An Implementation of the Behavioral Verilog Simulator in Ptolemy

Pai Chou

April 14, 1994

Abstract

Modeling and simulation of digital hardware at the behavioral level enable designers to experiment with a system without specifying implementation details. Verilog is a language that supports modeling and simulation at different levels of abstraction. However, all components in the same design must conform to a certain timing model, which may not be natural for all components. Ptolemy is a simulation environment for systems with heterogeneous timing models. This report describes the incorporation of the behavioral Verilog simulator into Ptolemy by a compiler, which expresses the simulation semantics of Verilog in C++.

1 Introduction

Behavioral modeling and simulation enable hardware designers to prototype systems without specifying implementation details. Several hardware description languages (including Verilog and VHDL) support behavioral specifications, with semantics tuned for simulation on a uniprocessor. Each simulator either assumes some timing paradigm or implicitly requires components of a given design to conform to a single timing model. This restriction can impose an unnatural timing model on the designer for some subsystems. Ptolemy is a prototyping and simulation environment designed specifically to address this problem by acting as a coordinator for different timing models.

Originally designed at UC Berkeley [1] for modeling digital signal processing (DSP) applications, Ptolemy is gaining applications in general digital hardware simulation. Ptolemy differs from most simulators because it does not impose any given timing model, but rather it coordinates the interaction between different models. Ptolemy is also well known for its sophisticated graphical user interface, including an integrated schematic editor, graphical library components ranging from scopes and chart plotters to video display windows.

Verilog is one of the major hardware description languages in use today [2]. Originally a proprietary language from Cadence Design Systems, and now openly available, Verilog supports mixed behavioral and structural descriptions of hardware. It supports concurrent programming constructs and hardware datatypes. Several commercial CAD tools also use Verilog for verification, timing analysis, and synthesis.

The ability to simulate Verilog in Ptolemy is useful for several purposes: it gives the Verilog simulator a graphical user interface, but more importantly, it enables designers to integrate components described in Verilog with those in different timing paradigms supported by Ptolemy, without having to describe them in an otherwise unnatural language for their application.

The paper is organized as follows: Section 2 gives an overview of Ptolemy; Section 3 gives an overview of the behavioral Verilog language and its simulation semantics; Section 4 explains the mapping of Verilog to C++; Section 5 describes the structure of this Verilog-to-C++ translator, which comprises the core of this project; and finally, Section 6 concludes this paper with an evaluation of this implementation and future work.

2 Overview of Ptolemy

Ptolemy is an object-oriented simulation environment that coordinates the interaction between multiple computation models. Each computation model is called a *domain*. Ptolemy uses astronomical metaphors to describe the hierarchy of the computational units, called *blocks*. A *star* is a leaf-level block belonging to a given domain. A collection of interconnected stars of the same domain forms a *galaxy*. A fully specified, executable galaxy is a *universe*. A *wormhole* is a block which appears as a star on the outside, but is actually a galaxy of a different domain on the inside. Wormholes are used to mix stars of different domains in the same system. Blocks consume, compute, and produce data; a block's communication interface is called a *porthole*. Two portholes are connected by a communication channel called a *geodesic*, which delivers *particles*, or data packets, from one porthole to another.

Each star is defined as a C++ class containing a constructor, a `start()` method, a `go()` method, and a list of variables. The constructor instantiates the star and its ports, the `start()` method is called once at the beginning of each simulation run to initialize local variables and states, and the `go()` method is called every time the star is scheduled to consume, compute, or produce particles. To run the simulation, Ptolemy compiles the C++ star descriptions and dynamically links them in with the the runtime library.

The domains currently supported are SDF, DDF, DE, Thor, and CodeGen. The SDF (Synchronous Dataflow) kernel schedules the block execution statically, while the DDF (Dynamic Dataflow) kernel schedules

Figure 1: Building blocks in Ptolemy

dynamically. The SDF and DDF domains do not have the notion of time, but the *order* in which the stars in these domains consume and produce the particles is significant. DE (Discrete Event) and Thor (CHDL) [3] are *timed* domains. DE particles correspond to events; Thor particles represent "values on pins." The particles in these domains have time-stamps for delivery. These stars can also schedule themselves to be invoked, or "fired," after a certain delay. The Verilog simulator described here uses the Thor domain's timing model.

Since specifying stars directly in C++ can be error-prone, and burdens the programmer with excessive detail, Ptolemy supports the use of preprocessors. These are invoked automatically to convert a star in any source language to C++ (in a format as expected by Ptolemy). SDF, DDF, and DE domains use the preprocessor language "ptlang," while the Thor domain uses "pepp" (Pthor Extended PreProcessor) to translate CHDL into C++. This project adds a preprocessor called "VCC" to the Thor domain to translate Verilog into C++.

3 Overview of Verilog

Verilog describes hardware as a set of hierarchical components called *modules.* A module interfaces with its environment via its *ports.* A module can either describe the *behavior* of a component as a program, or describe its *structure* as a set of nested modules, primitive components, and wires. This project uses Verilog to specify behavior and Ptolemy to capture the interconnection topology. Behavioral Verilog resembles an imperative programming language, with support for hardware data-types and concurrency.

Data-types in behavioral Verilog include input and output ports, registers, integer, and real. A *scaler* [*sic*] is a "single bit" quantity with four possible digital hardware states: 0, 1, **X** (unknown), and **Z** (floating). A *vector* is an array of scalers which represent a value as a group. For example, `4'b01xz` denotes a 4-bit vector constant whose scalers have values 0, 1, **X**, and **Z**, respectively. The *subrange* and *index* operators

```
always begin :R
        ... disable S;
end
always begin :S
        ...
/* thread R resets thread S */
```

```
while(cond) begin :S
        ... disable S ...
end

/* like continue in C */
```

```
begin :S
        while(cond) begin ...
                disable S
        end
end
/* like break in C. */
```

Figure 2: Example uses of **disable**

select a slice or an individual scaler from a vector. An `integer` models a 32-bit vector, and a `real` usually models a double precision floating point quantity.

The concurrent constructs include *continuous assignments*, the **initial** statement, and the **always** statement. *Continuous assignments* model combinational logic as an arithmetic/logical expression, with the input signals and possibly some registers as terms, where the result is assigned to an output port after some delay. An **initial** statement is executed once at the start of a simulation run and becomes inactive, while an **always** statement is executed repeatedly in an infinite loop. The **initial** and **always** statements are concurrent threads of control. During simulation, threads are scheduled round-robin until they block. The programmer is responsible for explicit synchronization between threads, and should not make assumptions about the relative order of execution between the threads at any given instant.

The statements in behavioral Verilog include those constructs commonly found in conventional C-like sequential programming languages, like **if/then/else**, **while** loops, **for** loops, **repeat**(n) loops, and **switch/case** statements. In addition, the ones specific to Verilog are **wait**, `@`'s that wait on a transition edge, `@`'s that wait on *named events*, `#`(*delay*), and **disable**.

The **wait**(*cond*) construct blocks until the boolean expression *cond* evaluates to true. The `@`(*expr*) construct blocks until *expr* changes its value. One can also specify a rising or a falling edge transition by qualifying *expr* with the keywords **posedge** and **negedge**, respectively.

Named events are abstract event types. The keyword **event** defines a name as the event. An event involves two threads, a sender and a receiver. The sender has the syntax `->`*name*, and the receiver `@`*name*. One can think of a send as depositing an event instance into a buffer, and a receive as removing the instance. If the buffer has no event, the receiver blocks. The sender does not block, because the buffer holds at most one instance. Any event instance already in the buffer is overwritten and lost.

The **disable** *S* construct interrupts the execution of a statement block named *S*. One thread can disable a named block in another thread, and this is often used to express a hardware reset. A nested statement can also disable an enclosing statement block. Figure 2 shows three example uses of **disable**.

4 Mapping Verilog to C++

This section describes the mapping of Verilog computation and control constructs into C++. *Inter*-module constructs are readily mapped to Ptolemy stars, galaxies, portholes, and geodesics. However, *intra*-module constructs must be implemented at the C++ language level.

4.1 Hardware Data-types

Scalers and vectors are implemented as abstract data-types in C++. They can be divided into three categories: literal values, storage references, and port references.

A literal value is implemented as an abstract data-type called `rVal` with a set of arithmetic and logical operators similar to those in C. The class `rVal` has three fields: *value*, *mask*, and *type*. A bit in *value*, and its counterpart in *mask*, encode the four possible values of each scaler. This compact encoding scheme allows the arithmetic and logical operators to be implemented efficiently in C++. The *type* field holds the size of the vector (i.e. number of scalers). A single scaler quantity can be represented as a vector of width 1.

Storage for a hardware value is implemented as a derived class of the `rVal` class. This allows all variables of type **reg** or **integer** to be used directly as literal values without type conversion. In addition, each variable must also yield *LValue*s so that the *subrange-selection* and *index* operators can both be assigned a literal value. For this purpose, a `RegRef` class is defined to create a reference object to a `reg` with an index or a subrange selection. One can either assign to or retrieve values from the `reg` via these reference objects.

To incorporate values from ports, a number of port-interface classes are also defined. Thor ports use C++ integers to represent scalers, but Thor's multi-ports do not allow specification of the bus bounds in the C++ description. Instead of modifying the Thor kernel, these port interface classes hide the conversion so these ports can be accessed just like registers.

4.2 Control Constructs

In Verilog, the **initial** and **always** statements are threads with special properties. They are cooperative threads that can **disable**, i.e. force exit, arbitrary named statement blocks. Most thread packages are both too general and not expressive enough for Verilog threads: Verilog does not have dynamic thread creation and has no preemption, and ordinary threads packages cannot express **disable**s. To satisfy the required properties, VCC implements the threads using a technique called *description by cases* [4].

It transforms Verilog a thread into a **switch** statement enclosed in a **do/while** loop as follows:

```
do {
        switch(point) {
        case 0: basic block 0...
        case 1: basic block 1...
        ...}
} while (runnable);
```

The **switch** statements contains all the basic blocks of the thread. Each basic block has a unique *label*, which is represented by the case's constant. The state of the thread is represented by two variables: the *point*, which is an integer variable containing the label of the basic block being executed, and the *runnable* flag, which is controls the exit condition of the loop.

The *point* and *runnable* can express the two most important thread primitives: arbitrary control transfer and blocking. Control transfer, either voluntary by the thread itself or by **disable**, is done by explicitly assigning the destination basic block's label to *point*. Blocking, or effectively a thread-*yield*, is represented by $runnable = \text{FALSE}$, causing the enclosing loop to exit. As a result, the next thread is executed. All Verilog constructs, including **wait**-on-conditions, wait-on-a-transition, named events, **disable**, and delays, can all be expressed in terms of explicit control transfer, blocking, and **if** statements.

To implement **disable**, the disabler thread checks the *point* of the disablee thread to see if it is in the range of the named block. If so, then the disabler forces control transfer by assigning the successor's label to the disablee's *point*.

The **wait**(*cond*) construct is implemented as an **if** statement which tests *cond*. The TRUE branch is a control transfer to its successor basic block, and the FALSE branch is a blocking statement.

The wait-on-transition statements use a temporary variable to stores the initial value of the expression, and go to a new basic block to test for transition. It compares the initial condition with the new value, and as a side effect, assigns the new value to the temporary variable. If the transition occurs, then control is transferred to the successor, else block.

To implement a delay, a temporary variable is allocated and is assigned the completion time; then control is transferred to a new basic block which effectively does a **wait** on the condition $completionTime \leq now$.

To implement named events, VCC allocates two integer variables, r for the receiver, and s for the sender. A *send* is implemented as incrementing s. A *receive* is mapped to a comparison of r with s: if s is larger, then an event has been sent since the last receive; otherwise the thread blocks. The event is received by

Figure 3: Structure of VCC

assigning $r = s$ and transferring control to the successor basic block.

The algorithms for transforming threads into the *description by cases* representation will be described in Section 5.3.

5 Structure of the Verilog-to-C++ Translator

The Verilog-to-C++ translator, VCC, compiles behavioral Verilog into a Thor star class definition in C++. The compilation process has three phases: the front-end, transformation, and code generation. In addition, a set of persistent data structures maintain the token values, attributes, and parse-trees for the three compilation phases. Figure 3 shows the structure of VCC.

5.1 Persistent Data Structures

The persistent data structures maintain the data objects and attributes used by all three phases of the compiler. These structures consist of the list management unit, token tables, and the symbol table.

The list is a fundamental data structure in VCC. As in LISP, atoms and lists are both represented using the same node structure. Each node contains an operator field which indicates the type of atom or list it represents. An atom node has a field which contains either the value or a pointer to the value. The routines for accessing the nodes are named like their LISP counterparts. For example, `atom`, `listp`, `cons`, `car`, and `cons`, are all implemented. Lists are also used as binary trees to represent the parse-trees.

The values of certain types of atom nodes are maintained in the token tables. Tokens are syntactic objects like keywords, identifiers, integer literals, floating-point literals, and strings. Each type of token has its own table: there is a string table, a vector table, and a floating-point table. Strings are hashed, and are referenced by the *string ID*, an index into the string table.

The symbol table serves two purposes: resolve static binding of identifiers, and maintain the attributes of the symbols. The symbol table can open and close a scope, define a symbol, and look up a symbol at any scope level. The symbol table also maintains the attributes for each symbol as a linked list. It can associate any number of attributes of arbitrary types with a given symbol without imposing a predefined format.

5.2 The Front-End

The front-end of VCC reads a Verilog file, defines the symbols, and outputs a partially decorated parse-tree. These tasks are divided between the scanner and the parser.

The scanner inputs a Verilog description file as a stream of characters, and outputs a stream of tokens to the parser. It removes comments and performs macro definition and expansion. It also maintains the line and column positions for error reporting. It uses a case statement to determine the token type, computes the semantic value of the token, and returns an atom node containing both the token type and a pointer to its semantic value in the token table.

The parser parses the steam of tokens from the scanner and performs semantic actions according to the syntax. Upon parsing declarations such as variables, named blocks, and ports, the parser calls the symbol table routines to define the symbols, or open or close a scope. This allows most static binding to be resolved as parsing progresses. To support use-before-define, the parser saves the scope of undefined symbols so the next phase will be able to look up their definitions. Upon parsing expressions and statements, the parser constructs parse-trees using lists as binary trees as described above. The grammar and semantic actions are specified in YACC, which generates the parser.

5.3 Transformation

The transformation phase converts the Verilog threads (represented by the parser-trees) into *description by cases*, namely basic blocks with explicit control transfers as described in section 4.2. VCC builds these basic blocks by calling the following set of operators:

`NewCaseLabel()`	allocate a new label;
`SetCurrCase(`*label*`)`	set the current basic block to the one pointed to by *label*
`AppendStmt(`*stmt*`)`	append the statement *stmt* to the current basic block
`GotoCase(`*label*`)`	create the parse-tree for the statement *point* = *label*.
`StartNewCase()`	allocate a new label and set it as the current block if non-empty
`Block()`	create a parse-tree for the statement *runnable* = FALSE

The **wait**(*cond*) construct is transformed by the following steps:

```
$L1$ = StartNewCase()
$L2$ = NewCaseLabel()
AppendStmt(if ($cond$) GotoCase($L2$) else Block())
SetCurrCase(L2)
```

These steps generate a basic block with the label *L1*. It evaluates *cond*. If false, it blocks. The next time this thread receives control, it will resume at *L1* and reevaluate *cond*. When *cond* evaluates to true, control is transferred to *L2*.

The `#`(*delay*) delay statement is transformed as follows:

```
AppendStmt($delayvar$ = CURRENT_TIME + $delay$)
AppendStmt(SELF_SCHED($delay$))
$L1$ = StartNewCase()
$L2$ = NewCaseLabel()
AppendStmt(if ($delayvar \leq$ CURRENT_TIME) GotoCase($L2$) else Block())
SetCurrCase($L2$)
```

The resulting C++ code first schedules itself to be refired after the delay by calling the Thor method `self_sched`(*delay*). It computes the completion time by adding the delay amount to CURRENT_TIME, the real-time clock defined by Thor. Control is then transferred to *L1* to test the completion condition. The comparison for completion time is necessary because if the star is fired before the completion time, it should continue to block. When the delay completes, control is transferred to *L2*.

The `@`(*expr*) construct expresses both waiting on a transition and waiting for a named event, including the composition of several expressions by the **or** operator. In both cases, this construct is transformed by the following steps:

```
AppendStmt($temp1$ = $expr1$, ...$tempN$ = $exprN$ ..)
$L1$ = StartNewCase()
$L2$ = NewCaseLabel()
AppendStmt(if (transition($temp1$, $expr1$) + ... + transition($tempN$, $exprN$)) GotoCase($L2$) else Block())
SetCurrCase($L2$)
```

The resulting C++ code stores the initial values of these expressions in temporary variables, and tests transition in a new basic block *L1*. Transition is tested by calling the boolean function posedge, negedge, or change, which compares the saved old value to the new value. The return value of transition and event receives are summed; any transition or event causes control to transfer to *L2*.

To implement **disable**, the transformation routine constructs a table which maintains the upper bound, lower bound, and successor labels for each named block. This algorithm guarantees that every named block always has a contiguous range of labels, so that **disable** can be expressed as
if (*lowerBoundLabel* $\leq$ *point* $\leq$ *upperBoundLabel*) *point* = *successor*.

5.4 Code Generation

The final phase of VCC generates a C++ text file using the basic blocks constructed in the transformation phase. In addition, the C++ file must include the appropriate header files in order to be compiled and linked by Ptolemy. VCC uses a template file to guide the code generation.

The template file contains two types of patterns. The first type of pattern is matched and substituted before it is written to the output file. The second type of pattern invokes a procedure in the code generator. The second type of pattern can also contain parameter strings which can be applied repeatedly. Those characters that do not match any pattern are copied directly to the output.

6 Conclusion and Future Work

This project has demonstrated the feasibility of adding new languages like Verilog to Ptolemy using a translator. The translator expresses the simulation semantics of Verilog in terms of the timing model in the Thor domain for *inter*-module communication and scheduling, and in terms of the C++ language for *intra*-module computation and scheduling.

Future work will include implementing the remaining behavioral constructs in Verilog, implementing modules containing other modules as galaxies. The Verilog constructs omitted here are the **switch** statements, **task**s and **function**s, RAM, bit arrays longer than 32 bits, deassignable continuous assignments, and module parameters. Most of these constructs can be viewed as syntactic convenience, and are expressible in terms of those already implemented. The others are relatively easy to add because the translator already parses all of behavioral Verilog, and the template-driven code generation back-end facilitates such extensions.

References

[1] *The Almagest: Manual for Ptolemy v0.3*, EECS Dept., Univ. of California, Berkeley, January 1992.

[2] Donald E. Thomas, Philip R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1991.

[3] "Thor Tutorial," VLSI/CAD Group. Stanford University, 1986.

[4] M. E. Conway, "Design of a Separate Transition-Diagram Compiler," *Comm. of the ACM*, vol. 6, pp.396-408, 1963.