Using Tcl To Build A Buzzword* Compliant Environment That Glues Together Legacy Analysis Programs

Carsten H. Lawrenz, Rajkumar C. Madhuram,
Siemens Westinghouse Power Corporation, Orlando, Florida
`{Carsten.Lawrenz,Rajkumar.Madhuram}@swpc.siemens.com`

Abstract. The Siemens Integrated Design (SID) Environment is a system that allows engineers to link together many legacy computer programs. This capability provides significant reduction in effort for defining the conceptual design of electrical generators. The SID environment is a generic tool for running all types of analysis programs (methods) as well as managing their associated data. Methods are plugged into the environment in a simplified fashion by using a well-defined interface. Any features that are added to the environment immediately benefit all methods. Data can be shared between remote sites through an in-house developed, java based, replication server. This paper discusses how Tcl was used to develop the SID Environment and why it was the best choice for our application.

***** *Buzzwords: Scalable, Multi User, Client Server, Distributed, Cross Platform, Customizable.*

1. Introduction

1.1 Business Case

Siemens Westinghouse relies on various complex computer calculations for designing electrical generators. Various computer programs (methods) written in FORTRAN and other languages have been used over time. Because each method addressed specific aspects of generator design, they existed as standalone entities. Running each method required the manual creation of input files and the manual extraction of output data to prepare it as the input for other methods. More complications arose when trying to incorporate the more contemporary functionality of spreadsheets. This process, and its many iterations, when performed by several different design teams, resulted in several manual hand-offs of information. Furthermore, this created an immense paper trail, data integrity issues, and time and effort spent trying to keep the design teams synchronized.

There was an effort to reduce the time required to conceptually design an electrical generator from 180 Days and 10 engineers to 18 Days and 3 engineers. The project was incrementally funded; therefore quick turn around time was required.

Although most of our user platform was UNIX based, we knew we may want to use the Windows platform some time in the future. The computer languages with GUI support that ran at the time (1995) on both platforms were limited. We evaluated a third party GUI library (Galaxy) which was C based. We found its use of coding too difficult to develop a prototype quickly. Java was still only known as Coffee. By chance, the authors discovered Tcl/Tk. Although Tcl was not officially available on Windows at the time, various Windows versions did exist.

Siemens Westinghouse designed an architecture that enabled methods to be easily plugged in, instantly sharing data among other methods. Using Tcl lent itself to fast prototyping and development. Six months into development, engineers were able to start using the newly created environment.

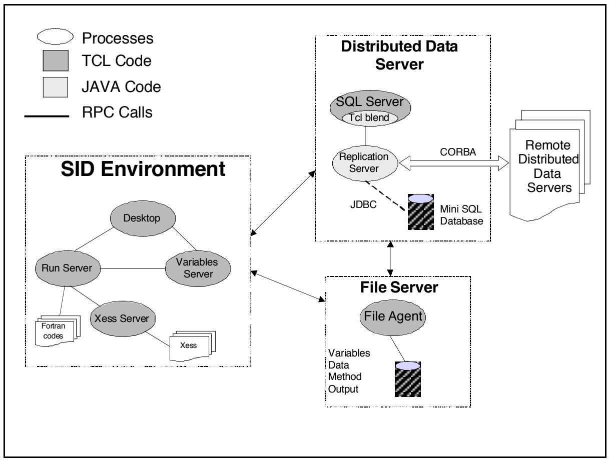


Figure 1: SID Architecture

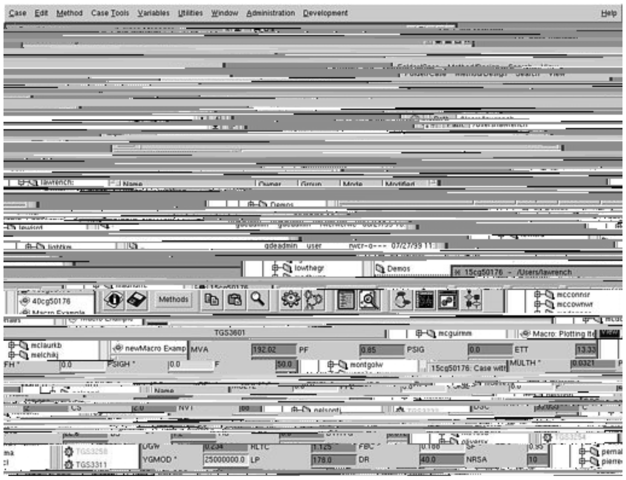


Figure 2: Snapshots of the SID Environment
(Left: Case Manager, Input Screen, and Variable table. Right: Dynamic Input Screen, Data Flow and Process Maps)

1.2 Basic Usage

The SID Environment simplifies the conceptual design process by automating and linking manual tasks. Users attend a one day training class during which they're ID's are registered in the SID database. Once registered, typing 'sid' in a command window launches the environment.

Each user works in their own unique data set that we call a case. These cases are collected in folders and are presented in a hierarchical format similar to a file manager. The folders and cases also mimic UNIX type file access permissions, which the user can modify. Any a number of methods are associated with each case. The user opens a case by selecting a method. The case is then locked to disable access by other users.

The case is displayed to the user as a window. Within this window the input screens for each associated method may be displayed. A method screen, by default, is a table of variable names. The environment allows customized input screens for each method as well. Users can select on any variable's entry box and change its value. Validations of input data for each variable are provided. A popup menu provides descriptions for each variable. The variables' descriptions, units and type are all defined in a global variable file.

The scope of each variable is global within the case, which allows methods to communicate. The user may select to run one method or several methods in series. As each method runs, the environment automatically updates the variables within the scope of the case. This transfer of data is accomplished by the use of input and output forms which are detailed below.

The environment also provides many utilities for viewing the various output types (PDF, postscript, plots, HTML) created by the methods. All output viewing is launched from a data viewer tool. This allows the less computer savvy users to be very productive.

2. Architecture

The architecture of SID Environment is outlined in Figure 1. The SID environment is a collection of servers and the desktop client. The desktop client (shown in Figure 2) handles most of the user interactions with the system. It communicates with several other servers for performing various functions.

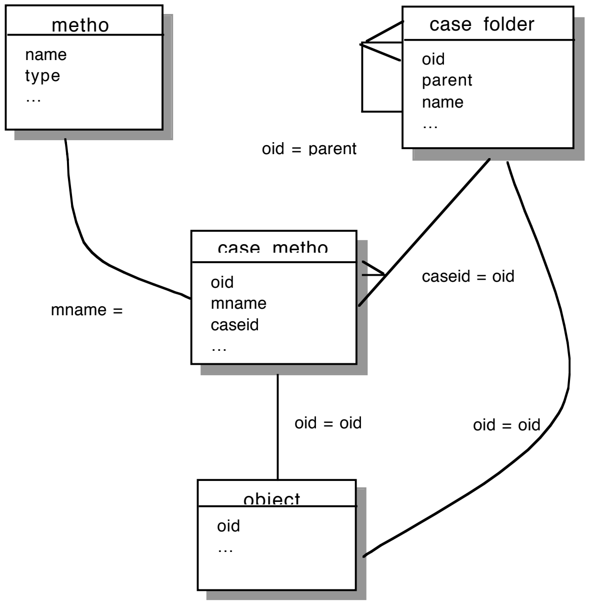


Figure 3: E-R diagram for case and method tables

Variables are the most widely used logical entities in the system. They are stored in a data file in key-value ordered pairs. Each method takes input from a set of variables and generates output that is returned. For every design calculation or analysis, engineers use a set of methods, which we group as a case. In essence, a case is the set of methods and their related variables.

Cases have to be managed in a reliable and fail-safe manner, especially in a multi-user environment. Hence, we employed a mini sql (mSQL) database [Hugh99] to hold the information about the cases, methods, users and the relationships between them. Cases are organized in the fashion of a UNIX file system, with a hierarchy of folders and permissions to control access. Each case and each method in a case are uniquely identified by an object id (oid) and its parents oid. The relations between these entities are shown in Figure 3.

As the need for exchanging data between engineers in remote sites surfaced, we wanted some of the folders to be shared between all the sites. In order to implement such a distributed system, we decided to use CORBA because it has established itself as a reliable and robust way to build distributed applications [Wolf98]. A replication server was coded in Java, which uses JDBC to communicate with the database. Tcl blend was used with the data server to access the replication server objects.

It was also required to port the system to Windows NT. In order to give uniform access to the case data, a file agent was required. It handles data movement to and from a case on behalf of the user into the central data repository. This mechanism also provides an added level of security, since all the data is owned by the file agent and access is only allowed through this file agent.

3. Software Development

3.1 Debugging

The fact that Tcl is as fully interpreted language lends itself well to software development. Many programmers fault Tcl because it does not provide syntax checking like other languages such as C. However, in the Tcl mode of programming the developer is able (with tools like TkInspect) to dynamically modify the code while the program is running. Furthermore, modified code can easily be reloaded into the interpreter by re-sourcing the code without having to exit the program. Also, bug fixes can be copied out to the production area without having to take the whole environment down. We found this approach to programming to be much quicker than the code, compile and debug method.

Tcl's error handling is more graceful than C or Java also. Most errors aren't fatal, i.e. the environment usually does not crash when errors are encountered. We overrode the error handler with a dialog box, that allows the user to email the developer a description of what caused the bug and a stack trace. This usually provides the developer enough information to determine the cause of the error.

3.2 Coding

SID requires relatively few lines of Tcl code. This is advantageous because the development team is only allotted 1_ man years per year for environment maintenance. The SID environment contains over 115 thousand lines of code and is maintained by only two developers. This smaller body of code also lends itself to utilities such as Concurrent Versioning System (CVS). Much of the coding is almost self-documenting; understanding code logic comes quickly. It's conceivable that to achieve this same functionality using C or Java could require about ten times as many lines of code.

The Tcl auto loading of methods also helps developers organize their source files in a comprehensive manner either by components or functionality. In the SID environment, our source directories are several levels deep.

3.3 Ease of Learning

The development team relied on contract labor to help meet the demands of the project in the early stages. The developers that were hired to program the environment did not have any prior knowledge of Tcl/Tk. However, it was easy to get motivated programmers up to speed quickly and start development. One observation is that experienced C programmers do not necessarily make good Tcl programmers. Strings and lists manipulations are so powerful and easily handled in Tcl. Tcl programming requires users to accept a paradigm shift while solving problems.

3.4 Reusability

Adding a method to the environment requires insertion of a record in the method table (Figure 3) and the creation of a file *(method.screen)* that lists all of the input and output variables used by that method. SID parses the list, and by default, a generic input screen is created for that method (See input screen in Figure 2). Several other applications source this same file. For example, there is a web server script that reads this file to create an input/output dictionary the methods online documentation. There is also an administrative application that reads every methods screen file and creates a matrix of all variables and how they are used by each method. This matrix allows SID to determine if other method's data has been invalidated due to the change of a variable's value. This simplifies method administration because all information is kept consistent and concurrent.

4. Most used Features of Tcl

4.1 Graphical User interface

The SID environment relies heavily on GUI components. The environment creates a large number of entry widgets, which are used for data entry. There are also a lot of bindings attached to these widgets to handle data validation. The GUI commands blend well with the source code because of their simplicity compared to other languages such as Java [WeFr97]. Very complicated and large input screens are easily handled by the Tk extension, as also witnessed in other large applications [Angel98] [DeCl97]. Our own experience has shown that dynamically creating an input screen with over 100 entries in Java required minutes compared to seconds in Tcl.

Some methods are preprocessors to complex Finite Element Analysis (FEA) models. These methods have customized input screens with simplified graphics of the various model components. These graphic elements may be adjusted manually to modify variables or they can redraw themselves to reflect the value of variables (See graphic input screen in Figure 2).

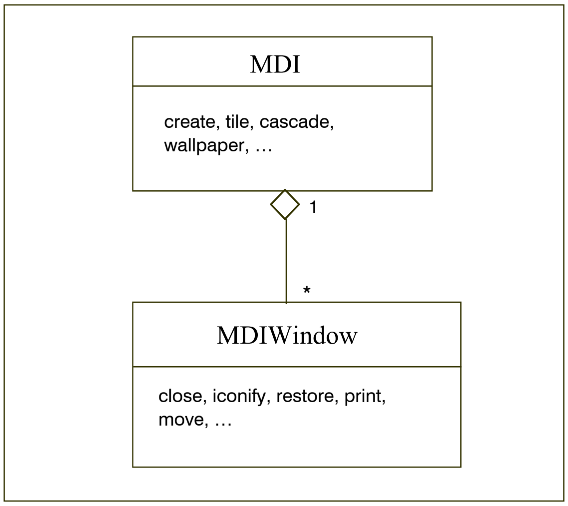


Figure 4: Tix MDI classes used in SID

We also created our own Multiple Document Interface (MDI) widgets on top of the Tix framework. The MDI system consists of the MDI widget and the MDIWindow widget (Figure 4). The MDI widget is a container widget that contains MDIWindow widgets. All the tools within SID environment are built using the MDIWindow widget. The advantages of such a scheme are many: 1. It enforces uniformity in all the windows and provides access to features provided by the MDI system and 2. It provides a compact environment where all SID related components are held together.

Users can create new customized input screens. By placing local versions of the screen files in a pre-defined directory. These files override the production version of the files. This allows developers to test functionality without having to check out a entire local copy of the environment. Once the new screen has been tested it can be copied out to the production area to be shared by all users.

The user can customize the look and feel of the environment. There are many configurable options like wallpaper, fonts, background color etc. In addition to

GUI, many options like viewers, sound etc., can be customized.

4.2 Strings and List Processing

We use the string processing capabilities of Tcl in SID. It is easy to create meta-languages for various purposes. For example, we developed a "forms" language, which is Tcl with a few abstractions that provide a generic interface to the legacy codes. An input form (iform) is a template for creating input (Figure 5) and an output form (oform) is used to get the output values back into SID. It is an easy and yet very powerful method of parsing the input and output.

Another example where the string processing capabilities were heavily used was in creation of a macro language. A macro is a sequence of SID actions that an engineer can use for design work. We created an environment where macros can be edited and run, complete with debugging options like stepping, watch and breakpoints. The language of choice for the macro was obviously Tcl and we supplemented it with some commands like RUN, GRAPH, CALL etc., to provide some higher level abstraction. For this, we used regular substitutions (`regsub` command). We avoided using slave interpreters since a tight integration with the data in the environment was necessary.

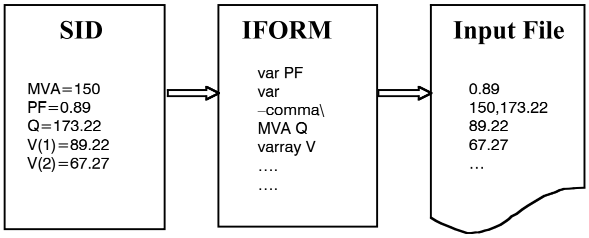


Figure 5: Mapping of SID data to Input File

In order to perform dynamic highlighting when a macro is run and also for debugging, a parser is needed. Conventionally, one would use lex and yacc to specify the grammar and then compile it with C to get a parser. However, we decided to use the powerful `regsub` command in an iterative fashion. Whenever a macro is run, it is first pre-processed in a two-phase method. We first substitute markers of the form @@*Mark[nn]*@@ instead of newline characters, where *nn* stands for the current line number. We also handle line continuations by introducing special markers. In the next phase, all the markers of the above form are substituted with a *macroPhase2* command and a check to see if it was halted. The command *macroPhase2* is called with the current line number and a pointer to the macro environment. It handles things like highlighting the current line in the editor, handling break points and also acts as a state machine to put the macro engine to the next state based on the user action (stop, reset, run etc.). Since a macro is typically a small script (less than 100 lines), the pre-processing is fast (~0.5 secs/100 lines of code) and hardly noticeable. Tcl enabled us to create a fairly robust macro system within a matter of days, which would not be possible had we chosen a different language.

Large lists are handled efficiently in Tcl from version 8.0 onwards. Consequently, we found a tremendous increase in performance of SID. The number of variables in a typical case can go above 5000. We store these variables in global arrays using `array set` command, which is many times faster than iterating through the list and storing them individually.

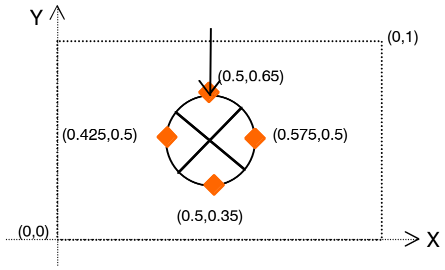
4.3 Global Variables

Many of the routines in SID rely on pointers, which are actually references to global arrays. Pointers are useful in creating complex data structures. They also provide an extremely convenient and clean way of keeping track of state information within an environment as large as SID. We use a single global variable GV that provides pointers to all the information about the current state of the environment. This makes it easy to organize the data in a hierarchical structure. For example, in order to determine if a module named tgs8000 has valid data in the active case, we could traverse like this

```
deref $GV(active_case) case;
deref $case(module_ptr) module_list
deref $module_list(TGS8000) module

if $module(valid) {
   ……
}
```

The pointer mechanism comprises commands `struct`, `alloc`, `free` and `deref`. The `struct` command can be used to create data types similar to that in C. Whenever the `alloc` routine is called, it creates a global variable of the form _mem<*nn*> where *nn* stands for a sequence number generated using a counter. A pointer is returned, which is of the form <level>#_mem<*nn*>. The `deref` command creates an alias for the global variable referenced by the pointer. In the absence of object oriented constructs (i.e, without Itcl), the pointer mechanism provided a way of neatly packaging different components inside SID. Also, we wrote a simple script that would go through all the `struct` definitions and create html documents. It serves as a good reference document to the internal data structures of the system.



```
deref [set OBJ(junc) [alloc array]] JUNC
.........
set JUNC(connPoints) {{0.5 0.35 v} {0.425 0.5 h} {0.5 0.65 v} {0.575 0.5 h}}
set JUNC(connRules) {
     {{in <= 3} {a maximum of only 3 inputs are allowed}}
     {{out <= 1} {you can have only one output from a junction box}}
}
```

Figure 6: Specification of a process chart primitive

Variable tracing is another aspect of Tcl that we used in SID. We use it in macros to associate the pseudo variable names with the real variables so that the internal mechanisms are hidden from the user. It is used in several places when we need to fill certain lists so that it remains consistent with the context of the application. One interesting lesson that we learned was to avoid variable tracing if it is needed only in certain instances when the variable in question is accessed. Trying to have a global variable that flags the tracing on and off creates problems that are hard to debug. One instance is when an error causes the interpreter to spiral out of a routine before the flag is not restored to its proper state.

4.4 Graphics Capability

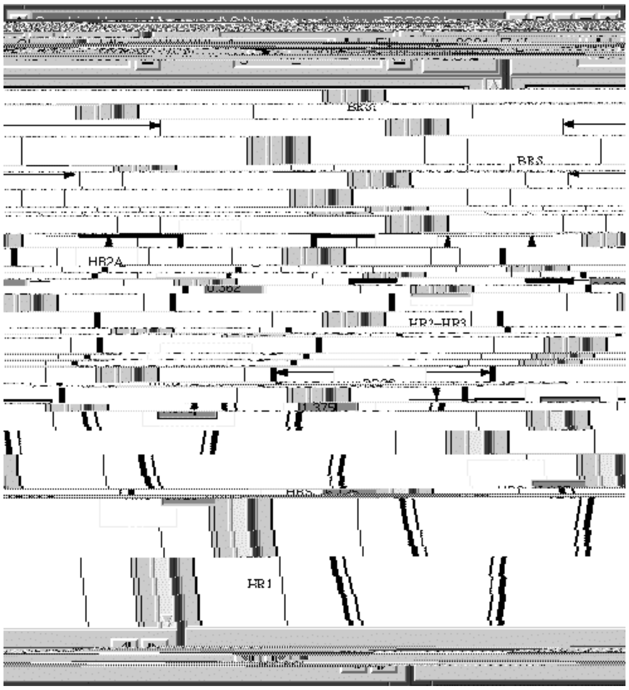


Figure 7: Snapshot of Graphical Input Screen

The canvas widget is the only one that provides drawing capabilities. Nevertheless, it is very powerful and we used it in components like process maps, data flow diagrams and custom methods. Process maps are basically flowcharts that represent an engineering process. The goal was to create and represent these processes inside the SID environment. First we investigated commercial charting programs, but these did not suit our requirements. Also, the Slate package [RL98] was not available at that time. Therefore, we decided to use the canvas widget to create one. We developed a set of primitives such as decision box, process box, sub-process box, etc. The whole process chart is represented as a flow chart structure. The sub-process primitives have pointers to the graphs of the corresponding sub-processes. Thus, the result was a hierarchy of graphs for a given process. All of these were neatly handled by the pointer mechanism that was discussed earlier.

Another interesting aspect of the process charts is the links that connect the different primitives. We introduced the notion of connection points, which are pre-defined positions around the primitive from which a link could be drawn. When a user drags the mouse to create a link, we search for the nearest connection point that is available. Also, every primitive has constraints on the connection points. For example, the start/end box could have only one output or one input and the decision box can have only one input and two outputs. Again, the scripting nature of Tcl “came to the rescue” in describing and checking the constraints. For example, the junction box object has a description similar to the one shown in Figure 6.

The co-ordinates of the connection points are relative to a hypothetical 1.0x1.0 bounding box of the primitive. The v and h indicates that only a vertical or horizontal connection is allowed at that point respectively. While parsing the rules, "in" is substituted with the number of total inputs (+1 if the current connection point is being considered for an input) and "out" is substituted correspondingly. The rule engine goes through each rule and evaluates the rule expression. If it fails, it displays the corresponding error message and returns. Tcl makes the process very generic and simple. One could even have rules such as "in+out <= 1" (in case of start/end box).

Each of the primitives and links has a pointer to an array that contains information about the object. The tags for each are also named accordingly. The tag for a primitive may look like obj_0#_mem72 and that of a connection, like con_0#mem66. Using `regexp` and `deref` commands, we get a quick access to the properties of the object when it is selected for various reasons like cut/paste, delete etc. The editor also provides powerful features like cut/copy/paste and undo/redo operations.

Once we got the process map editor in place, incorporating it into the environment was simple. Next, we were also able to make the process "run". A flashing circle beside the primitive indicates that it is currently executing. The handling of multiple charts in each case was done using our MDI window widget.

We use the BLT graph widgets for plotting graphs within the environment. The package PlPlot [PL99] is used to generate high quality engineering graphs without having the need to display on the screen.

We also provide graphical input screens, which are entry widgets overlaid on a gif image of a drawing (Figure 7). This provides an intuitive interface, especially for the novice users to enter values for various input variables in the method.

4.5 Networking/Communications

Communication between the various networked components is done using the Tcl-Dp package. The RPC mechanism is robust and provides a simple interface for servers and clients.

The servers within the SID environment are spawned on every invocation. It is not possible to assign fixed port numbers to each server since more than one user can be using SID (with remote displays) on a same machine. In order to solve this problem, whenever a server is spawned, it searches for a free port and stores the port number information in a registry file. This file serves as a directory for the various network components.

Interactions between the servers can be quite complex. Figure 8 shows the events that take place when the user clicks on the RUN button. Communications between servers take place through RPC and the status of other processes are monitored by the UNIX signal trapping commands provided by extended Tcl.

The data server provides replication services. Because, replicating all the cases involves huge network traffic and is largely unnecessary, we identified a subset of folders that would be replicated. There are two

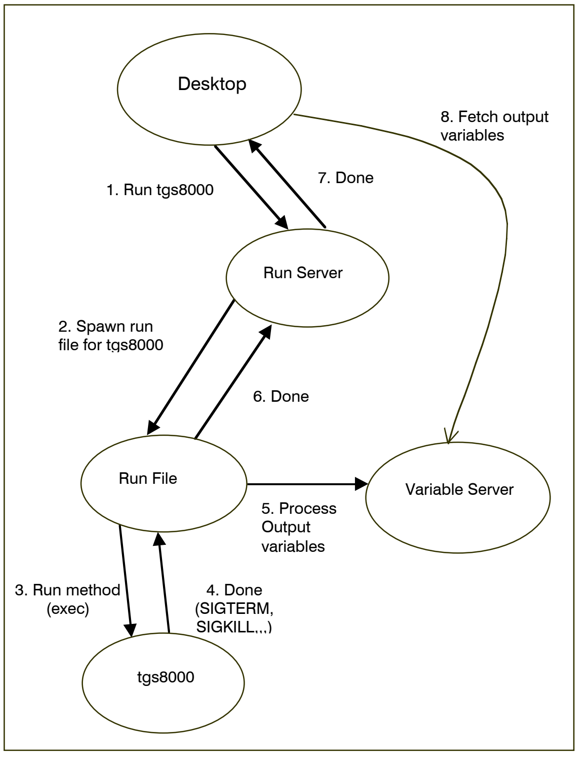


Figure 8: Sequence of events during a method run

replication modes, synchronous and asynchronous. In synchronous replication, the action has to be completed in all sites before proceeding. For example, when someone tries to open a case, it needs to be locked. The sql command to lock the record is passed on to all the sites. In each site, the command is supplied to the msql daemon, via JDBC. Only when it is successful in all sites, the locking is valid. On the other hand, asynchronous replication passes on the command and simply returns. Each site holds a command queue (java layer) which keeps trying until the operation succeeds. Replication of case data is handled using FTP. We tested the replication between three sites (Orlando and Charlotte in the U.S and Muelheim in Germany). We found that the network bandwidth, which we hope will improve, was the only bottleneck. It works exceptionally well and as planned, a testimony to Tcl's claim as a glue language. We use xess spreadsheets [Ais99] for some of our methods. There is a Tcl interface for the xess spreadsheets that we use to communicate between the spreadsheets and the SID environment in the unix platform. We communicate with Microsoft Excel spreadsheets on NT using the package tcom that exposes COM interfaces. We use a Tcl implementation of SMTP that we found in the newsgroup for all our mail purposes within the system. It works across platforms and is reliable.

4.6 Cross Platform Deployment

We recently started porting the SID environment to NT after all of the required extensions were available. Surprisingly, there were few coding changes required to bring up the environment. All GUI components worked extremely well on the first attempt. Early in the development, we made a conscious effort to remove as many `exec` commands with in the Tcl code. The majority of changes were related to file permissions and sharing between UNIX and NT. This problem was tackled by using SAMBA and by writing a small File Agent that handles most file duties on behalf of the user. This gave us an added benefit, the File Agent becomes the owner of all files in the repository thereby restricting access by general users.

We did notice some reduced performance when creating large input screens running on a Pentium II Windows machine. Also, some of our input forms would not display properly unless we added a few well placed `update` commands.

5. Conclusion

Our experience has been that we're able to add any desired feature into SID with Tcl. Most of our design was incremental and Tcl provided the flexibility to meet our goals. We could create prototypes quickly and incorporate them into the environment. Relying on many extensions can be a setback at times, especially when migrating to new versions of Tcl. But it certainly was not a showstopper since source code was freely available. SID was started with version 7.3 and now runs under 8.0.4. If we were to embark on another project of this scale today, it's our opinion that we would use Tcl again as opposed to the current trend towards JAVA.

Acknowledgements

We thank the open source community, the various tcl extension authors and Scriptics, who laid the foundations on which we could build a successful project of this magnitude.

References

[Ais99]	Applied Information Systems home page http://www.ais.com Valid as of 09/01/1999.
[Angel98]	Angelovich, Kenny and Sarachan, "NBCs Genesis Broadcast Automation System: From Prototype to Production", *Proc. Sixth Annual Tcl/Tk Conference*, pp. 1-9, San Diego, Calif.:USENIX, 1998.
[DeCl97]	De Clarke, "Dashboard: A Knowledge-Based Real-Time Control Panel", *Proc. Fifth Annual Tcl/Tk Workshop*, pp. 9-18, Boston, Mass.:USENIX, 1997.
[Hugh99]	The mSQL Home Page, http://www.hughes.com.au Valid as of 09/01/1999.
[PL99]	The PLPlot Home Page, http://emma.la.asu.edu/plplot Valid as of 09/01/1999.
[RL98]	Reekie H.J. and Lee E.A, "The Tycho Slate: Complex Drawing and Editing in Tcl/Tk", *Proc. Sixth Annual Tcl/Tk Conference*, pp. 37-46, San Diego, Calif.:USENIX, 1998.
[WeFr97]	Webster T. and Francis A., "Tcl/Tk for Dummies", pp. 324, IDG Books, 1997.

[Wolf98] Hans K.Wolf, "Java, CORBA, and Architecture", *Component Strategies*, September 1998, pp. 58-64.