Translating System Specifications to VHDL

Sanjiv Narayan Frank Vahid Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717, U.S.A.

Abstract

Languages based on hierarchical and concurrent state diagrams are powerful in specifying system level designs. Simulating such languages can be simplified by translating to a simulation language such as VHDL and then using available simulators. This paper describes system level abstractions commonly found in specification languages and presents semantic preserving VHDL implementations.

1 Introduction

A required component of system level design is a *simulatable specification*. VHDL is a candidate, but it is intended for hardware description and simulation, and thus can be cumbersome to use at the system level since many details related to hardware and/or simulation must be specified. For example, a CPU system is inherently state based, but VHDL contains no state abstractions. However, VHDL is a widely accepted standard, making simulators and other tools widely available. Our approach has thus been to create a language (SpecCharts) intended for system specification, and then to implement a translator to VHDL for simulation purposes. A concise, understandable specification can then be achieved, while still taking advantage of the simulation capabilities of VHDL. In this paper we present a translation scheme for the SpecChart language to VHDL.

Many others have taken a similar approach in designing languages for system specification [ArWC90, MaWa90, TiLK90]. Some of the most important constructs of such languages involve state diagrams with hierarchy and concurrency. We present a scheme for translating these constructs, avoiding problems found in existing schemes such as changing the execution semantics or assuming a minimum delay between external events. Issues that are often overlooked, such as details of deactivating a state, are discussed.

This paper will first introduce the SpecChart language for system level specification. It will then describe several abstractions/constructs found in the language, and present strategies for translating these into VHDL constructs.

2 SpecCharts : A Language for System Level Specification

The SpecChart language is introduced by an example. [VaNG90b] describes the language in detail.

Figure 1 shows a SpecChart description of a simple computer system, containing a clock generator, CPU, and memory. RESET (external reset signal) and DATA_BUS are declared as ports. SYSTEM is described using two concurrent substates, CLK_STATE and CPU_STATE. CLK_STATE is a *leaf* state which generates a 100 ns clock, described using VHDL sequential statements. The CPU_STATE receives the clock from the CLK_STATE via connected ports. CPU_STATE contains the memory and three registers declared as signals: PC (program counter), INSTR_REG (instruction register), and ACCUM (accumulator).

CPU_STATE is described as two sequential substates, RESET_STATE and ACTIVE_STATE. Initially the CPU is reset. Then, on the falling edge of the RESET signal, the active state is commenced. The first state is FETCH, which on the rising edge of the clock reads the current memory location into the instruction register. After this in done, DECODE extracts the opcode. EXECUTE then executes the instruction. After execution, FETCH is again commenced.

Each state's functionality can be described either by using VHDL sequential statements (a *leaf* state), *concurrent* substates, or *sequential* substates sequenced by arcs. Each state may contain VHDL declarations whose scope is all descendant states.

Also note that two different arcs are used for state transitions. An *exit-immediately* arc (or **EI arc**, drawn originating from the perimeter of the state) is traversed immediately if the arc condition becomes true. The EI arc shown pointing from the ACTIVE_STATE to the RESET_STATE means that regardless of whether the system is currently fetching, decoding, or executing, the CPU is immediately reset on the rising edge of the RESET signal.

An *exit-on-completion* arc (or **EOC arc**, originating from a dot inside a state) is traversed when the state has *completed* AND the arc condition evaluates to true. A leaf state is complete when all statements have completed and all signals have received their new value; a state with substates completes when control flows to the *stop* dot (see the *opcode=0* arc in the ACTIVE_STATE). An EOC arc by default has a condition of true. Note that EOC arcs are found exiting the fetch, decode, execute, and reset states.

3 SpecChart Abstractions

Since the SpecChart language is geared towards concise and understandable specifications of systems, it has several

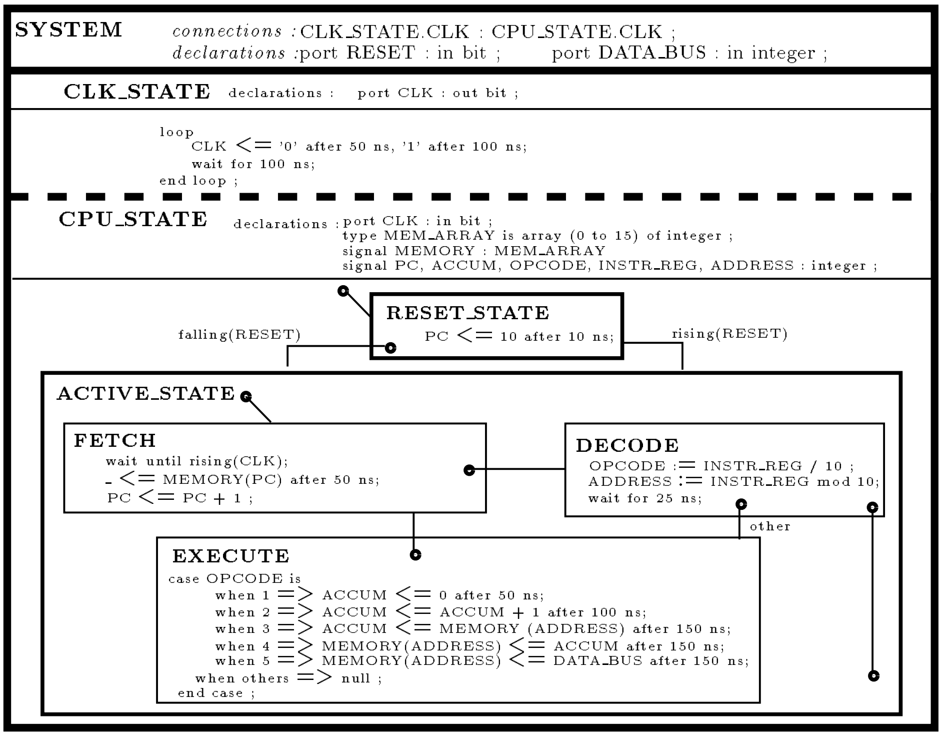

Figure 1: SpecChart of an example computer system

abstractions which are not found in VHDL. These abstractions are discussed below.

Hierarchical States and State Sequencing — State diagrams, when combined with hierarchy and concurrency, are a very powerful way to concisely specify real systems [Ha87]. A hierarchical state diagram implies that the functionality of a state may itself be described by another state diagram, containing sequential or concurrent substates in SpecCharts . This requires the concept of state decomposition. A second abstraction is that of an 'arc', or transition from one state to another.

Asynchronous Event Transitions - EI Arcs — It may be necessary to terminate a state machine or a process immediately on the occurrence of an event (e.g. a RESET signal in a computer). Checking for this event throughout all levels of hierarchy is cumbersome and causes an exponential blowup of code. In SpecCharts, the *exit-immediately* (EI arc) terminates state execution immediately (thus terminating all substates, or suspending code execution for a *leaf* state).

State Completion: EOC Arcs — Arbitrary transitions into and out of the states, or the presence of loops, wait statements and delayed signal assignments, make it very difficult to determine statically as to how long it would take for all computations made in a state to take effect. In SpecCharts, the *exit-on-completion* (EOC arc), causes a transition if all computations made by the current substate have completed, and the associated arc condition is true.

Global Variables — Variables, by removing the concept of 'time' and ensuring sequentiality of statements, lend themselves to simple descriptions. In VHDL, even if we were to ensure somehow that two processes could never be activated simultaneously, they would still not be able to share the same variable. SpecCharts allow variables to be global over sequential substates (states such that no two states are active at the same time), greatly increasing the situations in which they may be used.

Global Signals — Even if a VHDL design description is such that no two processes will drive the same signal at the same time, we still need to write resolution functions for the signals. In every process, we must enable and shutoff drivers as appropriate. This is not only cumbersome, but serves little purpose in that the resolution functions just return the value written by the only active process. SpecCharts allow a signal to be written to in several sequential substates, without the need for a resolution function or shutting off drivers for the signals when the states are not active.

Perhaps the two most powerful abstractions provided by SpecCharts are protocol based data transfer and arbitrated signals. The former permits data transfers to be specified using protocols, rather than specifying low level details (port and control signal assignments) that implement the protocol. The latter provides a concise representation of arbitrated access to a resource. Both abstractions are synthesized into the lower level details before converting to VHDL. They are thus beyond the scope of this paper.

4 Transformations

This section presents the transformations for translating the SpecChart abstractions to constructs found in VHDL.

4.1 Hierarchical States and State Sequencing

The activation scheme for VHDL maintains the same hierarchical model as in SpecCharts. The parent activates/deactivates its children, and a child tells its parent when it has completed. For example, when state SYSTEM in Figure 1 is activated, it immediately activates CPU_STATE and CLK_STATE. The former activates its first substates, the latter begins executing its code (since it's a leaf). This activation goes down the hierarchy until all appropriate leaf states have been activated.

```
SYSTEM : block
begin
    CLK_STATE : block
        code : process
            (CLK_STATE statements)
    CPU_STATE : block
        RESET_STATE : block
            code : process
                (RESET_STATE statements)
        ACTIVE_STATE : block
            FETCH : block
                code : process
                    (FETCH statements)
            DECODE : block
                code : process
                    (DECODE statements)
            EXECUTE : block
                code : process
                    (EXECUTE statements)
            control : process
                (ACTIVE_STATE control process)
        end block ACTIVE_STATE
        control : process
            (CPU_STATE control process)
    end block CPU_STATE
    control : process
        (SYSTEM control process)
end block SYSTEM;
```

Figure 2: General VHDL structure of state SYSTEM after translation

General VHDL Structure — Each state becomes a block. Substates of a *non-leaf* state in SpecCharts are mapped to sub-blocks of a block in VHDL. A *leaf* state containing VHDL process code becomes a block with a single process containing that code. A state's declarations become the block's declarations. Figure 2 shows the general structure of the VHDL produced when state SYSTEM is translated.

Control : State Activation/Deactivation — A state's main responsibilities include:

- Wait to be activated by its parent, and then begin execution of its code (if a *leaf* state), or activate/deactivate appropriate substates
- If deactivated by its parent, a state has the responsibility of terminating code execution (if a *leaf* state), or immediately deactivating all substates
- If the state completes while activated, it must inform the parent of its completion and wait to be deactivated

```
control : process

begin
    wait until state is being activated or deactivated
        OR EI_arc1_cond and arc's source substate is active
        OR .... for all EI arcs of all substates
        OR EOC_arc1_cond and arc's source substate is complete
        OR .... for all EOC arcs of all substates ;

    if state being activated (inState = true and not stable)
        if sequential substates, activate first substate
        if concurrent substates, activate all substates
    elsif state being deactivated
        deactivate all substates
    elsif EI_arc1_cond and arc's source substate is active
        deactivate arc's source substate
        activate destination substate
    elsif .... for all EI arcs
        ....
    elsif EOC_arc1_cond and arc's source substate is complete
        deactivate arc's source substate
        activate arc's destination substate
    elsif .... for all EOC arcs
        ....
    end if;
end process;
```

Figure 3: Structure of the control process generated for each non-leaf state

The implementation involves using two boolean signals for each substate. **'inState'** (where State is the substate's name) is used to activate/deactivate the substate, and **is set only by the parent's control process**. **'doneState'** is **set only by the child** to indicate completion to parent state.

In SpecCharts, a *non-leaf* state's arcs activate and deactivate the substates. In VHDL, we add a *Control Process* to each block representing the non-leaf state. Whenever the state is activated/deactivated, or transitions between substates occur, the control process appropriately activates/deactivates the sub-blocks. The structure of the control process is given in Figure 3.

In a *leaf* state, the state's block contains a single VHDL code process. The process initially waits to be activated and then executes the code. On reaching the end of the code it informs its parent state of completion, and waits to be deactivated. The *italicized* statements in Figure 4 show how state FETCH of the SYSTEM example is modified to handle activation and indicate completion.

4.2 EI Arcs : Immediately Terminating a State's Execution

When a *non-leaf* state is deactivated, the state's control process will perform an action which deactivates all substates (i.e. set *inState* to false for all substates) The control process then goes to its initial wait statement.

When a *leaf* state is deactivated, it must ensure that no future signal updates occur that were caused by signal assignments scheduled in this state (using an 'after' clause). Since the state may be executing a wait statement, it might miss the deactivation signal. To overcome this, we modify *all* wait statements in the process to stop waiting if inState becomes false. Each wait statement is then followed

```
FETCH

if guard then                                -- if activated
    FETCH_loop : loop
        wait until (CLK='1' and not(CLK'stable))
                                             or not(inFETCH);
        if not(inFETCH) then
            exit FETCH_loop;
        end if;
        INSTR_REG <= MEMORY(PC) after 50 ns;
        PC <= PC + 1;
        doneFETCH <= true;                   -- indicate completion
        wait until not(inFETCH);             -- wait for deactivation
        doneFETCH <= false;
        exit FETCH_loop;
    end loop FETCH_loop;
end if;
INSTR_REG <= null;
PC <= null;
wait on guard;                               -- wait for activation
```

Figure 4: State FETCH after being transformed to handle activation/deactivation and indication of completion

by a check to see if *inState* is false, in which case the wait statement was terminated because of state deactivation. Control should jump to the end of the code where all signal drivers are shut off to prevent future updates. Since there is no 'go to' command in VHDL, we enclose the code in a labeled loop, and an exit from this outermost loop will transfer control to the end of the code. The *bold* statements in Figure 4 show the *leaf* state FETCH modified to handle deactivations.

4.3 EOC Arcs: Calculating State Completion Time

Implementing EOC arcs for *non-leaf* states is simple: when an arc traversal reaches the *stop* dot, the control process informs the parent state and waits to be deactivated.

A *leaf* state is said to complete when execution reaches the end of its statements, AND all transactions scheduled by this state have completed. For example, a state containing a single statement, "X <= X + 1 after 10 ns", is said to have completed when X gets its new value, i.e. after 10 ns. The time for which execution has to wait at the end of the state (to allow signals to get their new values) must be dynamically determined, since it may differ each time the state is executed (due to loops, branches, and wait statements). Our implementation uses two variables of type time to keep track of the amount of time required for the last transaction to take place:

- *global_time*: Before every wait statement, it is set to the current simulation time - 'now'. After the wait statement, the value 'now' - *global_time* represents the time spent waiting.
- *remain_time*: After every signal assignment with an *after clause*, it is set to MAX(*remain_time*, *after clause* value). This maintains the amount of time into the future (relative to the current simulation time) when the latest transaction will occur. After every wait statement, *remain_time* is decremented by the time spent waiting.

At the end of the statements, we add the statement 'wait for *remain_time*;'. Figure 5 shows state FETCH modified to wait until the state has completed before indicating completion.

```
FETCH
declarations:
    variable REMAIN_TIME: time;
    variable GLOBAL_TIME: time;

    if guard then
        REMAIN_TIME := 0 ns;
        GLOBAL_TIME := now;
        wait until CLK='1' and not(CLK'stable);
        GLOBAL_TIME := now - GLOBAL_TIME;
        REMAIN_TIME :=MAX(0 ns,
                (REMAIN_TIME - GLOBAL_TIME) );
        INSTR_REG <= MEMORY(PC) after 50 ns;
        REMAIN_TIME := MAX(REMAIN_TIME, 50 ns);
        PC <= PC + 1;
        wait for REMAIN_TIME;
        doneFETCH <= true;
        wait until not(inFETCH);
        doneFETCH <= false;
    end if;
    wait on guard;
```

Figure 5: State FETCH transformed to *correctly* indicate completion (i.e. for EOC arcs)

4.4 Global Variables: Conversion to Signals

The declaration of the variable is modified to a signal of the same name. Then, in each *leaf* state of the SpecChart that writes to that variable, we declare a local variable (which *is* supported by VHDL) which is initialized to the global signal at the start of the leaf state. Every reference to the variable in the leaf state is now replaced with the locally declared variable. Also every write to the original variable is now followed by an update of the global signal (to ensure it holds the most recent value if the state is suddenly deactivated). The temporary variable preserves the variable semantics within the code, while the global signal makes the updated value visible to all the sequential substates.

4.5 Global Signals over Sequential Substates

Global signals over sequential substates need not be declared as resolved signals. Since SpecChart states are modeled as VHDL blocks, we need to have resolution functions for each signal declared in a *non-leaf* state, and assigned to by its sequential substates. To ensure that only one of the states is assigning a value to the signal at a given time, we declare a resolution function which simply returns the value of the first driver that drives that signal. Since only one of the sequential substates can be active at a time, the other non-active states must assign the value *null* to the signal. Thus at the end of each leaf state, we assign a null value to each signal assigned to by that state.

5 Results and Future Work

The translator has been implemented in C, running under UNIX. SpecCharts are entered via an X based graphical interface (or directly in their textual format using any text

Model	*Lines of code*
Armstrong's mixed block/process VHDL description of Controlled Counter [Ar89]	81
Lis' process VHDL description of Controlled Counter [LiGa89]	71
SpecChart model of Controlled Counter	67
SpecChart textual files (created by SpecChart X application) of Controlled Counter	139
VHDL generated by SpecChart to VHDL translator for Controlled Counter (in .3 seconds)	271
VHDL description of DRACO	392
SpecChart model of DRACO	226
SpecChart textual files of DRACO	268
VHDL generated by SpecChart to VHDL translator for DRACO (in 1.2 seconds)	506
SpecChart model of SYSTEM example	70
SpecChart textual files of SYSTEM example	118
VHDL generated by SpecChart to VHDL translator for SYSTEM example (in .5 seconds)	312

Table 1: Lines of code for various models, including generated VHDL

editor). Libraries exist for parsing and manipulating SpecCharts. The translator converts a SpecChart textual file to a VHDL entity, taking only a few seconds. The ratio of the amount of textual SpecChart code to generated VHDL code is approximately 1:3.

To simulate the generated entity, a new VHDL file is created in which the entity is instantiated as a component. The ports can then be assigned to, and assertions on output ports used to verify correct operation. We have automatically translated and simulated many examples – results are shown in table 1. Examples include a peripheral interface chip, a controlled counter, and the SYSTEM example introduced in this paper. All the translated SpecCharts passed simulation tests for correct functionality.

One interesting example is the DRACO peripheral interface [Rock89], an industry chip. A SpecChart model was created from Rockwell specifications, and the original test vectors used for testing (23,000 lines of process code). The textual SpecChart created contained 268 lines, and was translated to 506 lines of VHDL in 1.2 seconds.

Our current research focuses on developing and implementing algorithms for estimation of area, time, and pins for a given SpecChart, of partitioning specifications among chips, and of synthesizing interfaces between chips. Application of these synthesis steps to a SpecChart, results in a new SpecChart, that can at any time be simulated (via the VHDL translator) to observe and verify its behavior.

6 Conclusions

This paper introduced SpecCharts and discussed the abstractions built into the language which facilitate system level specification and synthesis. Due to the requirement that a specification language should be simulatable, and wanting to make use of the advantages provided by the standardization of VHDL, we have implemented a SpecCharts to VHDL translator. The results of detailed examples have demonstrated that the concept of a high-level language on top of VHDL is beneficial for the designer and does not decrease the efficiency of the simulation. The VHDL implementations of system level constructs detailed in this paper should prove useful to designers of translators of other state based languages to VHDL.

7 Acknowledgements

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #90-DJ-146). We are grateful for their support. We would also like to thank Nikil Dutt, Joe Lis, Tedd Hadley, and Rajesh Gupta for their suggestions.

References

[Ar89] Armstrong, J., "Chip-Level Modeling Using VHDL", Prentice-Hall, 1989.

[ArWC90] Arsenault, A., Wong, J.J., Cohen, M., "VHDL Transition from System to Detailed Design", VHDL User's Group Meeting, Boston, April 1990.

[Ha87] Harel, D., "Statecharts : A Visual Formalism for Complex Systems", Science of Computer Programming 8, 1987 pp 231-274.

[IE88] IEEE Standard VHDL Language Reference Manual, IEEE, March 1988.

[LiGa89] Lis, J., and Gajski, D., "Structured Modeling for VHDL Synthesis", U.C. Irvine Tech. Rep. 89-14, June 1989.

[MaWa90] MacDonald, R., and Waxman, R., "Operational Specification of the SINCGARS Radio in VHDL", AFCEA-IEEE Tactical Communications Conference, April 1990.

[NaVa90] Narayan, S. and Vahid, F., "Translating SpecCharts to VHDL", University of California, Irvine, Technical Report 90-21, July 1990.

[Rock89] Rockwell International, "DRACO Engineering Report", April 1989.

[TiLK90] Tikanen T., Leppanen T., and Kivela J., "Structured Analysis and VHDL in Embedded ASIC Design and Verification", EDAC, 1990.

[VaNG90b] Vahid, F., Narayan, S., and Gajski, D., "SpecCharts: A Language for System Level Specification and Synthesis", U.C. Irvine, Tech. Rep. 90-19, July 1990.