Debugging and Traceability

Debugging and Traceability

Table of Contents

List of Figures

List of Acronyms

API Application Program Interface. A set of routines and protocols for building software applications.

BNF Backus-Naur Form. A metasyntax used to describe formal languages, especially programming languages. Named after John Backus and Peter Naur.

DDD Data Display Debugger. A graphical front end that works with inferior debuggers such as gdb.

EBNF Extended BNF. An extension of the Backus-Naur Form created by Niklaus Wirth in order to promote readability and succinctness.

GCC Gnu Compiler Collection. A set of programming language compilers. Also common way of referring to the GCC 'C' language compiler.

GDB Gnu Project Debugger. A command-line, source-level debugger for compiled languages such as C, C++, Pascal, and Objective-C.

GUI Graphical User Interface. An interface to a software program that makes use of visual elements such as icons, windows, and other gadgets.

HDL Hardware Description Language. A programming language specifically designed for the formal description of electronic circuits.

IDE Integrated Development Environment. A set of related tools designed to assist computer programmers in developing software. Typical tools include a text editor, compiler or interpreter, debugger, and build automation tools.

OOP Object-Oriented Programming. A computer programming paradigm that emphasizes the use of objects – encapsulations of data and methods. Object-Oriented languages generally feature encapsulation, inheritance, polymorphism, and composition.

OS Operating System. A set of programs that forms the interface between the user of a computer, the hardware, and the other programs that run on the system.

RTL Register Transfer Level. A level of abstraction at which computation is described as transfers of data between storage units (registers) at clock-cycle level, where each transfer involves processing and manipulations of data.

SCC SpecC Compiler. The compiler for the SpecC SLDL.

SLDL System Level Design Language. A programming language designed specifically to be used as a tool for creating computer system specifications. Such a language provides support for executable specifications, well-defined architecture and implementation models, and a methodology for converting specifications into implementations.

SVC System Value Change. File format created to address the needs of trace logs for simulations of SLDLs.

VCD Value Change Dump. File format specified in the Verilog standard for logging value changes of variables during a simulation.

VHDL Very High Speed Integrated Circuit Hardware Description Language. An HDL commonly used for hardware design at RTL and logic levels.

Chapter 1: Introduction

System Level Design Languages and Simulation

In recent years, designers of embedded systems and Systems-on-Chips (SOCs) have come to study a new design paradigm that relies on a class of design languages called System Level Design Languages (SLDLs).

Background

This new paradigm has come directly as a result of the fact that the number of transistors available to computer system designers has increased dramatically over the years, while the ease of design has not increased as steadily [19].

As described in [19], the top-down methodology to System Design is a "set of models and transformations that refine an initial, functional system specification into a detailed implementation description ready for manufacturing." The top-level *specification* model is the most abstract model and does not contain detailed timing information, while the *implementation* model represents a clock-cycle accurate description of the system [19].

Figure 1.1 shows the SpecC design flow. The flow consists of three parts, *System Design*, *Validation Flow*, and *Backend*. At the top of the system design flow, the user captures the design in a specification model based on algorithms of his choice. The design is then refined through architectural exploration and communication synthesis, progressing through architecture and communication models. At each step of the refinement process, the designer must ensure that the design is correct. In order to do this, the validation flow is connected to the design flow.

Once the system design flow has completed , the design may be synthesized into hardware and platform software. Our work fits into the validation flow, with both the debugging support and the tracing support being useful at all stages.

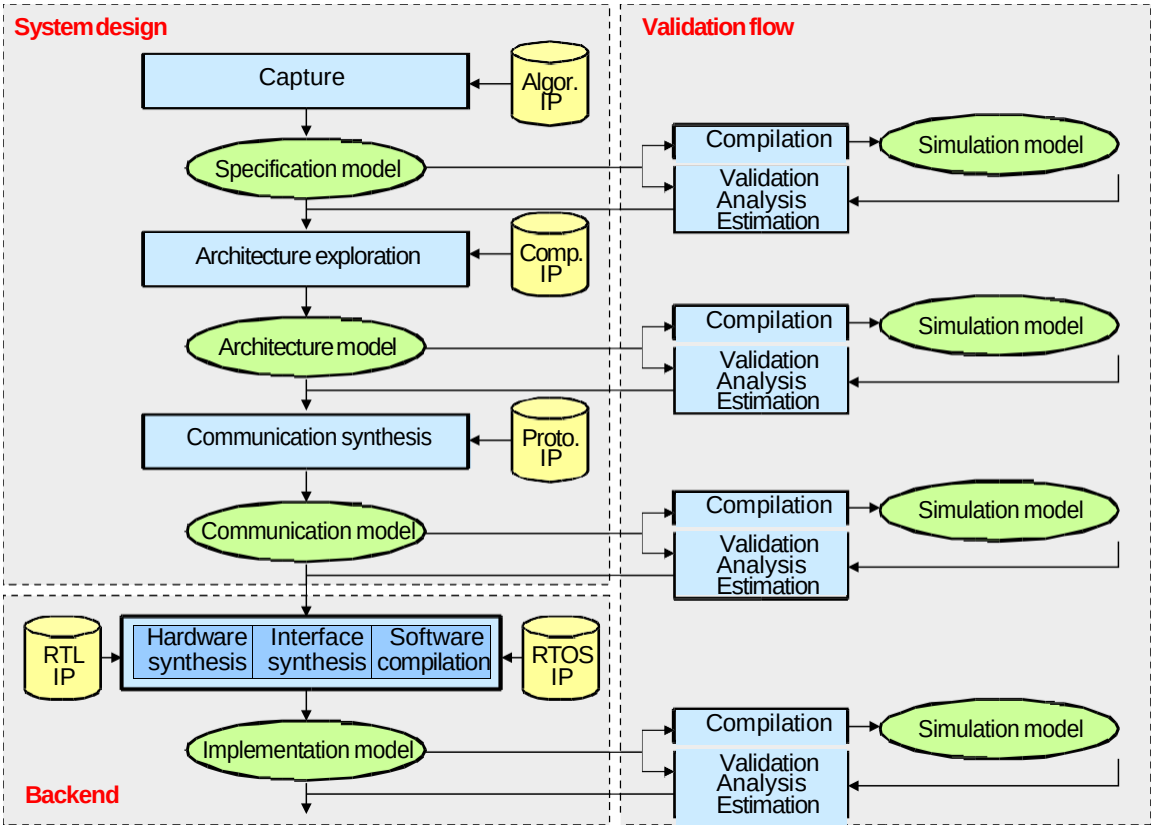

Figure 1.1: The SpecC Methodology [22]

SLDLs combine aspects of traditional high-level programming languages along with aspects of Hardware Description Languages (HDLs). In particular, the SpecC language

> *is a formal notation intended for the specification and design of digital embedded systems, including hardware and software portions. Built on top of the ANSI-C programming language, the SpecC language supports concepts essential for embedded systems design, including behavioral and structural*

hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing [1].

It is natural that users of SLDLs should expect similar tools for helping them debug their designs as those tools that software developers and hardware designers are accustomed to using. For software developers the primary tools are debuggers, and for hardware designers, log files and waveform viewers are the main tools for debugging design simulations.

An additional aspect of system design using SLDLs is that of architectural explorations [19]. An SLDL such as SpecC is designed to allows system implementers to try different architectures in order to achieve cost and performance goals. In addition to traditional software debuggers and hardware simulation traces, system designers desire new tools to aid them in the analysis of system architectures. For example, such analysis tools may help a system architect to understand which parts of the design are performance bottlenecks and which system components are underutilized, in order to provide direction in their architectural explorations.

Our work centers around providing users of SLDLs -- SpecC in particular -- new tools for debugging software aspects of designs, capabilities for creating simulation traces, and capabilities for performing system analysis.[1] Prior to this work, tools were available for compilation and simulation of SpecC designs, but debugging tools were limited and simulation trace tools were not freely available. The data provided by the simulation traces will allow us to develop further tools to provide designers with better feedback for design explorations and to remove some of the guess work that is required using today's tools.

Brief Introduction to the SpecC language

In this section we provide a very brief introduction to the SpecC language. Much more

1 Note that the techniques we describe could also be applied to other C-language-based SLDLs such as SystemC [27].

detailed descriptions of the language are provided in other sources [1][19][22], but it is necessary for the reader to understand a few of the basic concepts in order to understand the later chapters. We encourage the reader to examine to the more detailed references for a better understanding of the SpecC language.

As mentioned earlier, SpecC is built on top of ANSI-C and provides the necessary features for embedded systems design. At the heart of every SpecC program are *behavior*s, *channel*s, and *interfaces* [22]. Behavior classes are responsible for handling all computation and channel classes are used as a means for handling communication between behaviors. An important concept that comes from Object-Oriented Programming (OOP) is that of separating interfaces from implementations [5][10]. Interfaces provide a link between behaviors and channels and support reuse of IP and "plug-and-play" [19]. Both behaviors and channels may inherit from interfaces. Behaviors are considered *active* in the sense that they are responsible for handling computation, while channels are considered *passive* because calls to their methods must originate from behaviors.

Figure 1.2 shows an example of a channel, an interface, and two behaviors interacting in a design. Channel ***C1*** implements the interface ***I1***. The channel is used inside of behavior ***B*** to handle the communication between its two child behaviors ***b1*** and ***b2***.

```
interface I1
{
  bit[63:0] Read(void);
  void Write(bit[63:0]);
};

channel C1 implements I1;

behavior B1(in int, I1, out int);

behavior B(in int p1, out int p2)
{
  int v1;
  C1  c1;
  B1  b1(p1, c1, v1),
      b2(v1, c1, p2);

  void main(void)
  { par { b1.main();
          b2.main();
        }
    }
};
```

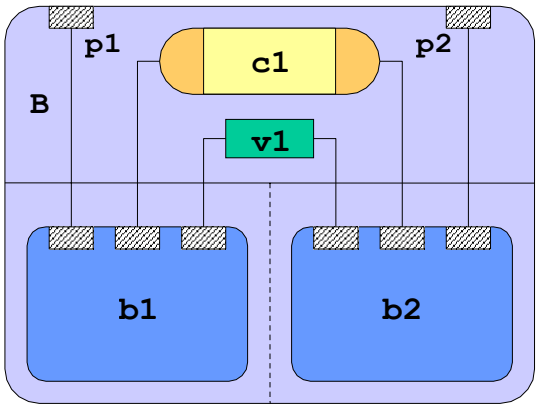

Figure 1.2: Basic Structure in a SpecC Program [22]

Inside of behaviors, the code may look much like ANSI C code, or it may take advantage of the more advanced features of SpecC. SpecC has special types such as *bool*s, *bit vector*s, and *event*s. *Bools* are boolean variables similar to those provided in other high-level languages such as C++ and Java. *Bit vectors* are vectors of bits of arbitrary length that support both standard operations such as logical, arithmetic, and comparisons; and special operations such as concatenation and slicing. *Events* in SpecC are used as signals for synchronization.

Similar to what are provided in HDLs, SpecC also provides a *signal* data type. *Signal*s are provided as a means of representing wires and buses in hardware designs [1]. A signal data type represents two values – one old and one new – and an event. As such, a signal may be used to to transfer data through assignment or mathematical operations or may be used in place of an

event. A clock signal is an excellent example: it has values associated with it – 0 or 1-- and a change in the clock signal will often act as an event to trigger other actions in the system.

SpecC also supports several forms of sequential and concurrent operation by use of the keywords *fsm*, *pipe*, and *par*. *Fsm* blocks represent finite state machines. *Pipe* blocks represent pipelined execution and *par* blocks represent pure parallel operation.

The fact that SpecC provides a means for specifying concurrency is the primary difference between SpecC and traditional programming languages like ANSI C. This difference is also the main reason that debugging of SpecC programs is more difficult than that of sequential programs and why different methods of debugging are required. HDLs have traditionally used waveforms for debugging, as they are an elegant method of visually expressing concurrency.

Figure 1.3 shows some of the ways of creating sequential and concurrent behavioral hierarchies. The leftmost example shows sequential execution, similar to what one sees in an ANSI C program. The next example shows finite state machine (FSM) execution. FSM execution is also sequential and can be modeled using *goto*s or function calls. The last two examples demonstrate concurrent execution. In parallel execution, all child behaviors start at the same simulation time and run concurrently. Pipelined execution models the type of execution that is often present in hardware. It is another form of concurrency where a pipeline stage may only start after the previous stage has already started. Support for interrupts and exceptions (not shown) is provided by means of *try*, *interrupt*, *trap* blocks.

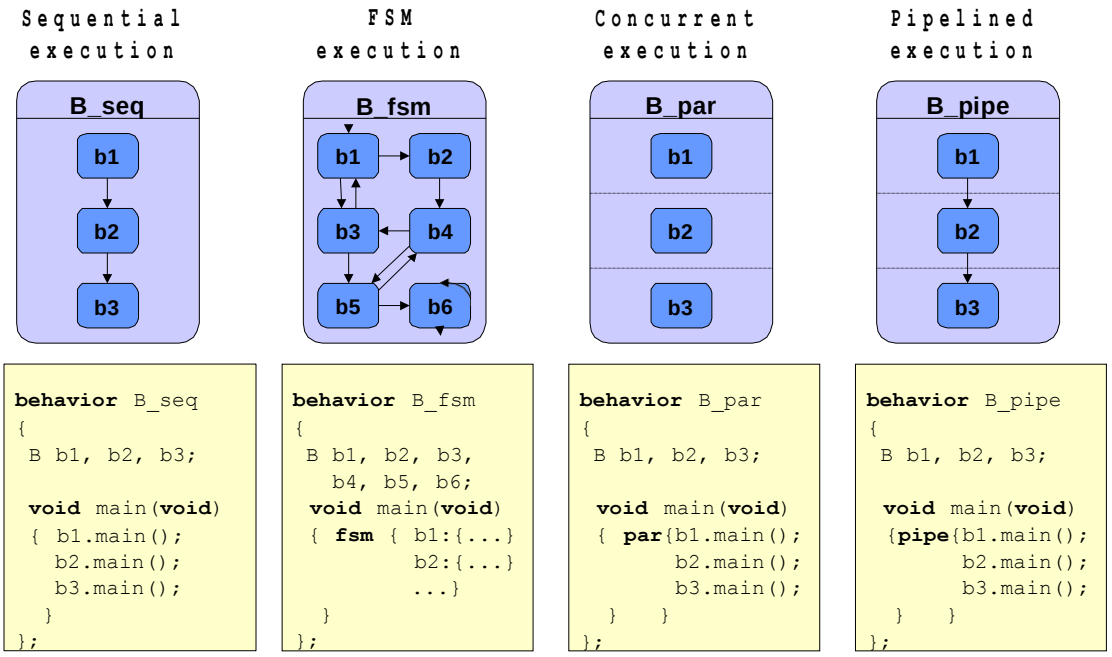

Figure 1.3: Examples of SpecC Behavior Hierarchies [22]

Motivation

Basic Debug Functions

When following a top-down approach to system design, as described in [1], [19], one starts with a specification model that does not provide any detailed timing information. Even at this early stage, a designer must be able to validate that the design is correct. In order to do this, she tests the design against a "golden" model to see that the results match. If the results are correct, she may move on to the next stage of model refinement or architectural exploration. If the model does not produce the correct results however, the designer needs a way to track down the bug(s) in the model. Normally, a software debugger provides the user with a way to trace the execution of the program and to examine the values present in variables at various points in time.

In this situation, a software engineer would typically fire up his favorite debugger and start to work tracking down the problem. Figure 1.4 below shows an example of ***Microsoft's Visual Studio*** [14] debugger in action debugging a C++ program. In the upper right corner, we the call stack, and "automatic" variables are displayed at the bottom of the window. The source code is seen in the main window.

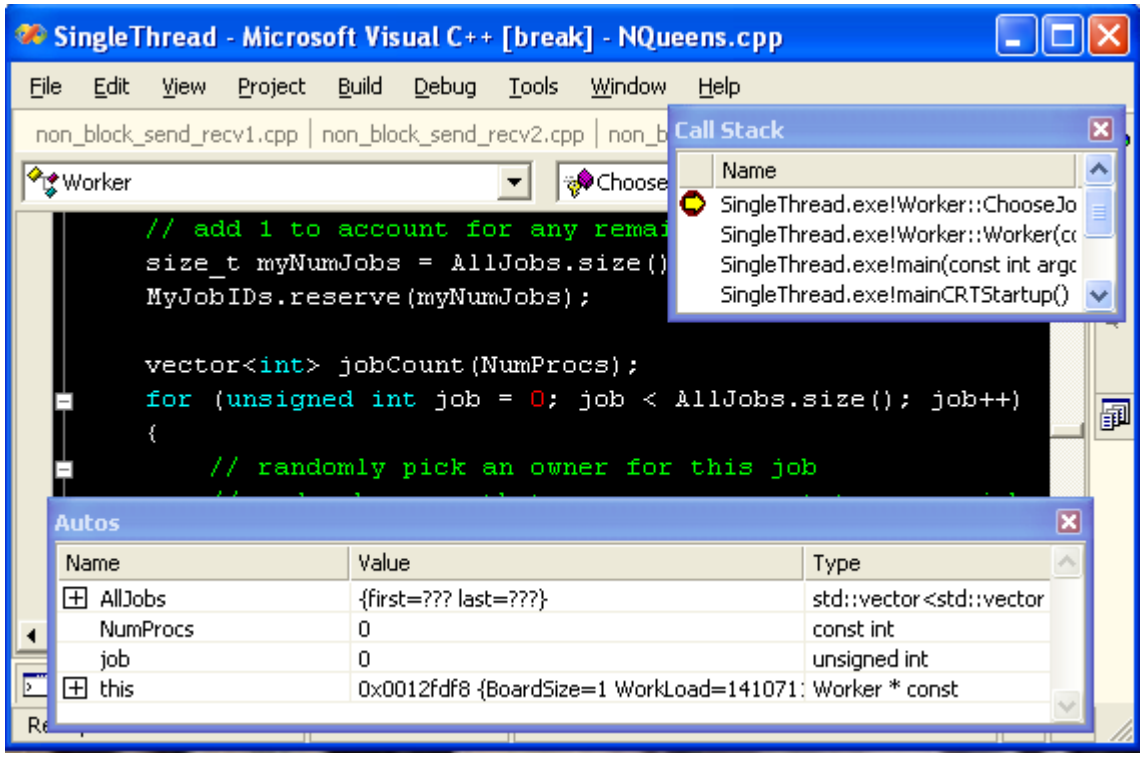

Figure 1.4: Screen Shot of the Microsoft Visual Studio Debugger in Action

While it is possible to use a standard debugger, on a SpecC design, there are several problems associated with this approach. We will begin by explaining what happens when a SpecC program is compiled to form an executable design simulation.

Like writing a 'C' program, a system designer writes SpecC code and then compiles it to form an executable program. As an intermediate step, the reference compiler (***scc)*** generates C++ code from the SpecC source code and links the C++ code to a pre-compiled simulation library in

order to allow “Design.exe” to run as a stand-alone executable. Figure 1.5 illustrates the compilation process.

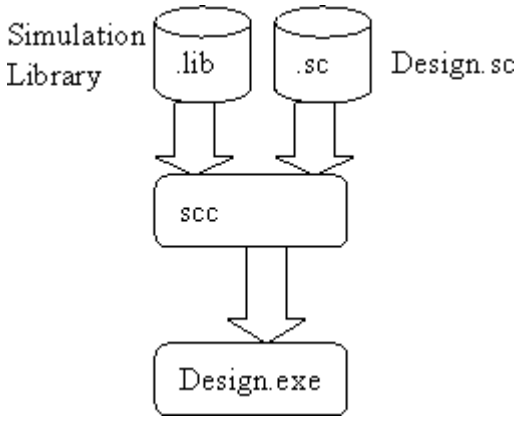

Figure 1.5: SpecC Design Compilation Flow

The simulator acts as a kind of “virtual platform” to simulate the effects of having multiple behaviors working simultaneously in pipelined or parallel execution. In contrast to a traditional 'C' program where one is able to step through code executing on the target hardware, a SpecC program must 'run' on hardware that does not exist in reality. Because of this key difference, traditional methods of debugging executables are not sufficient for SpecC programs.

One method of debugging is to use C's **`printf`** function in order to create a limited trace of the program's execution. This approach is less than ideal however in that it requires modifying the code that is being examined, is time-consuming, and often produces misleading results if the function calls are not carefully placed.

A second approach to debugging software problems is to compile the simulator and the generated C++ code with debugging flags enabled and step through the code using a traditional debugger such as ***gdb***. There are severals problems associated with this method of debugging. First, it requires the user to have a good understanding of the internal workings of the simulator

as well as understand code written in C++. The general SpecC writer should not need to be an expert in SpecC simulator internals. For that matter, he should not even be required to understand C++. Ideally, the SpecC writer does not even need to be aware of the fact that C++ code is generated for him. And since this is an implementation detail of the reference compiler, other SpecC implementations may choose a different method of generating an executable. It is also quite possible that a SpecC designer will not have access to the source code for the simulator and therefore will not be able to step through the simulator itself. Chapter 2 explains our approach to improve the tools available for debugging software issues in SpecC designs.

Simulator States

Because SpecC supports concurrency, the various behaviors in a design may take on complex combinations of states – running, sleeping, waiting, etc – at any moment in time during simulation. As a SpecC developer, there may be situations where one would like to have an understanding of the current states of a design's behaviors in the simulator. For example, if one's SpecC design is experiencing deadlock issues, being able to "peek inside" the simulator in order to determine the sequence of events leading up to the deadlock may be invaluable.

Both the SpecC Reference Simulator and the UCI CECS SpecC simulator implementations use threads to simulate concurrent behaviors [3]. Internally, they keep track of several queues of threads corresponding to the various states of a SpecC behavior's life cycle: *Ready, Running, Waiting For, Waiting, Notified, Trying,* and *Suspended*. The simulator is also responsible for updating the simulation time. The SpecC LRM [1], describes an "abstract simulation algorithm" that is close to what the reference simulator implements. Another paper, [4], has a detailed description of the execution semantics in terms of Abstract State Machines.

At the beginning of the simulation, only one ***Main*** thread exists. New threads are created as the simulation progresses whenever **par** or **pipe** statements occur. As these new threads are created, the simulator adds them to the *ready* list. When it is time for the simulator

to pick a thread to run, it may choose from among the ready threads in any order it chooses. Whenever a **`wait`** statement occurs in the design, the simulator moves the thread to the *waiting* list until an event occurs to move the thread into the *notified* list. In the case of **`waitfor`** statements, the simulator moves the running thread into the *waiting for* queue. A thread that has been interrupted moves into the *suspended* list and returns to its previous state at the return of the interrupt handler. If the interrupted thread is instead aborted, it is removed from all simulator queues.

The simulator is also responsible for maintaining the timing for the simulation. Simulation time cycles only advance due to **`waitfor`** statements. At the time that the simulator moves a thread into the *waitfor* queue, it calculates a *wakeup* time for the thread. When the simulator's thread scheduler has reached a point at which no threads should be notified or resumed, the simulator advances the simulation time. It picks the thread that has the earliest wakeup time (at the head of the *waitfor* queue) and advances the simulation time to this thread's wakeup time.

The fact that the simulator uses threads to keep track of the states of SpecC behaviors and to simulate concurrency is really just an implementation detail. Ideally, a SpecC writer should not be concerned with understanding thread execution. However, he would like to see "snapshots" of the states of a design's behaviors. Mueller et al [4] describe a high-level view of the thread life cycle as having four states: *running, waiting, completed,* and *interrupted* (see figure 1.6, below). In chapter 3, we present new debug functions to show the five states which roughly correspond to the aforementioned four states, but with some important differences.

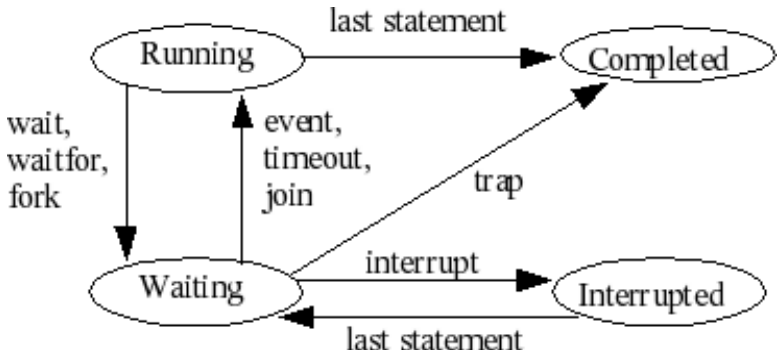

Figure 1.6: General Life Cycle of Behavior [4]

First, we added an additional ***ready*** state. The ready state is an acknowledgment that the simulator may not be able to run all available behaviors in parallel. In fact, the current reference simulator only runs one thread, and therefore one behavior, at a given time [3]. The ***sleeping*** state corresponds to behaviors that have encountered a SpecC **`waitfor`** statement. This is an extension of the **`waiting`** state described in [4]. Our ***waiting*** state corresponds only to behaviors that are waiting on SpecC events. We feel that splitting the general waiting state into ***sleeping*** and ***waiting*** states provides the user with additional useful information. Our ***suspended*** state is equivalent to the general "interrupted" state. We do not provide a means for obtaining a list of completed behaviors. Once a behavior has completed, the simulator know longer "knows" about it. In many situations, the user will be able to infer which behaviors have completed based on the knowledge of the other queues.

Tracing capabilities

As the design process continues through refinements and explorations of different architectures, the need for hardware-specific debugging tools become more important. Hardware designs nearly always contain a great deal of concurrency. For this reason, when designing hardware, it is often extremely valuable to be able to view the output of simulations in a

graphical form. The viewer displays a list of variables along one axis and time on the opposite axis. In this way, the designer sees signals changing as a function of time. Figure 1.7, below shows an example of a waveform produced from an RTL simulation.

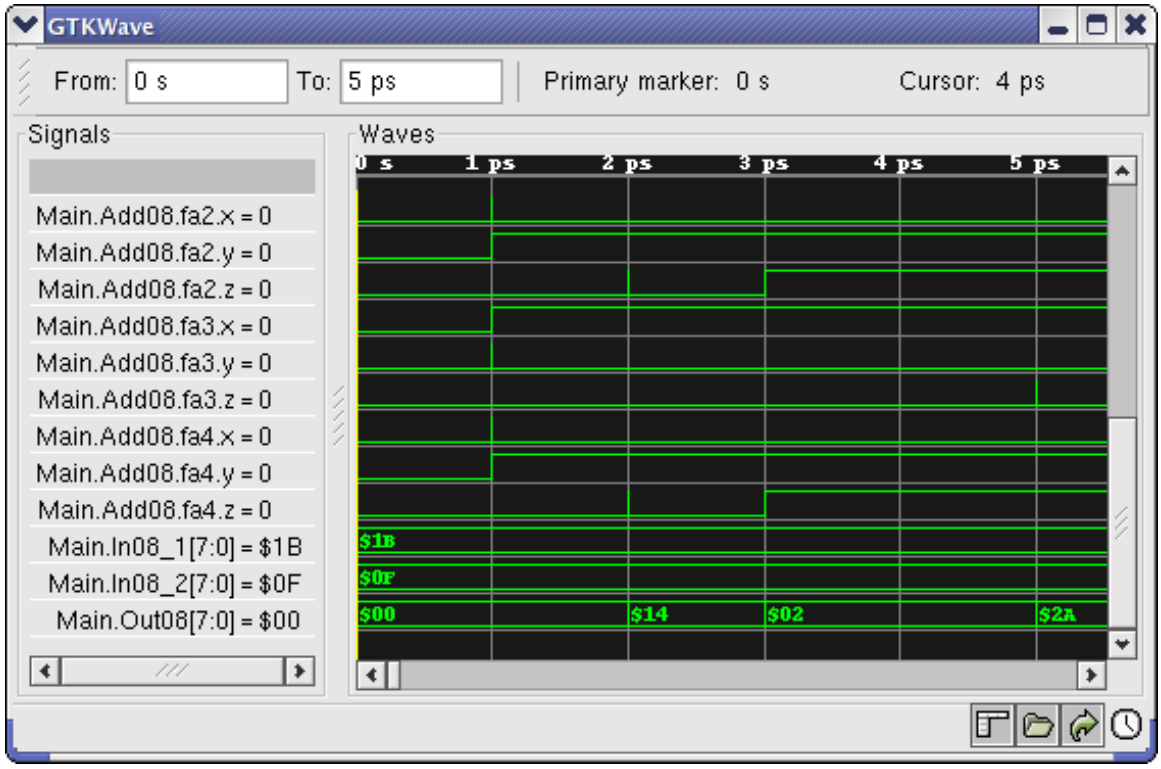

Figure 1.7: An example of a trace of an RTL model.

The information provided can be extremely useful for debugging, as well as for analysis. Simulators for languages such as VHDL and Verilog have supported this type of graphical waveform creation for many years. Popular examples of simulators with waveform viewing capabilities include ***Modelsim*** [24], ***NCVerilog*** [25], and ***BlueHDL*** [26]. A waveform viewer should have a least a few basic capabilities:

- the ability to zoom in and out in the time domain
- the ability to remove signals from view
- the ability to display variables in different numerical formats – binary, hex, etc

Prior to the work presented herein, the CECS group did not have any method of producing waveforms from SpecC simulations.

SLDLs, such as SpecC, also present new challenges for creating traces as compared to traditional HDLs, such as Verilog or VHDL. In a trace of an SLDL simulation, the designer would like to be able to understand the hierarchy of behaviors in a design as well as be able to understand how a behavior's state changes as a function of simulation time. In contrast, HDLs traditionally only support tracing hardware-specific signals such as *wires*, and not architectural entities such as *behaviors*.

In addition to being useful for debugging hardware design problems, graphical waveforms can also be extremely useful as system analysis tools. In particular, systems featuring a great deal of concurrency can benefit greatly by having tools that help the designer to detect bottlenecks and underutilized resources. Figure 1.8, below shows an example graph of a system with four behaviors: a parent and three child behaviors running in parallel. In this very simple example, a parent behavior ***B_par*** has three child behaviors running in parallel. Behavior ***b3*** takes longer than its siblings, so one could conclude that it is the "bottleneck". Of course, real systems are much more complex, but this example illustrates the usefulness of the analytical properties that an event graph provides.

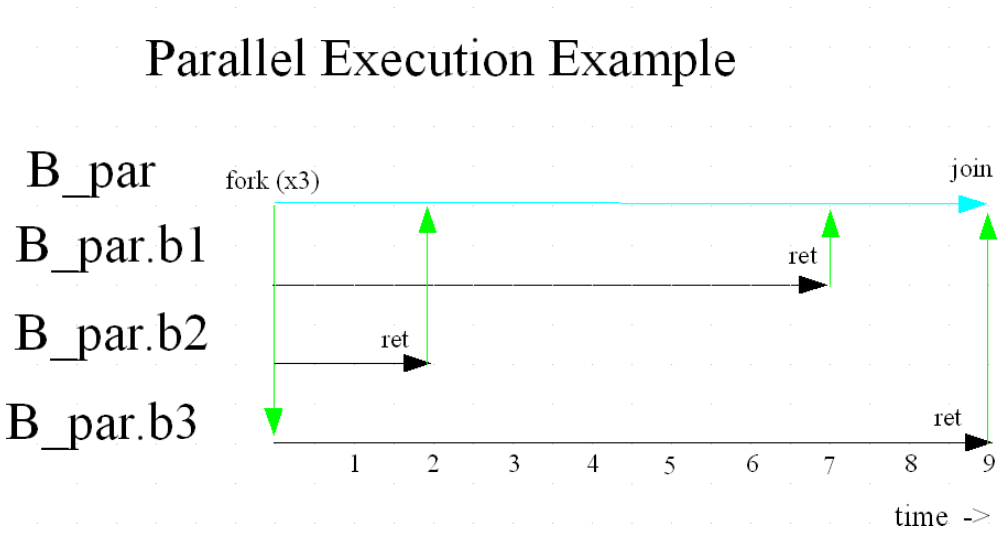

Figure 1.8: An example of how a simple design with 3 behaviors running in parallel might be visualized.

Production of Simulation Logs

Waveforms such as the one seen in figure 1.7, are created by waveform viewing programs from simulation trace files. HDL and SLDL simulators produce these trace files by creating a log of all significant simulation events (behavior state changes, method calls, event notifications, etc) and value changes along with time stamps to indicate when these events occurred during the simulation. These logs may then be read by waveform viewing programs in order to produce the graphical representations of simulation events and value changes as a function of time.

Goals for Tracing Implementation

After considering what features we would like to have for tracing system designs, we decided upon several important goals related to implementing tracing features for SpecC.

- Produce .vcd files (or something similar) when running a simulation in order to view waveforms
- Come up with a way to produce an event graph when running a simulation in order to view thread forking, parallel execution, pipelined execution, thread joining, etc. Also show states of behaviors -- waiting, running, etc.
- Allow for tracing to be enabled on per-signal, global, or at a module level. Ideally, this could be done after compilation.
- Allow for tracing to be enabled for certain time periods only. Ideally, this should be done post-compilation. It would be nice to allow for tracing to start, stop, and resume: i.e., "start 200, stop 250, start 3000, stop 5000"
- Tracing of "built-in" types (int, float, etc) should be considered a "nice-to-have" feature.
- Impact on non-traced signals should be minimized as much as possible. In particular, efficiency and memory use should be considered.
- Keep code generation simple.
- Keep the simulator code simple.
- Demonstrate implementation and benefits by use of realistic examples.

In chapter 4, we describe our approach to adding tracing abilities to the SpecC design toolkit.

Related Works

Within the computing industry and academia, several sets of debug, trace, and analysis tools have been developed. Here, we discuss some of these related works.

Software Debuggers

Software designers are accustomed to using programs called debuggers to assist them in finding errors in their programs. Debuggers may take several forms. On Linux platforms, the Gnu debugger, ***gdb***, and a graphical front-end, called ***Data Display Debugger*** (***ddd***) are quite popular [12][13]. On the Microsoft Windows platform, ***Microsoft's Visual Studio*** is a very popular IDE[14]. Other debuggers, such as ***Metrowerks' CodeWarrior*** and ***Eclipse*** from eclipse.org, work on several platforms including Windows and Linux [15] [16]. Some debuggers are implemented as separate programs from text editors while other debuggers are part of an Integrated Development Environment (IDE). Sometimes debugger users refer to *inferior* debuggers and *front-ends* to them. ***Gdb*** is considered an *inferior* debugger, since it does not have a graphical front-end and only responds to command line inputs. ***Ddd*** acts as a graphical front-end to *inferior* debuggers such as gdb.

A high quality debugger is rich in features and includes many of the following:

- the ability to break on a line of code and examine the state of the program
- the ability to view the values of variables during program execution,
- the ability to view the "call stack" or "back trace",
- the ability to view the contents of memory, and
- the ability to step through code one line at-a-time during program execution

Some debuggers also support advanced features:

- the ability to modify program variables and then resume execution of the program
- advanced graphical ways of viewing data, especially complex data structures
- "edit and continue" features that allow the programmer to actually modify the program during execution and then resume the program with the changes now in place
- remote debugging which allows the programmer to execute the program that is being debugged on one machine while executing the debugger on a different computer
- multi-thread support

Hardware Debuggers

There are a number of commercial simulators available for Verilog and VHDL that create log files that may be examined using a graphical viewer. The simulators make records of simulation time changes as well as changes to values of variables in the system. The waveform viewers know how to parse these simulation logs in order to create a graph of the variables values as a function of time.

Verilog simulators produce files in a format called "VCD", or "Value Change Dump". The IEEE 1364-2001 standard for Verilog [8] describes the details of the VCD format. Various companies and academic institutions have created GUI viewers for .vcd files. We have used one particular free viewer, called ***GTKWave*** in our experiments. ***GTKWave*** was written and is maintained by the Advanced Processor Technologies Group at the University of Manchester [9].

System Analysis Tools

In recent years, operating systems designers and users have come to make use of tools such as ***Linux Trace Toolkit*** (***LTT***) [6] in order to view the state of processes in the system plotted against time. ***LTT*** actually works in a way that is similar to HDL simulators to log events that happen in an operating system over a certain period of time. Another company, Wind River, has created an application called, ***System Viewer*** that serves similar purposes for their real-time OS, ***VxWorks*** [7].

The format of the logs are different from VCD files and the application of the information is somewhat different, but from a high-level point of view, the system of logging events and providing a graphical visualization tool are very similar.

One key difference between these operating systems traces and the types of tracing required for HDLs and SLDLs is that in an operating system for a single-processor computer, no two tasks actually run in parallel. Instead a multi-tasking OS runs a scheduling algorithm to switch between the processes running on the system. Even the majority of modern day systems using multiple processors use a small number, perhaps 2 or 4 processors, so the level of true parallelism is actually quite low. In viewing traces created for an SLDL, one must be able to visualize multiple behaviors running concurrently.

Chapter 2: Debug Functions

The first part of our efforts to improve the capabilities for debug, trace, and analysis of SpecC systems, was to provide some basic software debugging facilities.

Possible Approaches for Software Debugging Capabilities

One can roughly divide the techniques for creating providing debugging capabilities into two categories, a symbol table-based approach and introspection-based approach. Here, we briefly describe each of the techniques as well as pros and cons to each approach.

Symbol Table-Based Approach

The symbol table-based approach to creating a debugger is commonly used for popular programming languages on popular platforms. Examples of this would be ***gdb*** on Linux/Unix for a number of languages such as C, C++, Fortran, and Objective-C; and Microsoft's ***Visual Studio*** debugger on Windows which also supports a number of languages such as C, C++, and Basic. Generally speaking, a particular compiler and debugger pair will work together to support this approach. For example, ***gcc*** and ***gdb*** work as a pair and Microsoft's compiler and debugger work together as a set. The debugger expects symbol table(s) to be provided along with the machine code for the program. Often times the symbol table will be included in the same file as the "executable" [17]. Symbol tables provide the debugger with information including line numbers, names, types, and scopes of variables; and names, parameters, and scopes for functions[18]. The

debugger is then able to correlate the source code and the executable in order to implement the features previously mentioned in section 1.3.1.

One point of confusion for many people is the notion of compiler optimizations and debug mode. Optimizations may be enabled in conjunction with generating debug information. The issue is that it may be difficult for a human being to understand the relationship between source code and an optimized executable. For this reason, programmers generally disable optimizations when compiling for debug mode.

Introspection-Based Approach

Several object-oriented languages such as Java, Objective C, and Python support a feature called *introspection.* This feature allows an object to know extra information about its own type and/or name. The introspection-based approach to debugging works on this same principal, such that instances of *channels* and *behaviors* know both their instance names and class types. Because of this, the introspection-based approach provides the important ability to distinguish between different instances of a class. This approach involves compiling extra information into the source code so that functions may be called during runtime to provide the user with debug information.

The primary advantages of this method are that no separate "debugger" program is actually required and the implementation is simpler. The major disadvantage is that this approach does not provide many of the features listed in section 1.3.1, such as the ability to step through code on a line-by-line basis or the ability to examine the values of variables.

When debug mode is enabled at compile time, the compiler inserts extra information and function calls in order to make debug information available at runtime. The user may also insert extra debug function calls into her program. In addition, the user may take advantage of both approaches by enabling the introspection-based debugging and the symbol table-based features and using a debugger such as ***gdb*** or ***Visual Studio***.

Comparison of Debug Techniques

The primary difference between the introspection technique and the symbol table-based approach is that in the former the debug information actually becomes a part of the executable program, whereas in the symbol table-based approach, the debug information is separate from the executable and is only used by the debugger. The symbol table-based approach requires two programs – the executable under debug and the debugger – along with support from the operating system to allow them to work together. The symbol table-based approach provides a very efficient implementation of debugging features.

The major advantage of the introspection-based debug approach is the relative simplicity of the implementation. For us, the time required to implement the introspection-based approach was measured in weeks, while we estimate the time to implement the symbol table-based approach would take months or years. A second advantage of the runtime approach is that it is completely platform independent. The implications of attempting to create a platform independent debugger should not be underestimated. Ideally one would like to work within the framework of an open source compiler/debugger platform such as ***gcc/gdb***, but ***gcc/gdb*** is not platform independent. ***Mingw*** [20] exists for the Windows platform and is based on ***gcc***, but integrating a new language such as SpecC into both ***gcc*** and ***mingw*** would likely require a great deal of duplicated work. For the reasons listed, we decided to pursue the introspection-based info approach.

Description of the Implementation

In the current implementation of the SpecC compiler, we convert SpecC into C++, and then compile the C++ code to form an executable, as shown earlier in figure 1.5. This executable may then be debugged using a C++ debugger such as gdb. If the SpecC source is compiled with the new debug option (-G) enabled, the SpecC compiler (***scc***) adds extra debug information.

The new debug features depend on the ability for instances of behaviors and channels to know their instance names as well as the ability for the simulator to know which behavior or channel is currently running in a simulator thread.

When converting from SpecC to C++, the code generator turns *behavior*s and *channel*s into C++ classes. All behavior classes inherit from a behavior base class, and all channels inherit from a channel base class. The primary difference between behavior and channel classes is that behavior classes must define a virtual function, **`void main(void)`**. Prior to adding the debug features, there was no common base class between channels and behaviors.

We accomplish the task of allowing behavior and channel instances to know their own names by storing a copy of the instance name within the channel or behavior class as well as a pointer to the parent instance. Each instance only stores its own local name and derives its fully-qualified name by walking parent pointers to build the full name. Figure 2.1 shows an example of a simple hierarchy of behaviors. ***Main*** has one child behavior, ***s1***. The behavior, ***s1,*** has three sequential child behaviors: ***b1***, ***b2***, and ***b3***. ***Main.s1.b1***, stores its local name, "b1" and a pointer to its parent, ***s1***. The behavior named ***Main.s1*** will store its local name, "s1" and a pointer to its parent, ***Main***. The full name for ***Main.s1.b1*** can be found by building up the string by walking the ancestor pointers. This method of building the name saves memory required to store the entire names of instances. Since a channel may have a behavior as its parent, we introduced a new class called **class_type** from which channels and behaviors inherit.

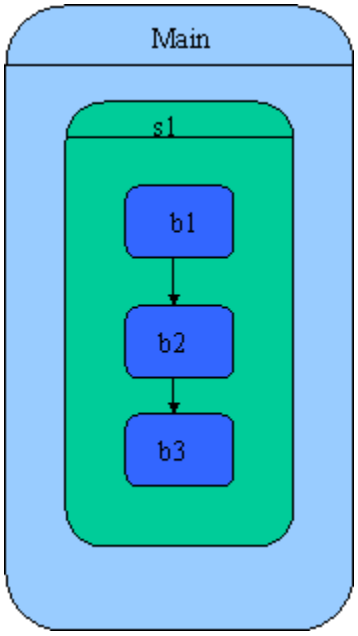

Figure 2.1: Naming of Behaviors

in a Simple Hierarchy

Figure 2.2 shows the class diagram for **class_type**, **behavior**s, **channel**s, and user-defined behaviors and channels. Each **class_type** object contains two strings: the class name and the instance name, as well as a pointer to its parent instance. In addition, the **class_type** class has three member functions for obtaining the class name, instance name, and fully-qualified instance name. The channel and behavior classes now have two constructors: one with name and parent arguments, and one default constructor that may be used in release mode, where instance and class names are not required.

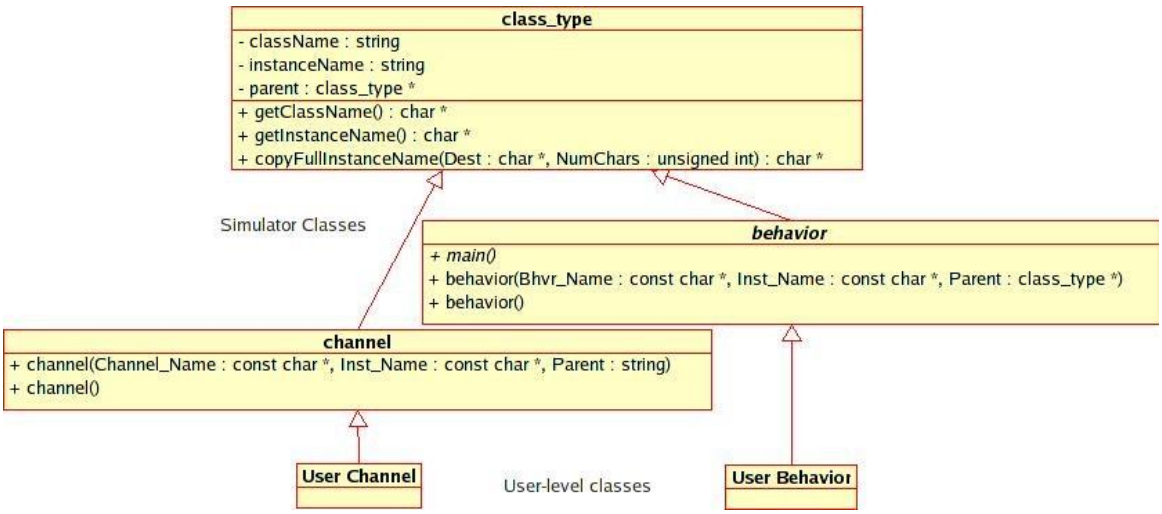

Figure 2.2: UML class diagram for channels and behaviors

In order for the simulator to keep track of which behavior or channel instance is currently running at any given time, the code generator automatically inserts method calls into behavior and channel methods in order to maintain the *current* and *active* instance and class context information. Four methods, **`_SetActiveInst`**, **`_RestoreActiveInst`**, **`_SetCurrentInst`**, and **`_RestoreCurrentInst`** have been defined for setting and restoring the active and current instance pointers in the SpecC simulator. At the beginning of any behavior "main" method, **`_SetActiveInst`** is called and before every **`return`** statement, **`_RestoreActiveInst`** is called. At the beginning of any other channel or behavior method, **_SetCurrentInst** is called and **`_RestoreCurrentInst`** is called before every **`return`.** As an optimization, any behavior methods other than "main" do not need to make these calls, since they are essentially "private" methods with respect to any other behavior or channel. The difference between *active* and *current* instances is that the *active* instance will always be a behavior, while the *current* instance may be a channel or a behavior. Recall from section 1.1.2 that behaviors are active and represent computational components, while channels are passive and represent communication components.

Once the SpecC design has been compiled with the new debug option, SpecC users may use a combination of runtime-supplied debug information and symbol table-supplied debug information to interpret the behavior of their programs.

The API

Our runtime-based debugging capabilities consist of an API providing 12 user-callable functions that give the SpecC user access to new debugging information. As a first step in providing a SpecC designer with debugging capabilities, we came up with an API consisting of 6 debug functions that a user may call to obtain copies of the instance and class names along with 6 more functions to print the names to the stderr stream. They are as follows:

```
const char * active_class(void);
const char * active_instance(void);
char * active_path(char * Dest, const unsigned int Length);
const char * current_class(void);
const char * current_instance(void);
char * current_path(char * Dest,
                    const unsigned int Length);

void print_active_class(void);
void print_active_instance(void);
void print_active_path(const unsigned int Length);
void print_current_class(void);
void print_current_instance(void);
void print_current_path(const unsigned int Length);
```

Figure 2.3: The debugging functions provided with scc -G option

The reader may refer to appendix A for a more thorough explanation of these functions.

Note that although the distinction between “active” and “current” may be confusing at first, if one remembers that behaviors are representations of *active* objects while channels are representations of *passive* objects, the names become clear [1]. *Active class* or *instance* always refers to a behavior, while *current class* or *instance* may refer to a channel or a behavior. In the

case where the debug functions are called within a behavior method, *current* and *active* will refer to the same instance – that of the behavior.

The first six functions provide the most flexibility to the user, but may require more effort to use compared to their "print" counterparts. The print functions are very easy to use, but always print to *stderr* and always print new lines following the text, which may not be what the user desires.

A SpecC writer may use these functions to help him determine which particular instance of a certain behavior or channel is currently executing. For example, there may be a behavior representing an OR gate that is instantiated several thousand times in a design. At a given point in time, it may be very useful to know which particular instance of the OR gate is executing. By making use of these new debug features, the SpecC writer can determine precisely which instance of the class is operating at any time.

There may be times where the specific instance name is not as important as knowing which class has called a certain function. For example, in the case of global functions that may be called by any behavior or channel, the current class name may also be a useful debugging tool.

Short Examples

Example 1:

The following example shows how a designer can make use of the **`print_active_path`** debug function to help trace the path of a design's execution. This examples is made up of two behaviors, a sender and a receiver, running in parallel and communicating through the use of a channel. By inserting a few lines of debug code in the channel's **`send_word`** and **`receive_word`** functions, the designer can see which behavior is calling the channel methods and at what time.

Figure 2.4 shows the **`main`** method for the example design. As we can see, the ***Main*** behavior contains two child behaviors, ***r*** and ***s***, and a single channel, ***c***, that is used for communication. ***Main*** starts ***r*** and ***s*** running in parallel.

Figure 2.5 shows how the designer could insert debug calls into the channel's **`send_word`** method. All of the debug code is wrapped in the conditional compile section that begins with **`#ifdef DEBUG`**. Here, we make use of a combination of the **`print_now`**, **`print_active_path`**, and **`print_current_path`** debug functions. Similar debug code is also added to the **`receive_word`** channel method.

```
behavior Main
{
      C     c;    /* using channel c    */
      R     r(c); /* connect a receiver r    */
      S     s(c); /* with a sender s    */

      int main(void)
      {
      sim_time_string buf;
      printf("Time =%5sns: Main::main(): Starting S and R in
              parallel...\n", time2str(buf, now()));

      par {
            s.main();  /* sender and receiver run in parallel */
            r.main();
      }

      printf("Time =%5sns: Main::main(): Exiting...\n",
            time2str(buf, now()));

      return(0);
      }
};
```

Figure 2.4: This example shows the Main behavior starting its child behaviors "r" and "s" running in parallel. The behaviors communicate by way of a channel.

```
void send_word(int Word)    /* internal sender routine */
{
#ifdef DEBUG
  // find out which instance of the C channel is being called
  const int MAX_LEN = 128;
  fprintf(stderr, "Time: ");
  print_now();
  fprintf(stderr, " ");
  fprintf(stderr, "in C::send_word\n");
  fprintf(stderr, "The active behavior is: ");
  print_active_path(MAX_LEN);
  fprintf(stderr, "The current channel is: ");
  print_current_path(MAX_LEN);
#endif

  while(ValidWire)
    {
      wait SyncWire;
    }
  DataWire = Word;
  ValidWire = true;

  waitfor SEND_CYCLE;

  notify(SyncWire);
}
```

Figure 2.5: A channel method with debug calls inserted. Similar debug code is also inserted into the channel's "receive_word" method.

With this debug code added, the designer can rapidly verify that the basic operations are working correctly. That is, that behavior ***s*** is acting as the sender and that behavior ***r*** is acting as the receiver, with channel ***c*** serving as the communication channel. Figure 2.6 shows an excerpt of the output of the design running with debugging enabled. As we can see, ***Main.s*** calls **`Main.c.send_word`** on a regular basis -- every 10 ps -- and ***Main.r*** calls **`Main.c.receive_word`** also on a regular basis. The design appears to be functioning correctly.

```
. . .
Time: 2270  in  C::send_word
The active behavior is: Main.s
The current channel is: Main.c
Time: 2270 in C::receive_word
The active behavior is: Main.r
The current channel is: Main.c
Time: 2280  in  C::send_word
The active behavior is: Main.s
The current channel is: Main.c
Time: 2280 in C::receive_word
The active behavior is: Main.r
The current channel is: Main.c
Time: 2290  in  C::send_word
The active behavior is: Main.s
The current channel is: Main.c
Time: 2290 in C::receive_word
The active behavior is: Main.r
The current channel is: Main.c
Time: 2300  in  C::send_word
The active behavior is: Main.s
The current channel is: Main.c
Time: 2300 in C::receive_word
The active behavior is: Main.r
The current channel is: Main.c
Time: 2310  in  C::send_word
The active behavior is: Main.s
The current channel is: Main.c
Time: 2310 in C::receive_word
The active behavior is: Main.r
The current channel is: Main.c
. . .
```

Figure 2.6: Excerpt from the output of the parallel execution design with debug functions inserted into the channel's send_word and receive_word methods

Example 2:

In this example, we show how the debug functions may be used from within a debugger such as ***ddd,*** that supports run-time evaluation of functions from the debugger itself. The advantage of this approach is that the user does not need to modify her source code, to use the debug functions. This example also demonstrates the technique of using a symbol-table based debugger along with the introspection-based debug functions.

Figure 2.7 shows a screen shot of the par3 example, from the SpecC distribution examples. In this screen shot, the debugger is stopped at a breakpoint with the *main* method of behavior ***A2***. Near the top of the debugger, we see the **`now`**, **`delta`**, and **`active_instance`** function outputs. From the information that these functions provide, we can tell that the simulation is in delta cycle 0 of simulation cycle 0, and the instance of behavior ***A2*** is named ***a2.***

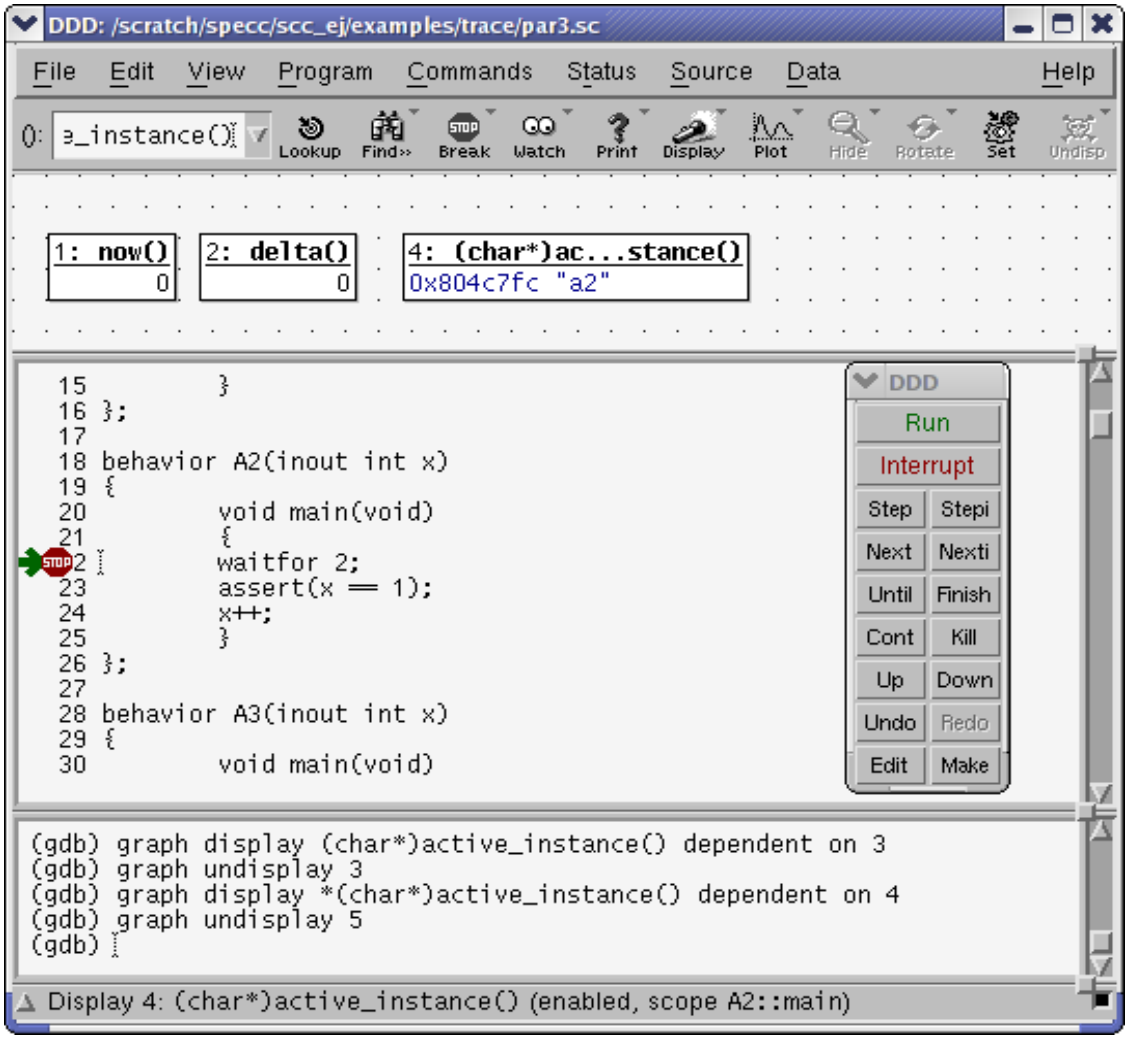

Figure 2.7: Screen Shot of ddd Debugger Updating Debug Functions

Chapter 3: Simulator State Functions

The next part of our efforts to provide debug capabilities for SpecC designs was to provide support for obtaining snapshots of the simulator's internal states. These functions differ from the ones mentioned in the previous section in that they require the user to know something about the working of the SpecC simulator. These functions are an improvement over earlier means of investigating the simulator states in several ways. Because they do not require a version of the simulator that has been compiled in debug mode, the user can run a "release" version of the simulator, which is much faster and it does not require that the source code for the simulator be available to the user. They also provide a more abstract view of the internal simulator states. The user does not require in-depth knowledge of the exact implementation of the simulator, rather general understanding of behavior states.

Behavior States

In chapter 1, we briefly discussed the different states that a behavior goes through during its lifetime. Here we extend figure 1.6 to show the additional ***parent*** and ***sleeping*** states. Figure 3.1, shows the six states that a behavior may assume during its lifetime. When a behavior is first created, it begins in the ***inactive*** state. When the behavior starts to execute its **`main`** method, it moves into the ***running*** state. If the behavior encounters a **`waitfor`** statement, it moves into the ***sleeping*** state. If instead, it encounters a **`wait`** statement, it enters the ***waiting*** state.

A behavior that contains a **`par`** or **`pipe`** statement, enters the ***parent*** state until all of its

children join. A behavior within a **`try-interrupt-trap`** block, will enter the ***interrupted*** state if it becomes interrupted by an event. After the interrupt has been handled by the interrupt handler, the behavior returns to its previous state. Finally, whenever a behavior is aborted from a **`try-interrupt-trap`** block or simply finishes its *main* method, it returns to the ***inactive*** state.

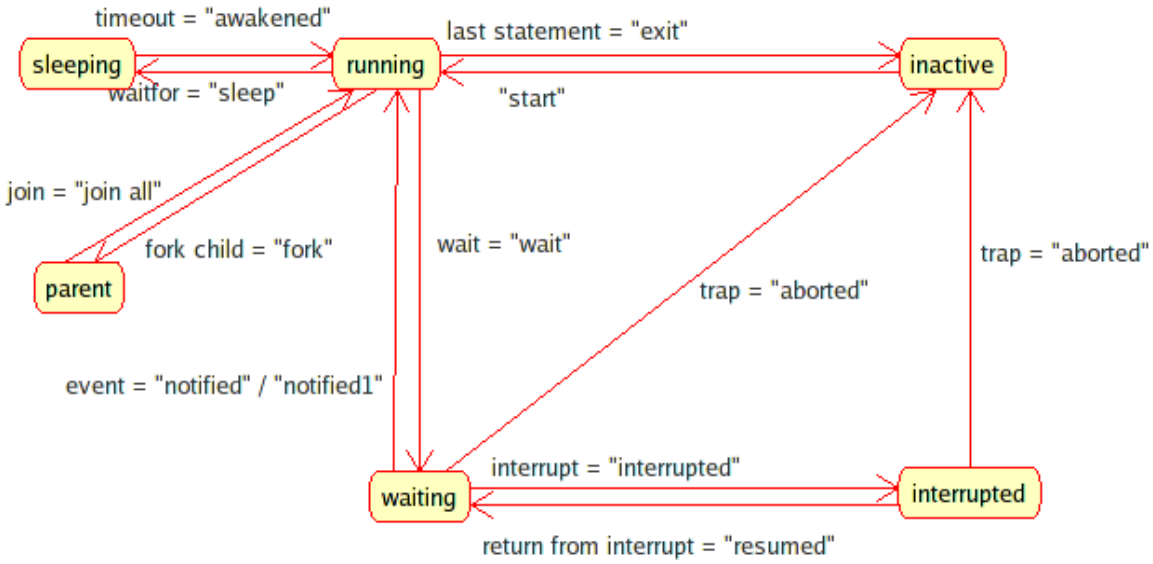

Figure 3.1: State Diagram for Behaviors in a System Design

Implementation of Simulator State Functions

In addition to the functions inserted by the compiler for setting and restoring active and current instance pointers described in chapter 2, function calls are also added to keep track of the method names. Whenever a behavior or channel method is called, **`_SetCurrentMethod`** is called with the name of the current method and **`_RestoreCurrentMethod`** is called before every **`return.`** Some of the simulator state debug functions use the method names when presenting the current state of the simulator as additional information regarding a behavior or

channel's progress.

The API

The first two functions provide the user with a quick snapshot of the current simulator state in an easy-to-use form. These functions are very powerful in that they condense a great deal of information into an easy-to-read format. The only difference between the two functions is that **`print_simulator_state`** prints the simulator time on a separate line before it prints the behavior states.

```
// print out a nicely-formatted list of states of behaviors
void print_process_states();

// same as above, but also prints the simulation and delta
// times
void print_simulator_state();
```

Figure 3.2: User-Callable functions for printing the simulator states

The following sets of functions provide the user with more flexible ways to see the state of the simulator.

Five functions provide a way to get the number of threads in each of the five simulator lists/queues. These counts can be used as inputs to the next set of functions.

```
//get the length of the current queue of ready threads in
//the simulator
unsigned int ready_queue_length();
unsigned int running_queue_length();
unsigned int waiting_list_length();
unsigned int sleeping_queue_length();
unsigned int suspended_list_length();
```

Figure 3.3: These five functions return the length of the various simulator thread queues and lists

The next five functions copy the names of the behaviors from each list or queue. The

user must provide a 2-dimensional array of characters for the names to be copied into and must provide the maximum number of names to copy as well as the maximum number of characters in each name string. The reason that the user must provide buffers for the strings is that these debug functions need to be reentrant to support concurrent execution.

```
unsigned int ready_queue(char * Names,
             unsigned int QueueLength,
             unsigned int StringLength);

unsigned int running_queue(char * Names,
             unsigned int QueueLength,
             unsigned int StringLength);

unsigned int waiting_list(char * Names,
                          unsigned int ListLength,
                          unsigned int StringLength);

unsigned int sleeping_queue(char * Names,
                            unsigned int QueueLength,
                            unsigned int StringLength);

unsigned int suspended_list(char * Names,
                            unsigned int ListLength,
                            unsigned int StringLength);
```

Figure 3.4: User-callable functions for copying the names of behaviors from the simulator's states lists and queues

The final set of 5 functions are more convenient, but less flexible counterparts to the previous 10 functions. Instead of returning a value or copying strings, they print lists or queues to the stderr stream.

```
void print_ready_queue(const char * Separator,
     unsigned int QueueLength, unsigned int StringLength);

void print_running_queue(const char * Separator,
     unsigned int QueueLength, unsigned int StringLength);

void print_sleeping_queue(const char * Separator,
     unsigned int QueueLength, unsigned int StringLength);

void print_suspended_list(const char * Separator,
     unsigned int ListLength, unsigned int StringLength);

void print_waiting_list(const char * Separator,
     unsigned int ListLength, unsigned int StringLength);
```

Figure 3.5: User-callable convenience functions for printing the simulator's behavior state lists and queues

Short Examples

Example 1:

This example demonstrates the easiest, but least flexible way of viewing the current state of the simulator. The call to **`print_simulator_state`** causes a table to be printed to the stderr stream. The table lists the active behavior instance, current behavior/channel instance, state, and the current behavior/channel method for each thread in the five simulator queues.

In this example, we have two behaviors, ***b1*** and ***b2*** running in parallel. As seen in figure 3.6, each behavior has a **`waitfor`** statement surrounded by calls to **`print_simulator_state`**. The simulator only runs one thread at a time, so if one examines the output in figure 3.7, one sees that in this case, the simulator picks ***b2*** to run first, then in the same delta cycle, 0:0, picks ***b1*** to run next and puts ***b2*** to sleep for 3 time units. At time 1:0, ***b1*** resumes and then at time 2:0, ***b2*** resumes.

```
behavior B1 ()
{
      void main(void)
      {
            print_simulator_state();
            waitfor 1;
            print_simulator_state();
      }
};

behavior B2 ()
{
      void main(void)
      {
            print_simulator_state();
           waitfor 3;
           print_simulator_state();
      }
};

behavior Main(void)
{
      B1 b1;
      B2 b2;

      int main(int argc, char **argv)
      {
          par
            {
                  b1.main();
                  b2.main();
            }
            return(0);
      }
};
```

Figure 3.6: Example code using print_simulator_state function

```
time: 0:0
Active Behavior Instance Current Instance          State Method
Main.b2          Main.b2                           Run   main
Main.b1          Main.b1                           Ready unknown
time: 0:0
Active Behavior Instance Current Instance          State Method
Main.b1          Main.b1                           Run   main
Main.b2          Main.b2                           Sleep main
time: 1:0
Active Behavior Instance Current Instance          State Method
Main.b1          Main.b1                           Run   main
Main.b2          Main.b2                           Sleep main
time: 3:0
Active Behavior Instance Current Instance          State Method
Main.b2          Main.b2                           Run   main
Main.b1          Main.b1                           Ready main
```

Figure 3.7: Output of print_simulator_state

Example 2:

The following example shows how a designer can view which behaviors are currently in the ready queue during runtime. In figure 3.8, we see that the code makes use of several variables, constants, and two function calls in order to print out the list of threads that are ready in the simulator. Figure 3.9 shows the output of these functions. Compared to the previous example, this debug code is much more intrusive, but it allows the user maximum control over the output.

```
. . .

// two-port exclusive-OR gate
behavior XOR2(in signal bit[1] a, in signal bit[1] b,
      out signal bit[1] c)
{
      void main(void)
      {
            #ifdef DEBUG
            unsigned int len;
            const bool Active = true;
            const unsigned int MAX_LEN = 80;
            const unsigned int MAX_Q = 16;
            char readyQueue[MAX_Q][MAX_LEN];
            unsigned int count;

            len = ready_queue_length();
            printf("Ready queue length: %d\n", len);

            len = ready_queue(&readyQueue[0][0], MAX_Q,
                              MAX_LEN, Active);

            printf("ready threads: \n");
            for (count = 0; count < len; count++)
            {
                  printf("%s\n", readyQueue[count]);
            }
            #endif // DEBUG

            while(true)
            {
            wait a rising, a falling, b rising, b falling;
            c = a ^ b;
            }
      }
};
. . .
```

*Figure 3.8: Excerpt from Adder2.sc. This code makes use of the **ready_queue_length** and **ready_queue** debug functions.*

```
. . .
Ready queue length: 6
ready threads:
Main.Add08.fa2.ha2.xor1
Main.Add08.fa0.ha1.xor1
Main.Add08.fa1.ha1.and1
Main.Add08.fa7.ha2.and1
Main.Add08.fa2.ha2.and1
Main.Add08.fa7.ha1
. . .
```

Figure 3.9: Output Showing Ready Threads

Chapter 4: Tracing

The third goal of our work was to create trace logs during simulation. As described in the introduction, trace files are useful both for debugging hardware-specific parts of the design as well as providing inputs for system analysis tools. One form of these trace files is very similar to trace logs created by VHDL and Verilog simulators, which when examined with an appropriate viewer, show values of signals changing on a graph plotted against time. We also added the ability for a design simulation to produce a new type of custom-format log file with the ".svc" file extension. We created the custom format to fulfill specific needs of system designs that the VCD format does not provide for. The tracing capabilities make use of the debugging information described in the previous sections, with additional support having been added to the code generator (part of the SpecC compiler) and the SpecC simulator as well.

In the introductory chapter (section 1.2.3.2) we listed a number of goals that we hoped to achieve in our implementation of tracing features for SpecC designs, including the production of log files, enabling/disabling of tracing on a per-signal basis, and tracing start/end times. Additionally we wanted to implement these features in an efficient manner that would not complicate the simulator or code generator implementations any more than was necessary. In order to meet these goals, we devised a system that modifies the SpecC compiler (scc), and adds a new ***Design.do*** file as an input during simulation to produce ".vcd" or ".svc" simulation logs. These log files may then be viewed in a GUI waveform or event viewer program.

Tracing Flow

Figure 4.1 below shows the new flow of the design compilation and simulation. The SpecC source (***Design.sc)*** is compiled and linked with the simulation library by the SpecC compiler (***scc***). ***Design.do*** acts as an input to the simulation to produce a log file (either ***Design.vcd*** or ***Design.svc)***. The log can then be examined using a graphical waveform viewer.

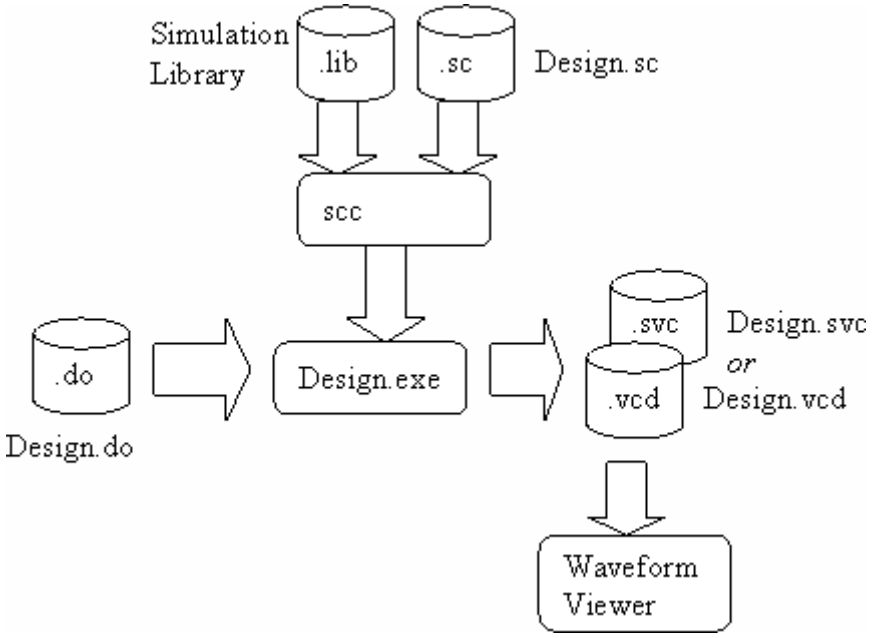

Figure 4.1: SpecC Tracing Flow

"Do" Files

We have introduced ".do" files as part of the design flow for tracing. "Do*"* files provide the design simulator with instructions that are required to produce a log of the simulation events. The required commands tell the simulator when to start and stop tracing, and what time scale (e.g., nano, pico, or fempto seconds) should be represented in log files. Optional commands tell

the simulator how to treat delta time and allow the user to selectively choose to trace particular variables (e.g., signals, behaviors, channels) or selectively disable tracing of these symbols.

Figure 4.2 shows an example of a .do file. The simulation start and end times, and time scale are all specified. Selected variables are enabled, then disabled, and ultimately enabled again. The .do file parser ensures that the correct variables will be enabled or disabled in the simulation.

```
# the start time for creating logs of the simulation
$start = 0;
# the end time for the simulation logs
# Note: the actually simulation may end before or after
# this time.  Any activity after this time will not be
# logged
$end = 1000000;

# indicate that we want to use pico seconds as our
# time unit
$timescale = ps;
# show changes to the delta time
$show_delta = true;

# enable the tracing of certain variables (signals,
# behaviors, channels, etc)
$enable = Main,
      Main.c,
      Main.c.SyncWire,
      Main.r,
      Main.s;

# now disable all variables that start with Main.c
$disable = Main.c*;

# now enable all variables
$enable = *;
```

Figure 4.2 – Sample contents of a ".do" file.

Code Generation

When a user wants to enable tracing, he must compile the SpecC code with one of the tracing flags (SVC or VCD) set. When tracing is enabled, the code generator inserts additional

code into the design executable. Tracing builds upon the debugging features described chapters 2 and 3 and also adds additional trace-specific code to the design. The code generated by the compiler does two things: it creates traceable symbols for system variables and it indicates whether or not the variable should be traced during simulation. The code generator also ensures that only one traceable symbol is created for each instance of a variable. For variables that have aliases, such as a signal that is connected into a child behavior through a port, the compiler creates a single traceable *symbol* and possibly multiple *references* to the traceable symbol.

In order to support tracing, we added a base class, **traced_object,** from which all *channel*, *behavior*, *event*, *signal*, *buffered*, and *piped* classes derive. The **traced_object** class holds an ID number, and a boolean flag to indicate whether or not the object is to be traced. In our implementation, each of the aforementioned classes inherits from **traced_object**, even when tracing is not enabled. It would be possible to provide separate libraries for tracing and non-tracing simulation if concerns about memory usage and execution overhead arise.

The trace *symbol*s and trace *reference*s that the code generator creates, supply extra information about each object that is to be traced during simulation and is required for creating the simulation log files. These trace symbols supply information to the log file writer classes such as the types of variables, name of the instance, number of bits (if a signal type), and whether or not the variable should be traced. Trace *references* are used when more than one instance name refers to the same object. After all annotations and *enable/disable* commands from the .do file have been considered, the simulator can decide which objects will be traced during simulation. Most of the extra information held in the trace *symbols* and *references* can be deleted once the headers for the vcd or svc files have been written.

Figure 4.3 shows an excerpt from the C++ code that is generated by scc when tracing features are enabled. In this example, a traceable symbol is being created for the *signal* "In08_1" The call to **_DupFullInstanceName** copies the name of the parent behavior and appends the name of the signal, "In08_1" to create a unique name for the *signal* instance. The numbers **7** and **0** indicate that the most significant bit is bit 7 and the least significant bit is bit 0. The call to **_AddSymbol** creates a traced symbol object and the call to **ToggleTracing** acts on the newly-created symbol.

```
In08_1._AddSymbol( _DupFullInstanceName("In08_1"), 7, 0,
_Trace::SYM_SIGNAL))->ToggleTracing(_Trace::_TRACE_TRUE);
```

Figure 4.3: Excerpt from the C++ code that is generated during compilation when tracing features are enabled.

It may seem odd that we have to explicitly enable tracing on a symbol once it has been created. The reason for this is that the SpecC compiler must also take *annotations* into account when generating the code. Persistent annotations are a feature of SpecC that are similar to comments, but are not removed by the preprocessor and are visible to the compiler and other design tools. They allow the user to specify additional information about a variable that is not explicitly part of the language [1]. The designer may decide to use the **_SCE_TRACE** annotation in the SpecC source to indicate that a variable is to be traced or that tracing should be disabled for a particular variable. Later, at runtime, the designer may then override the annotations by enabling or disabling tracing through commands in a ".do" file.

There are cases where an IP vendor would like to be able to supply IP code and want to explicitly disallow tracing for this code. The SpecC compiler provides an IP protection mechanism that secures SpecC designs against being copied, modified, or reverse-engineered

[19]. In accordance with this, the exception to the rule of using annotations and allowing tracing to be enabled through commands in ".do" files is that SpecC code compiled with the IP option set, is not a candidate for tracing.

Value Change Dump (VCD) Files

Before describing the functionality of the VCD file, it will be useful to describe the VCD file format itself. The VCD format was developed specifically for producing simulation traces of Verilog designs [8]. The VCD files consists of a few sections: the header, variable declarations, initial states of variables, and time/value changes. The header section provides the date, simulator version, and time scale of the simulation and looks something like the following:

```
$date
Fri Oct 7 15:00:48 2005
$end
$version
UCI SpecC VCD Writer v0.001
$end
$timescale
1ps
$end
```

Figure 4.4: Header Information in a .vcd File

The next section, the variable declarations, lists all of the variables that will be traced during the simulation as well as any aliases for the variables. An example may look like this:

```
$var wire 1 3 Main.Add01.fa0.ha1.and1.a $end
$var wire 1 4 Main.Add01.fa0.ha1.and1.b $end
$var wire 1 * Main.Add01.fa0.ha1.and1.c $end
$var wire 1 , Main.Add01.fa0.ha2.and1.c $end
$var wire 32 ( Main.b.z $end
$var event 1 . Main.e1 $end
$var event 1 / Main.e2 $end
$var wire 1 7 Main.Add01.c_out $end
$var wire 1 7 Main._scc_open_port_0 $end
$enddefinitions $end
```

Figure 4.5: Variable Definitions in a .vcd File

Each variable starts with the keyword, **$var** and ends with **$end**. The type field is specified next. In this example we see only 2 types: *wire* and *event*. For a complete list of types see [8]. Next comes the number of bits. In this example, the variable "Main.b.z" has 32 bits. The next field is an identifier code for the variable. Since VCD files quickly become quite large, the identifier is encoded using visible ASCII characters in the range '!' to '~' (decimal 33 to 126) [8]. As compared to encoding the names as decimal or even hexadecimal numbers, a great deal of compression can be achieved using this base-94 (33 through 126) encoding scheme. Next is the name of the variable to be traced. Names may be "fully qualified" as shown or may make use of **$scope** commands to declare variable names in a hierarchical fashion. Note in this example that the variables "Main.Add01.c_out" and "Main._scc_open_port_0" have the same identifier code -- "7". The actual signal is "Main._scc_open_port_0", while "Main.Add01.c_out" is a reference to this same signal. This way the value change only needs to be logged once, but both variables will appear to change in the waveform viewer.

The last section, contains changes in simulation times and changes in values of the variables. It looks something like this:

```
#0
0*
0+
srunning #
swaiting &
#1
swaiting #
srunning &
0*
1+
#2
1*
0+
#3
```

Figure 4.6: Value changes in a .vcd File

Simulation time changes are marked by the '**#**' character, followed by a value. Value

changes are listed on separate lines with the new value followed by the identifier code. In this example, we also make use of the fact that the GTKWave viewer supports strings. “srunning” means a string with the value “running”.

Figure 4.7 shows the wave form corresponding to the value changes listed in figure 4.6. The behavior ***xor1*** (ID string = '#') changes from *running* at time = 0, to *waiting* at time = 1, while the behavior ***and1*** *(ID string = '&')*changes from *waiting* at time = 0, to *running* at time = 1. The signal ***y*** (ID string = '*') changes from a value of 0 at time = 0, to a value of 1 at time = 2. Another signal, ***z*** (ID string = '+'), changes from 0 at time = 0 to a value of 1 at time = 1 and then changes again to 0 at time = 2.

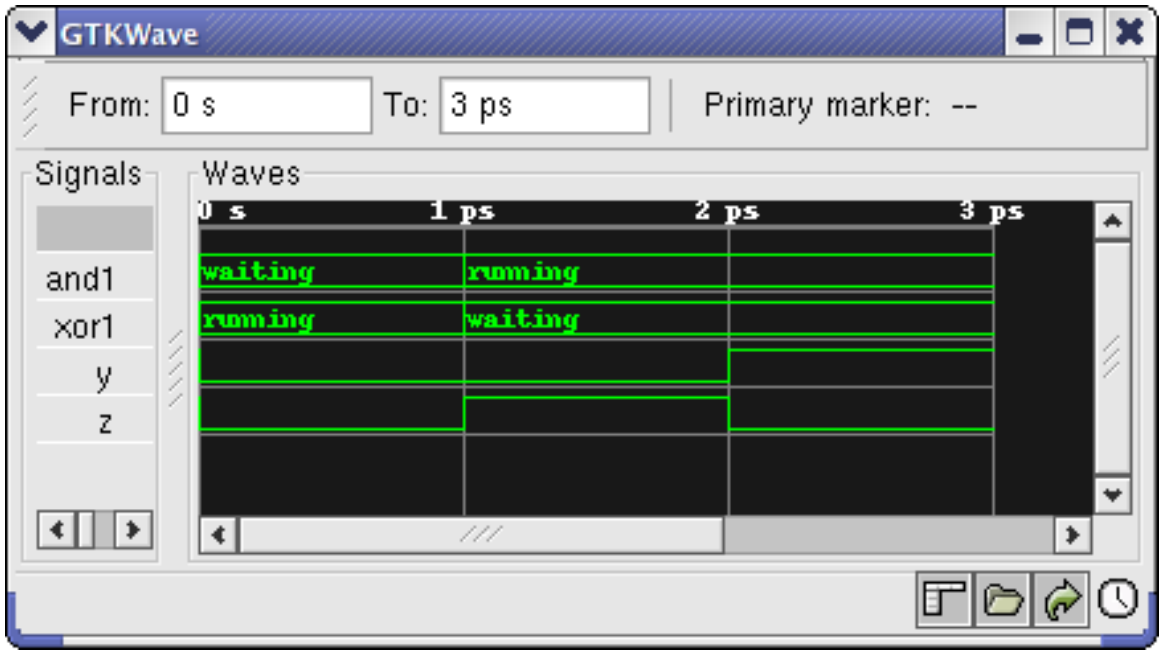

Figure 4.7: Waveform Corresponding to Events in VCD File

Initial values for variables are dumped in the same format as shown above, but are wrapped with the keywords, $**dumpvars** and **$end**. Again, for more details on the VCD format, please consult [8].

System Value Change (SVC) Files

SVC files are very similar in format to VCD files, but incorporate a few new features specific to system design and the SpecC language in particular. Most of the changes come directly from the fact that the VCD format was designed for HDLs and therefore lacks system design tracing features.

The new features supported by the SVC format are the following:

- strings
- delta time
- additional SpecC data types – *behaviors*, *channels*, *signals*, etc
- simulation events in addition to simple value changes
- support for signal slicing and concatenation.

Support for strings is a feature that some VCD viewing programs have. The viewer that we have used for our experiments, ***GtkWave***, supports strings overloaded on the *real* variable type, but the VCD specification does not include strings. We add strings as a variable type for SVC files. String values are represented by the letter 's' followed by a normal character string sequence.

Delta time is another feature missing from the VCD specification. One can workaround delta time by multiplying “real” simulation time by some constant (e.g., 10, 100, 1000) and add delta cycles on top, but this workaround has several problems associated with it. First, the times units are incorrect because the real cycles have to be multiplied by some number. For example, if the real time scale is in pico seconds and one multiplies by 1000 in order to allow for delta cycles, the time scale becomes nano seconds for real simulation time. As long as the designer can keep track of the fact that the units are incorrect, the workaround suffices, but it is somewhat confusing.

The second problem is that the number of delta cycles within real cycles will normally vary throughout the simulation. In some real cycles there may be no delta cycles or only a few, while in other cycles there may be thousands of delta cycles. This makes choosing an appropriate multiplier difficult and may lead to a great deal of wasted space in the viewer which has to make room for these delta cycles that may not occur.

SVC files support delta cycles by separating them from real cycles with a ':'. For example, "1000:2" would indicate delta cycle 2 of real cycle 1000. Ideally, an SVC viewer should support the notion of "elastic" time. In elastic time, the viewer will expand the time axis during regions of heavy delta cycle activity and contract the time axis during areas of little or no delta cycle activity. This allows for all delta cycles to be seen without unnecessarily wasting screen space and while using the correct time units.

We added several new variable types to support system level designs: b*ehaviors, channels, signals, buffered,* and *piped* types. Explicitly supporting these data types allows the SVC viewer to handle these types of variables in more intelligent ways. For example, a *signal* in SpecC acts like both an event and also has a value. The VCD format only supports value changes, so supporting *signal*s requires some workarounds. In the SVC format we can differentiate between a signals acting as an *event* and a signal acting as a variable with a value attached to it.

Rather than simply showing states for behaviors, the approach we take with the SVC format is to show *simulation event*s. The SVC viewer can easily derive the state of behaviors from these events. Along with these simulation events, we support parameters for the events. As an example, for a *trap* event, there are parameters for the *event* or *signal* that caused the trap, the exception handler, and the behavior that is being aborted. Other examples of simulation events would be *fork, join, start, wait, exit,* and *interrupt*. (See Appendix C for a complete list of the supported simulation events.)

Figure 4.8 illustrates how an SVC viewer can interpret these system events to derive the state of a behavior at any time during a simulation. With the exception of interrupt returns, all simulation events map to exactly one behavior state. As an example, a **`waitfor`** always causes a transition to the *sleeping* state. The return from interrupt event uses a parameter that the simulation engine must supply to indicate which state the behavior returns to following an interrupt. Note that in order to keep the diagram from becoming too cluttered, not all of the interrupt/ return from interrupt arcs are shown.

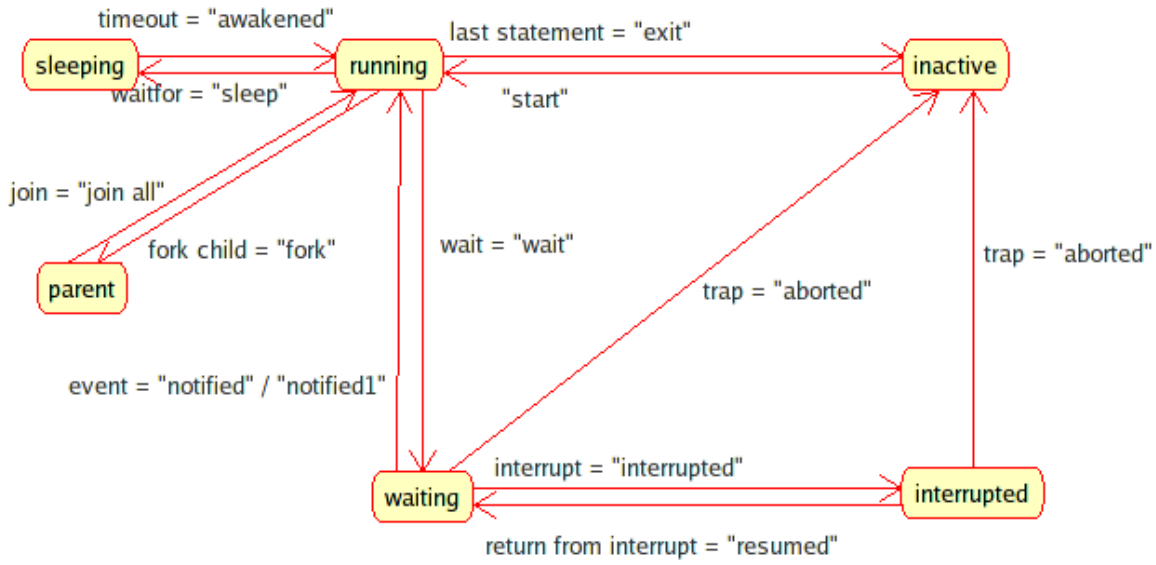

Figure 4.8: State Diagrams for Behaviors in a System Design

The SpecC language allows for *slices* of signals to be used in *port mappings* [1]. If a signal in a port map is a slice of another signal, it effectively creates an alias to certain bits in the second signal. Likewise, if a signal in a port map is a concatenation of slices from two or more other signals, it creates an alias to bits from both of the source signals. The VCD format does not support this notion, so supporting tracing of these port mapped signals would require considerable extra effort within the simulation engine to log value changes during simulation.

The SVC format provides syntax to express slicing and concatenation when declaring a variable, so an intelligent viewer can infer changes to the port mapped signals based on changes to any of the source signals.

Figure 4.9 shows an example of an SVC file. As one can see, the header and variable declaration sections look very similar to those in a VCD file. We see two of the new variable types in this example: piped and behavior. This example also shows delta time values and several simulation events -- *start, func_call, sleep, fork, join_all,* and *exit* – along with ordinary value changes.

```
$timescale
     1ps
$end

$var behavior 0 ! anonymous $end
$var piped 32 " Main.b.a_internal $end
$var behavior 1 # Main.b.b1 $end
$var behavior 1 $ Main.b.b2 $end
$var behavior 1 % Main.b.b3 $end
$var piped 32 & Main.b.x $end
$var piped 32 ' Main.b.y $end
$var piped 32 ( Main.b.z $end
$var behavior 1 ) Main.b $end
$var behavior 1 * Main $end
$enddefinitions $end

#0:0
$dumpvars
b0 "
b0 &
b0 '
b0 (
$end
$start *
$func_call * ) main
$start )
$fork ) #
$fork ) $
$fork ) %
$start #
$sleep # 1
#1:0
$awakened #
$exit #
$join_all )
b11111111 '
b111 (
$fork ) #
$fork ) $
$fork ) %
$start #
$sleep # 1
$start $
$sleep $ 1
#1:1
```

Figure 4.9: Example Contents of an .svc File

Design and Implementation of Tracing Features

The code changes required to implement the new tracing features can be roughly broken up into three areas: code generator changes, simulation engine changes, and the creation of SVC and VCD log file writers.

Code generator changes

The changes to the code generator (part of the SpecC compiler) were relatively minor. In addition to the code that gets added for the new debug features, the code generator must now create traceable symbols for variables that are to be traced and examine annotations to decide whether or not to enable tracing for each variable instance.

The compiler must detect all declarations for variables – **`signal`**s, **`piped`**, **`behavior`**, **`channel`**s, etc – as well as behavior port declarations. At the point where the the variable would be initialized – usually in a behavior's constructor – the compiler must insert code to created a traceable object. The compiler must examine the variable's type in order to determine the traced object type – **`signal`**, **`behavior`**, **`channel`**, etc -- , the most and least significant bits (if applicable), and the instance name. Then the compiler must examine the annotations, to see if an annotation has been attached to the variable that explicitly enables or disables tracing. The compiler then uses this information to set the appropriate flag to enable/disable tracing on the object.

Simulator changes

Creating logs of the events and value changes that occur during simulation requires the active participation of the simulation engine. Every time a value changes in an object of type **`signal`**, **`piped`**, or **`buffered`**, the simulator must create a log of this value change. Likewise every time a SpecC event occurs, the simulator must log this occurrence. Any time that a simulation event such as a behavior starting, exiting, or fork occurs, the simulator must log these events as well.

We created a tracing API, which has been included in Appendix F, to handle all of the logging calls that the simulator must manage. The API calls are then mapped to either a VCD file writer or an SVC file writer, depending on which type of writer has been activated at compile time.

Log File Writers

VCD file writer

The VCD file writer portion is used for creating value change dump files during a SpecC simulation. The basic function of the VCD file writer is to accept calls to log value changes and print these value changes in the correct format required for VCD files. Our implementation uses ideas from Anthony Bybell's implementation of a VCD file writing utility [11].

Our version of the VCD writer does differ from Anthony Bybell's implementation in several important ways. We implemented a different system for keeping track of symbol tables, our version uses an object-oriented design, and supports an event logging API that is specific to the SpecC language.

SVC file writer

The SVC file writer uses the same tracing API as the VCD writer, but handles the API calls in somewhat different ways than the VCD writer. As an example, the **`log_trap`** call has

3 arguments – *running ID*, *handler ID*, and *event ID*. The VCD writer is not able to write out all the information required to show the relationship between an event that caused a trap to occur, the behavior that was aborted due to the trap, and the trap handler. Instead the VCD writer only writes out the ID of the behavior that was aborted. Since the SVC file format allows for multiple arguments when logging simulation events, the SVC writer is able to show the chain of events leading to the abortion of the affected behavior.

Chapter 5: Experiments and Results

In this chapter, we discuss how the new debug and trace capabilities have been used by other researchers in the CECS group and how we were able to demonstrate that the new features work with an industrial-strength example.

We were able to use the experiences of two graduate students in the CECS group at UCI to examine the effectiveness of the new debugging features described in chapters 2 and 3. These graduate students used the debugging features just as any other user would in order to trouble-shoot their designs.

In order to test the completeness and usefulness of the tracing features described in chapter 4, we used the tracing option to help debug a student project that contained errors and also on an industrial-strength design of an MP3Decoder.

Experiences of debug feature users

Case Study 1

One user of the debugging features, Student A, was able to successfully use the new debug functions to troubleshoot his ARM core and AMBA models [28]. He made the following comments about his experiences,

It is always difficult to debug in an environment that supports concurrency like the SpecC simulator..... I could easily find all the behaviors that were waiting

and the active behavior. I was able to trace the behavior that never became active when it was supposed to be active. The conventional way was to add printf statements in order to get the feel of execution and then make a guess about the faulty behavior. [The new debugging features] definitely saved lots of time of mine, say around 2 to 3 hrs of debugging.

Case Study 2

Another researcher, Student B has been making use of the debug features for several months now and has made them a part of his regular debugging "toolbox". He has used the debug features while developing bus functional and TLM models for the AMBA [30] and CAN [31] buses, as well as for Result-Oriented Modeling of the AMBA bus [29].

He indicates that the debug functions that allow one to determine which instance of a behavior is currently active, are particularly useful when debugging bus models. Since bus models contains many instances of bus masters and slaves, being able to determine which particular instance is active is a critical task in debugging.

He has discovered an advanced technique for using the debug functions inside of ***ddd***, without needing to modify his SpecC code. He finds that being able to use the functions without modifying his code is especially useful and time-saving. ***ddd*** supports calling functions from within the debugger. He uses **`print_active_instance`** and **`print_time`** from within the debugger to determine the simulation time and the name of the instance that is currently active while he steps through code in the debugger. These two functions continually update, as he steps through the code – providing an always up-to-date snapshot of the current state. Figure 5.1 shows an example of a similar method of how a user may display the current simulation time, delta time, and the active instance in such a way that they update automatically as the user steps through the code.

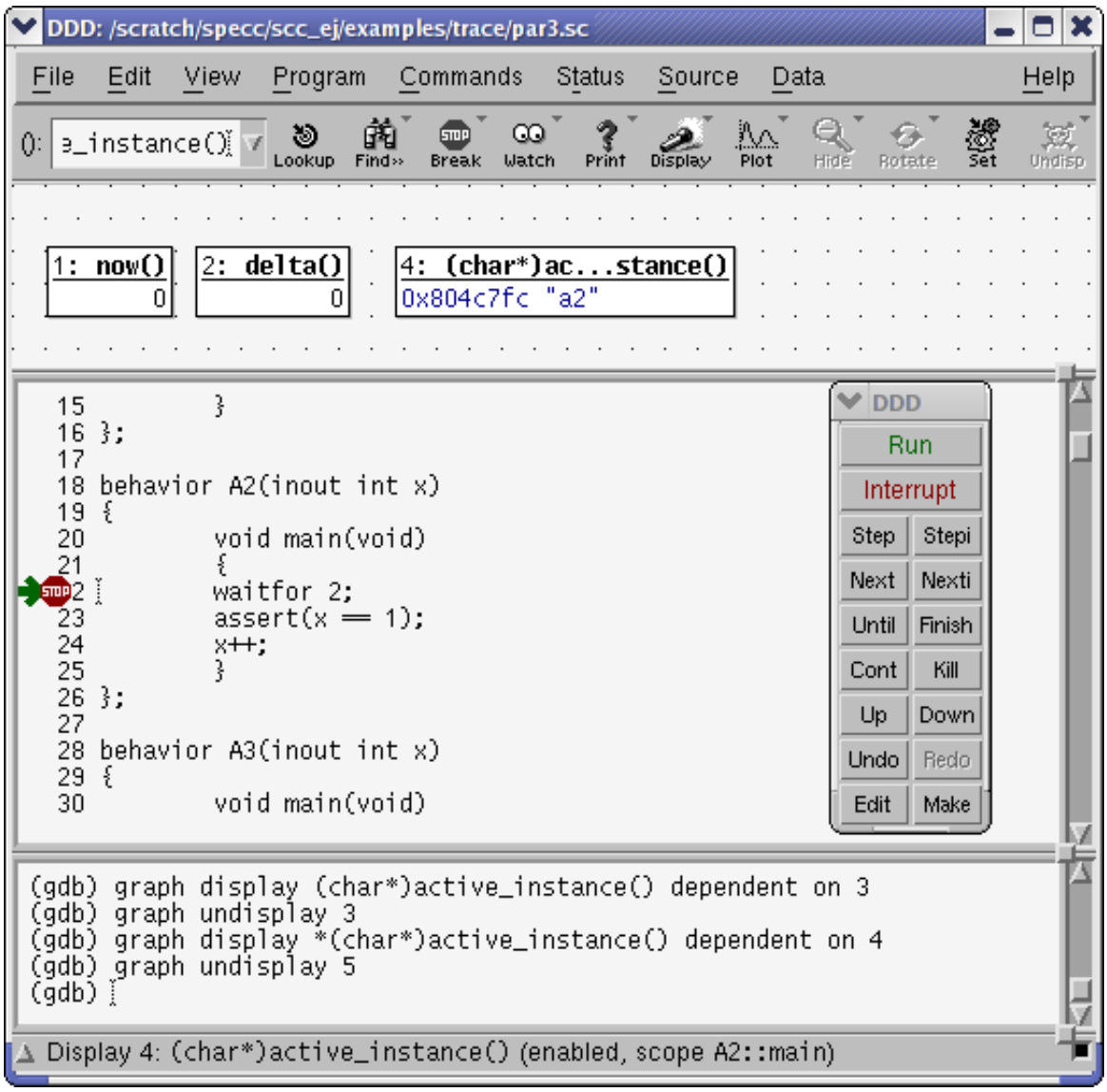

Figure 5.1: Screen shot of ddd with the Time and Active Instance Automatically Updating.

He uses **`print_process_states`** on an as needed basis from the ***ddd*** command line. Using this debug function, he is able to obtain a more complete snapshot of the current state of behaviors in the simulation. This ability is especially useful for quickly examining the effect of SpecC event notification on the various behaviors in the system.

Quantifying the time-savings of good debugging tools is difficult, but he was able to recall a particular instance where he estimates that the debugging functions saved him

approximately one day of effort. On this occasion, he was trouble-shooting a problem in his model and was pursuing a false lead when he noticed that the **`print_active_inst`** function was showing that an unexpected behavior was active. Observing this, he was able to abandon the false lead and quickly identify the real problem and fix it within a few minutes. Because of this experience, he has since added several shortcut commands to his ***ddd*** setup such that these functions are immediately available every time he uses ***ddd*** and are a standard part of his debugging process.

Tracing experiences

Student Project

We were also able to use the tracing features to help debug a student project. The student's program contained a small mistake that was not readily apparent. The design featured 2 buses and 2 behaviors. One behavior was to act as a slave, while the other was to act as a bus master.

The symptom was that the clock in the slave behavior did not appear to be toggling, but the root cause was unknown. After simulating the design with tracing options enabled, we were able to confirm that the slave behavior's clock was not toggling, but that the master behavior's clock was toggling as expected. Figure 5.2 shows a screen shot of the waveform produced during the simulation. Armed with this knowledge, we were able to inspect the model more closely to determine the root cause of the problem, that the two behaviors were unable to communicate because they were not properly connected through a channel (bus). The student had mistakenly attempted to use 2 buses, where only one bus was needed to properly connect the two behaviors. Using the trace output allowed us to quickly confirm the suspected symptom and then root cause the problem all in a matter of minutes.

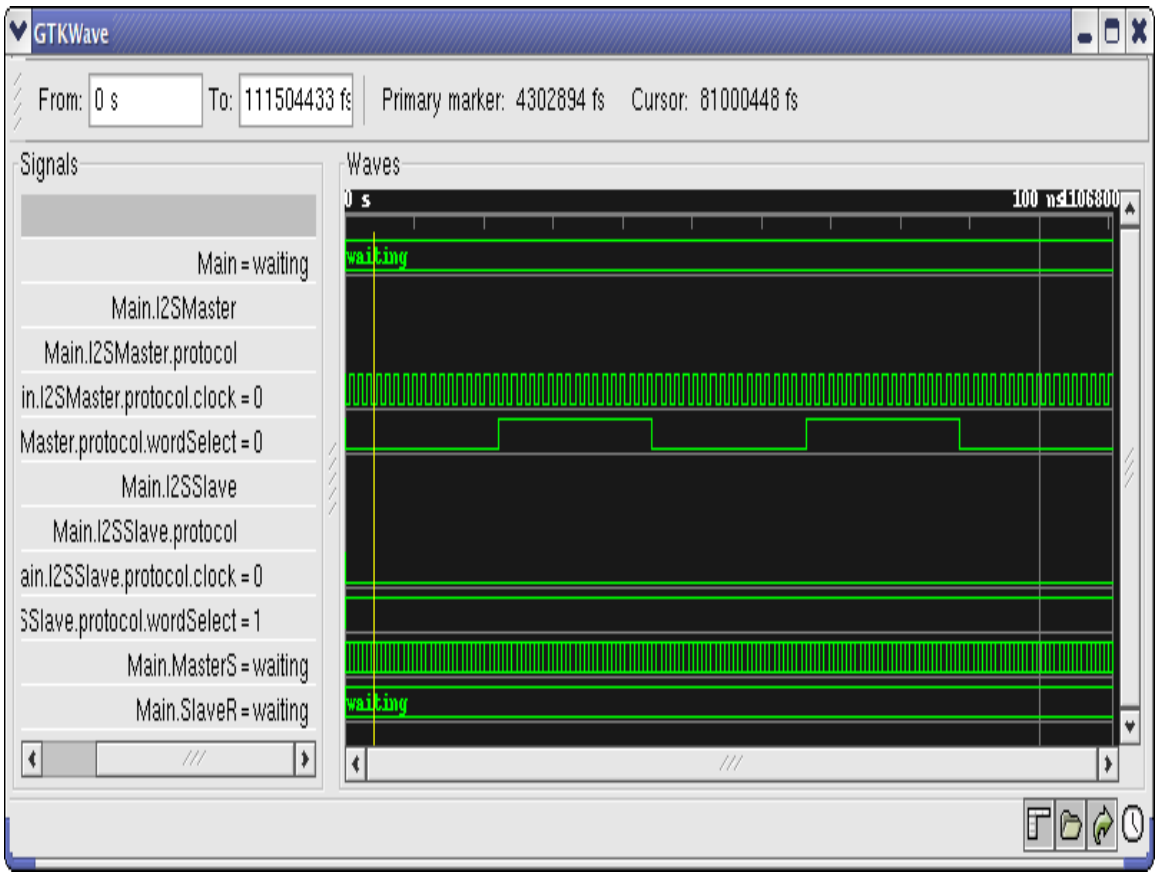

Figure 5.2: Screen shot from student example.

Mp3Decoder

One of our goals for creating tracing features was to demonstrate the new features using an industrial-strength, real world example. We were able to successfully use the tracing features to produce simulation traces for an MP3 audio decoder as described in [23]. This example allowed us to test the tracing features using various different types of models including specification, scheduler, network, transaction level, and communication. Figure 5.3 shows a screen shot of several behavior states as seen in a waveform viewer and figure 5.4 shows a screen shot of several signals from the MP3 decoder design changing in time.

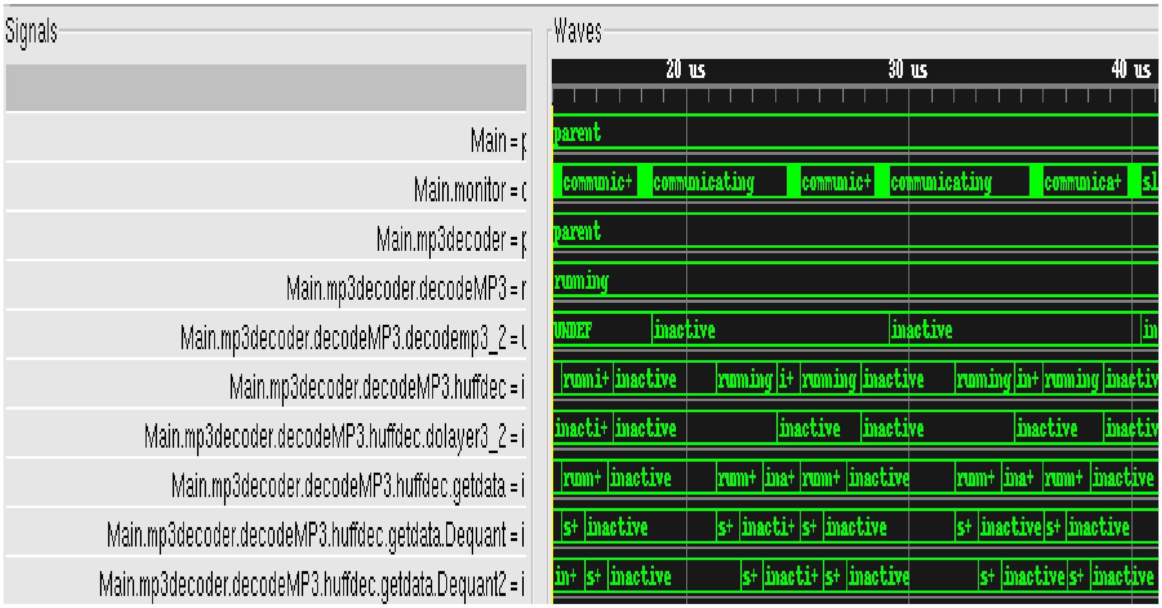

Figure 5.3: Screen shot of MP3 Decoder behavior states as seen in a waveform viewer.

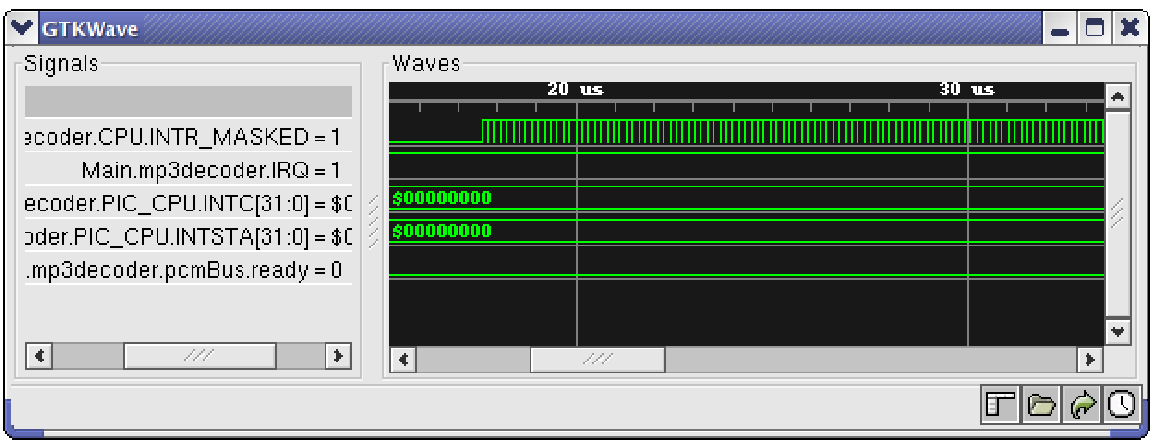

Figure 5.4: Screen shot of MP3 Decoder signals as seen in a waveform viewer.

Figure 5.5 Shows simulation times for the MP3 Decoder TLM Model compiled with no debugging, debug only, and tracing flags. Three times are shown where tracing was enabled. The time for the simulation with no debugging is our baseline for speed tests. We calculate the slowdown as a simple ratio T2/T1, where T2 is the simulation time being measured and T1 is the

simulation time of the baseline. This is simply the inverse of the accepted formula for measuring speedup.

Test	Time (seconds)	Slowdown
1) No debugging	111.09	1.00
2) Debug only	113.76	1.02
3) Tracing, but all symbols disabled in .do file; start / end times = 0	115.13	1.04
4) Tlm model tracing a subset of symbols; writing to /dev/null	250.53	2.26
5) Tlm model using subset of symbols; writing to real file	298.46	2.69

Figure5.5: Simulation Time Comparison

From the table of results, we see that enabling basic debug features (test 2), has little effect on the simulation time. Enabling tracing, but not actually tracing any variables and setting the start and end times for simulation both to 0 (test 3) also has little effect. Both of these measurements may be thought of as a reflection of the pure overhead due to debug and tracing features.

The overhead in tests 2 and 3 comes mostly from the extra work created to maintain the introspection features. As we described in sections 2.2 and 3.2, the code generator inserts extra function calls (**`Set/RestoreActiveInst`**, **`Set/RestoreCurrentInst`**, and **`Set/RestoreMethod`**) in each behavior and channel method to maintain the *active behavior, current class instance,* and *current method name*. The code generator ensures that the least amount of work is performed to set the three pointers required to maintain this information. As an example, “private” behavior methods, (i.e., methods other than ***main***) do not need to update

the *active instance* pointer, as it will not change as the result of the method call.

Test 4 illustrates that enabling tracing of a set of variables can have a significant effect on the simulation time. In this case, the effect is to slow the simulation by a factor of 2.26. This is to be expected, since the simulator has to note when significant simulation events have occurred, check to see if the relevant variable are enabled for tracing, and then must log the event to a text file. In this test, we write to a "null" file, to minimize the cost of file I/O on the measurements.

The final test case (test 5) shows the effect of tracing when writing to a real file. The cost of file I/O is significant, resulting in a slowdown of 2.69 as compared to 2.26 in the previous case. This case demonstrates the type of performance a real user might expect to see. Of course, the actual experience will depend on a number of factors, including the length of the simulation being traced, the number of variables being traced, the operating system and file I/O routines.

Chapter 6: Summary and Future Directions

Summary

In this report, we have described three new contributions to the SpecC design environment that aid the debugging and analysis of system-level designs. The three contributions are:

- Instance identification at runtime
- Simulator state observations at run-time
- Event logging for waveform displays after simulation

As System-on-a-Chip designs combine aspects of both hardware and software design, the new capabilities were required to aid both hardware and software development. As the results demonstrate, the new features have been integrated into the SpecC compiler and simulator in a such a way as to provide efficient debugging and tracing capabilities.

Instance Identification at Runtime

For the aspects of system design that are most like software design, we added a new API of 12 user-callable basic debug functions. This functions are mainly used to uniquely identify

instances of behaviors and channels in a design where multiple instances of a particular class may be present.

Simulator State Observations at Run-time

We also provided a second API consisting of 17 user-callable simulator state functions. These functions provide the user with a "peek" at the current state of the simulator. They allow the user to see how the states of behaviors in the system change during simulation and are especially useful for debugging deadlock situations.

Event Logging

For the aspects of the system design that are closer to traditional hardware design, we added the ability for the SpecC simulator to produce simulation traces. These traces may then be examined in a waveform viewing program. This fills a gap in the SpecC environment that users of HDL's expect to find in a design environment. Additionally, system traces may also be useful as design analysis tools. The logs may be used to by a system architect to determine which direction of architectural explorations will be most beneficial without as much guess work.

We have also demonstrated that these new capabilities have been applied to real SpecC designs by users at CECS for projects such as ARM processor models; AMBA and CAN bus models, and an MP3 Decoder.

Future Work

We feel that the work presented in here is a major step forward for users of the SpecC design environment, but alas improvements can always be made. Relatively minor changes can be made to better support less commonly used features of the SpecC language and to further improve efficiency during simulation. The next major goal to improve tracing and analysis of system simulations is to design and implement a custom viewer for the .svc file format.

In terms of supporting additional features of the SpecC language, we currently only support tracing of **`behavior`**s, **`channel`**s, **`event`**s and **`signal`**s, **`piped`**, and **`buffered`** variables of built-in integral types. The SpecC language also supports more complex data types such as *signal float* or *signal enum_type*. We have not attempted to support tracing of these types. For lack of time, we also have not implemented tracing of built-in types such as *int*s, *long*s, and *float*s, which could be supported in the longterm.

Also for lack of time, we did not attempt explore any methods for compressing simulation log files beyond what was already specified in the VCD file format.

Creating a custom waveform and behavior states analysis viewer is the next major goal for tracing tools. The SVC format was designed to provide additional information that the VCD format does not support. A custom viewer should be able to use this additional information to provide the user with powerful debug and analysis tools. In particular, being able to see the relationships among behavior states graphically as a function of time will provide the designer or system architect with meaningful information that will help guide the design process.

Bibliography

1. R. Doemer, A. Gerstlauer, D. Gajski. SpecC Language Reference Manual, v 2.0. SpecC Technology Open Consortium. www.specc.org. 2002.
2. H. M. Deitel, P. J. Deitel. C How to Program. Prentice Hall. Englewood Cliffs, New Jersey. 1992. p. 426-427.
3. CECS. Source code for the SpecC Reference Compiler and Simulator, version 2.0. June 24, 2004. http://www.ics.uci.edu/~specc/reference/ .
4. W. Mueller, R. Doemer, A. Gerstlauer. The Formal Execution Semantics of SpecC. Proceedings of the International Symposium on System Synthesis, Kyoto, Japan, October 2002.
5. Scott D. Meyers. Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition. Addison-Wesley. Boston, Massachusetts. 1997.
6. Linux Trace Toolkit Website. http://www.opersys.com/LTT.
7. Wind River Website. http://www.windriver.com/portal/server.pt .
8. IEEE. IEEE Standard Verilog, Std 1364-2001. IEEE. 2001.
9. GTKWave Website. http://www.cs.manchester.ac.uk/apt/projects/tools/gtkwave .
10. Bjarne Stroustrup. C++ Programming Language, Third Edition. Addison-Wesley. Boston, Massachusetts. 1997.
11. Anthony J. Bybell. Source code for vcd_write. http://www.ibiblio.org/pub/Linux/apps/circuits/libvcddump-0.1.2.tgz 2002.
12. Gnu Website. The GNU Project Debugger. http://www.gnu.org/software/gdb/gdb.html .
13. Gnu Website. Data Display Debugger. http://www.gnu.org/software/ddd/index.html .

14. Microsoft Website. Visual Studio Home. http://msdn.microsoft.com/vstudio .
15. Metrowerks Website. Metrowerks CodeWarrior Home. http://www.metrowerks.com/mw/default.htm .
16. Eclipse Website. Eclipse.org Main Page. http://www.eclipse.org .
17. Stan Shebs, John Gilmore. GDB Internals: A Guide to the Internals of the GNU debugger. http://www.gnu.org/software/gdb/gdb.html . 2004.
18. Julia Menapace, Jim Kingdon, David MacKenzie. The "stabs" Debug Format. http://www.gnu.org/software/gdb/gdb.html. 2004..
19. D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, S. Zhao. SpecC : Specification Language and Methodology. Kluwer Academic Publishers. Boston. 2000.
20. Minimalist GNU for Windows Website. http://www.mingw.org/.
21. Ines Viskic. Master of Science Thesis: Analysis and Conversion of C-based Architectural System Models. University of California, Irvine. 2005.
22. A. Gerstlauer, R. Doemer, J. Peng, D. Gajski. System Design: A Practical Guide with SpecC. Kluwer Academic Publishers, Boston, June 2001.
23. Pramod Chandraiah. Master of Science Thesis: Specification and Design of a MP3 Audio Decoder. University of California, Irvine. 2005.
24. Modelsim Website. ModelSim. http://www.model.com/.
25. Cadence Website. NC-Verilog. http://www.cadence.com/products/functional_ver/nc-verilog/index.aspx.
26. Blue Pacific Computing Website. BlueHDL. http://www.bluepc.com/bluehdl.html.
27. SystemC Website. Welcome to the SystemC Community. http://www.systemc.org/.

28. Gautam Sachdeva. Master of Science Thesis: Integration of an ARM core in a System Design Flow. University of California, Irvine. 2006.
29. G. Schirner, R, Doemer. Using Results Oriented Modeling for Fast yet Accurate TLMs. Center for Embedded Computer Systems, TR 05-05. 2005.
30. G. Schirner, R. Doemer. System Level Modeling of an AMBA Bus. Center for Embedded Computer Systems, TR 05-03. 2005.
31. G. Schirner, R. Doemer. Abstract Communication Modeling: A Case Study Using the CAN Automotive Bus. Proceedings of International Embedded Systems Symposium. Springer, Manaus, Brazil. August 2005.

Appendix A: API for Debug Functions

```
// obtain the class name of the last behavior that has called
//main()
const char * active_class(void);

// obtain the instance name of the last behavior that has called
// main()
const char * active_instance(void);

// obtain a copy of the full name of the active instance
char * active_path(char * Dest, const unsigned int Length);

// obtain the class name of the last behavior or channel to call
any function
const char * current_class(void);

// obtain the instance name of the last behavior or channel to
call any function
const char * current_instance(void);

// obtain a copy of the full name of the current instance
char * current_path(char * Dest, const unsigned int Length);

// print the class name of the last behavior that has called
// main()
void print_active_class(void);

// print the instance name of the last behavior that has
//called main()
void print_active_instance(void);

// print the full name of the active instance
// specify the max number of characters to print
void print_active_path(const unsigned int Length);

// print the class name of the last behavior that has called main
void print_current_class(void);

// print the instance name of the last behavior that has called
main()
void print_current_instance(void);

// print the full name of the current instance
void print_current_path(const unsigned int Length);
```

Appendix B: API for Simulator State Functions

```
// print out a nicely-formatted list of states of behaviors
void print_process_states();
// same as above, but also prints the simulation and delta times
void print_simulator_state();
//get the length of the current queue of ready threads in the
simulator
unsigned int ready_queue_length();
unsigned int running_queue_length();
unsigned int waiting_list_length();
unsigned int sleeping_queue_length();
unsigned int suspended_list_length();


// Names is an array of strings that the user supplies to copy
the
// ready queue into.
// StringLength is the max number of chars to copy for each
thread name
// QueueLength is the size of the string array that the user has
// provided
// return the number of items copied (<= QueueLength)
// Active = true means list active behaviors, while Active =
false means
// list current behavior or channel
unsigned int ready_queue(char * Names, unsigned int QueueLength,
unsigned int StringLength);

unsigned int running_queue(char * Names, unsigned int
QueueLength, unsigned int StringLength);

unsigned int waiting_list(char * Names, unsigned int ListLength,
unsigned int StringLength);

unsigned int sleeping_queue(char * Names, unsigned int
QueueLength, unsigned int StringLength);

unsigned int suspended_list(char * Names, unsigned int
ListLength, unsigned int StringLength);
```

```
void print_ready_queue(const char * Separator,
     unsigned int QueueLength, unsigned int StringLength);

void print_running_queue(const char * Separator,
     unsigned int QueueLength, unsigned int StringLength);

void print_sleeping_queue(const char * Separator,
     unsigned int QueueLength, unsigned int StringLength);

void print_suspended_list(const char * Separator,
     unsigned int ListLength, unsigned int StringLength);

void print_waiting_list(const char * Separator,
     unsigned int ListLength, unsigned int StringLength);
```

Appendix C: SVC Event Commands

The events logged in the svc files may be value changes for variables (signals, bits, etc), true SpecC events, actions that would cause a behavior to transition between states, or method call/returns.

Except for variable value changes, all events start with a '$' symbol, followed by a description of the event. Variable value changes follow the rules of VCD files.

The following is a description of the individual events and their parameters:

$fork parent_id_code child_id_code

$join_all parent_id_code

$start identifier_code **-- a behavior has started its main method**

$notified identifier_code event_id – note: a behavior may be notified, but not actually
awakened if it is using "wait and" semantics

$notified1 identifier_code event_id

$notified_awakened identifier_code – note: this means that a behavior was notified by an event and was actually awakened. No event id is necessary because the notify/notify1 log will already appear in the log

$awakened identifier_code – awakened after a timeout from a waitfor

$wait identifier_code event_id – **waiting on an event**

$sleep identifier_code time_delay – **corresponding to a waitfor statement**

$resumed identifier_code – **resumed from an interrrupt**

$exit identifier_code – **a behavior has exited from main**

$aborted running_id handler_id event_id – **a trap occurred**

$interrupted running_id_code handler_id_code event_id_code

$func_call caller_id callee_id function_name **-- a behavior or channel has called a method**

$func_return caller_id callee_id function_name – **a behavior or channel has returned from a method call**

$event identifier_code **-- a specC event**

// the following states should only appear at the beginning of the log as initial states

$waiting_state identifier_code

$running_state identifier_code

$interrupted_state identifier_code

$sleeping_state identifier_code

$parent_state identifier_code

Appendix D: SVC EBNF

```
value_change_dump_definitions ::=
      {declaration_command } { simulation_command }

declaration_command ::= declaration_keyword [ command_text ] $end

simulation_command ::=
      simulation_keyword { value_change } $end
      | $comment [comment_text ] $end
      | simulation_time
      | value_change

declaration_keyword ::= $comment | $date | $enddefinitions |
$scope | $timescale | $upscope | $var | $version | $slice

simulation_keyword ::=
      $dumpall | $dumpoff | $dumpon | $dumpvars

simulation_time ::=
      #decimal_number[:delta_time]

delta_time ::= decimal_number

value_change ::= scalar_value_change | vector_value_change |
command_event

scalar_value_change ::= value identifier_code

value ::= 0 | 1 | x | X | z | Z

vector_value_change ::= b binary_number identifier_code
      | B binary_number identifier_code
      | r real_number identifier_code
      | R real_number identifier_code
      | s string identifier_code
      | S string identifier_code

declaration_vars ::= $var var_type size identifier_code reference
$end
      | $slice var_type size num_slices {slice} identifier_code
reference $end

var_type ::= event | integer | string | real | signal | buffered
| piped | behavior | channel

size ::= decimal_number
```

```
num_slices ::= decimal_number

slice ::= reference

reference ::= identifier
      | identifier [bit_select_index ]
      | identifier [ msb_index : lsb_index ]

index ::= decimal_number

identifier_code ::= { ASCII character }
[Note: only the printable ASCII characters]

command_event ::= $fork parent_id_code child_id_code | $event
identifier_code
      | $join parent_id_code | $start identifier_code | $waiting
identifier_code
      | $running identifier_code | $ready identifier_code
      | $sleeping identifier_code | $suspended identifier_code
      | $resumed identifier_code | $abort identifier_code
      | $exit identifier_code caller_id_code | $continue
caller_id_code
      | $trap running_id_code handler_id_code event_id_code
      | $interrupt running_id_code handler_id_code event_id_code
      | $inactive identifier_code
```

Appendix E: .Do File EBNF

Commands:

```
$start “=”  number “;” .

$end “=” number “;”.

$timescale “=” (“s”| “ms”| “us” | “ns” | “ps” | “fs”) “;”.

$show_delta “=” (“true” | “false” | “0” | “1” ) “;”.

$delta_granularity “=” (“10” | “100” | “1000” | “10000” ) “;”.

$enable “=” extendedSymIdent { “,” extendedSymIdent } “;”.

$disable “=” extendedSymIdent { “,” extendedSymIdent } “;”.
```

Types:

```
extendedSymIdent = symbolIdentifier [ “.” “*” ].

symbolIdentifier = identifier { “.” identifier }.

identifier = letter {letter | digit}.

letter = “a” | “b” | . . . |“z”| “A” | “B” | . . . |”Z”.

digit = “0” | “1” | . . .  “9”.

number = digit { digit }.
```

Appendix F: Tracing API

```
// log the start of a main method call
int log_start(const int ID);
// log the end of main method
int log_exit(const int ID);
// log that a behavior is waiting on an event
int log_wait(const int ID);
// log that a behavior is waiting on a time interval
int log_waitfor(const int ID, const int DelayTime);
// log that a behavior has been notified by an event
int log_notified(const int ID, const int EventID);
int log_notified1(const int ID, const int EventID);
// log that a behavior has been awakened after some time period
int log_awakened(const int ID);
// log when a parent thread has joined all of it's child threads
int log_join(const int ParentID);
// log an interrupt occurrence
// RunningID is the ID of the behavior that was running when the
trap occurred
// HandlerID is the ID of the handler behavior
// EventID is the ID of the event that caused the trap
int log_interrupt(const int RunningID, const int HandlerID, const
int EventID);
```

```
// log that a behavior has resumed after being interrupted
// ID is for the behavior that is resuming
// PrevState indicates the state that the behavior is returning
to
int log_resumed(const int ID, const int PrevState);
// log a trap occurrence
// RunningID is the ID of the behavior that was running when the
trap occurred
// HandlerID is the ID of the handler behavior
// EventID is the ID of the event that caused the trap
int log_trap(const int RunningID, const int HandlerID, const int
EventID);
// log a parent behavior forking a child thread
int log_fork(const int ParentID, const int ChildID);
// The following 6 should only be used to show initial states
// log that a behavior is sleeping
// only as an initial state!!
int log_sleeping_state(const int ID);
// log that a parent behavior is waiting for its children to join
// only used as an initial state!
int log_parent_state(const int ID);
// log that a behavior is running
// only used as an initial state!
int log_running_state(const int ID);
// log that a behavior is waiting
// only used as an initial state!
```

```
int log_waiting_state(const int ID);
// log that a behavior is in the inactive state
// only used as an initial state!
int log_inactive_state(const int ID);
// log that a behavior is in the interrupted state
// only used as an initial state!
int log_interrupted_state(const int ID);
// the following 2 are for creating a function call trace
// log a function call
int log_func_call(const int CallerID, const int CalleeID, const
char * FunctionName);
// log a return from a function
int log_func_return(const int CallerID, const int CalleeID,
                const char * FunctionName);
// log a change in value for a signal
// return non-zero if operation succeeds
int log_value_change(const int ID, const int NewValue);
// overloaded version that takes a float
int log_value_change(const int ID, const float NewValue);
// etc.
```