Translating SpecCharts to VHDL

Authors
Narayan, Sanjiv
Vahid, Frank

Publication Date
1990-07-25

Peer reviewed

Translating SpecCharts to VHDL

Sanjiv Narayan
Frank Vahid

Technical Report #90-21
July 25, 1990

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

narayan@ics.uci.edu
vahid@ics.uci.edu

Abstract

SpecCharts is a new language intended for system level specification and synthesis. It is based on hierarchical state diagrams and VHDL, and posseses many constructs designed to facilitate ease of description. Since current requirements demand that a specification language be simulatable, an approach for simulating SpecCharts needed to be developed. Rather than taking on the major task of writing a new simulator, a translator from SpecCharts to VHDL was implemented. This permits making use of the advantages that accompany the standardization of VHDL, including use of powerful compilers and simulators, while maintaining the ability to describe systems concisely and perform system level synthesis steps. The SpecChart to VHDL translator must convert each SpecChart abstraction to functionally equivalent VHDL. This report describes each of those abstractions and their VHDL implementation. The system takes as input a SpecChart and outputs a VHDL file which, when compiled, is a simulatable entity that can be used as any other VHDL entity. Several examples display how the translator can be used to verify SpecChart models of systems, thus adding to SpecCharts capability as a system level specification language.

Contents

List of Figures

List of Tables

1 Introduction

The SpecChart language is intended to specify system level designs with emphasis on synthesis and ease of understanding. It permits description of a system as a hierarchical state diagram, where a leaf state's functionality is described with VHDL process code. It is a combined graphical and textual representation, intended to represent a design through many stages of system level synthesis. Due to its hierarchical state based representation and variety of high level constructs, the general behavior of a system can be easily discerned.

One of the requirements of system design is the existence of a *simulatable specification.* VHDL would seem to be a candidate specification language, since it is a widely accepted standard and thus provides a means for information exchange and the use of powerful tools (e.g. simulators, synthesizers, etc.). The language itself is excellent for creating structural and RTL descriptions, as well as simple algorithmic descriptions. However, VHDL can be difficult and tedious to use for many applications. For example, many higher level systems such as a CPU system are inherently state based, but VHDL does not provide any built in abstractions to simplify the description of such systems. The description of these systems are also greatly simplified by the existence of other system level constructs [VaNaGa90a], such as protocol based data transfer, which would require a change in the VHDL language definition. VHDL simply cannot be tuned for ease of use and understandibility for all possible applications. Thus, application specific languages, which provide simplicity of description and ease of understanding for particular problem domains, are evolving [DuHaGa89, Ha87, Lee89]. Many of these languages can be converted to VHDL to exploit several of VHDL's advantages, e.g. simulation. These approaches do not attempt to replace VHDL, but instead to exploit VHDL's full potentialities, and thus increase its usefulness and acceptance.

Our approach has thus been to create a language suitable for specification and synthesis of computer systems, and then to convert the specification to VHDL for simulation purposes. We have tried to use VHDL syntax wherever possible. This provides easier understanding since VHDL users will already have a good feel for the language. It also simplifies one possible synthesis approach which involves first performing system level sythesis steps, such as partitioning the specification among chips or synthesizing interfaces between components, and then converting each detailed chip or system component specification into VHDL for synthesis by existing VHDL tools [LiGa88].

This report will first introduce the SpecChart language and highlight several SpecChart abstractions that do not exist in VHDL. The majority of the paper will then describe the transformations needed to implement these abstractions in VHDL, and the overall translation scheme, followed by the results and status of the research.

2 SpecCharts: A Language for System Level Specification and Synthesis

The SpecCharts language will be introduced by an example. For a detailed description of the language, see [VaNaGa90a, VaNaGa90c]. The example by no means covers all the key features of SpecCharts, and is used only to give a general feel for the language.

Figure 1 shows a SpecChart description of a very simple computer system. The system described contains a clock generator and CPU, as well as a memory. The CPU normally executes instructions from memory, starting from address 10 until an external reset signal forces execution to begin again from location 10.

SYSTEM is a state which describes the computer system. Two ports are declared: RESET is the external reset signal, and DATA_BUS carries the input data used by some instructions. SYSTEM can be described using two concurrent states. CLK_STATE generates a simple 100 ns clock, described using VHDL sequential statements, for use by CPU_STATE and thus connected via ports in the *connections* section of SYSTEM. CPU_STATE contains the memory and three registers declared as signals: PC (program counter), INSTR_REG (instruction register), and ACCUM (accumulator). In addition, variables named OPCODE and ADDRESS are declared merely to simplify the code. Note that all these declarations are the same as VHDL declarations.

The CPU is always either in its normal active mode or is being reset, hence it is described as two sequential substates, RESET_STATE and ACTIVE_STATE. Initially the CPU is reset. After the reset is performed and on the falling edge of the RESET signal, the active state is commenced. When in the active state, the CPU is either fetching, decoding, or executing an instruction (each of these states are described with VHDL sequential statements). On the rising edge of the CLK signal, FETCH reads the current memory location into the instruction register. After this is completed, the opcode is extracted from the instruction. Based on this opcode, one of several instructions is executed. After execution, the fetch state is again commenced.

Note that the description is based on states. Each state's functionality can be described either by using VHDL sequential statements, concurrent substates, or sequential substates sequenced by arcs. Each state may contain declarations whose scope is all descendant states.

Also note that two different arcs are used for state transition. The arc originating from a dot inside a state is called an *exit-on-completion* arc, or EOC arc. Only when the state has *completed* (i.e. all statements have completed and all signals received their new value), *and* the arc condition evaluates to true, will this arc be traversed. An EOC arc by default has a condition of true; note that EOC arcs are found exiting the fetch, decode, execute, and reset states.

The second type of arc, an *exit-immediately* arc, or EI arc, is seen pointing from the ACTIVE_STATE to the RESET_STATE. It is drawn originating from the perimeter of the state, and means that whenever the arc condition becomes true, the arc should be traversed. Thus, regardless of whether the system is currently fetching, decoding, or executing, the CPU is immediately reset on the rising edge of the RESET signal.

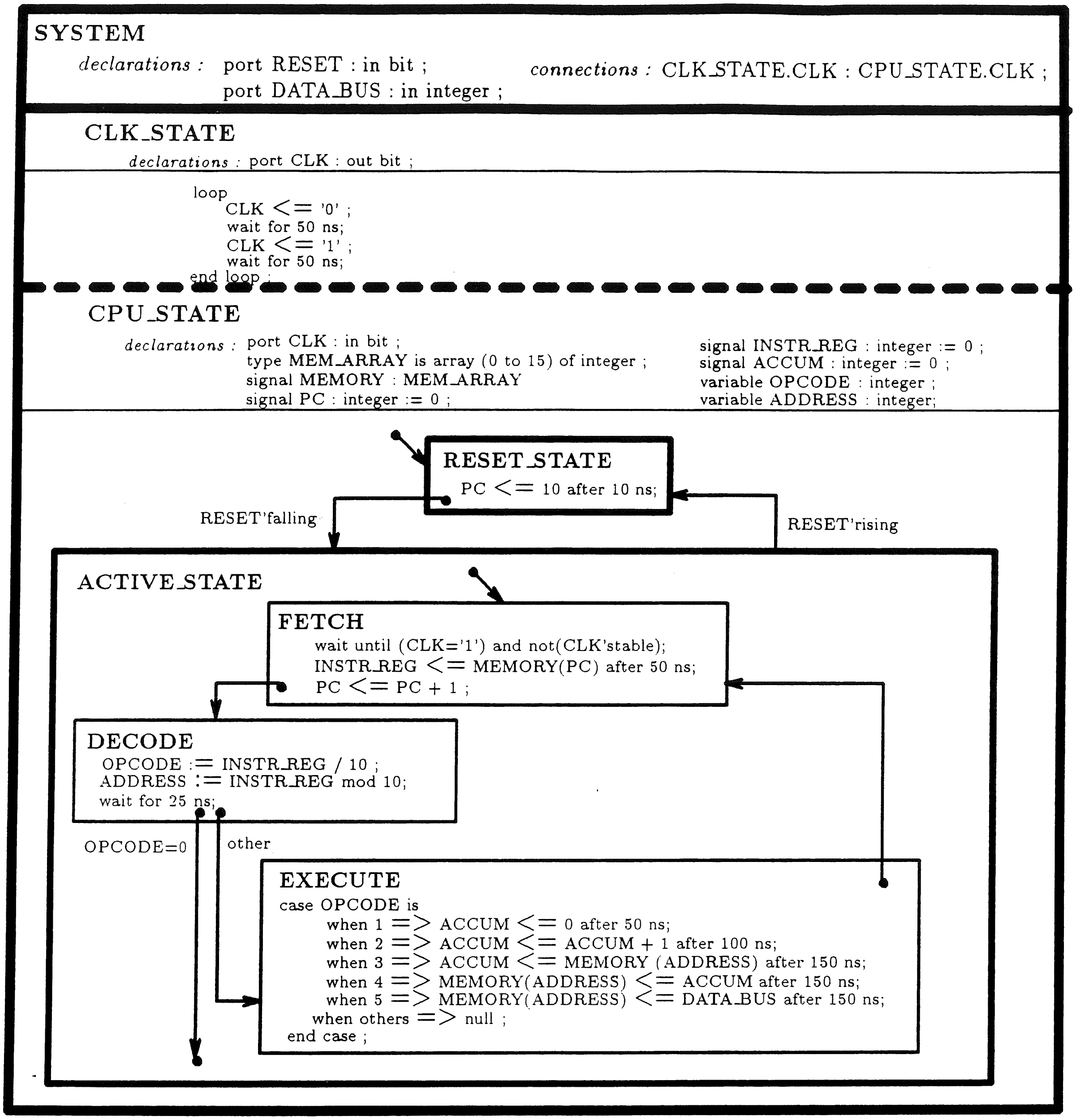


Figure 1: SpecChart of an example computer system

3 SpecChart Abstractions

The SpecChart language is geared towards concise and understandable specifications of the system being designed. To facilitate such a specification, the language has several abstractions or constructs, which are not found in VHDL. These abstractions are discussed below.

3.1 Hierarchical States and State Sequencing

State diagrams have long been popular for describing simple finite state machines, and when combined with hierarchy and concurrency, they are a very powerful way to concisely specify real systems. Any state diagram requires the concept of being 'in' a state, or with SpecCharts the concept of a state being active. Also, 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. Finally, the concept of an 'arc', or transition from one state to another, is needed.

None of these concepts are present in VHDL, since its basic units are not states, but instead blocks and processes, all of which execute in parallel and thus resemble hardware (however we are interested in systems).

The hierarchy (blocks) provided by VHDL is for scoping rules of declarations only. When used with block guards and control signals it can provide some functional hierarchy. Combined with guarded signal assignments, blocks are excellent for modeling at the register transfer level, but we assume the specification is at a more abstract level requiring the computational power of process sequential statements. Process hierarchies in VHDL are not permitted. Mimicking hierarchical states must be explicitly done by the use of blocks, block guards, control signals, process activation statements, and control processes to sequence blocks and processes and handle declaration scoping. It is the job of the SpecChart translator to do this automatically.

3.2 Protocol Based Interprocess Message Transfer

To facilitate easy specification of interprocess communication, SpecCharts support the concepts of channels and protocols. Channels are a high level abstraction used to avoid having to specify low level ports and data transfer statements for interprocess communication. By declaring a channel between two or more communicating processes and associating a protocol with it, the designer would have completely defined the interprocess communication mechanism.

While relatively simple protocols may be implemented in VHDL using procedures, representing most protocols with VHDL procedures would be cumbersome if not impossible. For example, the DMA transfer of data between an IO device and the memory of a computer system can be viewed as a protocol for communication between the three processes - memory, processing unit and the IO device. A DMA protocol may have several transfer and error-checking modes, and implementing it using a procedure would be very difficult. It is simpler to functionally split up such complex protocols into states, with transitions between them. This is exactly how SpecCharts represent protocols - *as a SpecChart itself.* This permits the protocol to be specified by using other SpecChart abstractions like state hierarchy and sequencing, which are built into SpecCharts. The task of translation involves implementing the protocol based message transfers of the design in VHDL.

3.3 State Completion: Exit-on-Completion Arcs

In a state based specification, 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. SpecCharts provide a type of transition arc called an *exit-on-completion* arc. An EOC arc from a substate that is currently active causes a transition to the next substate if and only if the current substate has completed execution, i.e. all computations made by the substate have completed, and the condition associated with the arc is true.

In VHDL, it is difficult to keep track of the signal assignments made in a process, and to suspend the process when all signals have received their new values (and is thus 'complete'). This is another task that needs to be performed while translating SpecCharts to VHDL - dynamic calculation of the state completion time.

3.4 Asynchronous Event Transitions - Exit-Immediately Arcs

While modeling a design in VHDL, it may be required to terminate a process immediately on the occurrence of an event. An example of such an event could be a RESET signal in a computer. However, if the VHDL process contains wait statements, the process could still be at a wait statement when such an event occurs and thus 'miss' the reset signal. Also, even if we stop execution of a process, signals may have been scheduled to receive values later (using an 'after' clause) which should no longer take effect. Writing code to terminate a process immediately makes the code very difficult to read and to synthesize from.

In SpecCharts, the *exit-immediately* arc provides this function. If we wish to terminate state execution immediately and proceed to the next state whenever a certain event occurs, we only need to include an EI arc between the two states, labeled by that event. Translation must ensure processes are terminated immediately when an EI arc is traversed.

3.5 Global Variables

Variables are powerful computational objects which remove the concept of 'time' and ensure sequentiality of statements. Using variables can greatly simplify descriptions by making them concise and easy to understand.

In VHDL, the scope of a variable is limited to the process in which it is defined. It is not possible for two processes to share the same variable due to its value not being defined over time. 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, on the other hand, allows variables to be global over sequential substates, i.e. states such that no two states are active at the same time. This relieves the designer of the burden of declaring local variables in each state. The translation of a SpecChart to VHDL necessitates eliminating global variables while providing the same functionality.

3.6 Global Signals

In VHDL, global signals can be assigned to by several processes. However, even if it is guaranteed that no two processes will drive the same signal at the same time, we still need to write resolution functions for the signals, and in every process, to enable and shutoff drivers as appropriate. This is not only cumbersome, but serves little purpose in that the resolution functions are not really resolving anything for each of those signals - they just return the value written by the only active process.

SpecCharts allow a designer to assign to a signal in any number of states. In case a signal is written to in several sequential substates, SpecCharts will permit this without the designer having to write resolution functions or shut off drivers for the signals when the states are not active.

In VHDL, if a signal could be assigned to by more than one process simultaneously, the resolution function resolves the signals based on the *values* that the processes are writing to it, rather than which processes are writing to it. SpecCharts have the concept of arbitration, where, in case it is possible for a signal to be assigned by two or more concurrent states simultaneously, some sort of a *priority* can be implemented between the states that are driving the signal. Thus a user could possibly specify a fixed or rotating priority (or an arbitrary complex priority scheme) between the several states writing to the same signal, or could simply resolve based on the values.

3.7 Other Abstractions

SpecCharts provide the concept of *timeout arcs*. These arcs are special EI arcs which limit the amount of time that can be spent in a state. The *other arc* is an EOC arc which has the condition true associated with it when the conditions associated with the rest of the EOC arcs emerging from that state are false.

SpecCharts are different from VHDL in another important respect. In VHDL, any signal or variable that has an initial value specified in the declaration itself will be initialized once when the simulation is started. However, in SpecCharts, initializations specified in a state are carried out every time the state is activated.

We have discussed above the abstractions de available by SpecCharts. The main purpose of these abstractions is to enable a design to be specified in a more concise and readable manner . Another purpose served by such abstractions is to provide more information to the synthesis tools. For example, consider the implementation of an exit-immediately arc using the 'wait until ..', 'if..then..else' statements (as explained in Section 4.6). It is nearly impossible for a synthesis tool to abstract these out of the VHDL model as being an exit-immediately arc. Using the abstractions enables better understanding of the true functionality of the design by both the designer and the synthesis tools. The task of translation is to represent each SpecChart abstraction in functionally equivalent VHDL so it can be simulated. We now proceed to discuss this task.

SpecChart Construct	VHDL Implementation	*Comments*
State	Block	
Substate	Sub-block	
State declarations and SpecChart scoping rules	Block declarations and VHDL scoping rules	Using nested blocks preserves hierarchy, simplifies translation
Active, complete, or inactive state	In each state declare two boolean signals per substate. *inState* (where *State* is the substate name) indicates if active or inactive, true meaning active. *doneState* indicates if complete (and waiting to be deactivated), true meaning complete. Add control process to non-leaf state's block. Modify code of leaf state.	Details of control process and code modifications follow
Leaf state code	Single process in state's block containing code, only executes when state is active	
Channels, protocols	Perform simple interface synthesis, expanding channels to ports, replacing channel calls with protocol SpecChart, and channel connections with port connections.	
Port connections	Declare unique signal for each net, replace all port accesses by the signal of its net.	
Global variables over sequential substates	Change global variable declaration to global signal. To retain variable semantics, declare local variable in each leaf writing to the global. Set to global at beginning, and replace references to global by local variable. Follow every write to local variable by updating the global with the value of the local variable.	Uses global signal to achieve global scope, local variable to retain variable semantics, result is same as global variable.
Global signals over concurrent states	Perform simple arbitration, where each concurrent state assigning to the the signal operates on its own copy, and an arbiter state sets the global signal with the appropriate value	
Global signals over sequential states	Set all signals assigned in leaf states to null at the end of the state. Declare a resolution function for each type which returns the first and only active driving value.	Performing arbitration guarantees to remove all global signals over concurrent states, only remaining global signals are over sequential states
EI arcs	SpecChartNon-leaf: When deactivated, deactivate all substates. *Substate sequencing:* When an EI arc condition is true and the arc's source substate is *active*, deactivate the source and activate the arc's destination substate. SpecChartLeaf: Modify all wait statements to stop waiting if the state is deactivated. Follow each wait by a statement checking if state is inactive; if so, jump to end code. Jump to end accomplished by enclosing code in loop, and executing exit loop. After the loop, set all signals assigned in this state to null.	Goal is to stop executing state *immediately*, be ready to be reactivated so don't miss any events
EOC arcs	SpecChartNon-leaf: When control flows to stop dot, indicate completion, wait until deactivated, reset the completion signal to false. *Substate Sequencing:* When an EOC arc condition is true and the arc's source substate is *complete*, deactivate the source and activate the arc's destination substate. SpecChartLeaf: Declare two variables, *global_time* and *remain_time* . Initialize *remain_time* to 0. Set *remain_time* to MAX(*remain_time* , after clause value) after every signal assignment with an after clause. Set *global_time* to 'now' before every wait, and to *global_time* - 'now' after every wait. Follow by setting *remain_time* to MAX(*remain_time* - *global_time* , 0). At end of leaf code, wait for *remain_time* .	Goal is to wait until all scheduled assignments have been made
Signal/Variable initializations in state declarations	Make the state's program a sequential substate. Precede it by a new sequential substate in which signals and variables are assigned their initial values. Add an EOC arc with condition true from this substate to the substate containing the program.	Since initializations occur every time state is activated, not just at beginning of simulation
Timeout(x) EI arc condition	Replace by *inState'stable(x)*	
'other' EOC arc condition	Replace by complement of ORing of the rest of the substate's EOC arcs	

Table 1: VHDL implementations of SpecChart constructs

4 Transformations

This section describes how each SpecChart abstraction is transformed into functionally equivalent VHDL constructs.

4.1 Hierarchical States and State Sequencing

In SpecCharts, the actual computations are made in leaf states, since it is there that the VHDL code exists; thus the main task is to activate (and deactivate) leaf states' code. We do not flatten the design. Instead, our activation scheme for VHDL maintains the same hierarchical model as in SpecCharts, which is that all state activation is between a parent state and its children states. The parent activates/deactivates its children, and a child tells its parent when it has completed. For example, consider a parent with two concurrent substates, each substate having two sequential substates. When the parent is activated, it immediately activates both substates, each of which then immediately activate its first substate, until the appropriate leafs have been activated.

4.1.1 General VHDL Structure

Each state becomes a block. In SpecCharts, a state may contain substates. In the VHDL, a block may contain sub-blocks. In SpecCharts, a state's arcs activate/deactivate substates. In the VHDL, a control process is added to the block to activate/deactivate sub-blocks. 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. The use of nested blocks preserves the hierarchy, permitting use of VHDL scoping rules. Thus the heart of a non-leaf state's block is its control process, of a leaf state is its code process. Figure 2 shows the general structure of the VHDL produced when SYSTEM is translated.

4.1.2 Control : State Activation/Deactivation

A state's main responsibilities include:

- Waiting until being activated by its parent, and then beginning execution of its code or activating/deactivating appropriate substates
- If deactivated by its parent, a state has the *important* responsibility of immediately deactivating all substates or terminating code execution
- If the state completes while activated, the state must inform the parent of its completion, and then wait until the parent deactivates it

As an example of how this scheme works, consider the case of a state being activated. It will activate the appropriate substates, which in turn activate their substates, and so on, until all the appropriate leafs are activated and have thus begun executing their code. Conversely, when a state is deactivated, it will immediately deactivate all its substates, which in turn will deactivate their substates, and so on, until all leaf states have been deactivated.

```
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

The implementation involves declaring two boolean signals in a state's declarations for each substate, initialized to false. **'inState' (where State is the substate's name) is used to activate/deactivate the child, and is set only by the parent's control process**, true meaning active. **'doneState' is set by the child to indicate completion to parent and is set only by the child**, true meaning completed and waiting to be deactivated.

Non-leaf states

In non-leaf states, the state's block contains nested blocks. A *control process* is added to this block. It waits until:

- a change on inState, which means its parent is either activating or deactivating it, OR
- an EI arc condition is true AND the arc's source substate is active, OR

- an EOC arc condition is true AND the arc's source substate is completed

If any of the above are true, the process performs one of the following actions:

- if the state is being activated (inState changing to true), activate the appropriate substates
  - if sequential substates, activate the first substate
  - if concurrent substates, activate all substates
- else if the state is being deactivated (inState changing to false), deactivate all substates
- else if a substate transition should be made due to an arc, deactivate the arc's source substate and activate the arc's destination substate. If the next state is 'stop', inform parent of completion, and then wait until deactivated (note: if the state has concurrent substates, then all substates with arcs pointing to the stop dot must have completed before indicating completion to parent).

After the action is performed, the process goes back to the initial wait. See figure 3.

If two possible actions could be taken, determinacy is provided by the if-then-else statement, which always gives activation/deactivation highest priority, followed by EI arcs in the order they were specified, followed by EOC arcs in the order they were specified.

Leaf states

In a leaf state, the state's block contains a single code process. The process should initially wait until being activated (inState changing to true). It should then execute the code; if deactivated it should immediately terminate this execution (see section 4.6 below). If the end of the code is reached, it should inform its parent of completion, wait until deactivated, and go back to the initial wait. Figure 4 shows how state FETCH of the SYSTEM example is modified. Note that the signal 'guard' is used throughout the code rather than the actual activation condition 'inState=true and not(inState'stable)'. This is purely for conciseness, and is achieved by making the leaf's block contain the activation condition as its guard.

4.2 Channels, Protocols, and Connections

To implement channels and protocols in VHDL, simple interface synthesis must be performed. This involves replacing each channel declaration with port definitions as defined in the protocol, and then replacing each channel call with the inline expansion of the protocol SpecChart. Connections between channels are then replaced by connections between ports. Other interface synthesis tasks, such as port optimization and protocol matching, are not performed as they are not needed for simulation of the current specification and would change the functionality.

Once all channels have been expanded, all connections of ports must be implemented in VHDL. This is done by declaring a signal for each net (connection list), replacing each substate use of a port by the signal, and removing the port declarations. Figure 5 shows the connection of the clock ports of CLK_STATE and CPU_STATE after ports are removed.

control : process

```
begin
    wait until state is being activated or deactivated
        OR EI_arc1_cond and arc's source substate is active
        OR EI_arc2_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 is true and not stable)
        activate first substate if sequential substates
        activate all substates if concurrent 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 arc's destination substate
    elsif .... for all EI arcs
        ....
    elsif EOC_arc1_cond and arc's source substate is active
        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

4.3 Global Variables: Conversion to Signals

SpecCharts allow a user to use global variables over sequential states while specifying a design. In VHDL, variables are permitted only in processes, i.e. the leaf states in our model. The global variables are transformed into global signals which are supported by VHDL , without altering variable the semantics of the given SpecChart.

To achieve this conversion, firstly, 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. This local variable is prefixed by the string *TempVar*.

The local variable 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. This has to be done every time and not just at the end of the state, because the state may be terminated prematurely due to a condition on an EI arc emerging from that state becoming true. The reason behind using a temporary variable is to preserve the semantics of variable use within the leaf state VHDL code. The global signal is needed to pass the updated value from one sequential substate to the next.

FETCH

```
if guard then
    wait until CLK='1' and not(CLK'stable);
    INSTR_REG <= MEMORY(PC) after 50 ns;
    PC <= PC + 1;
    doneFETCH <= true;
    wait until not(inFETCH);
    doneFETCH <= false;
end if;
wait on guard;
```

Figure 4: State FETCH transformed to handle activation and indicate completion

As an example, see figure 6. The variable OPCODE declared in state CPU_STATE has been converted to a signal declaration. Since OPCODE is written to in state DECODE, a local variable TempVar_OPCODE is declared in that state. It is initialized to the global signal OPCODE, each write to OPCODE is followed by an update of the global signal OPCODE, and all references to the variable OPCODE are replaced by TempVar_OPCODE. Similar changes would *not* be made in state EXECUTE where OPCODE is also accessed because the state EXECUTE does not assign to the variable OPCODE.

4.4 Global Signals over Concurrent States - Arbitration

A SpecChart can contain concurrent states which may assign a value to the same signal which is global to both of them. It might then be the case that two concurrent states assign to the same signal at the same time, and the decision as to which state actually succeeds in doing so is made by an *arbiter* state.

If a signal can be assigned to in two or more concurrent states, each such state is associated with its own copy of the signal. These multiple copies are declared in the same state as was the original signal. Also added as a concurrently executing state is the arbiter. Each state will now assign to its copy of the signal. The arbiter monitors all copies of the signal that are being assigned to in different states. Whenever a state assigns to its local copy, the arbiter accordingly updates the actual signal. In case two states assign to their respective copies of the signal simultaneously, the arbiter determines which assignment gets priority, or how to resolve the multiple values into a single value.

4.5 Global signals over Sequential Substates

After arbitration has been done, each signal in the SpecChart is now assigned by at most one state at a time. If in the original SpecChart, two states could assign a value to a signal at the same time, they now write to a private copy, while the arbiter process writes to the actual

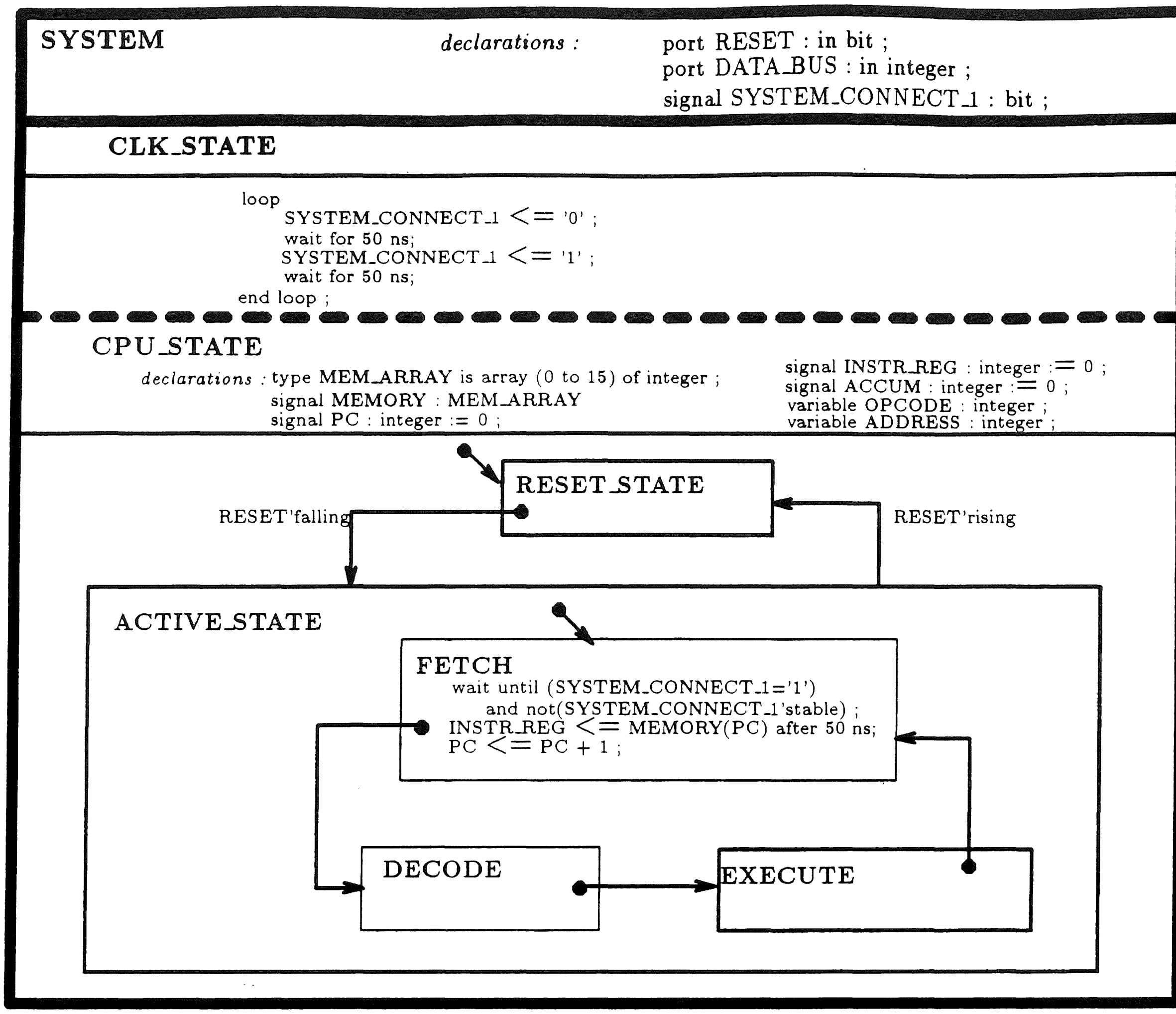


Figure 5: SYSTEM SpecChart after connections are transformed

signal. The only global signals left are those which can only be assigned over sequential substates.

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. Due to the fact that VHDL blocks are executed concurrently, substates (modeled as blocks) that originally intended to assign to the same signal exclusive of each other, would now, according to VHDL semantics, possibly assign to it concurrently. We need to ensure that only one of the states is assigning a value to the signal at a given time. To achieve this, we declare a resolution function which simply returns the value on 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.

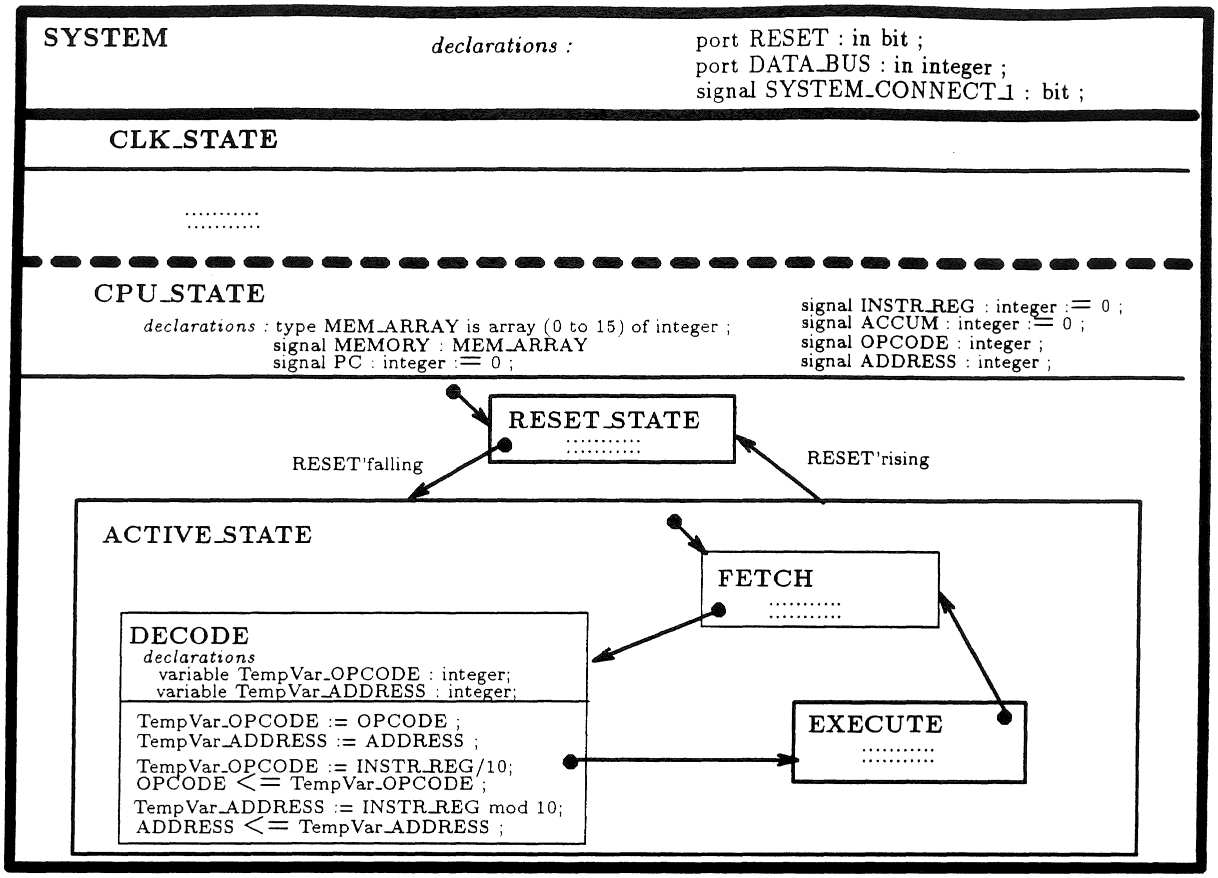


Figure 6: SYSTEM after global variables are transformed

4.6 EI Arcs : Immediately Terminating a State's Execution

Recall that when an active state is deactivated (inState goes false), it must terminate all computations being performed and immediately wait to be activated again. Doing so for a non-leaf state is simple: when deactivated, the state's control process will perform an action which deactivates all substates (i.e. set inState to false for all substates; see section 4.1.2). The control process then goes to its initial wait statement.

Leaf states are not so simple. At any point in the code execution, a state that becomes deactivated must immediately:

- Ensure no future signal updates occur that were caused by signal assignments scheduled in this state (using an 'after' clause)
- Be ready to be activated

This requires that the state can not be sitting at a wait statement, since it might miss the deactivation signal. Also, the code should immediately jump to the end where signals are set to null, thus shutting off drivers to prevent future updates.

The implementation of these requirements involves adding clauses to all wait statements which will terminate the wait if inState goes false (state deactivated). The clause added to a wait statement is determined as follows:

- if no on clause or until clause exists, create a 'not(inState)' until clause
- if no on clause exists but an until clause does, append 'or not(inState)' to the until clause
- if an on clause exists but no until clause does, add 'inState' to the on clause
- if an on clause and until clause exist, add 'inState' to the on clause, and append 'or not(inState)' to the until clause

Each wait statement is then followed by a check to see if inState is false, meaning the wait statement was ended because of state deactivation. If so, we want to jump to the end of the code. However, there is no 'go to' command in VHDL. This problem is solved by enclosing the code in a labeled loop. The statement 'exit loop label;' will then jump to the end of the code. The loop never really loops since an exit is added before the end of the loop.

For example, see figure 7. FETCH has been modified from figure 4 to take into account not just activation and indication of completion, but also deactivation. The wait statement has been modified and an if statement added. The code has been enclosed in a loop, and all signals assigned by the state are set to null just after this loop. Thus if a reset is performed, CPU_STATE will deactivate ACTIVE_STATE, which in turn will deactivate FETCH, which upon seeing it is being deactivated will immediately set all signals to null and wait to be reactivated (wait on guard, i.e. 'wait until inFetch and not(inFetch'stable)').

4.7 EOC Arcs: Calculating State Completion Time

Recall that a state must inform its parent when it has completed, and then wait until it is deactivated, after which it waits until it is activated again. Handling EOC arcs for non-leaf states is simple: when an arc is traversed which flows to the stop dot, the control process performs an action which sets doneState to true and waits until inState goes false (see section 4.1.2).

Once again, leaf states are not so simple. 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, if a state contains only a single statement, X <= X + 1 after 10 ns , then the state completes after X gets its new value, i.e. after 10 ns. However, in the more common case of leaf code which contains loops and branches, the time to wait at the end of the state must be dynamically determined, since it may differ each time the state is executed. Ideally this would be done by checking at the end of the statements the drivers for all signals assigned to in the state, finding the transaction scheduled to occur in the most distant future, and waiting for that amount of time. However, no VHDL facility exists to access the scheduled transactions of a signal driver. Our solution is to use variables to keep track of the amount of time until the latest transaction would take place.

The implementation consists of declaring two variables of type time:

FETCH

```
if guard then
   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;
      wait until not(inFETCH);
      doneFETCH <= false;
      exit FETCH_loop;
   end loop FETCH_loop;
end if;
INSTR_REG <= null;
PC <= null;
wait on guard;
```

Figure 7: State FETCH after being transformed to handle deactivation (i.e. EI arcs)

- *global_time*: before every wait statement, set *global_time* to 'now' ('now' is a VHDL defined time equal to the current simulation time). After every wait statement, set it to 'now' - *global_time*. The value tells us exactly how long we waited at the wait statement.

- *remain_time*: initialize to 0. After every signal assignment with an after clause, set it 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 (actually after the global time update that now occurs after every wait statement), set *remain_time* to MAX(*remain_time* - *global_time*, 0). This updates *remain_time* when we procede forward through simulation time, thus reducing the amount of time we'll need to wait at the end of the state.

At the end of the statements, we add the statement 'wait for *remain_time*;'. Note that this also handles the common VHDL problem of how to wait for signals to settle down with their new values in delta time. Even if *remain_time* is 0, this wait ensures that all delta time assignments are made by advancing to the next simulation cycle.

Figure 8 shows how state FETCH is modified from figure 4 to wait until the state is complete before indicating completion. Note that REMAIN_TIME will be 50 ns when the statement 'wait for REMAIN_TIME' is reached. If, however, the statement 'wait for 10 ns' was inserted between the assignment to INSTR_REG and the assignment to PC, then the final wait would be for 40 ns.

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(REMAIN_TIME - GLOBAL_TIME, 0 ns);
    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 8: State FETCH transformed to correctly indicate completion (i.e. EOC arcs)

4.8 Other Transformations and Details

In VHDL, signal/variable initialization occurs only once, when the simulation is started. In SpecCharts, the initialization should occur every time a state is entered. This requires that an initial sequential state be added which sets the initial values. This is implemented by enclosing a state's program section in a new substate, preceding this by a new sequential substate which contains assignments to the signals/variables with their initial values, with an EOC arc (condition 'true') pointing to the next substate (which the original program is in), and removing the initializations from the declarations. See figure 9 for an example.

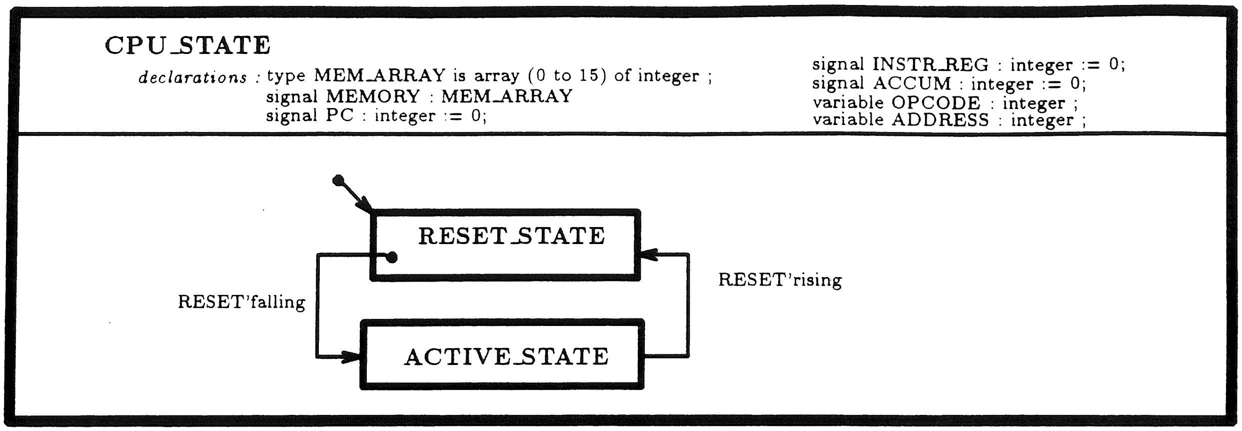
When transforming timeout arcs, which must have the form "timeout(x)" where x is a time, we can take advantage of the 'inState' signal of the arc's source substate, replacing "timeout(x)" by "inState'stable(x)". This means that if the state has been active for time x, traverse the arc, which is exactly the meaning of the timeout arc.

The condition 'other' on an EOC arc is just a notational convenience which is equivalent to the complement of the ORing of all the conditions of the remaining EOC arcs. The 'other' condition is replaced with this expression.

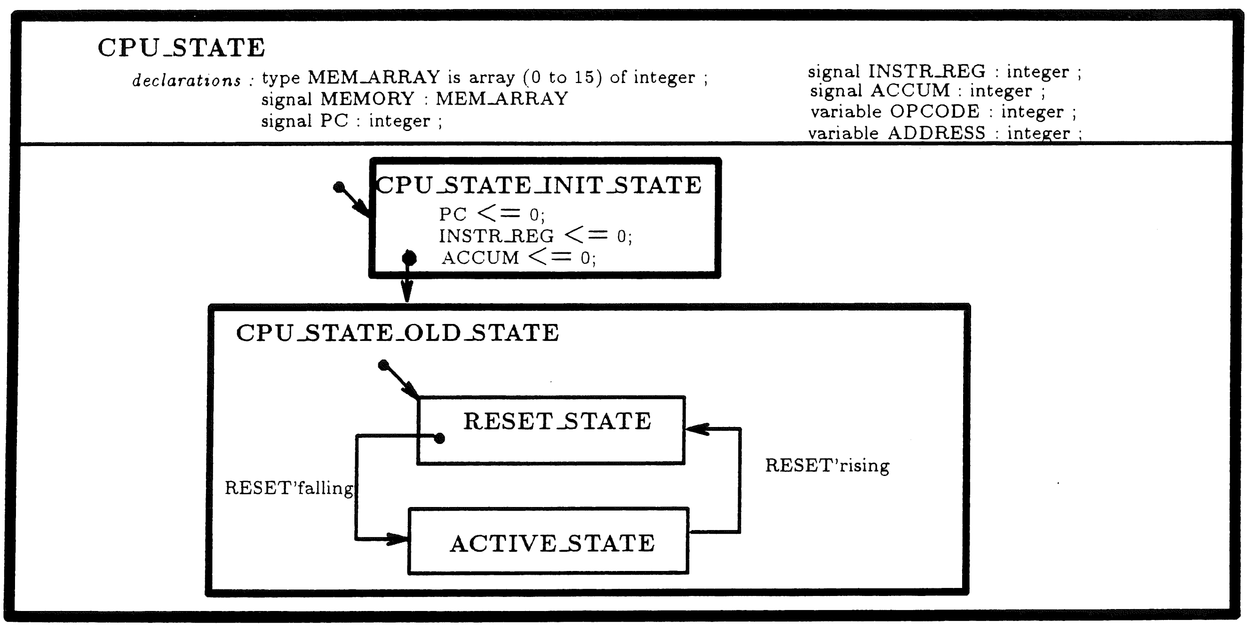
A detail concerning ports should be mentioned. The topmost state (i.e. the state being converted to VHDL for simulation) probably contains port declarations. These will become the ports of the entity created by translation. However, note that ports can be global over concurrent and sequential states. Since output ports can be written to, they may require arbitration and resolution functions, just as global signals did. The solution is to add a new signal for each output port to the top state. All port occurrences in the SpecChart are replaced by this new signal. The signal will then be treated like all other signals, i.e. it may be arbitrated, get resolution functions added, etc. Then, in the VHDL architecture, a concurrent signal assignment is added which simply assigns the signal to the port; thus the same output is achieved.

Another detail involves procedures. Since procedures can themselves contain wait statements and signal assignments, they too need to be operated on to handle deactivation and completion (EI arcs and EOC arcs). Thus, their wait statements must be modified, checks placed after them that jump to the end if deactivated, global_time and remain_time must be updated after waits and signal assignments, and the body of the procedure must be enclosed in a loop (see section 4.6). Also, all subprocedures declared in this procedure must similarly be operated on. Finally, the formal parameter list must have remain_time and inState added to them, as must all procedure calls.

Yet another detail relates to drivers in VHDL. According to [IEEE88], a process which drives a subelement of a resolved signal of composite type must drive every scalar subelement of that signal. We thought this was taken care of by the $X <= null$; statement added to each leaf's code. However, for reasons we have been unable to determine, the simulator we use [Zyca89], after a driver is shut off by setting it to null, will not turn the driver back on unless all subelements are subsequently assigned, not just one of them. We do not know if this is VHDL semantics or just a quirk with the simulator; in either case, we have found that setting all composite signals to themselves at the beginning of the leaf code will turn the driver back on without changing any functionality.



(a) CPU_STATE with initialized signals in the declaration section



(b) Modified SpecChart after addition of the initialization state

Figure 9: Transforming a state to initialize declarations every time it is activated

5 Translation

The general transformations that need to be made have been discussed. This section describes the order those transformations are made and the steps that are taken to create the output of the translator, a VHDL file which can be compiled into a VHDL entity.

Firstly the transformations are carried out on the SpecChart. Then the entity containing the port declarations is written to the output VHDL file followed by the beginning of the architecture. Then, starting with the top state and proceeding recursively in a depth first order, we write out the block level VHDL code for each state. This is followed by the addition of concurrent signal assignment statements to update the local signal representing each of the output ports. Finally, a process which simply sets the inState signal of the topmost state to true is added and the architecture is ended. The translation algorithm with references to sections where the details are discussed is shown below.

5.1 General Algorithm

```
Copy the SpecChart, so the translator can modify it
Perform the transformations on the SpecChart (sec. 4)
    Add signals for the top state's output ports, replace these ports' accesses by the signals (sec. 4.8)
    Perform simple interface synthesis (sec. 4.2)
    Replace connections by signals (sec. 4.2)
    Replace global variables by signals and local variables (sec. 4.3)
    Assign composite signals to themselves in accessing leaf states (sec. 4.8)
    Perform simple arbitration (sec. 4.4)
    Transform initializations into initial states (sec. 4.8)
    Add resolution functions for all signals (sec. 4.5)
    Declare control signals in each state for each substate (sec. 4.1.2)
    Declare a MAX function in the top level (sec. 4.7)
    Convert timeout arcs (sec. 4.8)
    Convert 'other' arcs (sec. 4.8)
    For each leaf state do
        Declare the signal remain_time (sec. 4.7)
        Declare the signal global_time (sec. 4.7)
        Update remain_time after signal assignments (sec. 4.7)
        Set global_time before and after all waits (sec. 4.7)
        Initialize remain_time to 0 at the start of the code
        Add the final wait statement to the end of the leaf code (sec. 4.7)
        Modify wait statements with clauses for deactivation (sec. 4.6)
        Add statements after all wait statements that exit the state loop if state is inactive (sec. 4.6)
        Add handshake statements to the end of the code that handle completion (sec. 4.1.2)
        Enclose the entire code, as it now exists, in a loop (sec. 4.6)
        Enclose the entire code, as it now exists, in an 'if guard' statement (sec. 4.1.2)
        Set all signals assigned in the state to null at the end of the code (sec. 4.5), (sec. 4.6)
        Add a wait on guard statement to the end of the code (sec. 4.1.2)
Create a new file, and write an entity containing the port declarations
Write the start of an architecture
/* Starting with the top state, each state is written as VHDL */
Make the current state the top state
    /* Beginning of state writing algorithm */
```

```
    Start a block having the state name as its label.
    If the state is a leaf, add the guard condition (sec. 4.1.2)
    Write non-variable declarations as the block's declarations
    If the state is a leaf state,
        Write a process , it's declarations being the state's variable declarations,
        its statements being the state's statements
    If the state is a non-leaf state,
        Recursively call this state writing algorithm for each substate
        Add the control process (sec. 4.1.2)
    End the block
    /* End of state writing algorithm */
Add the concurrent signal assignments for the output ports (sec. 4.8)
Write a process which merely sets the inState signal to true for the top state
End the architecture and close the file. The translation is now complete.
```

For clarity, the transformations needed for procedures ((sec. 4.8)) were not included in the algorithm above. To handle procedures, the following should be added at the beginning of the leaf state for loop:

```
Add to procedure calls the inState parameter (sec. 4.8)
Add to procedure calls the remain_time parameter (sec. 4.8)
```

and add the following after the leaf state for loop:

```
For each procedure do
    Add inState formal parameter (sec. 4.8)
    Add remain_time formal parameter (sec. 4.8)
    Declare global_time variable (sec. 4.7)
        Add inState parameter to procedure calls (sec. 4.8)
        Add remain_time parameter to procedure calls (sec. 4.8)
        Update remain_time after signal assignments (sec. 4.7)
        Set global_time before and after all waits (sec. 4.7)
        Modify wait statements with clauses for deactivation (sec. 4.6)
        Add statements after all wait statements that exit the state loop if state is inactive (sec. 4.6)
        Enclose the entire procedure body, as it now exists, in a loop (sec. 4.6)
        Recursively apply the algorithm to all procedures declared by this procedure (sec. 4.8)
```

5.2 Reducing the Amount of Generated VHDL Code

The algorithm above produces functionally correct VHDL code, but may contain unnecessary statements that makes reading the code more difficult. The following can reduce the amount of generated code:

- Calculating the remaining time in leafs and then waiting for that amount of time at the end of the leaf code (sec. 4.7) is only needed if an EOC arc exits the state, otherwise all those calculations go unused. Thus, if no EOC arc exits the state we merely add the statement 'wait;' to the end of the leaf, since deactivation does not depend on completion (the wait is needed since the leaf code gets statements added to the end that should only be executed when the state has been deactivated).

- Checking for deactivation throughout leaf code (sec. 4.6) is only necessary if an EI arc exits the leaf state or any of its ancestors, otherwise the checks are not added.
- If no EOC arc exits a state, the EOC handshake (sec. 4.1.2) need not be added, and the 'doneState' control signal need not be declared.

Adding the above checks to the translator consistently reduced the amount of output VHDL code by 30% of its original size.

6 Related Work

One approach [TiLeKi90] to specify and simulate the whole behavior of systems and ASICS at a high level was the graphics oriented Real Time Structured Analysis / Structured Design (SA/SD) method. The SA/SD description consists of hierarchical data flow diagrams, state transition diagrams and textual minispecifications. The functionality of the design is represented as textual minispecifications while the control behavior is specified using graphical state transition diagrams. The high level specification is converted automatically to behavioral VHDL for simulation purposes using a rule based Sokrates-SA compiler. The transformation of the specification of a fluid level controller (consisting of three dataflow diagrams and one state transition diagram) took fifteen minutes to translate into VHDL, producing 400 lines of code.

A methodology was presented [MaWa90] which describes how to translate a set of statecharts [Ha87], derived from the system requirements document, to VHDL. Their translation scheme employs a number of basic constructs like nested blocks to model hierarchy and processes to represent VHDL statements of the specification, quite similar to the translation scheme for hierarchy and leaf state VHDL code presented in previous sections of this report. This similarity is not surprising, given that SpecCharts are identical to statecharts as far as representation of hierarchy and concurrency is concerned, and nested blocks with leaf processes seems to be the simplest way to implement those constructs in VHDL . The translation time and size of the generated code are unavailable since the authors describe a methodology only.

7 Results and Future Work

The translator is implemented in C and runs on Sun3/Sun4 workstations under UNIX. The current graphical interface is an X widget based application. A complete SpecChart graphical interface (i.e. graphical arcs, automatic placement of states on the screen, etc.) has not been implemented. SpecCharts can be entered via the graphical interface and stored in files in a textual format, or directly entered in their textual format using any text editor. A library of C routines has been written for parsing the textual format, maintaining and manipulating the SpecChart internal representation, and performing basic error checking, thus providing a somewhat object oriented environment. This library is intended for use by various SpecChart applications, such as the SpecChart to VHDL translator. The SpecChart to VHDL translator takes as input the SpecChart textual files and outputs a VHDL file. It consists of approximately 2700 lines of C code. 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. After port mapping signals to the component, the ports can be assigned values with signal assignment statements, in any combination of blocks and processes. We generally use processes, assigning values to the inputs and making assertions about the outputs to ensure correct operation of the entity.

We have translated and simulated many examples. One detailed example is the Controlled Counter [Arms89, NaVaGa90b]. Entering the SpecChart via the graphical interface

Model	*Lines of code*
Armstrong's mixed block/process description of Controlled Counter	81
Lis' process description of Controlled Counter	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	271
[GuDu90] 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	506
SpecChart model of SYSTEM example	70
SpecChart textual files of SYSTEM example	118
VHDL generated for SYSTEM example	312

Table 2: Lines of code for various models, including generated VHDL

required entering 67 lines of code. The textual SpecChart created by the graphical application contained 139 lines, and was translated to 271 lines of VHDL in .3 seconds. The test file was a set of UCI CADLAB test vectors used to verify correct operation of the Controlled Counter, and the simulation passed all checks.

A second detailed example is the DRACO peripheral interface ASIC [Rock89, GuDu90, NaVaGa90b]. This example is particularly interesting as it is a real existing chip used in industry. The specifications were provided by Rockwell International, and a detailed model was created from this specification and from information provided by several Rockwell engineers [Sito90, Pase90]. To ensure that the model was correct and complete, we used Rockwell's original test vectors, consisting of 23,000 lines of process code (the process statements were converted from VTI format statements using an automatic convereter written here). The simulation verified that the SpecChart model of DRACO behaved correctly. Entering the SpecChart via the graphical interface required entering 226 lines of code. This can be compared to a manually written VHDL description of DRACO written for another project that contained 392 lines of code. The textual SpecChart created contained 268 lines, and was translated to 506 lines of VHDL in 3 seconds. The compilation on Zycad of the DRACO entity took 3 seconds, and of the 23,000 line stimuli VHDL file took 9 minutes. The simulation then takes about 2 minutes.

The example computer system used in this document consisted required entering 70 lines of code to the graphical interface. The textual file was 118 lines, and was converted to 312 lines of VHDL in .5 seconds.

Table 2 summarizes the translation results for the previous examples.

The current translator does not perform interface synthesis, and does not permit user defined arbitration schemes as of yet. These will be added as SpecSyn, a tool for system level synthesis that uses SpecCharts as its specification language, is developed [VaNaGa90a].

Some improvement in the readability of the generated VHDL code can be made by eliminating unnecessary code. For example, if a global signal is only written by one state and it never needs to be deactivated (due to an EI arc), then a resolution function is not needed for that signal. There are several such situations where modifications need not be made.

Experience will show if it is feasible to debug a model from the VHDL code directly, i.e. using a VHDL source level debugger to find problems with the SpecChart description. Perhaps a debugger will need to be written for SpecCharts, itself possibly using a VHDL debugger.

The current SpecChart compiler only performs checks that the VHDL compiler does not. We thus use the VHDL compiler to perform many checks such as type checks, undefined signals, etc. The feasability of this approach in an industrial environment is questionable, but in our development environment there is no reason to duplicate the work done by the VHDL compiler.

8 Conclusion

This report 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 two detailed examples demonstrated that the concept of a high-level language on top of VHDL is beneficial for the modeler and does not decrease the efficiency of the simulation. The translation resulted in more VHDL code than manually written code, but simulation took the same amount of time and verified the correct functionality of the SpecChart models.

9 Acknowledgements

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #89-DJ-146). We are grateful for their support. We would also like to thank Joe Lis and Tedd Hadley for their advice and suggestions.

10 References

[Arms89] Armstrong, J., "Chip Level Modeling Using VHDL", Prentice-Hall, 1989.

[DuHaGa89] Dutt, N., Hadley, T., and Gajski, D., "BIF: A Behavioral Intermediate Format for High Level Synthesis", University of California, Irvine, Technical Report 89-03, September 1989.

[GuDu90] Gupta, R., and Dutt, N., "Behavioral Modeling of DRACO: A Peripheral Interface ASIC", University of California, Irvine, Technical Report 90-13, June 1990.

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

[IEEE88] IEEE Standard VHDL Lanuage Reference Manual, IEEE, March 1988.

[Lee89] Lee, E., et al, "Gabriel: A Design Environment for Programmable DSPs", DAC, 1989.

[LiSU89] Lipsett, R., Schaefer, C.F., and Ussery, C. "VHDL : Hardware Description and Design " Kluwer Academic Publishers, 1989.

[LiGa88] Lis, J., and Gajski, D., "Synthesis from VHDL", ICCD, 1988.

[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., "Modeling with SpecCharts", University of California, Irvine, Technical Report 90-20, July 1990.

[Pase90] Dave Pasela, Rockwell International, private communication, 1990.

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

[Sito90] Johnny Sitou, Rockwell International, private communication, 1990.

[TiLeKi90] Tikanen T., Leppanen T., and Kivela J., "Structured Analysis and VHDL in Embedded ASIC Design and Verification", EDAC, 1990.

[VaNaGa90a] Vahid, F., Narayan, S., and Gajski, D., "Synthesis from Specifications: Basic Concepts", University of California, Irvine, Technical Report 90-03, January 1990.

[VaNaGa90b] Vahid, F., Narayan, S., and Gajski, D., "SpecCharts: A Language for System Level Specification and Synthesis", University of California, Irvine, Technical report 90-19, July 1990.

[Zyca89] Zycad Corporation, Menlo Park, CA 1989