A Tcl-based Self-configuring Embedded System Debugger

Dale Parson, Paul Beatty and Bryan Schlieder
Bell Labs Innovations for Lucent Technologies

A Tcl-based Self-configuring Embedded System Debugger

Dale Parson, Paul Beatty and Bryan Schlieder (dparson@lucent.com)

Bell Labs Innovations for Lucent Technologies

Abstract

The Tcl Environment for Extensible Modeling is a software system from Bell Labs for the simulation, hardware emulation and debugging of heterogeneous multiprocessor embedded systems. These embedded systems contain one or more digital signal processors or microcontrollers that execute real-time software written in assembly language and C. Tcl provides an environment in which embedded system designers can interact easily with their designs. Tcl serves as a processor query language, a modeling language for connecting and scheduling processors, an extension language for adding both model and environment enhancements, and as a user interface implementation language. Tcl's C API and calling conventions provide C and C++-level standards and portable libraries. The Tcl interpreter extends readily into a self-configuring simulation-emulation-debugging tool set. This tool set can use new processor types and new processor arithmetic without tool recompilation. This paper looks at exploitation of Tcl from a system perspective, and at some technical problems and solutions in applying Tcl.

Introduction

Our group in Bell Labs builds software generation and execution tools—compilers, assemblers, linkers, simulators, hardware emulators and debuggers—for a variety of Lucent Technologies embedded digital signal processors (DSPs). Some processors vary at core architectural levels, while other processors differ only with respect to I/O circuitry or memory configuration.

Tcl [1] provides a means for separating processor-specific details from the debugging environment of our simulation and emulation tools. Our Tcl Environment for Extensible Modeling (TEEM) couples a Tcl/Tk-based user interface to a C++ queryable model technology. TEEM configures itself at startup time to support a user-specified set of processors. It queries its processor models to determine processor-specific signals, registers, I/O and memory configurations, and debugger arithmetic semantics.

Tcl finds productive application throughout our environment. From large-scale architectural issues to localized implementation considerations, Tcl provides structure and code that considerably enhances the power and maintainability of our tools.

Section 1 describes where TEEM fits into our system for development of embedded software. Section 2 discusses how Tcl is connected to processor models to support embedded simulation. Section 3 shows how hardware emulation fits into the system. Section 4 discusses the multi-process TEEM graphical interface and our strategy for minimizing application knowledge and communication overhead. Section 5 shows how TEEM may be coupled into third-party environments as a coroutine. Section 6 summarizes our discoveries.

1. Embedded system software generation and execution

Figure 1 is a bird's-eye view of Lucent's Wireless and Multimedia software tools for embedded system programming and debugging. A compiler or assembler translates source code into an object format that a linker binds into an executable file. Compilers for different processors can produce output with different object file formats (COFF, ELF, etc.). Executable file format is one processor-specific system parameter.

The software generation system of Figure 1 is processor-core-specific. TEEM, on the other hand, grows to handle execution of different processor types. TEEM is at the heart of the embedded execution system of Figure 1.

TEEM is *tclsh* (Tcl shell) extended with processor modeling commands from four categories listed below. Tcl procedures can extend each category.

- **Model Management**: The **pssr** command queries available model types, constructs one or more processor instances and deletes model instances. Tcl callbacks that run when a processor is created provide a mechanism for loading and setting up the processor.
- **Model Access**: These commands initialize, examine and modify model instance state (registers, memory, buses and pins). Tcl procedures extend basic query mechanisms to provide information in an application-oriented format.
- **Model Control**: Execution and breakpoint commands drive simulation at the C++ level until breakpoints or exceptions occur. Tcl breakpoint callback procedures can copy data between registers and memory of different processors to model interconnection. Callbacks can also resume processor execution to

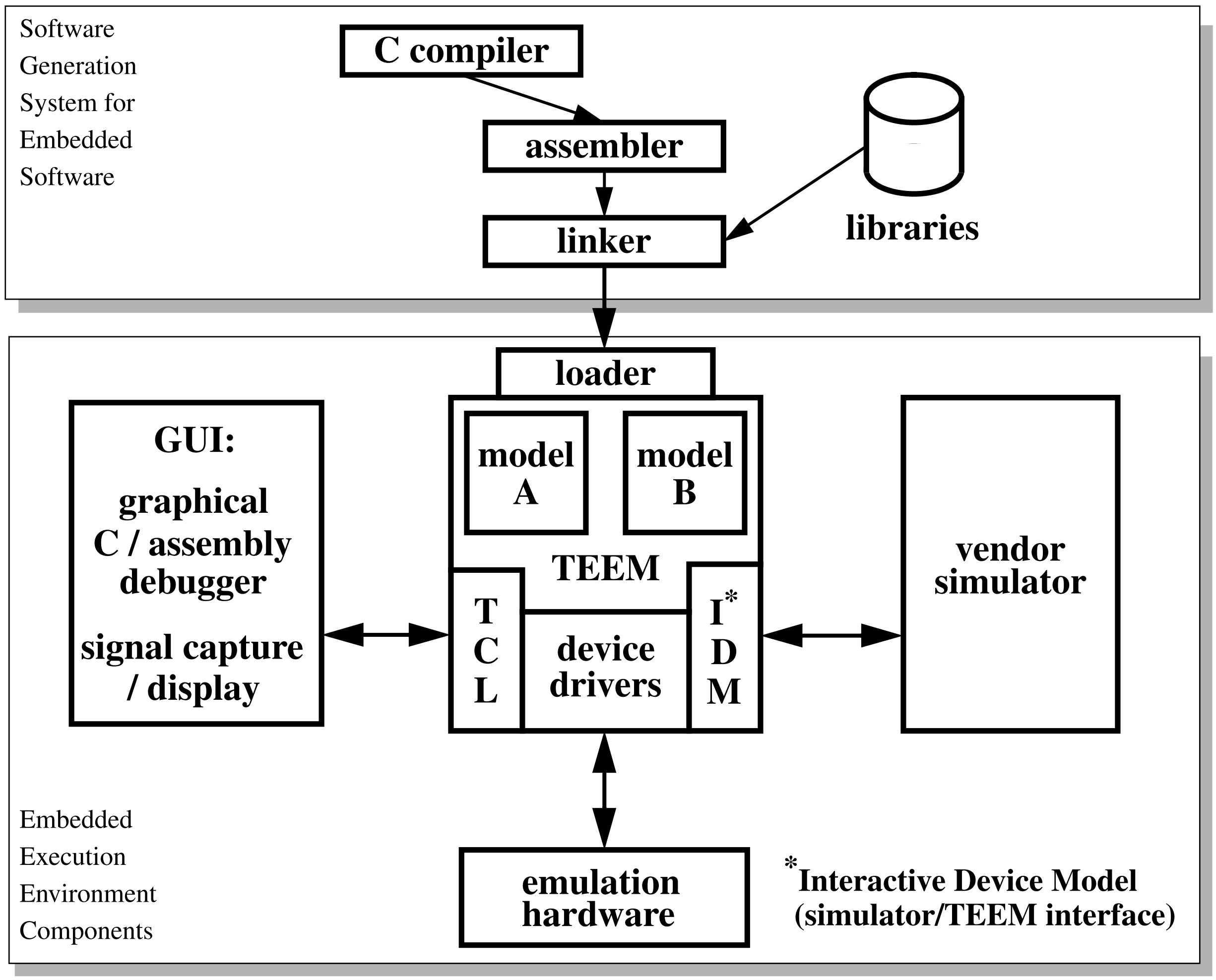

Figure 1: Embedded system software generation and execution tools

model scheduling in multiprocessor systems.

- **Model Input / Output**: These commands attach files or Tcl callback procedures to I/O activity. Tcl callbacks support simulation of I/O activity by sourcing and sinking I/O signals between data streams and modeled embedded system I/O.

TEEM operates as a stand-alone tclsh or in conjunction with one or more of the other components shown in Figure 1. In stand-alone mode TEEM supports interaction with an embedded system based on simulation models or emulation hardware. Emulation hardware can include processor evaluation cards or processors embedded in a user's prototype system. When TEEM emulates a processor via hardware, a model instance serves as a database for buffering processor state. Processor execution downloads model state to hardware when execution begins and uploads state to the model upon reaching a breakpoint. Section 3 discusses integration of hardware emulation into TEEM.

TEEM's Tcl command interpreter provides an ideal environment for batch execution, procedural extension and regression testing. However, textual commands give a low-bandwidth user interface, so typical interactive usage requires a graphical user interface (GUI). Instead of including GUI code, TEEM provides a generic socket interface to allow a client application to submit Tcl queries. The GUI process retrieves and updates model state via remote Tcl procedure calls. This optional C / assembly debugger and related graphical signal capture and display tool are discussed in section 4.

In addition to the stand-alone version, TEEM and its GUI may connect to vendor simulators via the IDM interface shown in Figure 1. Typical embedded systems contain analog components and may contain digital components that are not modeled in the TEEM environment. The integration of a vendor simulator such as Model Technology's VHDL circuit simulator or Math Works' Simulink® arithmetic function simulator

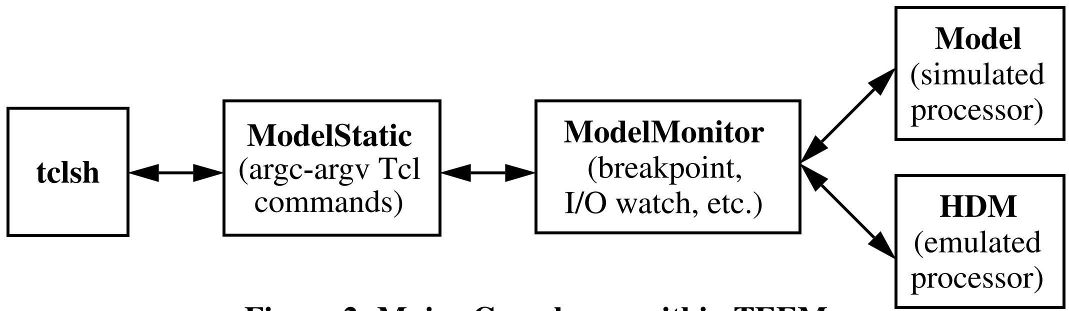

Figure 2: Major C++ classes within TEEM

provides a means for using TEEM models and TEEM debugging functionality in larger systems. While TEEM remains available for processor query and debugging, the external simulator supplants TEEM's built-in model driving functions. Section 5 discusses how TEEM is linked to a host simulator via the simulator's functional API.

2. Tcl Environment for Extensible Modeling = tclsh + processor models

2.1 Tcl-to-model interface classes

Figure 2 shows TEEM's internal structure. Arrows show inter-class communication paths between four C++ classes and the extended *tclsh*. ModelStatic is a Tcl-to-modeling interface class composed strictly of static functions and class-static variables. Each built-in modeling command takes the form of a ModelStatic function that complies with Tcl_CreateCommand's parameter conventions. ModelStatic uses no "this" object or inheritance. Its class variables manage the set of models.

Model is a processor modeling base class that runs simulated circuitry. Each Model-derived class adds processor-specific objects such as registers and I/O ports for storing state. It also adds simulation code for advancing processor state on clock transitions.

The hardware development mode (HDM) class provides access to a real, running processor. It retrieves processor-dependent configuration information from its Model. An HDM object also uses its Model object as a database that stores its processor state information.

Each Model object and HDM object ties to one ModelMonitor object. ModelMonitor is a base class with one derived class per processor core. ModelMonitor performs debugging-oriented monitor tasks that are not part of processor simulation or emulation. Key ModelMonitor jobs include stepping the processor, watching for breakpoint conditions in the Model, directing return to Tcl on uncaught exceptions, and directing callbacks to Tcl on caught exceptions. Both calls forward from tclsh to a ModelMonitor instance and callbacks from ModelMonitor to tclsh travel via class-static functions in ModelStatic.

2.2 Model management

Section 1 listed four categories of ModelStatic commands that manage a set of processors, query/update a processor, run a processor and simulate processor I/O. An important goal of TEEM is to enable debugging of multiple processor instances. Tcl's ability to support object-action command structure enables almost seamless transition of scripts from single to multiple processor debug environments.

The **pssr** model management command can create a processor instance (**pssr new** *modelName*), delete a processor instance (**pssr delete** *instanceName*), and query the set of available processors (**pssr modelTypes**).

Pssr gives access to a dynamically linked library of ModelMonitor C++ class constructors. Each constructed object returns a unique *instanceName* to Tcl. The *instanceName* becomes a Tcl command until the object is destroyed. When *instanceName* is called as a Tcl function, it saves a ModelStatic pointer to the *current processor instance* ModelMonitor object on the C++ stack, copies its own *instanceName*'s address into this *current instance* pointer, passes the remainder of the command to Tcl_Eval(), and restores the previous *current instance* pointer from the C++ stack on return. The next section explores the utility of this dynamic *current instance* pointer.

2.3 Model access

2.3.1 Fxpr: processor-oriented expressions

Assume **pssr new mydsp** creates model instances named "p1" and "p2" on two successive calls. TEEM changes *current instance* twice to interpret the expression:

expr "[p1 fxpr r0] == [p2 fxpr r1]"

First TEEM sets current instance to p1 and calls ModelStatic's **fxpr** with register name r0 to retrieve the r0 value from the *first* instance of mydsp. Next TEEM sets current instance to p2 and calls fxpr with register name r1 to retrieve the r1 value from the *second* instance of mydsp. Finally, the results of both fxpr queries pass to Tcl's expr to compare the results.

Fxpr accesses a target ModelMonitor using ModelStatic's current instance pointer. Fxpr is syntactically compatible with expr, but its arithmetic semantics are determined by the current processor's ModelMonitor. For example a fixed-point digital signal processor supplies fixed-point semantics, while a floating-point microcontroller supplies floating-point semantics. Fxpr builds and evaluates a processor-neutral parse tree. Parse tree evaluation consults virtual functions in the current ModelMonitor to evaluate numeric constants, model state references (e.g., register r0), variables and arithmetic operations.

Fxpr's use of the current instance pointer typifies the remainder of ModelStatic functions. The processor query, processor execution and I/O attachment commands interact with a model instance whose identity is determined from the current instance pointer. Therefore any remaining ModelStatic function can be prefixed by an *instanceName*. Entering *instanceName* without a command suffix makes *instanceName* the default current instance for all commands that lack an *instanceName* prefix.

Tcl programmers can write processor-neutral model manipulation scripts that do not specify *instanceName*. Their procedures operate on the current processor, using its model-specified semantics transparently. Alternatively, these programmers can prefix specific commands with an *instanceName*. Application of *instanceName* has stacked, dynamic extent. A major advantage of choosing this object-action convention over the alternative action-object convention [reference 1, section 28.3] is that scripts and adjunct tools can prefix an entire set of commands with object *instanceName*. Tcl *eval*s the remainder of a prefixed command within *instanceName*'s scope as part of the *instanceName* command. The trailing commands affect the intended processor even though they contain no processor-specification code. The action-object convention would require each Tcl action command to specify its processor, making multi-processor generalization of Tcl commands very difficult to achieve.

Fxpr syntax is a superset of expr. Fxpr adds an assignment operator for copying a value into model state. Model state includes registers, I/O ports and other model signals housed in Model, as well as user-declared signal variables housed in ModelMonitor. Fxpr also has memory vector operations for loading and copying sequences of model memory contents within and between processor Models.

Fxpr provides partial compatibility with an earlier, non-Tcl command line debugger. Users are accustomed to typing expressions such as "r0 = r1 + 3" into the debugger. We did not wish to require that "fxpr" be prefixed to every register/signal access and update (e.g., "fxpr r0 = r1 + 3"), so fxpr is linked to Tcl's **unknown** command. Unknown procedures (such as "r0") call the fxpr parser to determine if the leading token is a symbol for a unit of model state in the current instance. If a valid symbol is recognized, the entire command passes to the fxpr parser, otherwise Tcl's default unknown handler is called. Tcl scripts typically make the "fxpr" prefix explicit to speed processing and eliminate any potential ambiguity.

2.3.2 Other query / update commands

There are several other query commands. The **?** command includes options for determining the names, types, and properties of user-accessible elements within a model. Example names are "r0," "pc," "time." Example types are register, I/O port, memory block, user-declared signal variable. Example properties are signal width and memory width, allocation size and base address. The **width** command retrieves signal and memory width. The **alloc** command includes options for determining embedded memory configuration and for allocating memory for simulation or emulation. A user or TEEM client process can use **?** to find out what objects are inside a processor model. The user or client can then use **fxpr**, **width** and **alloc** to retrieve and update the state of those objects.

Each user command that creates a unit of debugger state returns an instance-handle string to Tcl that identifies that unit of state. Name-to-element bindings are unique within a Model. Each object of class Model includes a C++ symbol table that binds each Model-unique name to an element's type and its defining object within the Model. Each ModelMonitor object houses several symbol tables for administering breakpoints, exceptions, I/O connections, and other user-defined debugging state. In addition to Model state, all ModelMonitor debugger state—the state of breakpoint triggers, user-installed breakpoint and exception callbacks, and the state of Model I/O monitors—are available for Tcl-based query. The accessibility of declarative information about Model and debugger state combines with the interpretive nature of Tcl to support very powerful methods for extension and customization.

Tcl's hash tables and string library functions make provision of platform-independent symbol table objects a simple exercise. TEEM runs on several UNIX® platforms as well as Win32™. At the C++ level TEEM uses only ANSI C libraries and the Tcl library to achieve portability. A considerable portion of the queryable model infrastructure works on top of the Tcl C library.

Query commands determine both identity and content of state-bearing elements in a processor. Query results return Tcl strings or lists. ModelStatic command functions, Tcl extensions, and ancillary tools that interact with models can avoid hard-coded processor specifics.

The **mload** command demonstrates processor independence. Mload loads model memory from an executable file. Recall from Section 1 that different processor cores use different executable file formats. Different compilers or assemblers may pass different debugging information. Within the execution environment ModelMonitor integrates over these variations in file structure. Each core-specific ModelMonitor codes for its processor's executable file structure in a virtual mload helper function. The mload command loads Model memory and ModelMonitor debugging tables for any TEEM processor.

Processor-neutral syntax and processor-interpreted semantics extend naturally to C language expression handling for debugging. We are investigating a model-directed approach to C debugging comparable to the processor-independent techniques of ldb [2] and cdb [3]. These debuggers use processor-independent symbol table information compiled into the executable program to achieve processor independence. A TEEM-based approach houses similar symbol table information in its queryable models and ModelMonitor loaders.

2.4 Model control

ModelStatic control commands start and stop execution. Tcl callback procedures enable multiprocessor scheduling.

The **reset** command resets the current processor's state, and the **step** [**n**] and **resume** commands advance its state. Model-ModelMonitor pairs cooperate in advancing processor state and monitoring breakpoint and exception status. Step or resume returns processor halt status to Tcl only when an uncaught breakpoint or exception arises in a processor.

The user can set breakpoints on program locations, assorted program-memory interactions, or on the successful, non-zero evaluation of any fxpr expression within a model. Some processor models may augment the default set of breakpoint types, and hardware emulation may restrict available types of breakpoints. A Model may also assert a variety of exception conditions of four severities—note, warning, error and fatal. The list of exceptions is available for query from Tcl. Default processing of a breakpoint successfully returns a breakpoint identifier string (and TCL_OK) to Tcl. Default exception processing prints messages via **puts**, and errors and fatal exceptions return failure diagnostics (and TCL_ERROR) to Tcl.

A Tcl callback procedure name may be supplied as a handler for a breakpoint or exception. An empty string signifying "ignore" may be supplied for any non-fatal exception. When a handled breakpoint or exception arises during processor execution, ModelMonitor calls Tcl_Eval() (via ModelStatic), passing the Tcl callback procedure name and event-identifying parameters that the callback uses as event keys. The current instance is set to the interrupted processor, and the callback procedure has access to the full range of Tcl commands for passing information between processors and reading and writing files and other data streams. If the callback does not call **step** or **resume**, ModelMonitor returns to Tcl upon completion. If the callback does call **step** or **resume**, ModelMonitor resumes execution of the processor after the callback completes. Only fatal errors force a break.

Breakpoints and exceptions can trigger user-defined extensions to a processor model, transparent simulation of processor I/O events, processor state logging and multiprocessor synchronization. A processor scheduler can be as simple as the following Tcl loop:

```
proc sched {pssrlist} {
    # pssrlist is a list of processor instances
    while 1 {
        foreach p $pssrlist {
            $p resume } } }
```

Sched schedules simulation of the processors named in $pssrlist in round-robin order. Each processor resumes execution until it reaches a breakpoint, at which time *sched* schedules the next processor. Within the action of "$p resume" a breakpoint handler can pass information between processors; the handler has the option of resuming its processor without letting control return out to *sched*.

As written above *sched*'s outer loop iterates until an error occurs within one of the "$p resume" actions. Any such error propagates TCL_ERROR out of *sched* in the normal Tcl manner.

2.5 Model I/O

Model and ModelMonitor support attachment of any Model I/O port to a text file. Alternatively Model-level I/O operations may call a user-specified Tcl procedure. A processor input action from an I/O port can cause a call to a Tcl procedure that returns a value for that port. A processor output action to an I/O port can cause a call to a Tcl procedure that takes the output value as an argument. The combination of breakpoint, exception and I/O event callback procedures allows the user to design fully customized, event-driven, multiprocessor simulations within Tcl. The execution of the callback procedures is encapsulated as part of Model execution.

In fact, a processor designer can prototype a model in Tcl by writing only three C++ virtual Model functions as callbacks to Tcl. We normally write final processor Models in C++ for speed, but we have implemented partial Models using Tcl to allow concurrent engineering of the Models. For example, we have written a partial Model that houses only memory in order to test the loader of its companion ModelMonitor class. Unrelated Model functions are stubbed out by binding them to Tcl callback procedures. Using Tcl callbacks as stubs to support incremental construction of partial Models has been very useful.

3. Emulation access to processor hardware

The HDM class shown in Figure 2 provides communications between TEEM and each target processor through an IEEE 1149.1-compliant "JTAG" serial interface. HDM control of JTAG goes through a PC ISA bus card; optional networked access is via a TCP/IP socket interface. The HDM class hides physical communication details by providing a set of generic block transfer functions such as memory upload/download, register upload/download, processor reset, step and resume, and event monitoring. Plans for more sophisticated controller-based PCI and PC Card interfaces require that physical implementation details be encapsulated in the HDM class hierarchy.

Tcl-queryable models again play an important role. An HDM object determines processor details by querying its Model at startup. With emulation enabled, each HDM object intercepts calls to its corresponding Model, delegating non-emulation work back to the Model. Tcl continues to interact with processor state by querying and updating a Model. HDM uses the Model as a database for storing processor state.

4. Relational Tk mega-widgets and database-event-driven graphical update

The graphical C / assembly debugger of Figure 1 is a good demonstration of the expressive power of Tcl/Tk. In about 5000 lines of Tcl code it replaced a C-based approach to an earlier debugger that had about 45,000 lines of C GUI code. The Tcl/Tk GUI has far greater functionality thanks to Tcl as a query language. The Tcl/Tk debugger has a processor source window and command line history and search processing that use straightforward application of Tcl programming and Tk library widgets. The most significant custom savings for Tk came about through the construction of a relational mega-widget that leverages the queryable nature of TEEM. Any queryable data set in TEEM—i.e., the entire model and debugger state—can be displayed in the familiar row-column format of a relational database. We constructed an instantiable relational mega-widget in about 1300 lines of Tcl/Tk. The mega-widget's constructor includes parameters for domain names and associated properties, including formatting and editing callback procedures. User editing of fields and complete tuples is parameter-driven. Mega-widget instances were useful in building display / interaction windows for the following TEEM element types: registers, pins / buses, signal variables, breakpoints, memory vectors, I/O connections, and a spreadsheet-like watch window that can use user Tcl procedures to create customized views into model data sets. Figure 3 gives an abbreviated look at a register window and a relational watch window.

Performance was an issue of concern at the outset of GUI design, especially for PCs. Our solution was to minimize GUI-TEEM communication by doing as much work as possible within TEEM. Tcl "watchdog" procedures query active models periodically within the TEEM process. These procedures send display update messages to the GUI process only when widget contents need to change. This approach eliminates system call and inter-process communication overhead that would be required by GUI-driven model query. This approach makes good use of the interpreted nature of Tcl. No knowledge of the GUI client process is coded into the TEEM server process, but the GUI can source client Tcl "watchdog" code into the TEEM process for efficiency. Efficiency is gained without a loss in modularity.

5. Host simulators and Tcl as a coroutine

5.1 Simulator / TEEM interface

An embedded system designer may wish to model and debug one or more of our processors in a vendor simulation environment in order to gain access to tool capabilities, circuits or arithmetic functions modeled in that environment. A typical simulator architecture requires the user to start a simulator process and specify models as data. The simulator then loads models from

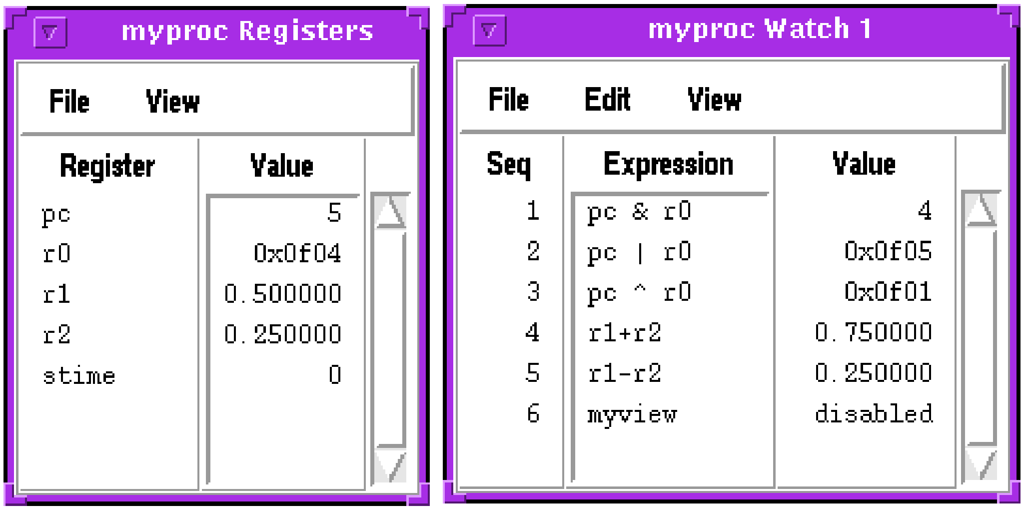

Figure 3: Relational watch window gives spreadsheet-like views into a model

object files via a dynamic, incremental loader. The models must supply certain access functions specified by the simulator as part of its C application procedural interface (API). The models may call C simulator functions that are also part of the C API.

The Interactive Device Model (IDM) interface of Figure 1 is part of ModelMonitor of Figure 2. It achieves a great deal of leverage from both Tcl and the queryable nature of Model. When a vendor simulator initializes its first TEEM Model after the incremental load of TEEM, ModelMonitor starts tclsh and constructs the communication paths of Figure 2. Thereafter ModelMonitor knits each additional Model into tclsh.

Previous IDM design required a great deal of processor-specific hand code in order to integrate a new processor into a given simulator. At about 800 lines of code per processor variant x 5 variants per year x 2 simulators, we were averaging around 8000 lines of processor-specific IDM code per year, and were limited in the number of additional processors and simulators we could support. With TEEM the IDM can query both the models and the simulator environments, and the number of lines for new processor variants drops to a constant 0. Any processor modeled in TEEM is immediately available to the IDM interface. A new simulator takes a constant of about 100 lines to integrate with TEEM.

5.2 Tcl as a coroutine

Figure 4 illustrates the control problem we encountered when attempting to integrate Tcl as an incrementally loaded subroutine beneath a host simulator. Arrows represent subroutine calls. The first call to a TEEM model initializer calls Tcl_Eval as part of tclsh. Suppose in the process of initializing TEEM, Tcl_Eval sources a commands file that calls Model **step** or **resume**. The external simulator requires that a model initializer or evaluation function must return to the simulator to advance the simulation, i.e., to bring about the **step** or **resume**. Unfortunately if Tcl_Eval returns, the context in which Tcl was sourcing its file will be lost. Tcl cannot simply return to the host simulator.

In this environment Tcl must run not as a subroutine, but as a coroutine peer of the host simulator. With a coroutine organization the Tcl thread has its own execution stack. During Tcl execution this thread is active and the simulator thread is blocked. The Tcl thread can access Models and ModelMonitors, performing queries and other model interactions. Only when Tcl calls **step** or **resume** is it necessary to block the Tcl thread and resume the simulator thread. Step and resume require coroutine resumption logic when run under a host simulator.

Lightweight threads under UNIX® and Win32™ should be able to supply the coroutine mechanism. Coroutines implemented using lightweight threads do not violate Tcl's single-thread limitation, since only one thread is active within Tcl at any given time. TEEM initialization starts a second thread for tclsh and yields control to that thread. ModelStatic's step / resume manages thread execution; synchronization code blocks the simulator thread in order to return from **step** or **resume** to Tcl, and it blocks Tcl in order to initiate a **step** or **resume** within

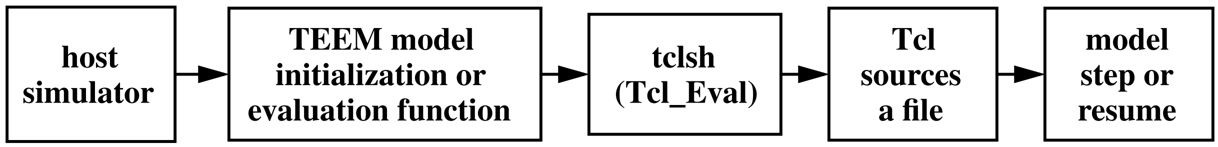

Figure 4: External simulator to model calling stack

the simulator. This scheme worked fine with Tcl and some toy applications, and it worked with Model Technology's VHDL simulator. Unfortunately another simulator not named in this report had mysterious failures when we used lightweight threads. Because we have no means to debug this simulator-specific problem, we switched to a safer but more cumbersome multiprocess approach to coroutines. The blocking threads remain a viable option for cooperative applications, but the external simulator did not cooperate with us.

Our multiprocess solution runs the host simulator and tclsh in separate process spaces. It distributes the class-static functions of class ModelStatic of Figure 2 over an inter-process communication (IPC) medium, and places ModelStatic's data in the appropriate processes. These functions all use the Tcl_CreateCommand-compliant argc-argv calling convention. Consequently distributing these functions across IPC required engineering only one distributed function. The tclsh process houses the main Tcl interpreter, but the simulator process also houses a Tcl interpreter used for decoding and calling the correct ModelStatic functions received via IPC. All user-initiated Tcl interpretation occurs in tclsh, and all ModelMonitor and Model objects reside in the simulator process. Tcl callbacks reverse the order of IPC remote call, calling from ModelMonitor to tclsh. We use an IPC-medium-independent C++ class we already had in hand to transport the remote calls and data, binding it to sockets.

6. Conclusion

Tcl plays many important roles in the TEEM processor modeling environment: 1) Tcl is the model query language that the debugger and ancillary tools use to retrieve processor configuration data. 2) Tcl is a modeling language that supports interconnection of processor instances and prototyping of connecting circuitry. 3) Tcl supports model extension and error handling through breakpoint-triggered, exception-triggered and I/O-triggered callbacks. 4) The Tcl C API and calling convention provide the local and remote procedure calling standard for C and C++-based system components. 5) The Tcl C library provides portable library functions for all required machine/OS platforms. 6) Tcl's well-defined representation of success and error status and return value from a procedure extends readily to a clear notion of a transaction between a tool and a queryable model. Delimited transactions achieve tool synchronization. 7) Each queryable model connects easily to a set of Tk relational mega-widgets that reflect model contents in tabular form. Event-driven widget update, where the change of a widget-displayed datum constitutes an event, minimizes inter-process communication overhead between the modeling and graphical debugger processes. Tcl as the query language supports efficient model event detection in the modeling process. 8) Tcl can integrate as a coroutine into external simulators, providing TEEM capabilities in many contexts.

7. References

1. John K. Ousterhout, *Tcl and the Tk Toolkit*. Reading, Ma.: Addison-Wesley, 1994.

2. Norman Ramsey and David R. Hanson, "A Retargetable Debugger," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 27(7), 22-31 (1992).

3. David R. Hanson and Mukund Raghavachari, "A Machine-Independent Debugger," *Software—Practice and Experience*, Vol. 26(11), 1277-1299 (November, 1996).