

Extension Language Automation of Embedded System Debugging

Dale Parson, Bryan Schlieder and Paul Beatty

This extended version of the paper from AADebug'2000 (<http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html>) appears in *Automated Software Engineering* 9(1) (January, 2002), pp. 7-39.

Extension Language Automation of Embedded System Debugging

Dale Parson
dparson@agere.com

Bryan Schlieder
bryanschlieder@agere.com

Paul Beatty
pebeatty@agere.com

All authors: Agere Systems
1247 South Cedar Crest Blvd.
Allentown, Pa. 18103 USA

ACCEPTED SUBMISSION
AUTOMATED SOFTWARE ENGINEERING

Keywords

extension language, embedded system, debugger, reflection, Tcl/Tk

Abstract

Embedded systems contain several layers of target processing abstraction. These layers include electronic circuit, binary machine code, mnemonic assembly code, and high-level procedural and object-oriented abstractions. Physical and temporal constraints and artifacts within physically embedded systems make it impossible for software engineers to operate at a single layer of processor abstraction. The Luxdbg embedded system debugger exposes these layers to debugger users, and it adds an additional layer, the extension language layer, that allows users to extend both the debugger and its target processor capabilities. Tcl is Luxdbg's extension language. Luxdbg users can apply Tcl to automate interactive debugging steps, to redirect and to interconnect target processor input-output facilities, to schedule multiple processor execution, to log and to react to target processing exceptions, to automate target system testing, and to prototype new debugging features. Inclusion of an extension language like Tcl in a debugger promises additional advantages for distributed debugging, where debuggers can pass extension language expressions across computer networks.

1. Introduction

Embedded system debugging involves more conceptual layers of a target system than debugging for time-sharing systems. Consider the case of debugging a C program within a time-sharing system. User-debugger interaction occurs almost entirely at a C language level of abstraction. Descent into assembly language and machine code representations of a target program is rare. Suspicions about a compiler bug may require inspection of generated assembly code. Inadvertent stepping into an optimized library subroutine leads to display of assembly mnemonics and binary numbers. Debugging concurrency problems in multi-threaded programs entails cognizance of time, but well-structured multi-threaded programs limit temporal awareness to a few, isolated regions where multiple threads are allowed to interact. These examples are exceptions, and most programmers can debug their programs exclusively from a source language perspective.

Embedded systems add several dimensions to debugging. Embedded systems include programmable physical devices that have no direct language counterparts at higher levels of abstraction. Their programming requires direct manipulation of registers and state machines. Assembly language programming is common for performance-critical modules. Temporal determinacy is fundamental to a real-time embedded system, eliminating the possibility of constraining temporal awareness to a few, isolated regions of code. Multiple, heterogeneous processors operating at different levels of abstraction, for example a microcontroller running Java™ or C++ teamed with one or more digital signal processors (DSPs) running a mix of C and assembly code, are commonplace within some classes of embedded systems. Mixing abstractions within design and debugging is typical.

This paper is about the practical application of an additional layer of language abstraction, an

extension language abstraction, to the aforementioned layers of embedded systems. An extension language serves a tool such as a debugger by providing a programming language, typically an interpreted language, for extending tool capabilities via composition at tool usage time. Tool users can automate repetitive tasks and enhance tool features by writing extension language *scripts*. Basic tool capabilities ultimately constrain the power of extensions that users can compose.

Luxdbg, the LUXWORKS tool suite's debugger for embedded systems (Parson, et al., 1997; Parson, et al., 1999), exposes multiple layers of target system abstraction to debugger users and auxiliary tools. Agere Systems (the former Lucent Microelectronics Group) provides a production version of Luxdbg in support of Agere's digital signal processors and mixed microcontroller-DSP systems for embedded telecommunications and related applications (Lucent, 1998). Luxdbg is implemented in C++, and it uses Tcl as its built-in extension language (Welch, 1999). A Java-based research version of this tool explores loading alternative extension languages such as Python (Beazley, 2000) at run time via dynamic class loading (Parson, 1999).

Luxdbg application space has primarily been in the area of embedded telecommunications signal processing, a huge area that is seeing rapid growth in multiprocessing. Cellular basestations — the electronics connected to the towers one sees while traveling along the road — are coming to employ large banks of two- and three-core DSP chips that put hundreds of individual processing cores into a system. Customers designing circuit boards for these systems require a debugger that can connect to about one hundred processors at a time. Not all processors are being debugged at any given time, and many of the signal processing algorithms running on each processor involve only that processor. The DSPs process signals for parallel voice and data channels. Nevertheless, all processors are running at the same time, and any processor can exhibit

a bug at any time, so the debugger must be capable of connecting and interacting with one or more of the one hundred at any time.

At the other end of the cellular continuum, a cellular handset (cell phone) often contains a DSP and a microcontroller. The DSP handles signal processing for the voice and data channels within that handset, while the microcontroller controls the system and provides user level IO capabilities. The handset presents only two processors, but they are heterogeneous, i.e., they provide different programming architectures and instruction sets. They may be programmed in different languages, e.g., assembly language or C for the DSP and C, C++ or Java[™] for the microcontroller. Handset debugging does not require the massive multiprocessor debugging features required by a basestation, but it does require the ability to debug heterogeneous instruction sets and languages.

A detailed account of the signal processing and related control architectures of Luxdbg's target embedded systems is outside the scope of this paper, and indeed it is unnecessary. Most embedded systems consist of an admixture of high level language processes, assembly code for performance-critical tasks, and hardware registers and special-purpose devices for accepting sensory input and producing sensory or sensorimotor output. Communications channels can be considered artificial, special-purpose sensory realms. Most of the programming examples in this paper come from multiprocessor signal processing systems, where a given processor reads an information-bearing signal frame, processes that frame (e.g., extracts information, encodes information, superimposes signal content on a carrier, encrypts, decrypts, removes noise or otherwise filters, etc.), and then sends the resulting frame of data on to the next stage. Human users usually attach at the endpoints of these distributed signal flows. Debugger users attach anywhere a bug surfaces.

This paper constitutes an experience report in effective uses of an extension language within a multiple abstraction embedded system debugger. Section 2 gives an overview of using an extension language within an application. Section 3 examines the layers of embedded system abstraction for which Luxdbg supports debugging. Section 4 surveys the classes of extensions that users can employ in extending Luxdbg and its target embedded processors. Section 5 discusses two strategies for employing an extension language in evolutionary development of Luxdbg. Section 6 discusses related work. Section 7 gives conclusions and directions for upcoming work.

2. Extension language systems

Figure 1 diagrams the interactions of an extension language in a system. The main system components are a *client* such as a user, extension script or auxiliary tool (e.g., GUI), an *extension language interpreter* such as Tcl (Welch, 1999) or Python (van Rossum, 1995), and an underlying *application* such as a debugger. Interaction begins at initialization time, when the application registers a number of primitive functions with the extension language interpreter. The interpreter adds these functions to its native command set.

<<FIGURE 1 GOES ABOUT HERE.>>

After initialization the system enters its main interaction loop. It uses a *command path* to pass client commands to the extension language interpreter. The user may enter commands via a textual or graphical interface, or the user may write and invoke an extension language program (a.k.a. *script*). Built-in extension language primitives include control flow and data structuring constructs. The extension interpreter parses commands and executes scripts by invoking both its own primitives and application primitives. Typical debugger primitives include data retrieval, data modification and execution control for target processing systems.

Figure 1 also shows a *callback path* from the application to the extension language interpreter.

In an event-driven system it is possible for a user to associate an extension language expression with an event in the application layer. For example, in Luxdbg a user can associate a Tcl expression with a target processor breakpoint. When the breakpoint occurs, the debugger calls back to the Tcl interpreter, passing the expression to be evaluated and an identifier for the breakpoint as parameters. The interpreter evaluates the expression and returns the result to the debugger. Expression evaluation may include retrieval and modification of processor state. During a callback the extension language interpreter acts as a servant for the application layer, a reversal of their normal roles. Whereas the command path of Figure 1 allows the extension language to build extensions out of primitives, the callback path allows the extension language to extend primitives. For Luxdbg this means that users can extend the built-in debugging layers discussed in the next section.

Extension languages support automated software engineering in the same way that higher-level languages have always supported automation of lower-level programming, by automating the fine-grain steps of lower-level programming. Just as a FORTRAN or C compiler automates many of the steps of assembly programming, so a Tcl or Python interpreter automates many steps of equivalent FORTRAN or C programming. Construction and manipulation of associative mappings in Tcl, for example, take the form of simple expressions within the programming language, while in C they take the more laborious form of manipulation of C data structures via pointers and library procedure calls. An extension language augmented with application primitives becomes, in effect, a high-level, application-specific language, e.g., a debugging language.

An extension language supports automation of a tool such as debugger in other ways. Users can automate repetitive command invocations by writing extension scripts that invoke those

commands. Interactive automation is assisted by the fact that the extension language is interpreted, as well as the fact that the original repetitive commands are extension language expressions. Users can embed commands within control constructs without a change in language. For example, a Luxdbg user can embed a parameterized repetitive command within a Tcl looping construct. Suppose the user wishes to set a series of breakpoints in certain C functions, every time she loads a new compilation of a target program. Rather than repeatedly typing “stop in f ,” where f represents a function name, she can create a script file (e.g., *stopfile*) with the following lines:

```
foreach f {functionA functionB functionC ...} {  
    stop in $f  
}
```

She then invokes “source stopfile” every time she reloads the target program. A user can create such command constructs interactively, without loading a file or defining a procedure, when those constructs are needed only once.

Complexities of script files increase as users gain experience. They begin to create extension procedure libraries with parameterized procedures and perhaps object classes. Extension languages that have access to graphical libraries such as Tcl’s Tk GUI tool kit allow graphical extension, e.g., user-defined buttons for stereotyped commands or graphical animation of application monitoring.

Eventually this increase in script complexity brings us to a third form of automation, in addition to programming language automation and repetitive command automation, that of tool feature prototyping. Extension languages are environments for experimenting with new tool features. Both users and tool developers can prototype new capabilities, using the extension language to combine existing capabilities with those of imported libraries. Prototypes become

stable features via consistent usage. Some may be rewritten as compiled application code if efficiency is important, but many features remain useful in their original, extension language form. An extension language supports incremental, layered extension of basic tools features, thereby automating portions of the tool development process. Without an extension language, feature development consists of paper specification followed by conventional coding. With an extension language, feature development consists of interactive explorations of feature space.

3. Luxdbg layers of processor abstraction

Figure 2 illustrates the layers of abstraction available to a Luxdbg user. The *extension language processor layer* of Figure 2 is the extension language interpreter of Figure 1, augmented with Luxdbg debugging primitives. The remaining layers of Figure 2 constitute the application module of Figure 1. Each layer provides a C++ API that allows outer layers to build upon it. Luxdbg supports concurrent debugging of multiple, heterogeneous *virtual processors*, where each virtual processor is a hardware processor, a processor simulation model, or an operating system process that implements the layers of processing abstraction of Figure 2. Luxdbg represents each target processor as a C++ object with these layers. Each object includes public methods for access, modification, and execution control at these layers.

<<FIGURE 2 GOES ABOUT HERE.>>

The *circuit layer* represents integrated circuit pins, processor registers, memory regions, peripheral devices and timing information. This is the most physical layer, closest to the hardware. Embedded system programming entails access to devices such as digital-to-analog converters that control physical devices (e.g., speakers, heaters or motors), analog-to-digital converters that allow the system to monitor sensors (e.g., microphones or temperature sensors), coder-decoders (*codecs*) that translate signals between computational and communications-

oriented representations, and other devices that manipulate physical signals in a variety of ways. An embedded processor interacts with physical devices via registers that it retrieves and modifies using dedicated processor register slots and dedicated IO instructions, or by using memory-mapped registers, conventional fetch-store instructions and direct memory access (DMA). Embedded devices house state for a program that is outside the bounds of memory-oriented program variable state. An embedded system debugger must provide access to this state.

The circuit layer in Luxdbg may be embodied by C++ circuit modeling objects such as memory or register models in a processor simulation model. A simulation model allows users to simulate execution of the instructions and IO operations of an embedded processor system before hardware is available. A simulation model supports debugger inspection of target system internal state and target system timing that may be inaccessible in a real, hardware system because it is hidden inside of a complex integrated circuit. The circuit layer may also be embodied by electronic circuits in a hardware processor. Hardware target systems typically provide dedicated debugger access pins and registers (hardware monitors) as well as target-resident debugger access library routines (software monitors) that allow a debugger to monitor and to control program execution and target system state. Luxdbg users can interactively read and write circuit registers, memory, and other scalar and vector values using Luxdbg primitives.

Currently all of the simulation models and hardware processor interface drivers used by Luxdbg are written by Luxdbg software engineers and processor architects who work closely together. However, Luxdbg loads target models and drivers as dynamic link libraries (DLLs), and we have plans to support our target processor API so that processor designers can attach new targets to the debugger without direct involvement of Luxdbg engineers.

The *machine code layer* adds the concepts of an *instruction stream* and a *system clock*.

Artifacts such as an instruction pointer (a.k.a. program counter), program memory, hardware interrupts and breakpoints become evident at this level. This layer consists of a programmable processor and the state contained in its circuit layer of abstraction, devoid of symbolic debugging information supplied by compilers, assemblers and linkers. With the machine code layer come Luxdbg primitives for allocating program memory and for determining execution location, primitives for setting and clearing breakpoints at machine program and data addresses, primitives for specifying breakpoint and processor exception handlers, and primitives for resetting, resuming, and interrupting program execution.

System clock-based synchronization of multiple processor cores on a single chip also comes as part of the machine code layer. Timing is an important element of real-time embedded systems. Modern telecommunications embedded systems often employ multiple processors within a single silicon *system on a chip* (SoC). Each SoC houses multiple machine code processors. A SoC simulation model distributes a system-level clock that maintains precise timing relationships among contained processor cores. SoC hardware provides debugger access to execution cycles and instruction counts. Luxdbg can gain access to sub-instruction timing granularity, for cases of debugging interrupt latency or side effects of a visible instruction pipeline, when working with simulation models that represent these precise degrees of timing granularity.

While the previous two layers reside in a target processing system that is being debugged, the remaining layers of Luxdbg reside in the debugger.

The *assembly code layer* adds symbolic interpretation on top of programs running within the machine code abstraction. This is the first layer to relate binary run-time information to build-time source information. Unlike time-sharing systems, embedded systems typically do not carry much symbolic, source code information in the run-time environment. Often the run-time system lacks

a loader; programs then reside in ROM. Luxdbg's assembly layer adds a loader, symbol resolver and assembly expression evaluator to each machine being debugged. Primitives at this layer translate program symbolic names to machine code layer memory addresses and contents.

The *procedural code layer* is a more powerful variant of the assembly code layer, adding constructs such as stack frames, data structures and objects that come with source languages such as C and C++. Both the assembly and procedural layers map user-supplied commands that are specified in source code terms to machine addresses and binary values. A C procedural layer, for example, maps a reference to a local C variable to a memory offset from a processor stack pointer in the machine code layer. These layers also map machine code event parameters such as breakpoint addresses up to source code terms such as source file-line number pairs and data structure member names, types and values.

The *extension language layer* is the home of the extension language interpreter. The command set of the extension language layer includes extension language operations and Luxdbg primitives for the inner layers. By supplying primitives from all of these layers to the extension language layer, Luxdbg gives users and scripts access to several perspectives of a target processor. A target embedded processor is simultaneously a collection of embedded circuits, machine registers and memory locations, symbolic data structures and executable functions, and extension language operations.

There are four categories of Luxdbg Tcl primitives.

Processor management primitives allow Luxdbg to connect to a set of processors. These primitives include:

- primitives to query the set of available processor simulation models,
- primitives to query hardware debug servers for connected hardware processors,

- primitives to construct a C++ processor model or to reserve a hardware debug connection, and to connect Luxdbg to one or more of these target processors,
- primitives to launch processes and to attach to existing processes within an operating system, and
- primitives to disconnect Luxdbg from target processors and processes.

Process access primitives allow Luxdbg users to read and write target processor state at all layers of abstraction.

- An *expression evaluator* at each layer reads and writes target processor state, and combines state values within arithmetic expressions. Luxdbg's *fxpr* expression evaluator primitive uses machine code level entities (e.g., registers and memory locations) and assembly level symbols (e.g., labels) to retrieve, compute and store values at these levels. The *ce* expression evaluator evaluates C expressions in the context of the procedural layer, translating symbolic references to machine references. Tcl provides its native *expr* expression evaluator with interactive C-like operations. Tcl can combine results from lower level expression evaluators using *expr* and other Tcl primitives.
- *Query primitives* allow users to determine the identity of state-bearing entities within each layer of abstraction. Circuit level query allows Luxdbg users and Tcl scripts to determine the identity of registers, circuit signals, and blocks of physical memory in a specific processor. Machine code level query identifies the name of a processor's program counter, a processor's byte order, a processor's native word size, and a processor's program and data memory. (Program and data memory are distinct address arenas in many DSP architectures.) Unlike many debuggers, Luxdbg does not hard code processor-specific details like those listed above, but instead it queries processor simulation models

to determine these details at run time. The assembly and procedural levels follow suit by allowing users to query for the identity and type of program symbols.

- Signal logging primitives for simulation models store precise time-value transition tables for pins, registers, and other signals into log files. Time-triggered Tcl procedures can log procedure level information as well.

Processor control primitives direct program execution.

- There are primitives to set breakpoints, clear breakpoints, query breakpoints, reset processors, resume execution, stop execution, and to synchronize starting and stopping of multiple processors in a target system.
- Control primitives accept both numeric addresses for the machine code layer as well as labels, function names, source file-line number pairs and data names from the procedural layer.

Processor IO primitives connect processor models and hardware processors to data sources and sinks.

- A loader primitive loads binary values into processor memory regions and loads symbol table information into assembly and procedural debugger layers.
- There are primitives for connecting low-level device IO ports to debugger files and to Tcl callback procedures. IO at the procedural layer usually occurs via library calls, and Luxdbg can insert breakpoints into these calls and redirect library-based IO flow to and from files or Tcl procedures.

Each layer of Luxdbg supports *reflection* (Parson, et al., 1999). Reflection refers to the ability of client code to interactively inspect the underlying contents of server code, in order to determine unique capabilities of a specific server at run time. Reflection forms the basis for Luxdbg query

primitives. For example, Luxdbg allows users to query for available processor model types, and it supports queries for properties of a specific processor object such as its registers:

```
luxdbg: pssr models ; # query available processor models
```

```
16210i 16270i 16410c arm9 ...
```

```
luxdbg: ? R ; # query register {name value type bitwidth} properties in this processor
```

```
{p0 0 s 32} {p1 1000 s 32} {r0 5 u 20} ...
```

Reflection allows Luxdbg to adapt its behavior as well as its user interface to a particular processor at run time. An outer layer can determine and manipulate not only the state of an underlying layer, but through reflection it can also determine the identity of state-bearing entities within that layer. A client of the circuit layer uses reflection to determine the identity and properties of pins, registers and memory blocks within that layer. A client of the machine code layer uses reflection to determine the name of the program counter register, program memory, and byte ordering within a multiple-byte instruction stream. Reflection for assembly and procedural layers exposes symbol names, types and addresses, stack frame conventions and source file identities. Reflection for an extension language such as Tcl provides information on the state of the interpreter, for example:

```
info commands ; # queries Tcl command set
```

```
next logsigs down resume ...
```

Implementation support for reflection within Luxdbg takes the form of C++ abstract classes with symbol table query operations. *InterfaceProcessor* is an abstract class that represents a machine code virtual machine (Parson, et al., 1999). *InterfaceProcessor* includes query operations for determining available registers, memory arenas, signals, peripherals and their properties, and for retrieving and modifying their state. The primary client module for *InterfaceProcessor* is the

Tcl interface, a layer of C++ primitives that translate Tcl commands such as “? R” into InterfaceProcessor queries such as register reads. This Tcl interface also converts results from such C++ queries into data formats suitable for return to Tcl. Concrete C++ classes implement InterfaceProcessor’s abstract query operations by accessing a simulation model or remote hardware processor. These concrete classes include machine-level symbol tables. The Tcl interface is largely insensitive to the implementation details of a particular concrete processor class. The Tcl interface serves to make the abstract operations of InterfaceProcessor available to users and scripts as interpreted, symbolic commands.

At a higher level of target abstraction, Luxdbg uses C++ abstract class *DebugInfo* as a wrapper for symbol table information read from a conventional object file. Like other debuggers, Luxdbg uses this symbol table information to interpret target machine code state in terms of assembly, C or C++ target program constructs. Unlike other debuggers, Luxdbg connects this symbol table information to Tcl query commands. A user or script can interactively query the symbol table to determine functions, data structures, variables, etc., along with their machine code address bindings.

Luxdbg uses reflection to adapt its operations to each target processor and program. Such flexibility is essential in debugging multiprocessor systems that have come to include an array of heterogeneous processor types and programming languages.

Luxdbg’s *expression evaluators* rely heavily on reflection. Luxdbg’s *fxpr*, the machine code and assembly language expression evaluator, uses fixed syntax that is compatible with *expr*, Tcl’s built-in arithmetic expression evaluator. *Fxpr* operands, however, come from machine code entities (e.g., registers and memory) and assembly language symbols that *fxpr* identifies and reads or writes via reflection. *Fxpr* can thus adapt itself to different processor architectures and target

programs by querying their reflection interfaces for operands available to `fxpr`. Likewise, Luxdbg's C expression evaluator, `ce`, queries target programs for available types, variables, functions, etc. The extension language layer has access to all of this reflection and the primitives built using it.

Each processor object has a textual *instance name* in Luxdbg. A user can invoke the *processor new* command to create a new processor simulation model or to connect to a remote processor or process, and to associate this processor with an instance name such as "dsp1" or "controller" or any user-selected name. Thereafter the user can use an instance name as a prefix to debugging commands. A Luxdbg primitive uses an instance name to set the *current processor* reference in C++. Commands nested within the dynamic scope of an instance name go to that processor object. For example, the Tcl command "p2 fxpr r0 = [p1 r3] * 2" retrieves the value of register r3 from processor p1, multiplies it by 2, and stores the result in register r0 of p2. Fxpr determines the semantics of its arithmetic operators by consulting its target machine code processor. Fixed-point DSPs supply fixed-point semantics while microcontrollers supply mixed floating point / integer semantics similar to C expressions.

A Luxdbg Tcl expression can defer specification of a target processor. Listing 1 shows an example Tcl procedure *logRegisters*, written without a processor prefix, that writes properties for all processor registers to standard output. The query "? R @x" returns a list of 4-tuples, formatted in hexadecimal, each of which gives the name, value, machine code type (e.g., signed or unsigned integer) and width of its register. Tcl's *foreach* construct iterates over the list. The bottom of Listing 1 shows a portion of the output for an Agere DSP16210.

<<LISTING 1 GOES ABOUT HERE.>>

Suppose that Tcl procedure *logAll* were written, again without a processor prefix, to invoke

logRegisters along with some other log procedures. Now suppose that Luxdbg is connected to a DSP instance that a user has named “dsp5,” and the user invokes “dsp5 logAll” in an interactive command or from within a Tcl script. Procedure *logRegisters* works with Luxdbg’s *current processor*, in this case the processor named *dsp5*. Primitives such as *fxpr query dsp5* for state-bearing entities and their contents. The Tcl procedure *logRegisters*, while not a primitive, uses primitive “? R” to determine the identity and contents of registers within the current processor, without any hard-coded knowledge of that processor’s registers. A later invocation of “controller2 logAll” for microcontroller “controller2” would perform similarly for controller2, which might be an entirely different sort of processor. As long as *logRegisters* and *logAll* are written to use the reflection interface and target-neutral primitive commands, and to avoid making target processor assumptions, they can work for a variety of processors, selected by a user or script via a processor name prefix at run time.

4. Luxdbg avenues of extension

So far we have described the basic extension language, virtual processor and reflection machinery that provides the basis for Luxdbg’s debugging power. This section catalogs a number of ways in which Luxdbg developers, field support staff and users can employ this machinery in extending the capabilities of Luxdbg as well as its target processors. This section concludes with an examination of performance considerations for these extension mechanisms.

4.1 Command and callback path mechanisms

The *command path* of Figure 1 initiates activity in Luxdbg. All interactions start when a user, a Tcl script or an auxiliary tool issues a set of textual commands to Tcl. Tcl, in turn, examines and modifies target processor state. Tcl sets breakpoints via breakpoint primitives; it causes target processor execution by calling the *resume* primitive. Tcl again interacts with processor state upon

processor arrival at the next breakpoint. These are all examples of command path control mechanisms. Upper, client layers direct control.

Callback path control comes about when a target processor object reacts to an event in the target processing environment by invoking a callback procedure. A callback may be a Tcl expression set by a user, script or auxiliary tool. A callback may also be a C++ method, built into Luxdbg, that reacts to target events. A target event causes its processor to stop at a breakpoint, and a callback can interact with the stopped processor as well as with any other Luxdbg processor that is at a breakpoint. Luxdbg sets the default *current processor* to the triggering processor during a callback. A single processor may provide multiple target events, thereby triggering multiple callbacks, during a single interaction with the debugger. If all events connect to callbacks, and if all callbacks invoke *resume*, then upon completion of all callbacks, the target processor resumes execution as though no breakpoint had occurred. Callbacks form the basis for *conditional breakpoints*.

Command and callback operations have one important difference, part of which we have just seen. Invoking *resume* as a command operation causes a halted processor to begin execution. Invoking *resume* as part of a callback operation schedules that processor for resumption upon completion of all callbacks. A single callback invocation that does not invoke *resume* causes a break to the outer, command-invoking user, script or auxiliary tool upon callback completion. An outer, command *resume* invocation can start any processor, while a callback *resume* can schedule only the processor that triggered the callback.

The major net effect of callbacks is the extension of target processor primitives at any of the layers of Figure 2. From the perspective of a command *resume* invocation, execution of a callback can appear as a seamless part of target processor execution. All of the examples of this section use

callbacks to extend processor capabilities.

4.2 Conditional debugging and assertions

Callbacks support incremental extension of conventional debugging activities, from providing small helper functions that eliminate manual activity, to providing sophisticated assertion checking mechanisms. Conditional breakpoints provide one example. Suppose a target C function exhibits an error after many invocations of that function. A simple breakpoint that stops on every function invocation becomes annoying because of the number of times that the debugger user must manually resume execution. Some debuggers provide breakpoint counters that a user can set in order to skip over breakpoints, but these debuggers do not provide means to determine the counter value in the first place. They also do not provide a means for sampling the triggered breakpoints, for example stopping on every tenth breakpoint. A user could insert extra debugging code into a target function to keep track of invocation counts, but this insertion involves time-consuming compilation, and it shifts the target program address space, possibly masking the error being debugged.

Listing 2 shows Tcl procedure *setCount* that counts the number of invocations of target C function *targetFunc*. Assume that the user has issued the command “stop in targetFunc setCount,” which sets a breakpoint in targetFunc and associates setCount as the callback operation to pass to Tcl. Every target invocation of targetFunc triggers the breakpoint, which stops the target and invokes setCount via the callback path. Tcl interprets setCount, incrementing its counter and resuming target execution. Eventually the target error occurs, causing a breakpoint for which the user has not provided a callback, so control returns to the user. Now the user can interactively retrieve the counter value by issuing the “puts \$setCounter” command to Tcl.

<<LISTING 2 GOES ABOUT HERE.>>

With a `setCounter` value in hand (e.g., 1233), the user could rewrite the “resume” line of procedure `setCount` to be a conditional: “if {`$setCounter < 1230`} resume.” Since Tcl is interpreted, the user can rewrite `setCount` interactively and use its new definition immediately to stop function `targetFunc` in the desired invocation by restarting the target application.

Now `setCount` is the simple counted breakpoint feature built into some debuggers. Sampling every tenth breakpoint could be achieved by using Tcl’s modulo (“%”) operator:

```
if {($setCounter % 10) != 0} resume
```

In general, any callback expression of the form

```
if {predicate} resume
```

supports conditional breakpoints, where “predicate” could be the invocation of a Tcl procedure.

Execution breaks to the user when a predicate is false. Predicate testing can occur at any level of abstraction of Figure 2. For example

```
if {[fxpr r0 != endlocation]} resume
```

uses `fxpr` to compare machine register `r0` to the address of assembly label `endlocation`. Predicate

```
if {[ce head_of_list != NULL]} resume
```

uses the C expression evaluator to compare C variable `head_of_list` to the NULL value. Predicate

```
if {[info commands $pname] == ""} resume
```

uses Tcl’s `info` command to test for the existence of a command whose name is contained in variable `pname`.

This technique is much more powerful than standard conditional breakpoints, because the latter allow tests only on the target program. A Luxdbg user can extend the test space available to predicates. The test space could be the saved results of previous target runs, allowing breakpoint callbacks to monitor regression test results.

The test space may also be a set of assertions that a programmer expects to see upon entry or exit of a target function. By setting breakpoints at function invocation and exit points, and by coding assertions as Tcl predicate expressions, the programmer can check assertions without modifying target code, halting execution only when an assertion is violated. Since the assertion checker is a piece of Tcl code, it can print a message identifying the cause of failure to the user (e.g., print the assertion expression as a string). When control returns to the user, source code display identifies the location at which the assertion failed.

Conditional breakpoints and assertions that break to a debugger for evaluation can have serious effects on target processing performance. We consider performance issues in Section 4.6.

4.3 Extended input-output

Callbacks support input-output extension for all of the layers of Figure 2. At the lowest layer, a processor simulation model can include *peripheral models* that take the place of actual IO hardware. Each peripheral model contains simulated registers and memory that represent their hardware counterparts. In addition, each peripheral model contains a callback hook that a user can connect to a Tcl procedure. Suppose a target processor executes the machine code instruction “ $r0 = pio1$,” where *pio1* is a parallel IO port register. In real hardware the processor would latch the value on the input pins of *pio1* into a register and transfer that value to register *r0*. In a Luxdbg simulation where the user has connected *pio1* to a Tcl procedure by using the *srcfn* command (to “source” input from a function):

```
srcfn pio1 myInputProc
```

Luxdbg invokes a callback to the Tcl procedure *myInputProc*. The procedure can retrieve a value for *pio1* from many places — a file, the user, an output port from a different processor instance, or from a random number generator. The return value from the Tcl procedure finds its way into

simulated register `pio1`, and the simulation continues without interruption.

Reversing the machine code instruction to its output equivalent, “`pio1 = r0`,” copies `r0` to a parallel output port in real hardware. In a Luxdbg simulation where the user has connected `pio1` to a Tcl procedure by using the `sinkfn` command (to “sink” output to a function):

```
sinkfn pio1 myOutputProc
```

Luxdbg again invokes a callback. This time the callback procedure `myOutputProc` receives the output value as a parameter, and it can write the value to a file, user, or input port of another processor instance.

Input-output redirection is not restricted to simulation models. Both models and real hardware can use memory access breakpoints (a.k.a. data breakpoints or watchpoints) to simulate *memory-mapped IO*. A read or write for a memory location with a breakpoint results in a callback that simulates memory-mapped IO. The `stop` command can specify breakpoints at the machine code, assembly or procedural levels of abstraction, supporting memory-mapped IO simulation at any of these layers. For example, if a C program contains a global variable called `output`:

```
volatile int output ;
```

that the C program uses to simulate output, Luxdbg can set a data-write breakpoint on this variable:

```
stop output -write outputHandler
```

and Tcl procedure `outputHandler` can use `ce` to retrieve `output` for writing to a file:

```
puts $logfile [ce output %d]
```

Comparable machine level memory-mapped IO on address `0x2000` would use “`stop 0x2000 - write handler`” to set the breakpoint and “`fxpr *0x2000`” within procedure `handler` to retrieve its value.

These trivial examples only hint at the possibilities. We have constructed a simulation model for an entire application-specific IC block (ASIC) written as a set of Tcl functions. A multiple-processor chip could use such a hardware ASIC for inter-processor communications. We can simulate this interaction in Luxdbg by running real, discrete hardware processors, routing their communications through the Tcl ASIC model via breakpoint callbacks.

Luxdbg supports IO redirection at higher levels of processor abstraction. A target embedded system might not support textual IO, but C *printf* statements remain a popular method of monitoring and debugging program execution, even in the presence of powerful debugging utilities. Because of the usefulness of streaming textual IO during software development, Luxdbg supports *semi-hosted libraries* for many of the standard C IO system calls and library functions. A semi-hosted library function consists of a compiled function stub in a target system and a corresponding Tcl implementation procedure in the debugger. The debugger initializes the library by loading the library into the target, loading the Tcl procedures into itself, and then setting a breakpoint in the target library. Thereafter, whenever the target stops at this library breakpoint, the Tcl procedures upload the relevant state and execute library Tcl code on behalf of the target. The Tcl procedures then return any results to the target and resume its execution.

Listing 3 shows the stub portion of *printf* as implemented for a target DSP16000. This assembly code links directly into an embedded application running in hardware or in a processor simulation model. A caller pushes arguments for *printf* to the stack and invokes *printf* as it would on any system. The *printf* stub works by putting the *printf* function identifier into the processor's *i* register, then invoking the *icall 0* system call instruction. This system call sends target execution to a software interrupt handler that consists of a *noop* instruction, on which Luxdbg has placed a breakpoint.

<<LISTING 3 GOES ABOUT HERE.>>

At the time that the target processor triggers this breakpoint, its stack holds the arguments to `printf` and its `i` register identifies `printf`. The breakpoint causes a Luxdbg Tcl callback to take control. The callback retrieves the value of the `i` register and uses it to select the appropriate library function handler via a switch statement.

```
switch -exact [fxpr i @d] {  
  
    $SHLIB_PRINTF handle_printf  
  
    $SHLIB_SCANF handle_scanf  
  
    etc.  
  
}  
  
resume ; # target processor resumes after handler completes
```

Listing 4 shows the beginning and end of the `handle_printf` event callback handler. It reads the `printf` format string from the top of the DSP16000 stack memory, iterates through a loop that reads value parameters from the stack and concatenates an output string (details not shown), sends this string to the debugger's standard output, and finally returns the number of characters printed back to the target environment in register `a0`. Upon completion of printing, `handle_printf` returns to the callback procedure that selected it. This procedure resumes target execution, an action that flushes any register and memory state changes to the target, including `printf`'s result in `a0`. The target now returns from its interrupt handler, through the `printf` stub of Listing 3, to the caller, who receives the return value set by `handle_printf`.

<<LISTING 4 GOES ABOUT HERE.>>

Input-output handling at the client layer of Figure 1 takes the form of Tcl/Tk GUI extensions. Luxdbg gives users access to the full power of the Tcl/Tk graphical tool kit (Welch, 1999), with

which they can create custom graphical widgets and bit-mapped graphical canvases. As with the other levels of IO redirection, users can connect event callbacks to GUI extensions, updating their graphical creations with processor information as changes occur.

4.4 Multiple processor scheduling

Luxdbg becomes involved in processor scheduling when it is used to simulate communications within a multiprocessor system. Luxdbg loads one or more frames of input data to the first processor in a pipeline or other multiprocessor topology, then uses *resume* and breakpoint callbacks to run that processor until it fills an output buffer. Filling an output buffer triggers a breakpoint, and a breakpoint callback procedure copies the output buffer to the input buffer of a receiving processor. A Tcl level scheduler can then *resume* the receiving processor's execution, causing it to contribute its part to system data flow.

Luxdbg also becomes involved in processor scheduling when debugging concurrent threads of execution that interact by accessing shared resources in *critical sections* of code. The debugger can set breakpoints around known or suspected critical sections, resume threads concurrently until they reach critical section breakpoints, and then resume them serially until they reach breakpoints following the critical sections. In most concurrent systems users can achieve concurrency control only by coding and compiling synchronization primitives into target programs. In Luxdbg users can achieve and modify concurrency control interactively in the debugger as well.

Processor input-output handling works using the callback path of Figure 1. Processor scheduling takes us back to the top-down command path. Recall from Section 4.1 that invoking *resume* from above causes processor execution, while invoking *resume* from within a callback avoids taking an interactive breakpoint. Avoidance of callback *resume* is the key to Tcl schedulers. A breakpoint that has no Tcl callback, or a callback without *resume*, causes processor

execution (initiated via *resume*) to stop, returning control to the outer *resume*'s caller. This outer *resume* may have come from a user, or it may have come from a Tcl scheduler script.

Listing 5 shows a simple round-robin, cooperative scheduler for a list of processors passed as a parameter to this Tcl procedure. Each processor in the list runs until it hits a breakpoint, and the scheduler iterates over the list *N* times. Presumably the breakpoint is triggered on the completion of some task, e.g., completion of processing a data flow. A scheduler with uncooperative processors could substitute *stepi \$count* for *resume*; the *stepi* command steps a target processor some number of machine level instructions; it is guaranteed to return.

<<LISTING 5 GOES ABOUT HERE.>>

Listing 6 gives a scheduler variation, this time a scheduler driven by processor output events. The scheduler starts out using parameter *startproc* to identify the processor to start, then it consults a global Tcl variable *nextproc*. Output breakpoint callback procedure *output_callback* is coded to write its output value to a neighboring processor's input port — obtained via associative array *neighbor*, set up by the scheduler's Tcl script, which is indexed on the current processor name obtained via primitive *processor name* — and then pass that neighbor's name to the scheduler for execution via *nextproc*. The lack of *resume* in the callback guarantees return of control to the command path scheduler. Scheduling terminates when the last processor in the sequence is reached, signified by a blank slot (“”) in the neighbor table.

<<LISTING 6 GOES ABOUT HERE.>>

Luxdbg originally provided only the blocking *resume* and *stepi* commands for execution resumption, allowing only one processor or multi-processor chip to execute at one time. Serial scheduling was the only possibility. Luxdbg now includes the ability to run target processors in the background without blocking the debugger via the *resume &* command, named after the

UNIX use of `&` for background execution. We have added a *wait* command that blocks until specified background processors have stopped at breakpoints. The combination of *resume &* and *wait* gives Luxdbg logical *fork* and *join* operations for concurrent target processors and processes. With these commands we are able to write concurrent schedulers that interleave target processor execution, blocking the debugger until it is safe for it to act.

4.5 Exception logging and testing

Luxdbg treats exception events in a target processor similarly to the way it treats breakpoints. Each exception has a unique identifier to which a Luxdbg user or script can attach a callback. Exceptions come in four levels of severity — note, warning, error and fatal. Default behavior for notes and warnings is for Luxdbg to print a message and issue *resume*, while default behavior for errors and fatal errors is to print a message and stop the processor. Callbacks can be attached to any exception type, and any non-fatal exception callback can successfully invoke *resume*. Users can use this mechanism to shut off unwanted notes and warnings, to log specific exceptions, and to perform processor exception handling in Tcl similar to breakpoint handling already discussed.

Among other uses, an extension language such as Tcl provides an ideal basis for regression test machinery. Tcl has a complete set of file manipulation operations, making it possible to set up tests and compare results from files of text tables. Tcl provides a *catch* instruction that allows Tcl-level handling of failed instructions, so it is possible to set up degenerate conditions in a test script and test for proper Luxdbg reactions without crashing the script if Luxdbg operations fail. Luxdbg testing uses all of the extension language machinery discussed above within its regression test suite.

4.6 Performance considerations

We have discussed using conventional execution breakpoints, processor simulation models

and extension language callbacks to support conditional breakpoints, simulated input-output extension, scheduling of multiple target threads and processors, exception logging and testing. Stopping a target processor to extend its capabilities affects the performance of that processor. Using an interpreted extension language rather than the debugger's compiled implementation language further affects debugger performance. This section looks at the temporal effects of stopping target computation and of using an interpreted extension language.

A user typically specifies a conditional breakpoint by using an efficient triggering mechanism such as a program execution breakpoint in combination with an expression to be evaluated when the breakpoint trigger is satisfied. A debugger can evaluate a conditional expression after the primary breakpoint occurs, or a debugger can patch code into the target system to perform expression evaluation. Target-resident code patching techniques show orders of magnitude performance improvement over debugger-resident conditional expression evaluation (Keppel, 1993; Kessler, 1990; Wahbe, et al., 1993).

Here we measure two temporal effects, those of evaluating a conditional breakpoint within Luxdbg, and those of using Tcl to perform the evaluation. We use a three-line loop compiled without optimization for an Agere DSP16210 to measure these effects.

```
int j = 0 ;  
while (j < N) {  
    j++  
}
```

The number of iterations N depends on test conditions. We place a conditional execution breakpoint on the “j++” instruction, using an expression that always evaluates to false, i.e.,

```
stop 'timeloop.c'6 -when { j == -1 } ; # line 6 is “j++” line
```

to measure the effects of conditional tests that do not result in breakpoints.

The left column of Table 1 shows execution of this loop under four conditions. “No breakpoint” signifies execution without a breakpoint. Time is measured from within the debugger, beginning when the debugger issues a *resume* command just before entering the loop, and ending when the debugger detects an unconditional breakpoint event just beyond the loop. “Target patch” simulates target-resident expression evaluation by means of a hand-coded conditional test compiled into the target code. “Debugger test” shows the result of testing the condition in the debugger by evaluating the conditional expression using the debugger’s compiled C expression evaluator. “Tcl test” shows the result of testing the condition in the debugger by invoking a Tcl callback procedure as discussed in Section 4.2.

<<TABLE 1 GOES ABOUT HERE.>>

The center section of Table 1 shows the results of running these four test conditions on an Agere DSP16210 processor with a 10 MHz. clock. Luxdbg runs on a 360 MHz. Sparc processor with the Solaris 5.7 operating system, which is connected via a 100 Mbit./sec. Ethernet connection to a 200 MHz. Pentium processor with the Windows 95 operating system, which is connected via a 765 Kbit./sec. serial debugging connection to the target DSP. The latter serial connection is the debugging system’s bottleneck. Such a multipartite cross-debugging system is not unusual for embedded systems. The Solaris machine acts as the software development environment, and the Pentium machine acts as a hardware interface controller. The target DSP has almost no memory available to assist debugging. The slow, serial nature of the 765 Kbit./sec. debugging connection is symptomatic of the facts that IC designs allocate few (hence serial) pins to debugging, and bit rates along these pins are slow. Pin count and speed are minimized to reduce cost per unit and power consumption. Here we are interested in comparative measurements for the

rows in Table 1, rather than absolute measurements of time. The conditions of Table 1 are worse than those for running a similar benchmark on a time-sharing system where the debugger and target process reside within the same computer.

The large iteration count N of 10^7 for the first two hardware rows of Table 1 is necessary because, at lower powers of 10, the time the debugging system takes to download state, start the loop, and handle the breakpoint after loop exit dominates the measurement. With $N \gg 10^7$ the execution time scales linearly with N .

Table 1 shows that a target-resident patch within a hardware target runs about 42% slower than the no breakpoint loop. Performing the condition test is almost half as costly as performing the other simple work of this loop. In this example we have patched the in-line conditional test by hand. An automated patch would entail jumping to condition testing code (Kessler, 1990), adding additional overhead. In more realistic scenarios with a higher ratio of application work to conditional breakpoint evaluation, conditional tests running in a target without breaking to an operating system or debugger incur relatively little overhead.

Table 1 shows a staggering 16,294,242% increase in execution time per loop in going from no breakpoint to the debugger test condition; code with the conditional breakpoint runs almost 163,000 times slower. Bear in mind that this test is a worst-case scenario for conditional breakpoints. The execution breakpoint occurs in a tight inner loop of only eight machine code instructions, and we are measuring only the execution time of that loop. A conditional breakpoint at a less heavily traveled location can still be useful when debugging a hardware system. Nevertheless, we see that the costs of stopping the processor and uploading some of its state across multiple delays is significant for this cross-debugging system.

The good news for the Tcl test of Table 1 is that, while it, too, runs about 163,000 times

slower than the no breakpoint condition, Tcl interpretation adds a mere .1% overhead when compared with the debugger test condition. Tcl adds insignificant additional overhead, but it adds tremendous additional flexibility. Whereas conventional conditional breakpoints evaluate simple target state conditions, Tcl tests can include comparisons to expected test results as well as complex assertions. The specification of an embedded system could include constraints on major interfaces expressed as Tcl assertions. It is not unusual to employ over-night and even multi-day simulation runs in verifying the hardware design of an embedded system. It would be equally useful to employ assertion checking as part of final system integration testing.

The right side of Table 1 shows the more promising results of running the same test conditions on an instruction-accurate simulation model of the DSP16210 using the 360 MHz. Sparc processor. The debugger, the simulation engine and the model reside in a single process and they communicate via function invocations. Again time is measured within the debugger from entry of the loop until its exit. With eight machine code instructions in the loop and 415.8 microseconds per iteration, each instruction simulates in about 52 microseconds, so this instruction-accurate simulation model can simulate about 19,240 instructions per second when run on this Sparc processor. In this benchmark the simulation model runs about 219 times slower than the real processor.

Keppel mentions target simulation as a classic way to implement data breakpoints (Keppel, 1993), and Table 1 shows why processor simulation remains popular for debugging embedded systems where access to hardware is costly. Target-resident condition testing is about 50% slower than the no breakpoint condition, on par with the 42% figure for hardware. Perhaps surprisingly, the debugger test of the conditional is actually less expensive than the target patch, incurring 46% overhead. This result shows that Luxdbg's compiled implementation of expression evaluation is

marginally more efficient than this DSP16210 simulation model's fetch-decode-and-execute logic for this example.

The simulation of debugger test conditional evaluation executes 511 times faster than the hardware execution of this condition. A user can leverage a processor simulation model to locate data-sensitive bugs in reasonable time.

The temporal effects of using Tcl are somewhat more pronounced in the simulation results. Tcl adds 31% overhead over simulation debugger test, compared to .1% for hardware execution, and Tcl adds 91% overhead over no break simulation, up from debugger test's simulation overhead of 46%. Here we see the difference between compiled and interpreted tool code. The effect is incremental rather than staggering, and Tcl proves to be a viable means for extending simulations.

Both the range of simulated processor performance and the range of related literature are very wide, and a detailed examination of processor simulation technology is beyond the scope of this paper. We can summarize by saying that as the fidelity of an embedded processing simulation model with its simulated system increases, its performance degrades. It is common to find that a detailed, circuit-level model that simulates appropriate state changes for a processor and its bus and peripheral components on every transition of a simulated clock, written in a circuit description language such as VHDL (Bhaskar, 1998), executes on the order of tens of instructions per second. Instruction-accurate C/C++ models that simulate the fetch-decode-and-execute logic of a processor via interpretation of op codes execute on the order of tens of thousands to hundreds of thousands of instructions per second (Jeremiassen, 2000). The 19,240 instructions per second measurement of the DSP16210 model of Table 1 is typical. Between these two model types are cycle-accurate and phase-accurate C/C++ models that reflect internal state changes with clock

cycle and clock transition fidelity respectively (two transitions constitute a cycle); these models execute hundreds to thousands of instructions per second.

At the low fidelity / high performance end of the spectrum lie so-called *compiled simulations* that simulate, not a target processor executing op codes, but rather a target application (Cmelik and Keppel, 1993). Simulation compilers compile target applications into op codes for a simulator machine (e.g., Sparc) rather than a target machine (e.g., DSP16210), and the execution of these op codes simulates changes in target state. These models simulate the effects of hundreds of thousands to millions of target instructions per second, but they cannot readily simulate multiple target instruction streams that have nondeterministic temporal interactions, e.g., interrupt-driven interactions, and they cannot deal with self-modifying code caused by target mechanisms such as overlay loaders.

With its ability to debug multiple levels of target abstraction, Luxdbg can debug software running on all of these simulation model types. Given the wide range in simulation performance and fidelity, Tcl extension of target simulations as presented in this section has reasonable performance. As the lower-right entry of Table 1 indicates, Luxdbg users can employ Tcl to extend simulated input-output and multi-processor scheduling without significant performance penalty. Tcl provides an environment where users can explore simulated extensions interactively, rewriting them as more efficient compiled code only when their design becomes stable.

While debugger-resident Tcl extension is appropriate for simulated real-time systems, the results of Table 1 indicate that not only Tcl, but interactive debugging itself is not appropriate for non-obtrusive monitoring of *real* real-time systems. Interactive debuggers stop target execution, changing its real-time properties. Debugger-target interaction can occur before and after, but not during temporally significant processing in real targets.

Many of the embedded processors with which Luxdbg works include processor-specific data breakpoint matching hardware and execution trace buffers. Triggering stimuli for matching hardware do not readily map from arbitrary boolean-valued conditional expressions in a high-level language. We have found it useful to open “back doors” for users, making matching hardware accessible via Tcl manipulation of circuit-level match registers. Sophisticated users analyze bus conditions and then initialize match registers via Tcl, creating custom breakpoint types. Similarly, users can configure and access trace buffers using Tcl. Tcl-based access gives users the ability to participate in exploring the command space appropriate to specialized breakpoint circuitry. As productive usage patterns for custom hardware breakpoints become clearer, we can migrate interaction from custom scripts to built-in debugger commands. Thus, while performance precludes Tcl from playing a large role in extending the run-time behavior of real hardware, Tcl plays a significant role in providing access to custom debugging hardware for which, at present, there are no general-purpose debugging commands.

5. Evolution of an extensible debugger

While the previous section concentrates on the extension mechanisms that tool developers and users can employ in extending Luxdbg’s capabilities, this section examines two strategies for employing those mechanisms in expanding the scope of the tool set. The first strategy is one of dynamically adding higher layers of processor abstraction to those built into Luxdbg, while the second is one of prototyping new features in the extension language.

5.1 Application-specific debugging

Figure 3 extends the layers of Figure 2 by adding another layer, the *application language layer*, between the procedural and extension language layers. The key to Figure 3 is the idea that programmers can write new layers in the extension language. Extension language procedures

become new “primitives” for these layers. Application-specific reflection mechanisms may be part of these layers (e.g., implemented as Tcl data tables). An application layer encapsulates its virtual machine view, and the extension language continues to reside at the outermost layer, capable of extending the layers within it.

<<FIGURE 3 GOES ABOUT HERE.>>

One example application-specific layer for Luxdbg is a signal processing system composed of two digital signal processors connected as a pipeline, that transforms signals flowing through it by invoking library routines. Figure 4 illustrates Luxdbg interaction with such a system. Processors 1 and 2 are DSPs. They may be hardware processors or simulation models. The *mailbox* is a Tcl simulation model of application specific hardware that supports inter-processor communication.

<<FIGURE 4 GOES ABOUT HERE.>>

Luxdbg interacts with this target system in the following ways. A *temporal breakpoint* callback handler, written in Tcl, supplies input signals to an input port of Processor 1 at regular intervals. Luxdbg’s *at* command supports scheduling breakpoints at temporal intervals. This callback handler simulates input circuitry external to Processor 1; it provides input signals to Processor 1 by obtaining values from a Tcl data file and writing them to a memory-mapped IO location of Processor 1. The Tcl input handler stimulates a hardware interrupt in Processor 1, and, for each input signal value, it computes an expected output signal that it passes to the output handler, which is also written in Tcl.

A Tcl simulation scheduler resumes execution of Processor 1 for a period of time during which Processor 1 receives and processes incoming signals, after which it writes them to the simulated mailbox. A data structure in Processor 1’s memory space simulates memory-mapped mailbox hardware. When the amount of data in the mailbox exceeds a mailbox *data threshold*

parameter set by Processor 1 or 2, a Tcl callback for a *data breakpoint* (a.k.a. watchpoint) transfers the state of the mailbox to the memory of Processor 2, locking a semaphore within the mailbox memory of Processor 1 to disable Processor 1 from using the mailbox during Processor 2's access. The scheduler resumes Processor 2, which drains the mailbox, performs its signal processing, and writes a result to its memory-mapped output port. Another Tcl callback for a *data breakpoint* on Processor 2 fires whenever this output port is written. This breakpoint handler compares actual output to expected output computed by the input handler, asserting an error that breaks to the user when these values differ.

The solid arrows of Figure 4 indicate Tcl extension of the target processing system. Three Tcl breakpoint handlers simulate input circuitry, interprocessor communication circuitry, and output circuitry. The scheduler simulates temporally accurate execution of the target system. The dashed arrows of Figure 4, on the other hand, indicate Tcl extension of the debugger. Tcl handlers for additional data breakpoints watch for application bugs such as buffer overflows or lost signal samples.

Figure 5 helps make the transition from a view of Figure 4 as a piecemeal collection of Tcl scheduler and breakpoint handler procedures to a view of an application virtual machine layer. Figure 5 is a virtual machine — it is a state machine for Processor 1's application algorithm. Execution begins in the start state *waiting*. When an input signal arrives at Processor 1, it moves from the *waiting* state to the *working* state, where it performs signal processing. If the mailbox is available, i.e., not in use by Processor 2, Processor 1 then writes a result to the mailbox in the *mailing* state and returns to the *waiting* state. If, on the other hand, the mailbox is not available because Processor 2 has its semaphore locked, Processor 1 enters the *buffering* state, where it saves its output signal in a local buffer and enables mailbox interrupts. Again Processor 1 enters

the *waiting* state. At a later time when available space in the mailbox exceeds a *space threshold* parameter set by Processor 1 or 2, access to the mailbox returns to Processor 1, asserting an interrupt. Processor 1 then enters the *mailing* state where it drains its local buffer into the mailbox, then returning to the *waiting* state.

<<FIGURE 5 GOES ABOUT HERE.>>

Most debuggers have no means for utilizing information at this level of abstraction. Luxdbg's extension language supports interaction at this level of abstraction by allowing a user to create Tcl commands and breakpoint monitors specific to this application view. Figure 6 shows a textual representation of application state. Tk GUI text widgets display the state of Processor 1 as *p1state*, capacity statistics on Processor 1's local buffer and the mailbox as *bufpercent*, *buffermax*, and *mboxpercent*, statistics on the average and maximum time for Processor 1's response to the arrival of input events as *plavg* and *plpeak*, and an assortment of other application-specific values.

<<FIGURE 6 GOES ABOUT HERE.>>

Figure 7 shows an interactive Tk window that corresponds precisely to Figure 5. The current state, which is *mailing* for Figure 7, appears in gray. Textual monitors near the *buffering* and *mailing* states display Processor 1's local buffer statistics and mailbox capacity. Both the states and transitions of Figure 7 respond to mouse-driven input by toggling breakpoints on and off. The stop signs in Figure 7 show that a breakpoint is set for entry into the *mailing* state, and that a breakpoint is set for the *working-to-buffering* transition. These two breakpoints translate into a breakpoint on function entry for *mailing* and a breakpoint at a function invocation for the *working-to-buffering* transition. In order to support this style of interaction, the Tcl scripts that control the window of Figure 7 must have access to the state-to-function and transition-to-

invocation symbolic information used in converting the state diagram to a target C program, as well as function-to-state symbolic information for highlighting the current state.

<<FIGURE 7 GOES ABOUT HERE.>>

Several commercial CASE tools for embedded systems support generation of code from assorted diagram types (I-Logix, 2000; Project Technology, 2000; Rational, 2000). These tools include their own debuggers that are limited to operation strictly at the diagram level of abstraction. A single debugging session cannot span levels of abstraction, but spanning levels of abstraction is important for embedded system debugging. The extension language layer of Luxdbg allows users to automate debugging of CASE diagram (and higher) abstractions via extension scripts. If a commercial CASE tool publishes the symbolic information that relates graphical diagrams to generated code, then Luxdbg extension scripts can map run-time code state up to diagram state, supporting debugging for multiple CASE tool abstractions. By positioning the extension language at the outer layer of Figure 3, Luxdbg becomes a debugger for unbounded levels of abstraction.

5.2 Prototyping via an extension language

Section 2 introduces extension language prototyping of tool features as one form of automation associated with embedded extension languages. Rather than limit feature engineering to paper specification and compiled code, an extension language supports interactive, incremental exploration of feature spaces.

Application-specific debugging provides one example of a Luxdbg feature set prototyped in an extension language, that is now headed into the core of the debugger. Luxdbg has always hard coded the layers of Figure 2, with the extension language supporting creation of additional layers of abstraction as discussed in Section 5.1. A new version of Luxdbg is adding support for open-

ended layers of machine abstractions within compiled debugger code. Initially the debugger connects to a set of abstraction levels L_0 of a target embedded processor, typically having the machine code layer as the set's innermost layer, via a *connection factory*. This factory manufactures a connection from the debugger to a target. An alternative L_0 set would consist of a Java source-level abstraction atop a Java Virtual Machine using the Java Debug Interface (Sun Microsystems, 2000). Thereafter a *loader factory* adds an additional set of layers L_1 on top of L_0 , containing for example assembly and C symbolic information for a machine code L_0 . Higher order loaders can load higher order L sets by loading symbolic information (e.g., CASE tool diagram-to-code relations) into the debugger, and optionally by attaching to debugging hooks in targets. The extension layer L_{ext} always resides at the outermost level, with access to the inner layers.

This version of Luxdbg uses Java's dynamic loading capabilities to load debugging layers at run time, based on arguments to connection and loader factories. The generic debugger uses the *Chain of Responsibility* Design Pattern (Gamma, et. al., 1995), diagrammed in Figure 8, to route commands to the appropriate layer of abstraction. Each layer houses its own expression evaluator. A layer that implements a specific command, e.g., the *stop* command that sets a breakpoint, inspects the syntax of the command arguments to determine if it is appropriate for that layer. If it is, that layer acts on the command. Otherwise that layer delegates the command to the next lower layer, e.g., L_2 delegates to L_1 , repeating this pattern of interpretation. If a command does not match the requirement of any layer, then L_0 throws a command exception.

<<FIGURE 8 GOES ABOUT HERE.>>

At present we have a working version of the language-neutral portion of this debugger, along with dynamic connection factories and language-specific modules for two L_0 virtual machine

types, a Java VM accessed via the Java Debug Interface, and a machine code VM accessed via our simulation modeling technology. The generic debugger encodes no knowledge of Java, of a processor's machine code, or of our simulation technology. It interacts with an abstract interface for requesting breakpoints in symbolic terms, for handling breakpoint events triggered by these requests, for stepping, resuming and stopping target execution, and for querying target state. Three commands — *syminfo*, *expression*, and *dump* — suffice for querying a layer's reflection API for state-bearing entities (*syminfo*), evaluating an expression using a layer's syntax and semantics (*expression*), and dumping a layer's state (*dump*). Each layer has language-specific command keywords, for example *syminfo -classes* for determining the classes in a target Java application, or *syminfo -registers* for determining the registers in a machine code target. *Syminfo* returns lists that can be examined in scripts. Each layer has a unique name that is open to reflection, e.g., *je* for the Java debugger and *me* for the machine code debugger, that a user or client code can query and use to determine appropriate command keywords. We are in the midst of building a loader and L_1 VM for C/C++ debugging atop the machine code L_0 VM. We have proven the feasibility of this approach by using it to debug two extremely disparate virtual machines, a Java VM and a DSP machine code VM.

A more mundane example of feature prototyping in the extension language comes from Luxdbg's GUI. This Tk-based GUI updates display windows when it receives display update events from the debugger. The original GUI loads the debugger with Tcl scripts that test for changes in target processor state by extracting and comparing state as Tcl strings. This technique, while effective, becomes slow when applied to a large number of target processors, because of the overhead of string comparisons and Tcl interpretation. A new version of Luxdbg is re-implementing state difference detection and display update event propagation in compiled code

for efficiency.

A third example feature prototyped in the extension language is a processor-neutral scheduler. Originally Luxdbg employed two types of schedulers, those designed into multi-processor simulation models and chips, and those written in Tcl as discussed in Sections 4.4 and 5.1. A C++ multi-processor simulation model consists of an outermost *driver processor* that schedules the clock logic of the individual processors that it contains by calling their respective *strobe* methods according to some timing ratio (Parson, et. al., 1999). A given driver processor's strobe method is coded to call the contained processors' strobe methods. It mirrors the chip-specific clock logic found in real hardware.

Tcl processor schedulers operate at the opposite end of the scheduler spectrum. They can be configured to work with any collection of processors at run time, as our examples of Section 4.4 indicate, but they operate slowly. Based on Tcl prototyping, our solution has been to design a special purpose driver processor that does not hard code its contained processors, but instead allows a user to select them and establish their strobe ratio at run time. Figure 9 shows a UML class diagram for the *Composite* Design Pattern (Gamma, et. al., 1995) around which these dynamic processor groups are designed (Parson, et. al., 2000). Any Processor is either an InstructionStream that executes a sequence of programmed instructions, or a ProcessorGroup that contains and synchronizes a set of InstructionStreams and ProcessorGroups. A user can create a new ProcessorGroup and aggregate existing Processors within it at run time, using primitive Tcl commands. This approach keeps the flexibility of Tcl in dynamically selecting the processors to schedule together, but it uses the more efficient mechanism of the C++ ProcessorGroup simulation model to schedule the fine-grain temporal steps of the simulation.

<<FIGURE 9 GOES ABOUT HERE.>>

These examples of extension language prototyping of features that may later migrate into compiled tool code share a common architectural characteristic. They all operate at the extension language - compiled language boundary. As Figure 10 suggests, there are multiple strata within both the compiled code and extension language layers. Luxdbg layers of virtual machines are examples of the former, and a prototyped application-specific machine layer or a Tcl scheduler within the extension language are examples of the latter. When an extension language stratum becomes sufficiently stable, with sufficient demand for efficiency improvement, it can migrate across the boundary into compiled code. Layers above it continue to use it largely as they have in the past. With compilation it becomes more efficient and less flexible. One could view Figure 10 as a “sedimentary” form of tool development that proceeds by depositing newer, softer layers of functionality on top of those that have been hardened and proven by usage over time.

<<FIGURE 10 GOES ABOUT HERE.>>

6. Related work

Several other debuggers have used debugging languages or extension languages. The *deet* debugger for C programs (Hanson and Korn, 1997) comes closest to Luxdbg in its use of Tcl as the user interaction language, its use of a user-extensible Tk GUI, and its application of Tcl for conditional breakpoints and testing. Deet builds atop *cdb*, an earlier, machine-independent debugger for C programs (Hanson and Raghavachari, 1996). Luxdbg and *deet* differ in their use of Tcl primarily with respect to Luxdbg’s user association of arbitrary Tcl expressions with breakpoints and exceptions for callback interactions. Deet supports conditional breakpoint expressions that it evaluates using Tcl procedures, but *deet* appears not to have Luxdbg’s more general target processor event handling machinery. Much of this Luxdbg machinery finds application beyond debugging, for example in simulation, testing and prototyping, but event-

driven callbacks are also useful for automating reactions to bugs.

SmartGDB is another Tcl-based debugger (University of Kansas, 2000). In addition to adding GDB commands as primitives to a Tcl interpreter, SmartGDB provides the ability to associate a Tcl procedure with a breakpoint event. SmartGDB does not distinguish between the command path and callback path of Figure 1. Invoking the GDB *continue* primitive from within a breakpoint procedure causes immediate resumption of processor execution, resulting in loss of breakpoint context as well as loss of any concurrently triggered breakpoint procedures. Because of the lack of distinction between command and callback invocations of the Tcl interpreter, SmartGDB does not support seamless extension of debugger primitives via resume-bearing callbacks.

DUEL (Golan and Hanson, 1993) and *ACID* (Winterbottom, 1994) are two debuggers with embedded, special-purpose debugging languages. These languages are not general-purpose scripting languages such as Tcl, but rather are dedicated to debugging specific types of target systems and languages. As with deet, both use top-down, command interactions from the debugger to a target system.

Generalized path expressions (Bruegge and Hibbard, 1983) and the Formal Annotation Language (FORMAN) (Auguston, 1994) are two query-oriented debugger languages that operate on program execution traces. Path expressions provide a special purpose notation for querying program execution paths and for specifying ordering and other constraints. FORMAN uses target processing events and a general event grammar to provide a language for automation of debugging, assertion checking and profiling. Both approaches operate at a higher level of abstraction than Luxdbg, providing mechanisms for searching and testing the execution history of a running process.

In the issue of language selection, there is a trade-off between language applicability and

language accessibility to a large programming population. Use of a special purpose language requires learning yet another programming language that is dedicated to a single stage of software development, debugging. Use of Tcl, on the other hand, transfers into areas beyond debugging, such as testing, user interface design, and other applications. Popular extension languages come with large utility libraries, commercial support, and a large user community from which to draw information. With modest efforts a user can build or collect Tcl procedures for Luxdbg that achieve some of the benefits of the debugger-specific languages. We have found that field application engineers and expert users of Luxdbg have built sophisticated, application-oriented debugging infrastructure using Tcl that easily puts Luxdbg on par with dedicated debugging languages with respect to expressiveness. Tcl could provide a base for building automated debugging extensions similar to those of dedicated debugging languages.

The Coca debugger has used a more widely accepted, interactive language — PROLOG — as the basis for a powerful engine that searches program trace information during target program execution (Ducassé, 1999). Coca uses PROLOG's backtracking on failure mechanism very effectively to allow users to explore a set of program execution states that satisfy query constraints. Coca avoids the storage-intensive performance problems of some relational debugging systems by using breakpoints and target system-resident program event extraction and analysis modules to perform its queries at target run time. With the ready availability of PROLOG interpreters, it would be possible to replace Tcl with PROLOG in a research version of Luxdbg that supports run-time selection of the extension language (Parson, 1999). A novel requirement of Coca is extension of the symbol table information of a target program with annotations concerning various target programming constructs such as the location of *for* loops and *if* conditional constructs for language construct-driven specification of queries and corresponding

placement of breakpoints.

The *ldb* debugger uses PostScript as an extension language for communicating target program symbol table information among the compiler, target application and debugger (Ramsey and Hanson, 1992). It does not bring the extension language out to the debugger user.

Luxdbg differs from all of these related projects in several regards. Luxdbg uses processor / process instance names, as Tcl command prefixes, to delimit the dynamic scope of debugging commands. Luxdbg provides debugging at the multiple layers of abstraction of Figure 2. It includes dedicated expression evaluators *fxpr*, *ce* and *expr* to delimit the scope of symbols within an expression. DUEL and ACID work directly in target system scope, at a single level of abstraction, while *deet* employs a single level of namespace indirection for C variables. The other debuggers cited use explicit scope to access variables at a single level of target language abstraction. Consequently Luxdbg could be viewed as a series of debuggers, for different levels of processor abstraction, with an expression evaluator for each. Commands that interact with more than one level of abstraction, such as *stop* for setting breakpoints at machine, assembly or procedural levels of abstraction, include command syntax for disambiguating context. Commands infer a default context of either a procedural language or assembly / machine code, based on a processor's current breakpoint address and corresponding program scope.

Many of the features of Luxdbg are orthogonal to the issue of extension languages. Virtual processor layers, reflection, multiple processor debugging and heterogeneous processor debugging are examples. Yet this orthogonality is a property that gives an extension language considerably more power in Luxdbg. An extension language user has access to all of these orthogonal capabilities, and a user can combine these capabilities via extension language scripts to create powerful custom debugging features.

7. Conclusions and directions

Luxdbg is being enhanced, supported and deployed by a team of engineers in Agere Systems. The generic debugger / simulator consists of about 115,000 non-comment lines of C++ code, and hardware control comprises another 22,000 lines. The GUI consists of 5700 lines of C++ code and 34,000 lines of Tcl. A single simulation model adds anywhere from 10,000 lines to over 100,000 lines, depending on its fidelity with hardware. Tcl and Tk version 8.3.3 contribute 100,000 and 123,000 non-comment lines respectively. Clearly we are getting a great deal of leverage in terms of source lines by using Tcl as Luxdbg's command interpreter.

Luxdbg's Tcl extension language features are very successful. All of the aspects of Tcl discussed in this paper — its use in mixed C-assembly-machine code-circuit debugging, its use in customizing embedded input-output operations and exception callbacks, its use in processor scheduling, its use in testing, its use in constructing custom development environments, and its use in extensible GUI construction — find practical application in the field. Luxdbg's application of Tcl remains one of the strongest aspects of this debugger.

Tcl extension of Luxdbg and its processing targets is a success primarily among Agere field application engineers (FAEs) and one or two expert users per typical customer site. They write extensions that other users use. Extensions include the following categories, developed independently or in collaboration with Luxdbg engineers:

- custom test beds for regression testing of customer embedded systems
- pre-run loading and post-run analysis and storage of time-dependent test / profiling data
- custom Tk GUIs for visualization, e.g., visualization of multi-processor data flow
- circuit-level interaction with processor-specific breakpoint hardware to provide fast breakpoint facilities for which there are no equivalent high-level debugging commands

- exploration of possible mechanisms for debugging optimized code

FAs have used Tcl to extend Luxdbg for marketing demos, extending the appeal of the tool. These demos have in several cases grown into full-fledged features that FAs assist in supporting at their customer sites. The most recent example (May, 2001) pertains to debugging optimized code. It is typically not possible to map optimized machine code to C source code for symbolic debugging, especially on highly parallel target processors. Time-dependent problems often disappear when optimizations are turned off, so it becomes necessary to debug these real-time problems using assembly-level debugging coupled with additional diagnostic information on optimizations provided by the C compiler. An Agere field engineer is exploring using Tk to highlight lines of C source code entailed in the execution of a single, variable-length execution set of instructions (VLES) at the machine level. A VLES packet contains multiple instructions, possibly derived from different lines of C source, that execute in parallel. An assembly source window steps sequentially through machine code that includes VLES instruction packets, and one or more source windows highlight the lines of C code contributing to the current assembly instruction. After several rounds of customer interaction, we hope to have an approach to debugging optimized real-time code that we can incorporate into Luxdbg.

Ongoing work includes investigation of Java for upper debugger layers in order to take advantage of Java networking, dynamic loading, reflection and class library infrastructure. Luxdbg has used Java to parameterize the extension language, so that languages such as Python can be loaded at run-time in the place of Tcl (Parson, 1999).

Distributed debugging supported by distributed extension language interpreters is an area for research (Parson, et al., 2000). Passing expressions across the Internet to remote debugging sites is easily achieved with interpreted extension languages. The *send* command provides the basis for

remote interpretation in Tcl (Welch, 1999). By combining the debugging primitives of Luxdbg with communications primitives such as send, extension languages provide a powerful base for distributed debugging research and application.

8. References

- Auguston, M. 1994. A Language for Debugging Automation. *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, pp. 108-115.
- Beazley, D. 2000. *Python Essential Reference*. Indianapolis, IN: New Riders.
- Bhaskar, J. 1998. *A VHDL Primer*, Third Edition. Upper Saddle River, NJ: Prentice Hall.
- Bruegge, B. and Hibbard, P. 1983. Generalized Path Expressions: A High-Level Debugging Mechanism. *The Journal of Systems and Software*, Elsevier, Volume 3, pp. 265-276.
- Cmelik, R. and Keppel, D. 1993. Shade: A Fast Instruction-Set Simulator for Execution Profiling. University of Washington Technical Report UWCSE 93-06-06.
- Ducassé, M. 1999. Coca: An automated Debugger for C. *Proceedings of the 21st International Conference on Software Engineering*, ACM Press, pp. 504-513.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Golan, M. and Hanson, D. R. 1993. DUEL—a very high-level debugging language. *Proceedings of the Winter 1993 USENIX Annual Technical Conference*, San Diego, CA, pp. 107-117.
- Hanson, D. R. and Raghavachari, M. 1996. A machine-independent debugger. *Software—Practice and Experience*, 26(11), pp. 1277-1299.
- Hanson, D. R. and Korn, J. L. A Simple and Extensible Graphical Debugger. *Proceedings of the*

- USENIX 1997 Annual Technical Conference*, USENIX, Anaheim, CA.
- I-Logix. 2000. I-Logix home page, <http://www.ilogix.com>.
- Jeremiassen, T. 2000. Sleipnir - An Instruction Level Simulator Generator. *Proceedings of the 2000 International Conference on Computer Design*, IEEE Computer Society, Austin, TX.
- Keppel, D. 1993. Fast Data Breakpoints. University of Washington Technical Report UWCSE 93-04-06.
- Kessler, P. 1990. Fast Breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, NY.
- Lucent Technologies. 1998. *DSP16000 LUxWORKS Debugger*, Luxdbg Version 1.7.0.
- Parson, D., Beatty, P., Schlieder, B. 1997. A Tcl-based Self-configuring Embedded System Debugger. *The Fifth Annual Tcl/Tk Workshop '97 Proceedings*, USENIX, Boston, MA, pp. 131-138.
- Parson, D., Beatty, P., Glossner, J., Schlieder, B. 1999. A Framework for Simulating Heterogeneous Virtual Processors. *Proceedings of the 32nd Annual Simulation Symposium*, IEEE Computer Society / Society for Computer Simulation International, San Diego, CA, pp. 58-67.
- Parson, D. 1999. Using Java Reflection to Automate Extension Language Parsing. *Second Conference on Domain-Specific Languages Proceedings*, USENIX, Austin, Texas, pp. 67-80.
- Parson, D., Herrera-Bendezú, L., Vollmer, J. 2000. Distributed Source Code Debugging for Embedded Systems. *Proceedings of the 2000 International Conference on Parallel and*

- Distributed Processing Techniques and Applications*, CSREA (Computer Science Research, Education and Applications) Press, Las Vegas, Nevada, Vol. V, pp. 2409-2415.
- Project Technology. 2000. Project Technology home page, <http://www.projtech.com>.
- Ramsey, N. and Hanson, D. R. 1992. A Retargetable Debugger. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 27(7):22-31.
- Rational. 2000. Rational home page, <http://www.rational.com>.
- Sun Microsystems. 2000. Java Platform Debugger Architecture, <http://java.sun.com/products/jpda/>.
- University of Kansas. 2000. SmartGDB project, <http://hegel.ittc.ukans.edu/projects/smartgdb/>.
- van Rossum, G. 1995. *Extending and embedding the Python interpreter*. Amsterdam: Stichting Mathematisch Centrum, 1995, also at <http://www.python.org/doc/ext/ext.html>.
- Wahbe, R., Lucco, S. and Graham, S. 1993. Practical Data Breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN'93 Symposium on Programming Language Design and Implementation*, Albuquerque, NM.
- Welch, B. 1999. *Practical Programming in Tcl and Tk*, Third Edition. Upper Saddle River, NJ: Prentice Hall PTR.
- Winterbottom, P. 1994. ACID: A debugger built from a language. *Proceedings of the Winter 1994 USENIX Annual Technical Conference*, San Francisco, CA, pp. 211-222.

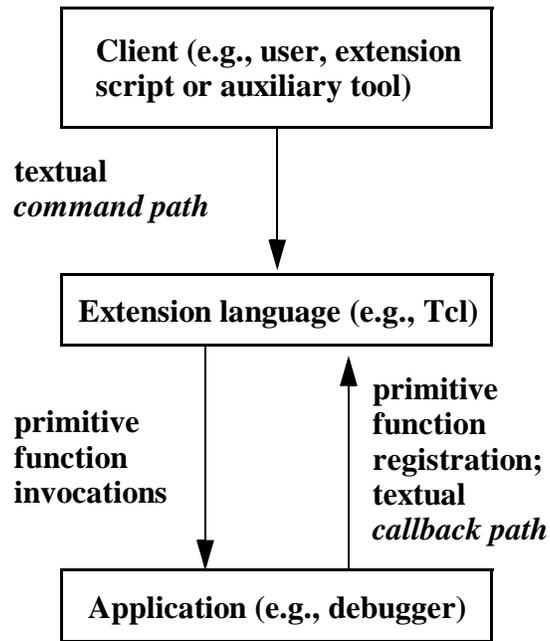


Figure 1: Extension language interactions

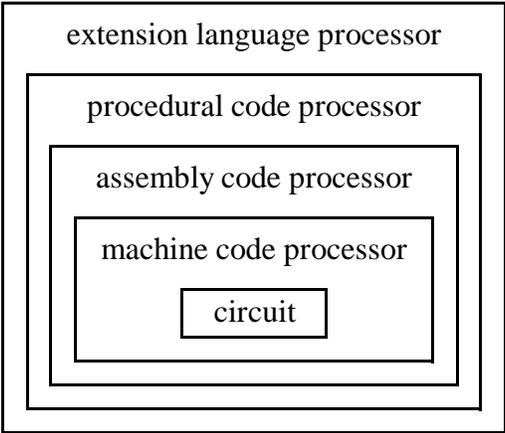


Figure 2: Layers of Luxdbg virtual machines

```
proc logRegisters { } {
    foreach reg [? R @x] { # query all registers in hex
        puts $reg # write to stdout
    }
}

logRegisters
a0 0x00000005 s 32
a0g 0x00 s 8
...
y1 0x00ff s 16
```

Listing 1: A processor-neutral procedure to log registers

```
set setCounter 0 ; # initialization
proc setCount args {
    global setCounter
    incr setCounter ; # bump counter
    resume ; # continue target processor execution
}
```

Listing 2: Recording a target invocation count

```
#include "shlib.h"

.rsect ".text"
.align
function __printf(
    // IN: *(sp) = format string
    // IN: *(sp+N) = other arguments (if used)
    // OUT: a0= output length on success, else -1
) {
    i = _SP_FUN_CODE(_SHLIB_PRINTF)
    icall 0
    return
}
```

Listing 3: Printf hook in a DSP16000 target

```

proc handle_printf {} {
    set top [sp @rd]; # arguments are on stack
    a0=-1 ; # default return value into register a0
    set sFormat [readstring [list ymem \
        [readlong "ymem $top"]]]
    incr top 2; set skipme 1; set formatargs ""
    foreach arg [split $sFormat %] {
        ... (Implementation not shown)
    }
    set output [eval format \${sFormat} $formatargs]
    send_to_stdout $output
    a0=[string length $output]
}

```

Listing 4: Printf implementation snippet in Tcl

```
proc scheduler {processors N}{ # command path
  for {set i 0} {$i < $N} {incr i} {
    foreach p $processors {
      $p resume
    }
  }
}
```

Listing 5: Round-robin, cooperative scheduler

```

proc scheduler { startproc } { # command path
    global nextproc
    $startproc resume
    while { $nextproc != "" } {
        # terminate on empty processor name
        $nextproc resume
    }
}
# output callback is attached to each processor's
# output breakpoint
proc output_callback args { # callback path
    global nextproc neighbor
    set nextproc $neighbor([processor name])
    if { $nextproc != "" } {
        # copy my outport to neighbor's inport
        $nextproc fxpr inport = [fxpr outport]
    }
    # no resume means break to scheduler
}

```

Listing 6: IO event-driven scheduler

Table 1: Effects of conditional breakpoint mechanisms

condition	hardware execution		simulated processor	
	time per loop	N iterations	time per loop	N iterations
no breakpoint	1.9 microsec.	10^7	415.8 microsec.	10^4
target patch	2.7 microsec.	10^7	624.9 microsec.	10^4
debugger test	309,592.5 microsec.	10^2	605.6 microsec.	10^4
Tcl test	309,897.5 microsec.	10^2	792.9 microsec.	10^4

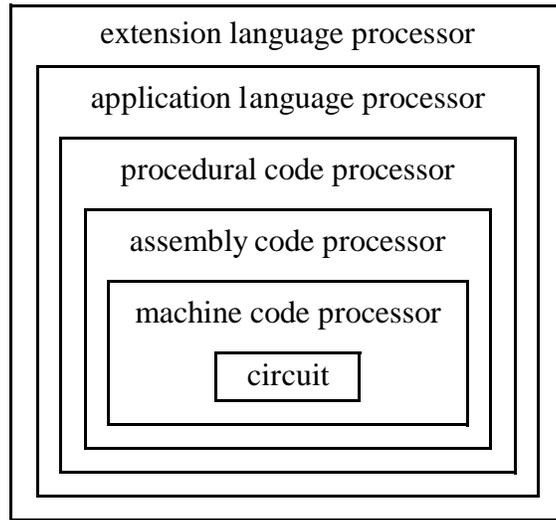


Figure 3: Application virtual machine layer

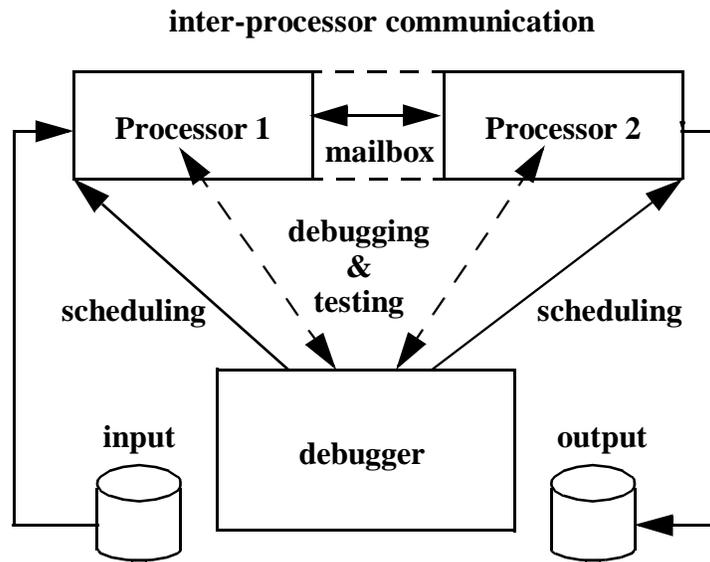


Figure 4: A signal processing application virtual machine

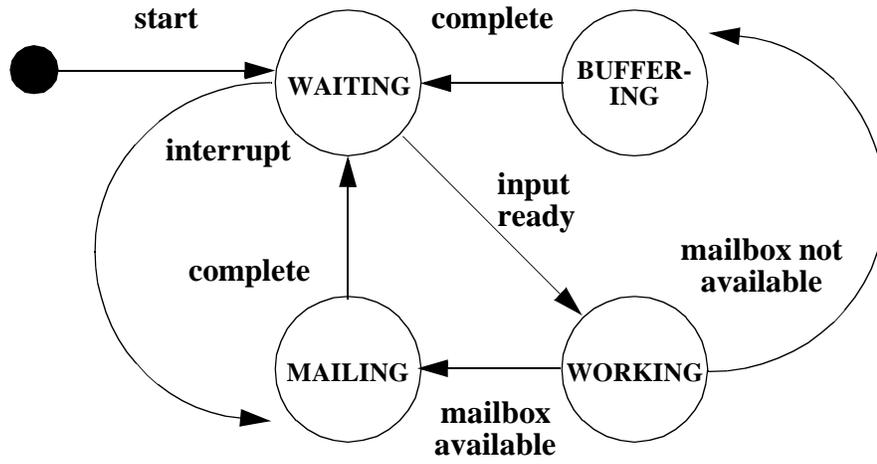


Figure 5: State transition diagram for Processor 1 — an application virtual machine

monitor mailbox									
input_ready	0	p1state	MAILING	mboxowner	p1	p2state	WAITING	output_ready	0
input_data	2	p1peak	2013	mboxpercent	18.8	p2peak	32040	output_data	28416
time1	110752	p1avg	1993	runner	p1	p2avg	32017	time2	98927
		bufpercent	0.0						
		buffermax	12.5						

Figure 6: An application level view via Tk

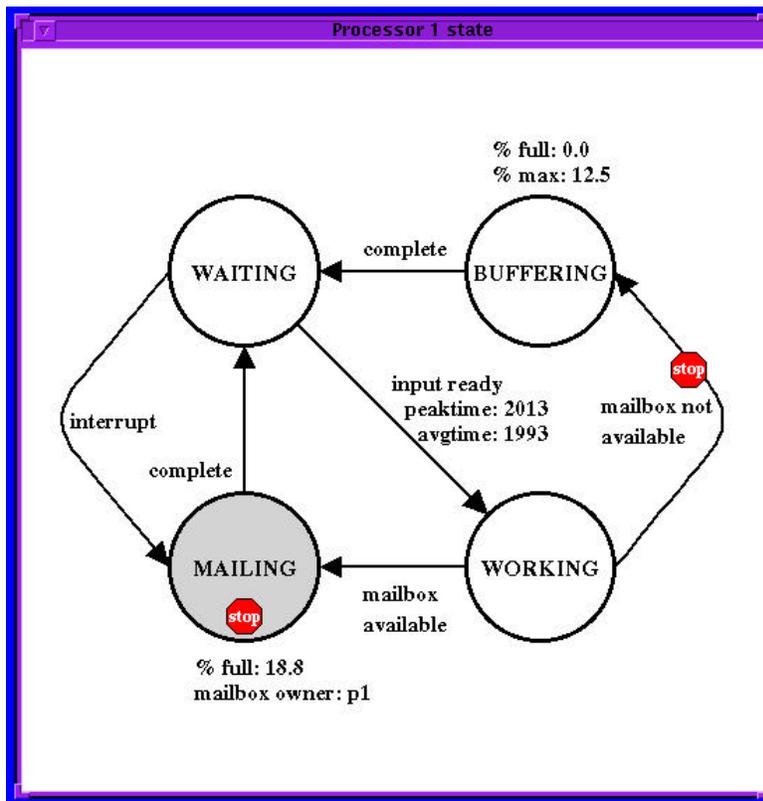


Figure 7: An interactive debugging window for Processor 1's application-level state machine

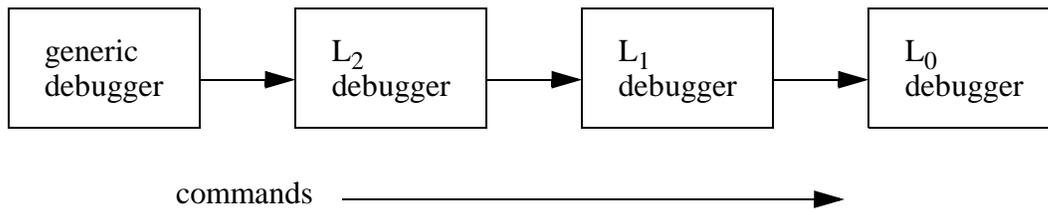


Figure 8: Chain of Responsibility delegates commands among layers of machine abstraction

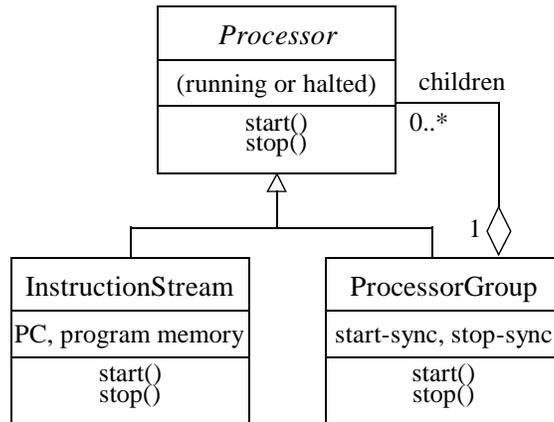


Figure 9: Classes of Luxdbg processor containment as the Composite Design Pattern

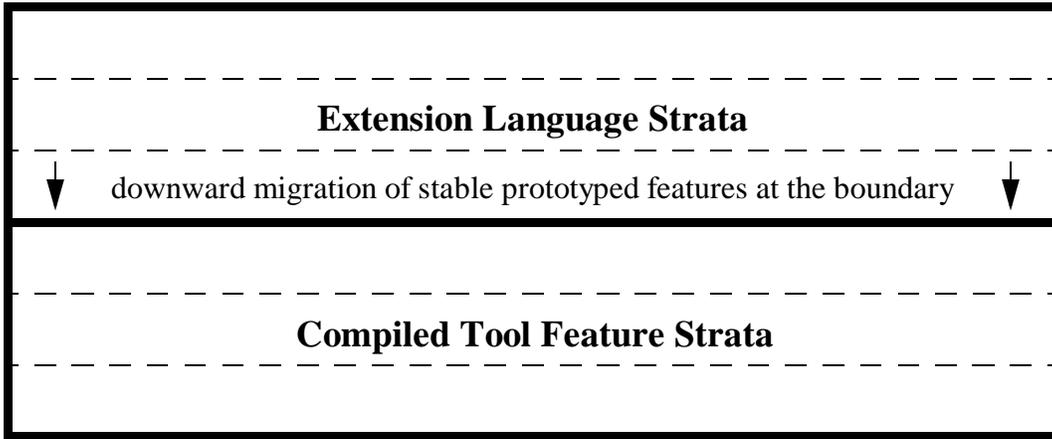


Figure 10: Prototyping as downward migration of boundary extension language strata