A Framework for Simulating Heterogeneous Virtual Processors

Dale Parson
Lucent Technologies
Allentown, Pa. 18103
dparson@lucent.com

Paul Beatty
Lucent Technologies
Allentown, Pa. 18103
pebeatty@lucent.com

John Glossner
Lucent Technologies
Allentown, Pa. 18103
glossner@lucent.com

Bryan Schlieder
Lucent Technologies
Allentown, Pa. 18103
bryanschlieder@lucent.com

Abstract

This paper examines the layered software modules of a heterogeneous multiprocessor simulator and debugger, and the design patterns that span these modules. Lucent's LUxWORKS simulator and debugger works with multiple processor architectures. Its modeling infrastructure, processor models, processor monitor / control, hardware control, vendor simulator interface and Tcl/Tk extension layers are spanned by the following design patterns: 1) build and extend abstract virtual processors, 2) build reflective entities, and 3) build a covariant extensible system. Together these modules and patterns define a processor execution architecture that encourages reuse and dynamic extensibility.

1. Introduction

Our group within Bell Labs provides software tools for cosimulation of embedded hardware and software systems [1], and for the debugging of distributed applications running on simulated and actual hardware, in communication systems using Lucent digital signal processors and assorted microcontrollers. Since 1995 we have been designing and building a software framework that addresses the needs of the following groups of users:

- Processor architects who use simulation to evaluate prospective processors.
- IC designers who combine processor cores, custom input-output circuitry and interprocessor communication circuitry into custom integrated circuits.
- System developers who build and debug distributed and embedded applications.
- Software tools engineers who build processor models and develop new tools features.
- Field application engineers who need to prototype custom circuitry and custom tools features rapidly by using the framework's Tcl extension language [2,3].

Lucent Technologies markets the current product implementation of this framework as *luxdbg*—the LUxWORKS simulator / debugger for Lucent embedded processors [4,5]. Luxdbg satisfies the definition of a framework as "a system that can be customized, specialized, or extended to provide more specific, more appropriate, or slightly different capabilities." [6] The framework solves two problems often found in other processor simulators. First, processor simulator architectures usually over-couple a specific processor's simulation model to a simulator and debugger. Over-coupling yields a simulator that can control simulations of only one processor architecture. Over-coupling also yields a debugger that can debug programs for only one architecture. When a simulator / debugger must support a new processor architecture, the only form of reuse available consists of ad hoc, cut-and-paste source code hacking. This hacking produces yet another processor-specific simulator / debugger. Subsequent maintenance of cloned, near-duplicate copies of code requires duplication of effort. We call this *the reuse problem*.

The second problem is *the extensibility problem*. In addition to new instructions, a new processor may add new features never before seen in supported processor architectures. A Harvard memory architecture, which separates program memory from data memory, is one example. Hardware support for software multi-threading is another. New processor dimensions such as these do not fit the cut-and-paste style of simulator / debugger evolution. There may be no place to put new feature dimensions in a static, hard-coded simulator / debugger. New processor dimensions require modular extension interfaces.

We have solved the reuse and extensibility problems for multiple processor architectures by designing a simulator / debugger that configures itself to

heterogeneous multiprocessors at run time. Luxdbg builds on the abstraction of a *virtual machine* [1, p. 15-17]. This framework is structured as a set of simulation, debugging, profiling and hardware-monitor-control modules that operate on an abstract virtual processor. A processor designer refines the definition of an abstract virtual processor into a concrete processor via C++ inheritance and composition. When the designer has added these refinements, that designer is rewarded with a powerful simulator, profiler, graphical debugger, and interfaces to vendor circuit simulation frameworks.

This paper unfolds as a series of examinations of luxdbg's framework *modules* and *design patterns*. The layered modules provide the major capabilities of the system. The design patterns provide the rules for connecting modules and designing new modules. These patterns solve the reuse and extensibility problems. Without the patterns this framework would become gridlocked and brittle over time.

We treat luxdbg as a matrix of modules and design patterns. Section 2 lays out the horizontal strata of modules and their interactions within luxdbg. Sections 3 through 5 explore design patterns as vertical spans of the strata. Section 6 summarizes our results and examines future directions.

2. Luxdbg framework overview

Figure 1 shows the major modules and their interactions within the luxdbg framework. Down arrows and right arrows indicate flow-of-control in client-to-server transactions. Labeled up arrows indicate server-to-client callbacks. Numbers give cardinality of associations of instantiated objects, where "0..1" signifies zero or one object and "*" signifies zero or more objects. This section starts at the bottom of Figure 1 and works its way up.

Modeling infrastructure is a C++ class library. It provides base classes that support the abstract virtual processor pattern through inheritance. It also provides utility classes for building processor signals, registers, memory, IO ports and monitor / driver probes through composition. A processor architect or other modeler builds a *processor model* by using the classes of modeling infrastructure, along with C or C++ operations and other libraries. A specific processor is not a hard-coded part of the framework.

Luxdbg processors are *reflective*. Reflection is a mechanism that allows client code to query a processor at run time to discover the identity of state-bearing infrastructure objects such as registers within the processor. The client can then retrieve and set values for these objects. *Processor monitor / control* is a class library

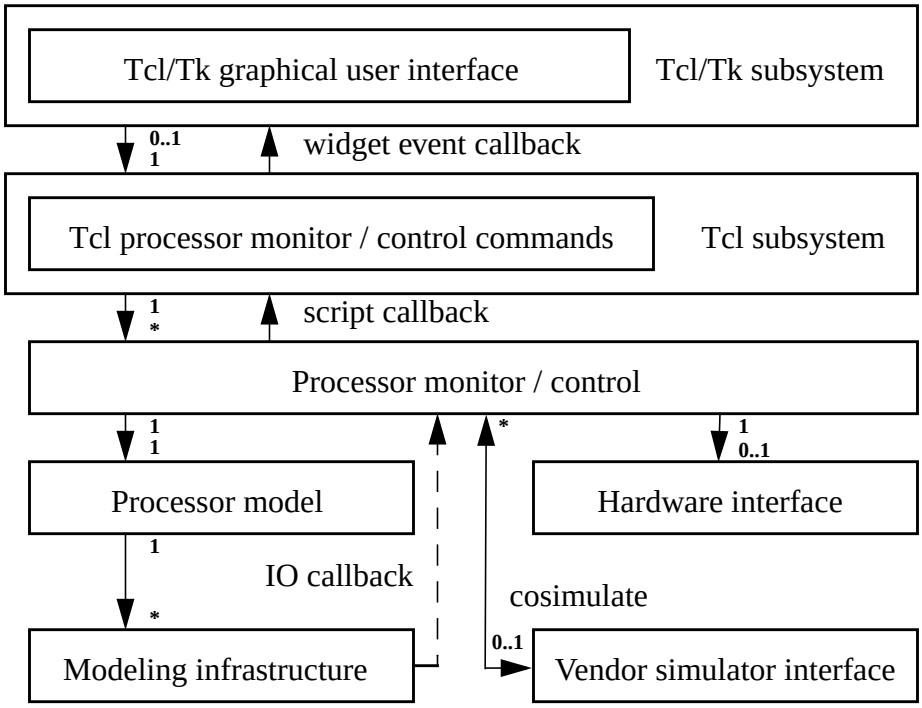

Figure 1: Luxdbg simulation and debugging architecture

for simulating, debugging and profiling processor objects through the reflective processor query interface.

A processor model acts both as a *simulation state machine* and as a *database*. As a state machine it simulates processor execution. As a database it stores state information for its query interface. Processor monitor / control can supervise execution of real processor hardware via the *hardware interface* module. When a hardware processor reaches a breakpoint, monitor / control uploads hardware state into a model. A user can then interact with state by interacting with its model. When the user has completed interaction, monitor / control downloads state from the model to hardware and resumes hardware execution.

The *Tcl subsystem* provides luxdbg's interpreted extension language [2,3]. Tcl supports inclusion of application-specific primitive commands written in C or C++. Luxdbg's *Tcl processor monitor / control commands* provide a textual command layer on top of processor monitor / control. Users, Tcl extension scripts, and auxiliary tools use Tcl as a query language for interaction with processors via monitor / control. Processor events can trigger recursive callbacks into Tcl for nested script evaluation.

The *Tcl/Tk subsystem* is a separate process that houses our *Tcl/Tk graphical user interface* (GUI). This optional client tool uses Tcl queries to communicate with luxdbg's main process. It provides a source debugging window for each processor instance in a luxdbg session, as well as optional register, memory, breakpoint and data watch windows. A watch window can use user-supplied Tcl expressions to provide derived views of processor data.

The *vendor simulator interface* is an optional procedural interface for integrating luxdbg models into vendor simulation frameworks such as Model Technology's VHDL circuit simulator or Math Works' Simulink® arithmetic function simulator. Luxdbg models can run as components within these frameworks. Luxdbg serves as a loader, debugger and model extender. Users employ the capabilities and model libraries of these environments while retaining luxdbg for interaction with Lucent processors. The vendor simulator advances model state via this interface.

Luxdbg's normal flow-of-control goes down from layered client modules to their supporting server modules as shown in Figure 1. Clients invoke servers. Servers do not contain hard-coded dependencies on specific clients. A client may request a *callback* from a server when the server detects an event. The *IO callback* of Figure 1, for example, occurs when a read from an input port object or a write to an output port object takes place within a model. These port objects are from modeling infrastructure. If monitor / control has registered itself for a port callback, then an input port will fetch a value from monitor / control via the callback, and an output port will send a value to monitor / control via the callback. In this way monitor / control can route processor IO events to file sources and sinks opened by Tcl commands, or up a *script callback* to a Tcl extension procedure. IO callbacks eliminate the need to model all connecting circuitry. Port infrastructure objects act as generic circuitry, and attached custom files or Tcl procedures manipulate input-output values at a high level. Embedded application developers can concentrate on programs while ignoring the details of connecting circuitry.

3. Design pattern: Build and extend abstract virtual processors

This design pattern is the main element of our solution to *the reuse problem*. Luxdbg has distilled properties that are common to all instruction stream processors, and placed them into a set of virtual processor base classes. This pattern solves problems that are *common* to all processors. Subsequent patterns solve problems that are *not-common*, i.e., problems whose solutions vary as processor architectures vary.

3.1 Virtual processor infrastructure

Figure 2 shows the layers of processor models that a modeler can build from the modules of Figure 1. Each processor appears as a nested set of machine definitions. Luxdbg users have interactive access to these processor layers.

A processor at the lowest level is a *circuit machine*—a collection of signals, registers, memory cells, and transition rules for advancing processor state. The *modeling infrastructure* module of Figure 1 supplies building blocks for constructing a circuit machine. At the next level is the *machine code processor* or *instruction stream machine*. It fetches, decodes and executes a series of programmed binary instructions. A *processor model* of Figure 1 is a machine code instruction stream processor. Next, *assembly code* and *procedural code processors* result from augmenting a bare machine code processor with symbolic programs. *Processor monitor / control* of Figure 1 contains loader and symbol table objects that implement these symbolic machine layers. At the top, the *Tcl subsystem* of Figure 1 supports the *Tcl extended processor.* A user or tool can invoke Tcl procedures that extend and provide views into lower-level machines.

Each processor *instance name* becomes a Tcl command prefix that sets the *current processor* to the named processor for the duration of that Tcl command.

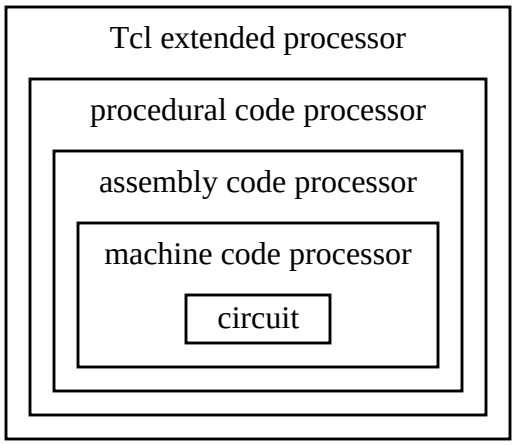

Figure 2: Layers of luxdbg virtual machines

For example, in command “p1 load myprog.e,” “p1” is a processor instance into which luxdbg loads program “myprog.e.” The Tcl command may be a Tcl extension procedure that temporarily changes the current processor to a different named processor. For example, “p1 r0 = [p2 r1]” retrieves the value of register “r1” from processor “p2” and stores it into register “r0” of processor “p1.” When the instance name prefix is missing from a command, that command uses the current processor already in effect. A user or tool can set the current processor at a top level, and all commands and scripts that do not temporarily override this processor, will simply operate on this processor. In this way generic scripts that apply to multiple, possibly heterogeneous processor instances become possible. Commands that explicitly name processor instances can achieve interprocessor communication.

The remainder of this section focuses on abstract processor base classes from the infrastructure library that support the machine code virtual processor abstraction. Later sections on reflection and covariant extension show how luxdbg supports the outer layers of Figure 2.

Figure 3 shows two luxdbg virtual processor modeling hierarchies. The left side shows a class *inheritance hierarchy.* Derived class InterfaceProcessor inherits from Circuit, and class DriverProcessor inherits from InterfaceProcessor. Circuit, InterfaceProcessor and DriverProcessor are *abstract classes* from the infrastructure library. They specify behavior and provide machinery for reuse by processor-specific *concrete classes*. Figure 3 shows the place of processor-specific concrete classes.

The right side of Figure 3 shows a processor object *containment hierarchy.* A concrete DriverProcessor object contains nested concrete InterfaceProcessor and Circuit objects. C++ composition of modeling objects into a containment hierarchy supports building block-based construction of processor and multiprocessor models. Hierarchical composition is similar to netlist construction in circuit design languages such as VHDL [7]. Luxdbg’s C++ model composition is similar to structural VHDL composition in the interconnection of circuit building blocks.

Circuit represents an executable block of circuitry. A modeler defines a block of circuitry by deriving a circuit-specific class from Circuit and implementing the circuit’s *eval()* function. Eval() reads circuit-specific input and state and writes circuit-specific output and state. Detailed representation of input, state, and output are left up to the modeler. They may be simple C integers. A Circuit client calls eval() after changing circuit inputs or state variables.

InterfaceProcessor represents an instruction stream processor. A modeler defines a specific instruction set by deriving an instruction-specific class from InterfaceProcessor. InterfaceProcessor adds three main mechanisms to those inherited from Circuit. First, an instruction-specific processor always contains a register that the processor identifies as its *instruction pointer* (IP, also known as *program counter*). With the IP comes identification of a *program memory.* This memory holds an executable stored program, and the IP gives the location of the next instruction to begin execution.

Second, class InterfaceProcessor contains a *query table* and C++ query interface into it. The query interface supports *reflection*. A modeler connects model state within a specific InterfaceProcessor or contained Circuit—signals, registers, pins and memory—to the query interface by attaching named probes provided by the infrastructure library to the contained state-bearing objects. The simplest state-bearing object is a C integer. In

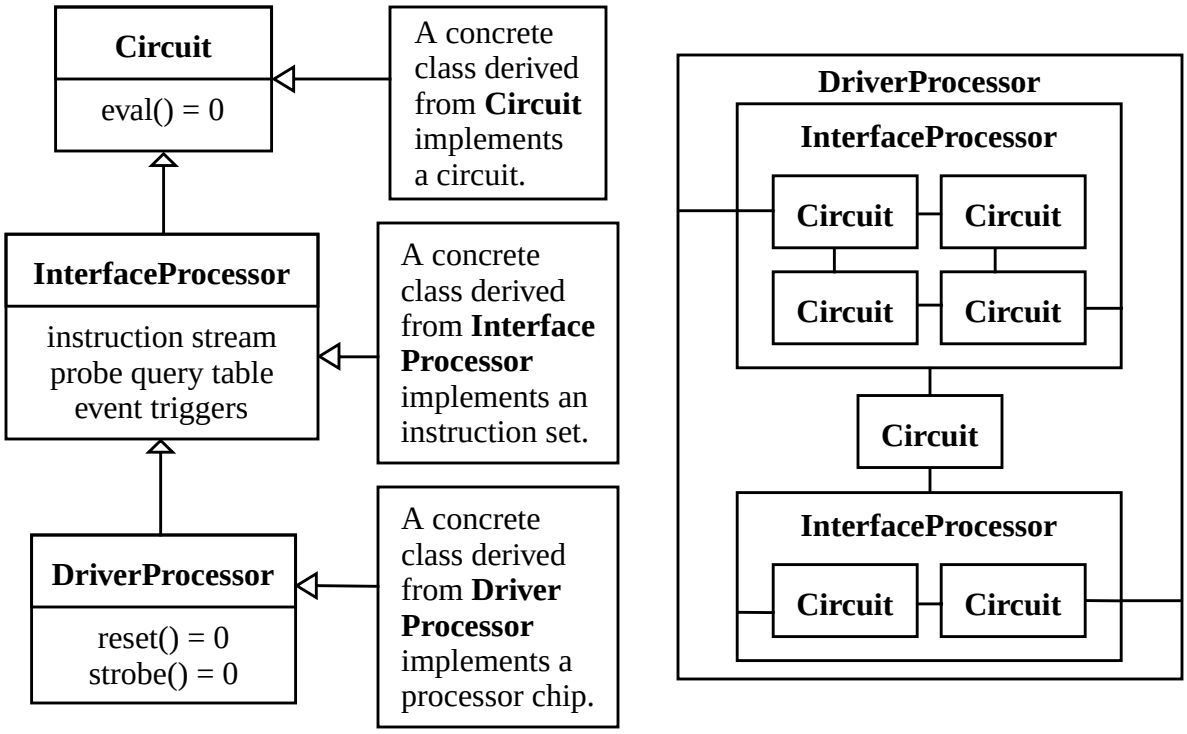

Figure 3: Virtual machine inheritance hierarchy (left) and containment hierarchy (right)

the query table the probe provides a symbolic name for its state-bearing object (e.g., "r0"), type information (e.g., register, input pin, memory), and methods for client read and write access. The query table also collects exception messages from within an InterfaceProcessor for user communication and methodical exception handling.

Third, class InterfaceProcessor provides an *event trigger interface* for simulation control, profiling and debugger breakpoints. Method "virtual eventHandle triggerEvent(unsigned long type, long count, long *triggers)" provides basic breakpoint capability. InterfaceProcessor supports a set of predefined event *types* such as TRIGGER_IP, which triggers an event on an instruction pointer value. InterfaceProcessor also implements IP range events for entering or leaving a section of program, as well as data access events. Array *triggers* supplies actual values for testing and *count* supplies their count. A specific class derived from InterfaceProcessor can redefine triggerEvent() to handle event types supported by a custom model or hardware processor.

DriverProcessor represents a *uniprocessor* or *multiprocessor chip*. In clock-based models DriverProcessor controls global clock and reset logic. DriverProcessor specifies abstract function *reset()* that resets a chip to its starting state, and abstract function *strobe()* that advances the chip one state quantum. A modeler defines a specific processor chip by deriving a chip-specific class from DriverProcessor that implements reset() and strobe(). For a simple uniprocessor instruction model, a modeler can derive an instruction-set model directly from DriverProcessor. The modeler codes the eval(), reset() and strobe() functions and attaches probes without using any hierarchical Circuit or InterfaceProcessor objects.

The right half of Figure 3 illustrates a more complex structure for a multiprocessor IC (a so-called *superchip* or *system on a chip*). Each InterfaceProcessor is really an instruction-set processor derived from InterfaceProcessor. They need not implement the same instruction set. Figure 3 also shows some connecting circuitry outside of these processor *cores* but within the superchip. Each InterfaceProcessor houses a query table for its respective internal circuitry probes. The outer DriverProcessor, by virtue of also being an InterfaceProcessor, also houses a query table. The nested InterfaceProcessors appear in the query table of the outer DriverProcessor.

Each DriverProcessor object is a Tcl-accessible processor instance. Nested InterfaceProcessors receive hierarchical names starting with the outer processor's Tcl name. Thus if a user constructs a superchip model called *p1* that contains two nested processors *dsp* and *uC*, the

user can access the dsp by using name *p1.dsp* and the microcontroller by using name *p1.uC*. Each InterfaceProcessor provides its own hierarchical scope for named probes.

The virtual processor infrastructure classes solve common problems so that modelers do not need to reinvent methods for circuit modeling, instruction stream debugging and multiprocessor synchronization. Modelers can concentrate on unique architectural issues like instruction set design, memory organization, interrupt handling, etc. Modelers do not need to resort to source code cut and paste or other ad hoc coding techniques to implement a simulator or debugger to control their models. Debugger "hooks" are an integral part of the infrastructure itself.

3.2 Multithreaded processor example

To explore the efficiency of various digital signal processor architectures, we have used luxdbg to experiment with both instruction set design and implementation alternatives. Figure 4 shows an example model organization.

A DriverProcessor encapsulates a number of nested InterfaceProcessors, each of concrete class *Context* ("ctx" in Figure 4). A Context is a hardware-supported thread unit. Each Context object supplies an application program instruction stream to shared, downstream processing circuits. A Context does not include shared circuits such as execution units, global instruction schedulers, global memory, or the control unit.

This model's Memory and Register classes derive from infrastructure library classes infraMemory and

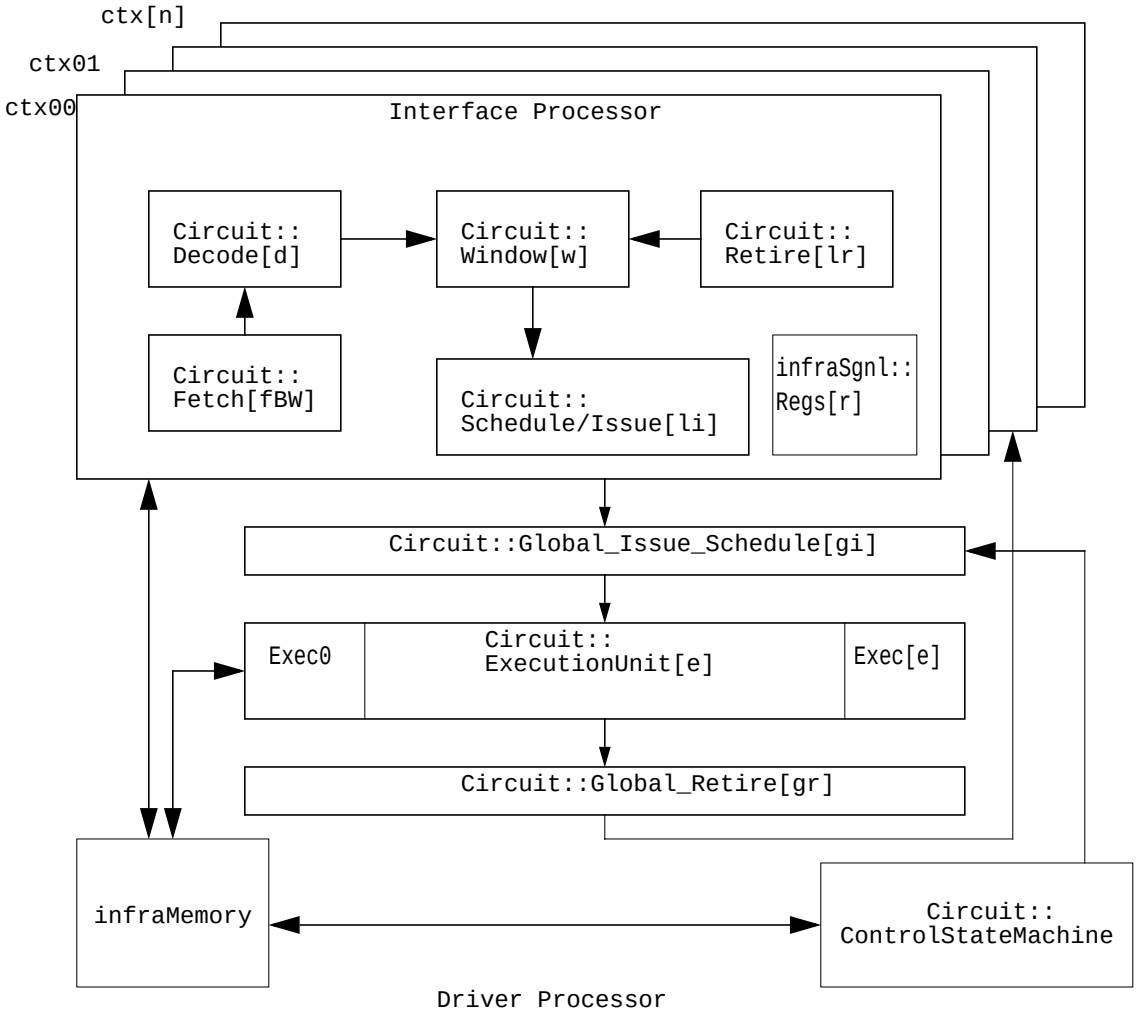

Figure 4: Multithreaded processor example

infraSgnl, respectively. These library classes provide memory probes, memory access breakpoint triggers and arbitrary-width signals. By using the built-in classes, a luxdbg user can set breakpoints, view contents and modify values. In addition, a probe connects to each architecturally visible register. This processor model's classes are template-parameterized to allow for easy porting to alternative register implementations. For example, `class ProbedRegisterFile <REG, PROBE> becomes ProbedRegisterFile <infraSgnl, infraSgnlMnDr>` via C++ template instantiation. The templates allow the register file's signal base type (infraSgnl here) and probe type (infraSgnlMnDr here) to *covary*, providing alternative implementation types without recoding. We use infraSgnl here because it supports generic arithmetic on bit-fields of variable width. This model contains 64-bit datapaths, and infraSgnl provides arithmetic on data types not present in most C++ compilers. Using infraSgnl produces some simulation overhead, but it provides for architectural design flexibility and a reference implementation of the architecture that both the hardware and fast instruction set model can use for verification. After simulation has provided sufficient architectural results to substantiate the efficiency of an architecture, the same implementation classes can be reused to implement a fast integer-based register file simply by changing the register implementation type and probe type of the template parameters.

To further demonstrate the flexibility of the built-in classes, we have derived a memory class from infraMemory that allows memory to be treated as a stream. This has proved useful for loading programs, transferring memory regions, and other house keeping functions.

In addition to extending the built-in classes through inheritance, we have designed these processor models to be dynamically configurable. For example, the simulator reads a configuration file during model construction that provides the number of Contexts, the number of instructions that may issue per Context, the latency of the execution units, the number of execution units, the number of instructions that may be retired globally and locally per cycle, the type of scheduler to be instantiated, etc. The generality of our processor models allows us to model processors as simple as a single-threaded single-issue controller through a full simultaneous multithreaded processor.

This complex example shows that there are many ways that we have attacked the reuse problem. Infrastructure signals, circuit probes and memory objects provide reusable building blocks. InterfaceProcessor provides means for loading and debugging a program running from a Context. DriverProcessor provides coordination that requires only connection of contained parts to manage global timing and circuit interaction. Every step of model design finds an assortment of reuse opportunities.

4. Design pattern: Build reflective entities

The last section showed how abstract processor infrastructure supports the *common* aspects of processor modeling. This section on *reflection* [8] examines a pattern that assists in capturing *not-common* aspects. Reflection is a mechanism whereby a processor-neutral tool, such as a simulator / debugger, can query a processor model about its unique elements. Query-based configuration supports reuse of self-configuring tools, thereby attacking *the extensibility problem*.

Reflective software entities supply information about their custom abilities to run-time clients. Clients can adopt custom behaviors after querying reflective entities. Section 3 discussed the InterfaceProcessor query tables that give access to state-bearing objects within a model. Infrastructure probes can attach to the most elementary circuit signals. The monitor / control module of Figure 1 feeds this access through to the Tcl subsystem. A user or tool can query a machine code processor to discover its registers, pins, signals and memory, assorted properties such as width, memory depth, and register arithmetic type (signed, unsigned, or floating-point), linear memory hierarchies (e.g., sequences of ROM and RAM within a program memory), disjoint memory hierarchies (e.g., separate program memory and data memory in a Harvard Architecture machine), IO port identities, and finally, state values. InterfaceProcessor provides reflection into the defining characteristics of a custom processor. There is no need for a modeler to write custom code in support of a debugger's interaction with a processor model.

The assembly and procedural code processor layers need additional symbol tables for entities found in assembly and procedural programs loaded into a processor. When luxdbg loads a program into an InterfaceProcessor, the loader defines a new virtual machine around that processor. The *processor monitor / control* module supports this level of virtual machine.

Figure 5 gives an overview of the main monitor / control classes. *MonitorControl* is a client of *InterfaceProcessor* and *ProgramSymbols*. There is one MonitorControl object for each InterfaceProcessor object, and when an assembly or procedural program is loaded, there is a ProgramSymbols object as well.

For a simple machine code processor, MonitorControl could direct all processor access and control to InterfaceProcessor and DriverProcessor mechanisms

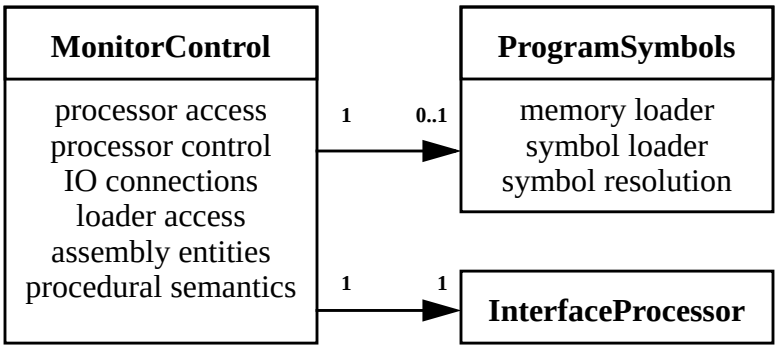

Figure 5: An overview of processor monitor / control classes

discussed in Section 3. MonitorControl maintains tables of breakpoints and profile triggers set by the Tcl client. It uses InterfaceProcessor trigger methods to relay these to its processor model. MonitorControl also supports nested *callbacks* to Tcl on processor *IO events*, processor *breakpoints* and processor *exceptions*. Tcl commands can set callbacks on any of these events, and Tcl can recursively query a model from within a callback procedure. MonitorControl records all trigger-callback pairs in an event table. All of these tables—break triggers, profiling, and IO, breakpoint and exception callbacks to Tcl—are available for query from the Tcl interface. Tcl commands can query for the existence and values of any virtual machine extensions they make via MonitorControl.

ProgramSymbols come into play when a symbolic program is loaded. A ProgramSymbols object includes code for loading memory and symbol table contents from an object file. MonitorControl transfers memory contents into an InterfaceProcessor's memory, and symbol information remains in its ProgramSymbols object. User and GUI requests concerning symbol scope (e.g., set the scope of symbol lookup to C function "foo()"), symbol attributes (e.g., what is the type of symbol "foo" in this scope), and symbol location (e.g., address in data memory of C integer "foo") go through MonitorControl to a ProgramSymbols object. When MonitorControl needs to examine or store a program variable's value, it resolves the variable to a storage or register location by consulting ProgramSymbols, then it accesses the machine storage or register via InterfaceProcessor. ProgramSymbols adds the assembly and procedural reflection layers by making the symbol information of a loaded object file fully visible to MonitorControl clients such as Tcl. Monitor / control's debugging commands configure themselves to a specific procedural program by consulting the procedural symbol bindings of ProgramSymbols.

Finally, both Tcl and Tcl/Tk are reflective. Just about anything that can be contained or added to Tcl/Tk is available for query by built-in commands such as "info" [2,3].

The *graphical user interface* of Figure 1 gives one example of the power of reflection. The GUI has no built-in knowledge of any processor's registers, memory configuration, or IO capabilities. Instead it uses Tcl commands to query processor configuration information, then it configures widgets accordingly. The GUI adapts itself to each processor at run time.

The *vendor simulator interface* of Figure 1 provides another example of the power of reflection for integrating models into both behavioral and algorithmic simulations. A *behavioral simulation* models circuit behavior [7]. Behavioral simulation requires that pin information for a processor must be represented and translated to and from an external vendor simulator's data format. In Figure 6 the vendor simulator interface attaches a luxdbg DSP16k pin-level model to a behavioral simulation at the circuit level. *Algorithmic simulation* is a much higher level of modeling. The modeler runs a complete algorithm (e.g., a C function) on a target processor to determine the functional behavior of that processor in a system that uses that algorithm. A modeler porting a floating-point algorithm for fast execution on a fixed-point digital signal processor, for example, might run a ported fixed-point function on a DSP model to determine the effects of fixed-point implementation. In Figure 6 the vendor simulation interface attaches a Fast Fourier Transform function (fft) in a block-diagram simulator to a fixed-point fft library function loaded into a luxdbg processor model at the procedural level.

Reflection assists integration into both types of simulation environments. For behavioral simulation, the vendor simulator interface uses InterfaceProcessor's query table to obtain pin information for a processor. This interface then maps that information to pin structures in the vendor simulator. The vendor simulator interface is a query-based *bridge* between luxdbg debugging and vendor

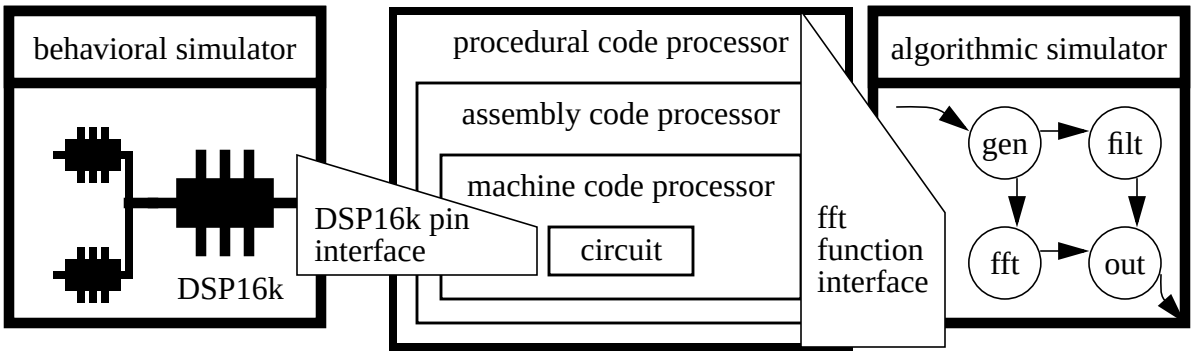

Figure 6: Vendor behavioral and algorithmic simulators attach via the vendor interface

circuit simulation.

For algorithmic simulation, the vendor simulator interface queries ProgramSymbol information to determine function addresses and parameter types and positions. The algorithmic interface gives a vendor simulator the ability to call functions on a target luxdbg processor without worrying about the processor architecture or compiler calling conventions. Luxdbg reflection allows both LUxWORKS and third party tools to connect easily to our processors.

Luxdbg's GUI interface and vendor simulator interface solve the extensibility problem by querying each processor's custom features. The GUI extends luxdbg by providing an adaptive visual front end. Vendor simulators extend themselves by loading models and luxdbg's extensive debugging capabilities.

5. Design pattern: Build a covariant extensible system

Covariance is a pattern that allows *not-common* processor dimensions to vary together. Object-oriented covariance refers in a narrow sense to the ability of a method's parameters and return types to vary from base to derived types as the class of the method varies from a base to derived type [9]. Classes and member function parameter types *covary.*

Luxdbg's covariance refers to the ability of processors with extended capabilities to accept extended commands. Recall from Section 3 that method InterfaceProcessor::triggerEvent() can accept a number of basic event types such as TRIGGER_IP, and that specific models and hardware processors derived from InterfaceProcessor can extend the range of legitimate triggers. Trigger types covary with processor types.

Luxdbg's most complex covariance relates to Figure 5 of Section 4. In addition to InterfaceProcessor, classes ProgramSymbols and MonitorControl are in fact abstract classes. Distinct ProgramSymbols-derived classes for ELF/DWARF, COFF, Java™ ClassFile, and custom object file formats are possible. ProgramSymbols symbol access methods are evolving to cover the set of language and file format-provided symbol capabilities that we anticipate.

Finally, each new instruction set processor core gets its own MonitorControl-derived class. That class covaries with unique capabilities in that core's models, file format, and embedded debugging hardware. MonitorControl methods called from Tcl interpret their own command-line arguments. Tcl merely passes textual arguments through. Consequently, if a MonitorControl-derived class wishes to extend a Tcl command, it redefines that command's virtual function. For example, if class DSPn derived from MonitorControl monitors a processor with a special counter breakpoint capability—e.g., count every access to data memory in this range and trigger a breakpoint on the 1000th access—then DSPn could redefine the setBreakpoint method to scan command line arguments for a "-count N" command switch. If DSPn::setBreakpoint fails to find its command switch, it calls MonitorControl::setBreakpoint to handle the command using default command parameters.

By using covarying virtual functions, a MonitorControl-derived class can add processor-specific command options without hard-wiring them into MonitorControl. Covariance attacks *the extensibility problem* without disrupting underlying, generic code.

6. Conclusions

This paper has shown some of the intersections of the main design patterns with the layered modules of Figure 1. Focusing on a clean distinction between an abstract virtual processor and a specific processor implementation maximizes the opportunities for reuse. Providing reflection for all levels maximizes the potential for self-

configuring clients that use those levels. Covarying processor-specific derived classes and processor-specific commands gives ready access to processor enhancements without perturbing generic base classes. The result has been ready adaptation to new processor architectures and tool feature requests as they arrive.

The biggest drawback in the current luxdbg implementation is close temporal coupling between the monitor / control and processor model modules of Figure 1. Monitor / control advances simulation state in a model a single step at a time, then it queries the model for a breakpoint event. The hardware interface, on the other hand, installs breakpoints in a remote hardware processor. It allows a hardware processor to run full speed across many instructions until the hardware detects a breakpoint. Only then does the hardware processor return a breakpoint event to monitor / control.

Simulation speed will be accelerated and modularity improved when we port this multi-step, high-speed execution approach from the hardware interface into processor modeling infrastructure. Models will run faster because they can use internal, optimized mechanisms for breakpoint detection. A model will return control to monitor / control only when the model detects a breakpoint. This change will unify monitor / control's view of processor models and processor hardware, simplifying the interfaces beneath monitor / control. Finally, we have plans to reuse this unified multi-step debugger - processor interface to attach luxdbg to remote processes running under operating systems. An operating system process will appear as another flavor of a *virtual processor*. We will organize models, processor hardware and operating system processes behind a Virtual Processor Software Component interface [10], with the upper three layers of Figure 1 partitioned into a Networked Debugger Software Component. Once we have partitioned luxdbg into these two software components, it will be possible to connect the debugger to a virtual processor anywhere on the Internet. We already have this capability for hardware processors, and the unification of the debugger - virtual processor interface will extend it to distributed models and processes.

7. References

1. S. Kumar, J. Aylor, B. Johnson and W. A. Wulf, *The Codesign of Embedded Systems*: *A Unified Hardware/Software Representation*. Boston, MA: Kluwer Academic Publishers, 1996.

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

3. Brent B. Welch, *Practical Programming in Tcl and Tk*, Second Edition. Upper Saddle River, NJ: Prentice Hall, 1997.

4. D. Parson, P. Beatty and B. Schlieder, "A Tcl-based Self-configuring Embedded System Debugger." Berkeley, CA: USENIX, *The Fifth Annual Tcl/Tk Workshop '97 Proceedings*, Boston, MA, July 14-17, 1997, p. 131-138.

5. *DSP16000 LUxWORKS Debugger*, luxdbg Version 1.4, Lucent Technologies, December, 1997.

6. Richard P. Gabriel, *Patterns of Software: Tales from the Software Community.* New York: Oxford University Press, 1996.

7. J. Bhasker, *A VHDL Primer*, Revised Edition. Englewood Cliffs, NJ: Prentice Hall, 1995.

8. J. Vlissides, J. Coplien and N. Kerth, editors, *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996.

9. Bertrand Meyer, *Object-Oriented Software Construction*, Second Edition. Upper Saddle River, NJ: Prentice Hall, 1997.

10. Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*. New York: Addison-Wesley, 1998.