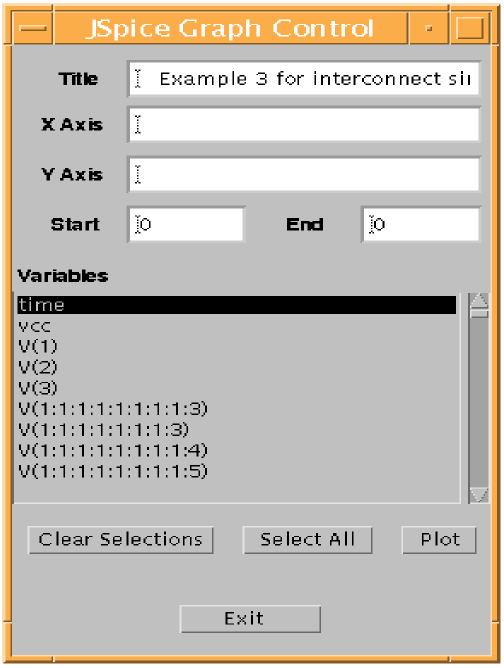

Figure 3: A snapshot of the Graph Control component of the JSPICE client.

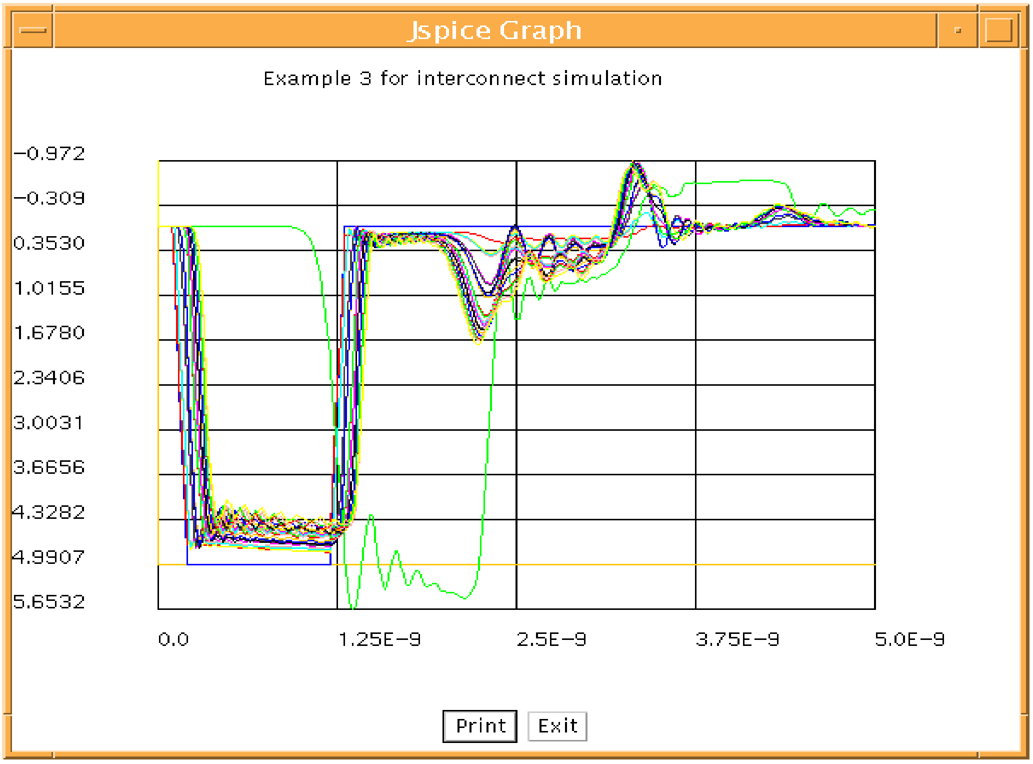

Figure 4: A Snapshot of JSPICE PlotSpice Javabean.

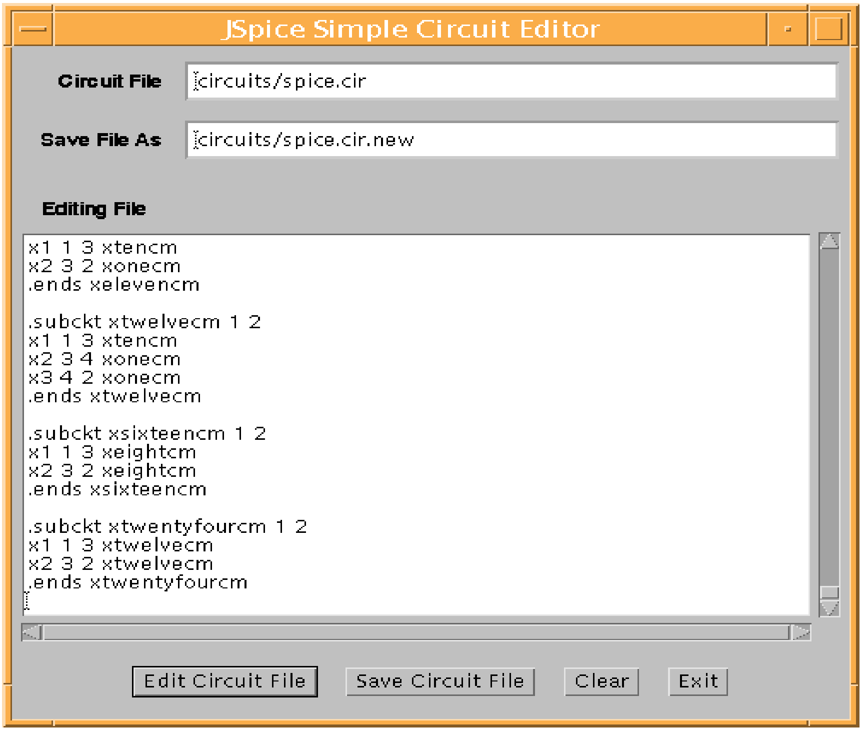

Figure 5: A snapshot of the Circuit Editor component of the JSPICE client.