Supporting Java Components in the SID Simulation System

Hasrul Ma'ruf*, Hidayat Febiansyah* and Jin Baek Kwon*

Abstract—Embedded products are becoming richer in features. Simulation tools facilitate low-costs and the efficient development of embedded systems. SID is an open source simulation software that includes a library of components for modeling hardware and software components. SID components were originally written using C/C++ and Tcl/Tk. Tcl/Tk has mainly been used for GUI simulation in the SID system. However, Tcl/Tk components are hampered by low performance, and GUI development using Tcl/Tk also has poor flexibility. Therefore, it would be desirable to use a more advanced programming language, such as Java, to provide simulations of cutting-edge products with rich graphics. Here, we describe the development of the Java Bridge Module as a middleware that will enable the use of Java Components in SID. We also extended the low-level SID API to Java. In addition, we have added classes that contain default implementations of the API. These classes are intended to ensure the compatibility and simplicity of SID components in Java.

Keywords—Embedded System, Simulation System, SID Simulator

1. INTRODUCTION

Embedded systems are growing richer in features. Both hardware and software aspects of embedded systems must keep pace with development before entering the market. Time-to-market remains the single most challenging issue that must be addressed by manufacturers of sophisticated electronic equipment to remain competitive. Simulation systems help developers to reduce time-to-market with rapid increases in system complexity and ever more stringent demands upon overall system quality.

Computer system research allows for the improvement of hardware design to attain the highest performance, operating system development, and application performance tuning to determine and implement the most effective code modifications [1].

Complete machine simulations can aid in understanding the behavior of modern computer systems. SimOS is a simulation environment that is capable of modeling complete computer systems, including a full operating system and all application programs that run on top of it [2].

※ This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2011-C1090-1131-0004)

Manuscript received June 28, 2011; accepted November 22, 2011.

Corresponding Author: Hasrul Ma'ruf

* Dept. of Computer Science, Sun Moon University, Cheonan, South Korea ({hasrulwho; havban; jin.b.kwon}@gmail.com)

However, SimOS is not open source, and at the time of this report, its development has been discontinued.

Based on a number of considerations, we have chosen SID as a framework for building computer system simulations [3]. The SID simulator consists of an engine that loads and connects simulated components of an embedded system using a configuration file, and runs simulation sessions [4]. SID is an open source and extensible simulation framework.

Virtual Development Environment for Embedded Software (VDEES) [5] is a virtual environment that provides a configuration tool, a code editor for writing simulated components, building tools for binary images, a debugger, and a system monitor for investigating the virtual target [6]. VDEES provides a full-featured Integrated Development Environment (IDE) for the SID framework employing Eclipse plug-ins [7].

SID provides a built-in system monitor, written in Tcl/Tk, to monitor running simulations. The system monitor lists the components in the active virtual platform, showing specified component attributes such as pins, registers, etc. As VDEES is based on the Eclipse framework, the system monitor should be formulated as an Eclipse plug-in, which must be written in Java. However, SID cannot directly support components written in Java without a Java bridge component.

Based on the possibility of integrating the system monitor into VDEES, we propose using Java as an alternative platform for developing components in SID. Performance improvements provide additional motivation, as SID components are currently written in C/C++ and Tcl/Tk, and our tests indicate that SID components written in Tcl/Tk run very slowly. Each touch input requires several seconds for evaluation and 1 second for display refresh [7]. Moreover, GUI development in Tcl/Tk is neither easy nor user-friendly as compared to more advanced languages, such as Java and .NET.

Previously, we briefly presented the main concept of Java Bridge along with a prototype system using Socket and JNI [8]. In this paper, we focus on the development of Java Bridge via the Java Native Interface (JNI). JNI was chosen because it does not have a duplicated cache, and involves no inter-process communication, thus offering better performance than any other Java–C++ bridging technique [9].

We also propose that all newly written Java components should follow a specific API, to ensure compatibility with SID. As Java supports object-oriented design, the API would also facilitate faster and easier development of SID components in Java.

2. RELATED WORK

There has been some previous research focusing on computer system simulations (e.g., SimOS). Red Hat's SID was chosen for our work because it is an active open source project, supported by a community and with readily available documentation. It offers extensibility, processor support, debugger support, IDE support, and good performance. VDEES was developed as an IDE for SID, to provide the ease of development, debugging, and the deployment of simulated systems. A GUI component was previously created in SID via Tcl/Tk. The use of Tcl/Tk is enabled by the existence of a Tcl/Tk bridge component in SID.

2.1 SID and VDEES

Red Hat's SID is a well-known framework for hardware virtualization based on components [3]. It offers the benefits of an active open source community and a configurable simulation environment. The open source community enables us to find and reuse existing hardware components.

SID is extensible due to its plug-in architecture. A SID simulation consists of independent components, and SID provides a simple but complete interface for creating and adding custom components to the existing library of built-in components. Therefore, we can either reuse the built-in components or create custom components to match specific requirements.

Prior to this study, SID components were written exclusively in C++ and Tcl/Tk. Most of the existing components were written in C++, and GUI components (such as LCDs and touchscreens) were written using Tcl/Tk [11]. The use of an additional language such as Tcl/Tk can be enabled by the existence of a bridge component, in this case a Tcl bridge component was used.

Virtual Development Environment for Embedded Software (VDEES) wraps SID in the Eclipse platform [12], thereby providing a GUI for managing component interactions and creating custom components. Using VDEES, developers can write software on top of new virtual components without waiting for implementation of the hardware associated with the physical components. This reduces the time, and minimizes both cost and risk. Furthermore, VDEES is open source, and can therefore be enhanced and modified as needed.

Currently, VDEES has the following four components: VDEES Binary Image Builder, VDEES Configuration Builder, VDEES Custom Component Wizard, and VDEES monitor. As an Eclipse plug-in, VDEES can extend CDT (C++ Development Tooling) [13] and JDT (Java Development Tooling) [14]. The custom component wizard extends CDT, as it is a C++ SID component builder.

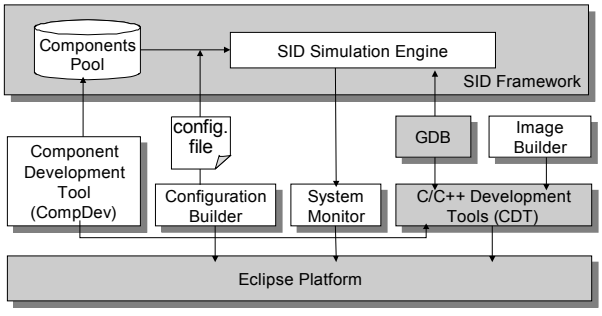


Fig. 1. VDEES Architecture

2.2 SID Tcl/Tk Component

The SID Tcl/Tk component is a non-native component in the SID environment, enabled by the existence of a Tcl/Tk bridge. The primary purpose of this component is to provide GUI building blocks to SID simulations. The bridge allows the transparent utilization of Tcl/Tk components by the SID user. The current VDEES version does not support the development of

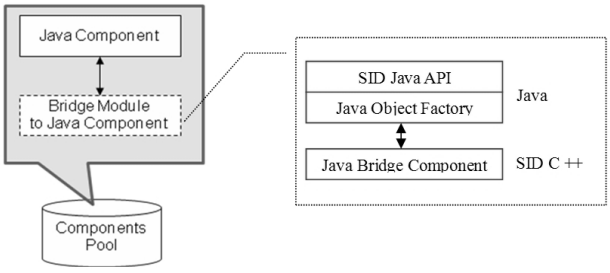


Fig. 2. Java Bridge Module Location in VDEES

Tcl/Tk components, and Tcl/Tk component development is accomplished using an external editor [7].

VDEES already extends CDT features in its custom component wizard. However, when we tried to create a GUI component with Tcl/Tk in VDEES, Eclipse did not really support Tcl/Tk. It would be very inefficient to create Tcl/Tk Development Tooling from scratch in Eclipse. Moreover, Tcl/Tk components are slow and perform poorly in simulations.

Rather than creating Tcl/Tk Development Tooling in Eclipse, and then a Tcl/Tk custom component wizard, we came up with the idea of using Java as an alternative platform for developing SID components. To accomplish this, a Java bridge component is needed.

3. JAVA BRIDGE MODULE

There are a number of software bridges that enable interoperability between different middleware environments. Our work focuses on the Java Bridge Module, which enables the SID simulation environment to use the Java Component transparently (see Fig. 3.1). The bridge module has three main components. The first of these is the Java Bridge Component, a native SID component that acts as a bridge to enable the loading and creation of a Java Component. The second is the Java Object Factory, which is a Java code that facilitates the creation of a Java component ordered by the Java Bridge Component. The third is a SID Java API, which standardizes the creation of SID components in Java (SID Java Component). In addition to providing a standard interface, the API also includes implementations to make SID component development in Java both faster and easier.

This chapter describes the basic features of the Java Bridge Module. Here, we describe the SID Interfaces that we interpreted from SID to Java, and vice versa. We discuss the theory of Tcl bridge components as a reference for constructing the Java Bridge components. A discussion of the Java Native Interface (JNI) and JNI wrapper is also included. We use JNI techniques to hook C++ calls to Java calls, and vice versa.

Later in this chapter, we explain our design and implementation of the Java Bridge Component, Java Object Factory, and SID Java API.

3.1 SID Interface

SID is an extensible simulation system. This property is a consequence of the underlying architecture of SID components. SID provides a simple, yet complete interface for creating and

adding custom components. This interface is defined in super-classes in the SID library package, and can be extended through C++ object-oriented programming.

Any component must implement the interfaces such as Components, Attributes, Pins and Buses, although most of the methods can be implemented trivially [15]. In addition, there are several mixing classes located in extra SID utility headers. These classes are not part of the official component API, but cover a wide variety of common component implementation strategies, and greatly simplify the task of writing a component. Programmers are encouraged to read these headers for further details.

The Bridge Module creates cross-language communication for the SID interface or SID calls. The bridge module contains codes in different languages (e.g., C++ and Tcl/Tk, or C++ and Java). These codes are compiled into libraries that can accept and forward the SID interfaces from one programming language to another.

3.1.1 Components

An SID component is a C++ object of type sid::component, which calls and is called by other SID components. A complete SID simulation is a set of connected components. Simulations are single-threaded, and there is no underlying "supervisory" simulation code. As the simulation runs, control repeatedly loops within a "root" component, calling into its connected components, which perform any local simulation tasks, and in turn then call their connected components before returning control to the caller. This cycle continues until the simulation is terminated. The Java Bridge Module contains an SID component that acts as a bridge component.

3.1.2. Attributes

SID components are equipped with a set of string-valued attributes, which can be queried and set during configuration or simulation, to inspect or modify the behavior of the component. Attributes generally provide information and control which is of interest to the simulation user, rather than to other components in the simulation.

Conventionally, attributes are grouped into named "categories" by role, although attributes may have no category, and may also be accessed without regard to their category. Categories merely assist in treating a group of attributes similarly; for example in a simulator's GUI.

3.1.3 Pins and Buses

SID models a component's communication connections as a set of buses and pins. These terms are intended to reflect the design of the hardware being simulated, and the SID's API presents these terms as the abstract C++ classes sid::bus and sid::pin. A SID component may have many pins and many buses, each of which represents the receiving end of a connection to another component.

A pin represents a non-refutable, one-way connection, through which an integer value may be "driven." Pins are best thought of as "triggers," which may carry an informative value, but which do not have any type of built-in request/response mechanism. A bus, on the other hand, represents a read/write interface, through which integer values and addresses are transmitted. Every bus request returns a sid::bus::status object, which describes both the success and failure of the request and the latency of the bus transaction, as modeled by the recipient. Buses are best thought of as small windows into the recipient's memory or register set, through which the sender may attempt to read or write values.

Pins and buses generally do not exist beyond the scope of the component to which they belong, and a component is usually responsible for managing the life cycles of its own pins and buses. Moreover, components seldom set up their own connections. Instead, each component presents a set of "factory methods," which can be called with string names of buses or pins. The component must then choose to either construct a pin or bus "on demand," serve a pin or bus from a fixed set compiled into the component, or respond with a NULL pointer, indicating that an unknown pin or bus was requested. A component may also be asked to place a pointer to a given pin or bus in a named slot within itself, which it will use as the sending end of a connection during simulation. Slots for buses are referred to as "accessors," whereas slots for pins are simply called "output pins."

3.2 The Bridge-Tcl Component Theory

A bridge-tcl component is a shell that hooks all SID API calls to an embedded Tcl interpreter, so that they can be handled as Tcl procedure calls. In addition, SID API calls are exposed to that interpreter, so the Tcl procedures can call back out to the C++ system. With these two capabilities, a user-provided Tcl package may become a first-class SID component.

Objects such as the bus, component, and pin pointers may be safely passed via Tcl scripts, as the bridging calls represent these as unique strings, and automatically convert them back into C++ pointers. Any pointers seen through incoming call arguments or outgoing call return values are transparently converted into unique long-lived opaque strings. In this way, C++ pointers can safely pass through the bridge-tcl in both directions.

Unlike C++ components, Tcl scripts that run in bridge-tcl do not have access to the sidutil group of utility classes. This means that only low-level operations are directly provided, and sidutil abstractions, if needed, would have to be rewritten in Tcl.

3.2.1 Incoming SID API Calls

Almost all incoming SID API calls are passed through verbatim to the embedded Tcl interpreter; exceptions are parameterized and noted below. Plain types are mapped according to the table given below: C++ object to Tcl for arguments, and Tcl to C++ for return values, see Table 3.1. If Tcl procedures are not loaded into the interpreter with the appropriate names by the time they are invoked from another SID component, a TCL_ERROR message is printed to cerr,

Table 3.1. Types conversion from C++ to SID, and vice‐versa

C++ Type	Tcl Type
string	string
vector<string>	list of strings
component,bus,pin pointer	opaque strings
{little,big,host}_int_{1,2,4,8}	numeric integer - care with 64-bit ints
component::status	string: ok, bad_value, not_found
bus::status	string: ok, misaligned, unmapped, unpermitted
vector<component*>	list of opaque strings
vector<pin*>	list of opaque strings
0 (null pointer)	""

Table 3.2. The multiple variation of outputs in sid::bus::read depends on which type of sid::bus

Incoming C++ call	Outgoing Tcl call
read(address,data)	read_h4_{l,b}Y $address ** return [list $status $data] **
write(address,data)	write_h4_{l,b}Y $address $data

and a function-specific error indication is returned.

Calls belonging to sid::pin and sid::bus are similarly mapped to Tcl procedure calls. The C++ pin/bus object in which they are called is passed to the procedures as an extra argument. C++ pin/bus objects may be constructed for a Tcl component through special callback functions, as listed below.

Functions with multiple outputs, such as the sid::bus::read reference arguments, a map to Tcl procedures, and returning a list with the mapped C++ return type as the first element and the output reference argument as the second element. Here is the example, in sid::bus,for host_int4 address and {big, little} int Y data types, it will use the read_h4.

3.2.2 Outgoing SID API Calls

Once a Tcl program is loaded into the interpreter, it is able to both make outgoing SID API calls and to respond to incoming calls. All SID API functions are exposed to Tcl as procedure hooks in a very symmetric way for the incoming calls. Each function in the incoming set has a shadow: sid::component::FUNCTION, sid::pin::FUNCTION, or sid::bus::FUNCTION, whichever is appropriate. Each outgoing procedure takes a receiver handle (the same opaque string passed in an incoming call) as its first argument.

There is no method of checking for preventing an outgoing SID API call from becoming recursive and referring to the originating component, either directly or indirectly. For all other components, the prevention of infinite recursion is the responsibility of the component author.

3.3 Java Native Interface

Java Native Interface (JNI) is a powerful feature of the Java platform. Applications that use JNI can incorporate a *native code* written in programming languages such as C and C++, as well as code written in the Java programming language. JNI allows programmers to take advantage of the power of the Java platform without having to abandon work that was written earlier in other languages.

As a part of the Java virtual machine implementation, JNI is a *two-way* interface that allows Java applications to invoke native code, and vice versa. Fig. 3 illustrates the role of JNI.

There are other mechanisms for integrating Java into C/C++ or legacy applications. A Java application may communicate with a native application through a TCP/IP connection, or through other inter-process communication (IPC) mechanisms. A Java application may connect to a legacy database via the JDBC API. It may also take advantage of distributed object technologies, such as the Java IDL API.

JNI enables Java to communicate with C/C++ in the same running process. Intuitively, this should provide better performance. This is where JNI becomes useful. For example, consider the following scenarios:

The Java API may not support certain host-dependent features required by an application. For example, it may be necessary for an application to perform special file operations that are not

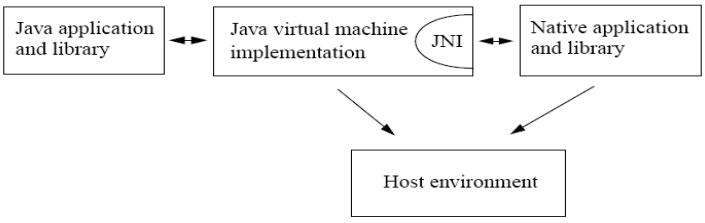


Fig. 3. Role of the JNI [16]

supported by the Java API, but it may be cumbersome and inefficient to manipulate the files via another process. It may be necessary to access an existing native library while avoiding the overhead of copying and transmitting data across different processes.

Loading the native library in the same process is much more efficient, and having an application span multiple processes could result in an unacceptable memory footprint. This is typically true if these processes need to reside in the same client machine. Loading a native library into the existing host process for the application requires less system resources than starting a new process into which the library is loaded.

It may be necessary to implement a small portion of time-critical codes into a lower-level language, such as an assembly language. If a 3D-intensive application spends most of its time in graphics rendering, it may be necessary to write the core portion of a graphics library in assembly code to achieve maximum performance.

3.4 JNI Wrapper

Although JNI is powerful, a JNI code can be complicated. However, there are a number of products and tools that can be used to help simplify the JNI experience. One major project is JavaOSG [17], a Bridge pattern-oriented solution that acts as a bridge to the native class to automatically generate a Java wrapper. It is sophisticated and robust, and to our knowledge is in the process of being open sourced. Unfortunately, it is not well documented, and the existing webpage is outdated.

Another such product is the JNIWrapper [18], which is the apparent proprietary market leader to date. It follows more of an Adapter pattern model, in that it attempts to adapt some native types so that native resources can be accessed through direct method calls. Unfortunately, this wrapper is not free.

Our solution is to use GIWS, a free script generator engine that encapsulates normal C++ into Java JNI calls [19]. Using GIWS, we can construct a C++ class that encapsulates a real C++ to Java call, so that a neat object-oriented code can later be used to call Java from C++. We used GIWS to create a library of Java object classes in C++.

3.5 Java Bridge Component

Java Bridge Component is a native SID C++ component that acts as a bridge to the Java Component. With it, the SID simulator can create a SID Component object from a Java class. In addition to being created, a Java component can also interact with other SID components via the SID interface. All of these features are seamlessly enabled by the Java Bridge Component, so

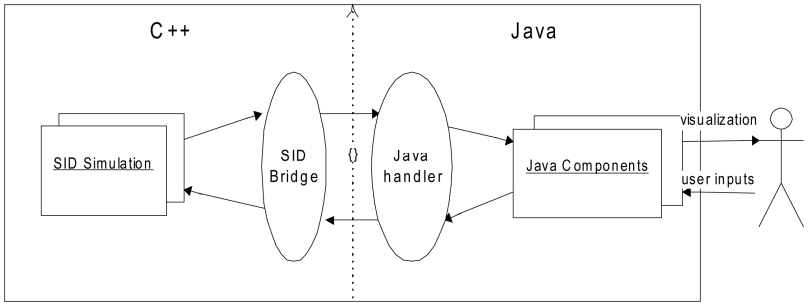


Fig. 4. Java Bridge Module Architecture [16]

that a normal SID user will not have to learn a new way to create or to use the SID interface in the Java Component.

The Java Bridge Component has a number of functions. The first is that this C++ component must extend sid::component, and must be exported as sid::COMPONENT_ LIBRARY_MAGIC, as with other native SID components. The JNI-based Java Bridge Component is encapsulated in libjavajniapi.la.

Second, the bridge must have the component list_types function. This is where a normal SID component begins to differ from a bridge component (e.g., Tcl/Tk-bridge or Java-bridge). Third, the bridge must implement the entire SID interface. This is also different from a normal SID component.

In this section, we explain the implementation of list_types and the SID interface in the Java Bridge Component.

3.5.1 Java Bridge Component's list_types

In a standard SID component, the list_types function contains only one name in its types vector. For example, liblcd2410.cxx has only one name (hw-lcd-s3c2410) in the types vector, and the name is written statically in the liblcd2410.cxx code.

On the other hand, in the Java Bridge Component, the list_types function can contain many component names in its types vector. It will execute JNI calls to the Java Object Factory and search for Java classes in specified directories and JAR files ending with "sid." Previously, it was necessary to specify these JAR files and directories in the SID_LIBRARY_PATH environment variable. After finding Java classes that match the criteria, it adds the class name(s) to the types vector.

The types vector is added to the SID runtime. All the names stored in the vector can subsequently be created or deleted via the bridge component. This implementation enables seamless loading, creation, and deletion in SID runtime and the SID configuration file.

3.5.2 Java Bridge Component's SID interface

Any component that extends the sid::component must implement the entire SID interface. However, whether a particular interface actually possesses some functionality is determined by the requirements of the component itself. In Java Bridge Component, the functionality of the SID interface is to call the corresponding SID interface in Java.

Before it is possible to call the SID interface in Java, a particular Java component must first

be created. This will be done if there is a new <java-component-name> syntax in the SID configuration file.

3.6 Java SID Object Factory

The SID Object Factory is a Java library that supports the creation of SID components. It receives a call from the Java Bridge Component to list all existing Java components. Subsequently, it can be instructed to create a particular Java component mentioned in the SID configuration file via the new <component library name> <component name> command.

The name of the object factory class in our implementation is SIDJFactory. When listing SID Java components, the object factory uses the FolderCompList and JarCompList classes. The FolderCompList lists all SID component classes in directories specified in the SID_LIBRARY_PATH. At the same time, the JarCompList will list all of the SID component classes in the .jar path(s) of the SID_LIBRARY_PATH. Below is an example of the SID_LIBRARY_PATH environment variable.

As indicated in the above figure, the environment variable can contain both directory and JAR file paths. After the object factory acquires a list of classes, it will send this list to the Java Bridge Component list_types function. The object factory stores the list and uses it to handle a createObject(String className) call from the Java Bridge Component. It then uses the Java reflection utilities from the java.lang.reflect package to create an instance of the required class in SID Java Component. The resulting Java object is then passed to the C++ Java Bridge Component for storage in Java Bridge Component as a compObj pointer.

```
labuntu:~$ echo $SID_LIBRARY_PATH
/opt/vdees/sid/lib/sidcomp:/opt/vdees/sid/share/sidcomp:/home/hasrule/lat_comp/lib/sidcomp:/home
/hasrule/NetBeansProjects/LCDComponent/dist/:/home/hasrule/NetBeansProjects/JavaBridge_jni
/dist /JavaBridge_jni.jar
```

Fig. 5. Example of SID_LIBRARY_PATH for Java Components

3.7 SID Java API

As mentioned above, the goal of the Java Bridge Module is to create a hooked API between the native SID in C++ and Java. We implemented the SID interface in the Java Bridge Component to make a call to the corresponding SID interface in Java, which is called the SID Java API.

SID Java API is implemented as abstract classes to confer abstraction on the child component. There are seven classes in this API. Three of these classes are the abstractions sid::component, sid::pin, and sid::bus in Java, and have no implementation. The others are classes that provide default implementations to facilitate faster and easier SID Java component development.

The three abstractions of SID classes are listed below.

And four classes that provide the default implementations are below.

First, we implemented these classes in Java. And later on, we used GIWS as a script generator to create C++ wrapper classes for these Java Classes so that we can simplify the calling of SID Java API from our bridge code.

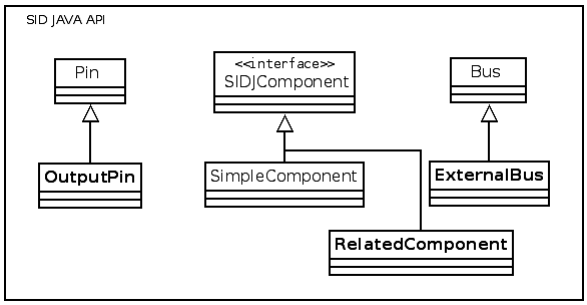


Fig. 6. SID Java API Class Diagram

Table 3.3. Abstract Classes of the SID Java API

Class Name	Description
Bus	This abstract class represents a sid::bus in the SID Java environment.
Pin	This abstract class represents a sid::pin in the SID Java environment.
SIDJComponent	This abstract class represents sid::component in the SID Java environment.

Table 3.4. Classes providing basic functionalities from SID Java API

Class Name	Description
ExternalBus	This class acts a pointer to the native sid::bus.
OutputPin	This class acts a pointer to the native sid::pin.
RelatedComponent	This class represents a native sid::component that has a "relate" function (relation-ship) with a SID Java Component.
SimpleComponent	This abstract class contains the default implementation for sid::component in a SID Java environment.

4. CASE STUDY

In this section, we present an example to illustrate the functionality of the SID Java Bridge Module and SID Java API. We cover all types of communication between components in SID. We also demonstrate the use of pin and bus mechanisms, as well as the attribute and relation mechanisms in Java Component, and show how the Java Bridge Module enables seamless communication between the Java Component and other SID components.

The example involves an LCD panel with touchscreen capability. It is assumed that a simple application is to be run on a typical embedded system board that includes a color LCD panel with touchscreen capability. It is also assumed that the application will run directly on the board, without any operating system in between. The LCD panel will be connected to an LCD controller for the display input (LCD images), and to an Analog-to-Digital (ADC) controller for the touchscreen capability.

For Java Bridge Module and API usage, we focus only on three components: the LCD panel, LCD controller, and ADC controller. The LCD panel is made up of Java components, and the other two are normal C++ components. In general, the LCD controller sends the data to the LCD panel, and the LCD panel will send data to the ADC (touchscreen) controller.

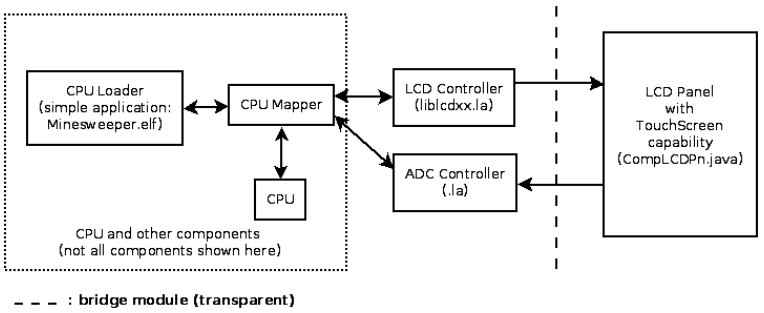


Fig. 7. Components' Interaction Scenario

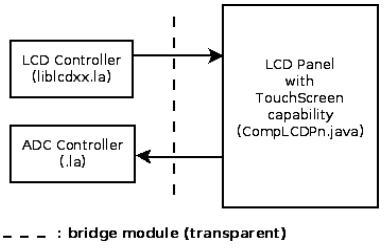


Fig. 8. LCD Panel Component Interaction

4.1. Basic Java Component Code

An SID component written in Java must extend the SIDJComponent parent class. An SIDJComponent is an abstract class that defines all SID communication APIs as abstract methods. Therefore, any class that extends it will have to implement an interface in the SID communication API.

We provide the SimpleComponent class to help the developer implement the Java Component. The SimpleComponent is a child of the SIDJComponent, and contains basic and default implementations of the SID communication API. By extending SimpleComponent, we can implement a new SID Java component with a few short lines of code. Here, we discuss how to create the LCD Panel by extending the SimpleComponent class, see Fig. 9.

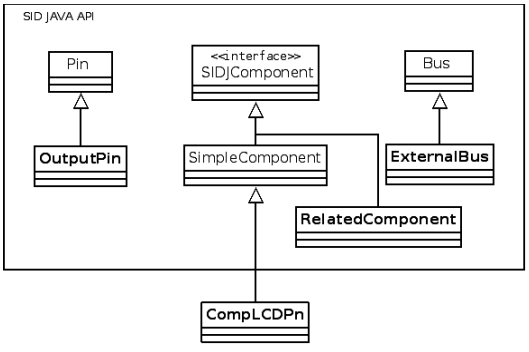


Fig. 9. New Component: CompLCPn.java

```
import sidcomp.*;
    public class CompLCDPn extends SimpleComponent{
    public CompLCDPn(){
            }
}
```

Fig. 10. CompLCDPn Class Basic Valid Form Extends the SimpleComponent

The simplest form of a SID Java Component is a class that extends the SIDJComponent or SimpleComponent. Rather than extending the SIDJComponent and reinventing or implementing many new methods, we prefer to extend the SimpleComponent as shown in Fig. 10.

Although short, the code in Fig. 10 represents a valid SID Java component. By building the component and inserting the build into the SID_LIBRARY_PATH, it can be loaded into the bridge and created in the simulation. However, in the absence of a SID communication API implementation, the new component cannot interact with other components. Thus, the above component is not yet of any real use in the simulation.

4.2. Component Loading and Creation

To create a C++ component in SID runtime, it is necessary to load the libraries of components in the configuration file. We can then create the component by using the new command. To create a Java component, it is only necessary to load the Java bridge library. There is no need to load the LCD Panel component library in the configuration file. Instead, we must insert the LCD Panel build into the SID_LIBRARY_PATH.

C++-based Component	*#include libraries* **load ADC_TS_Controller.la adc_component_library** **load liblcd2410Java.la s3c2410jlcd_component_library** *#touch screen* **new hw-adc-controller adc_cont** *#lcd controller* **new hw-lcd-s3c2410-j lcd**
Java-based Component	*#include bridge library only* **load libjavajniapi.la java_bridge_library** *#no need to include component as a single library* *#lcd panel creation* **new CompLCDPn display**

Fig. 11. Comparison Between the C++-Based Component and the Java-Based Component Loading Mechanism in the SID Configuration File

4.3. Component Communications

To make a component useful in a SID simulation, it is necessary to implement the SID communication API for the component. Once this API has been implemented, the component will have one or more of the SID communication mechanisms, which could be a pin, bus, or "relate" attribute. Relate attribute is a special attribute in component communications, by using it we can virtually relate one attribute in one component to an atttribute in another compoennt.

Java Component
LCD screen update connect-pin lcd mode -> display mode connect-pin lcd row-col -> display row-col connect-pin lcd frame -> display FR *#Touch screen update* connect-pin display xp-pin -> adc_cont xp-pin connect-pin display yp-pin -> adc_cont yp-pin connect-pin display down-pin -> adc_cont down-pin

Fig. 12. Java Component Attribute Access and Pin Connect Assignment in a SID Configuration File

Eventough the real hardware only communicates with Pins and Buses, relate could be a great shortcut to make component communications.

In the component interaction configuration, there is no difference between a Java component, a Tcl/Tk component, and a C++ component, and they can be used transparently as a single module of SID components. In this example in Fig. 11, we demonstrate the use of these mechanisms in a LCD screen update and the touchscreen input from a LCD Panel.

Simply adding the above pin configuration into the configuration file without modifying the new Java component (i.e., the CompLCDPn.java source code) will result in SID runtime errors. This is because we have not assigned pin objects to the Java component and all that is required is to create these pin objects.

There are two types of pin objects. The first is a normal pin object, which could be called an input pin. Ordinarily, this pin object drives data from the native SID to the Java Component. In this case, we should define the handling for data being driven to the component. For example, in the row-col pin, we must define what happens once data is driven to this pin. In our scenario, the handler for this pin is buffering the data to the Java image buffer. If the FR pin subsequently

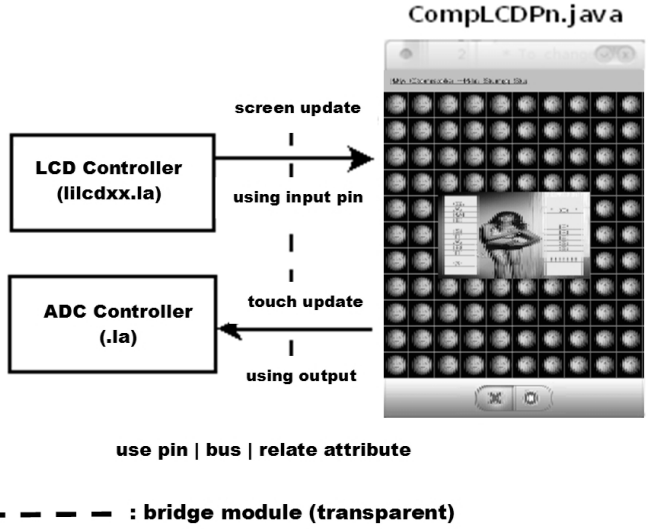


Fig. 13. LCD Panel Component Interaction with the Bacteria Wars Game

Table 5.1. Experiment Results on the Time Consumption for Function Calls and the Average Frame Rates

Parameter	Tcl/Tk	Java
The "driven pin" function Time (μs)	29.4	6.5
Frame rates (fps)	1.4	21.7
Simulation of the 50 loop time (second)	750	750

receives new data, one frame transfer has been accomplished and the LCD screen must be refreshed.

From the above example, we can override the method Pin::driven to do some specific task given the driven value as its parameter. On the other hand, we need only to create instances of output pins such as xp-pin, yp-pin, and down-pin in CompLCDPn.java. There is no need to specify any method nor implementation for these kind of pins, as the driven method in the OutputPin.class is implemented to the drive data to some remote component. The scenario is illustrated in Fig. 13 .

5. EVALUATION

In this section, we will compare the Java Component with the legacy Tcl/Tk Component. We only compare the Java Component with Tcl/Tk since we created the Java Bridge Module and API to be an alternative for Tcl/Tk and not for the C++ component.

5.1 Parameter Selection

Although we created Java Bridge primarily to enable components to be more compatible with the Eclipse environment, we also wished to improve speed using the Java Component rather than Tcl/Tk in the simulation process. Accordingly, we defined and measured three parameters in the case study discussed in the previous chapter.

- **Function time:** In our case study, this is the time required for one "driven" call using Pin (in microseconds). Each driven pin contains 2 pixels of information.
- **Frame refresh rate:** The number of frames refreshed in one unit of time (in fps). As our LCD resolution is 320 × 240, this will include 38,400 driven pin calls.
- **Total loop time:** The total simulation time required for sending a specified number of frames (in seconds). Here, the specified number is 50 frames.

5.2. Results

The comparison test was conducted using Ubuntu 10.04 (Lucid) Linux running on an Intel Core 2 Quad CPU, with a clock speed of 2.33 GHz for each core and 2 GB of memory. The performances of Tcl/Tk and Java in driving a pin from the native LCD controller component to the LCD component are compared in Table 5.1.

As we can see in Table 5.1, the Java-based component can perform almost five times faster on a single pin driven function execution and showe 15.5 times faster frame rates. As we mentioned in the previous section, Tcl/Tk takes much time in packaging and converting the binary object

into a string object in the bridge-tcl component and then parsing the string one by one on the Tcl/Tk-based component side. The Java-based component in the other sides has its native corresponding object type in C/C++ and Java using JNI. At the moment when the object from the C/C++ environment is being passed to a Java environment, a copy mechanism will be automatically performed by JNI so that we only need to ensure that a type of it will comply with the interface specified in both sides.

We provided interfaces for four primitive type objects (byte, int, long, and short) and four complex object types (SIDJComponent, Bus, Pin, and SIDJFactory). The prior objects are an adaptation of primitive variable types into Java object that are to be be transferred from C/C++ environment to a Java environment and vice versa. Meanwhile, complex objects are used to be able to complete a function call experienced by the Java Bridge, which are to be forwarded to the corresponding complex objects that reside in the Java environment, just like a mirror where one complex object in the C++ enviroment will be mapped to one object in the Java environment. Thus, the calling method will be simplified and the creation and destruction of objects will be consistent where as the creation of one complex object in the C++ environment will also be the creation of an object in the Java environment, as will be the same with destruction. The SIDJComponent is the blue print of a component in SID, consisting of required methods and attributes. The Bus and Pin are also the blue print of the Bus and Pin in a regular (C++-based) SID component. Meanwhile, the SIDJFactory is the creator of complex objects by request of the Java Bridge component. SIDJFactory employs Java Reflection API so it can create any new object, in this case, a SID Component, on the fly.

5.3. Qualitative Comparisons

In addition to its performance, the programming of the Java Component also offers some benefits. Java is a strong object-oriented language, and can therefore provide a considerable amount of extendibility to child classes that correspond to desired concrete Java components. In this section, we present a comparison of the coding effort associated with Java and Tcl/Tk components. Specifically, we compare the number of lines of code required to create a certain function.

5.3.1. Functionality

The Java Bridge has the equivalent functionality of the Tcl/Tk bridge, as it can create a foreign component and forward the SID interface to the foreign component. It can also execute a call in the opposite direction from a foreign component to a native SID component. Therefore, in terms of functionality, the Java and Tcl/Tk bridges both accomplish the same task (i.e., enabling a non-native component to be used in the SID simulation environment).

5.3.2. Convenience

Java Bridge is more convenient than a Tcl/Tk bridge. By using Java Bridge, we can create a component in Java, and thus immediately obtain all the benefits of Java.

Java is a strongly object-oriented language, and has strict rules for extending a parent class and implementing an interface. Due to these strict rules, all SID Java components are assured to be 100% compatible with the SID environment. Any incompatibilities in the code can be identified far in advance of the runtime, as a Java class cannot be constructed unless its inheritance rules are fully satisfied.

Java also has a better runtime bug-tracing mechanism than Tcl/Tk or even C++. With Java Stack Trace, we can obtain a user-friendly snapshot of the threads and monitors in a Java Virtual Machine (JVM). Depending on the complexity of an application or applet, a stack trace can range from 50 lines to thousands of lines of diagnostics. In contrast, a runtime error in Tcl/Tk or C++ will only display one level of error messaging.

5.3.3 Productivity

An object-oriented language provides the benefits of inheritance, which endows a child class (e.g., our new SID component) with the ability to reuse or extend the properties and behavior of the parent class. This ability results in faster development, leading to greater productivity.

Java is, by nature, a strongly object-oriented language. Our Java Bridge Module contains the SID Java API, which not only provides the set of SID interfaces that must be implemented in the Java Component, but also enables us to create default implementations of the SID interface. Thus, a created component can extend the class that contains default implementations (SimpleComponent.class) to deliver an efficiently written Java component. This will result in improved productivity in creating components compared to Tcl/Tk. Nevertheless, the user can still extend the basic SID API class (SIDJComponent.class) and reimplement the SID interface from scratch.

Use of the Java Component also provides an opportunity to improve productivity with the creation of an IDE that fully supports the development of SID Java API.

6. CONCLUSIONS

In this paper we proposed a Java Bridge as an alternative to the current Tcl/Tk Bridge in SID. The bridge between SID and Java component would be built by using JNI and a socket. Here we focus on using JNI since JNI is considered as the best in terms of performance and ease of use. Our tests have shown that the Java Bridge module using the JNI is giving a much better performance than its Tcl/Tk counterpart.

We also designed and implentend SID Java API as an insurance for a Java component to be compatible to SID. SID Java API also provides the basic functionalities so it can help the developer to reuse some default implementations of the SID interface. Hence, this will help the development of an SID component in Java to become faster and easier to use.

We hope this work can be continued with the development of the Integrated Development Environment that specifically supports the SID Java Component. This particular IDE can be an addition to the current Eclipse-based Virtual Development Environment for Embedded Systems (VDEES).

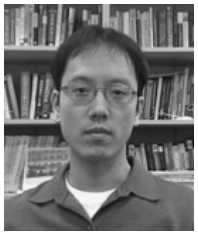
REFERENCES

[1] Stephen Alan Herrod. *Using Complete Machine Simulation to understand complete Machine Behavior*. February, 1998.
[2] *SimOS*: *The Complete Machine Simulator*. http://simos.stanford.edu .
[3] *SID System Simulator*. http://sourceware.org/sid.
[4] *SID Simulator User's Guide*
[5] *VDEES*. http://cslab.sunmoon.ac.kr/vdees/

[6] Hadipurnawan Satria, Budiono Wibowo, Jin. B. Kwon, Jeong B. Lee and Young S. Hwang. VDEES: *A Virtual Development Environment for Embedded Software Using Open Source Software. IEEE Transactions on Consumer Electronics*, May, 2009, Vol.55.
[7] Febiansyah Hidayat, Hadipurnawan Satria, Jin Baek Kwon, Software *Verification of a Virtual Development Environment for Embedded Software, In Proceedings of the 9th WSEAS International conference on Software Engineering, Parallel and Distributed Systems*, 2010.
[8] Hasrul Ma'ruf, Febiansyah Hidayat, Jin Baek Kwon. *Java Bridge Module and Java API for SID Simulator. In Proceedings of the 10th WSEAS International conference on Software Engineering, Parallel and Distributed Systems*, 2011.
[9] Mario de Sa Vera. "*JNI-bridged Java Desktop Application*."*Java Developer Journal*, published August, 1, 2006.
[10] Rod Fatoohi, Vandana Gunwani, Qi Wang, and Charlton Zeng. *Performance Evaluation of Middleware Bridging Technologies*. 2000.
[11] *Tcl/Tk*. http://www.tcl.tk
[12] *Eclipse Platform*. http://www.eclipse.org.
[13] *Eclipse C/C++ Development Tools (CDT)*. http://www.eclipse.org/cdt.
[14] *Eclipse Java Development Tools (JDT)*. http://www.eclipse.org/jdt.
[15] *SID Simulator Component Development Guide*.
[16] *Sheng Liang*. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley 1999.
[17] *JNIWrapper*. http://www.teamdev.com/jniwrapper/.
[18] *JavaOSG*. http://www.noodleheaven.net/JavaOSG/javaosg.html.
[19] *GIWS*. http://forge.scilab.org/p/giws/.

Hasrul Ma'ruf
He received a B.S. degree in computer science from University of Indonesia in 2007. He joined the M.S. degree program in computer engineering at Sun Moon University in 2009. His research interests are in the area of embedded system simulation, and green computing.

Hidayat Febiansyah
He received a B.S. degree in computer science from the University of Indonesia in 2007 and an M.S. degree in computer engineering from Sun Moon Univ. in 2010. Currently, he is enrolled as a Ph.D. candidate in Sun Moon University. His research interests are in the area of embedded system simulation, and multimedia systems. They include topics such as system simulation engines and broadcasting approaches for multimedia content distribution.



Jin Baek Kwon
He received his B.S in statistics from Hankuk University of Foreign Studies in 1998, and his M.S. and Ph.D. degrees in computer science from Seoul National Univ. in 2000 and 2003, respectively. Currently, he is an associate professor in Dept. of Computer Science and Engineering in Sun Moon University, Republic of Korea. Also, he is a project director of "Regional Leading Embedded Software Team" of Brain Korea 21, which is a national project. His research interests are operating systems, distributed systems, cloud computing, etc.