SpecCharts: A VHDL Front-End for Embedded Systems

Daniel D. Gajski
Frank Vahid
Sanjiv Narayan

Technical Report #93-31
June 23, 1993

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

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

Abstract

VHDL and other hardware description languages have become popular as system specification languages in top-down design. However, their constructs do not support the behavioral specification of embedded systems. We introduce a new conceptual design model, called Program-State Machines, that caters to embedded systems. We then describe the SpecCharts language, an extended version of VHDL, which supports capture of this design model. In conjunction with a translator to VHDL, SpecCharts can be easily incorporated into a VHDL design environment, with the advantages of significantly reduced specification time and fewer errors. The extensions introduced for VHDL are applicable to many other hardware description languages as well. We demonstrate the advantages of using SpecCharts for system specification capture through several experiments.

Contents

List of Figures

1 Introduction

Top-down specify-and-design approaches are gaining popularity in system design. In such approaches, one first specifies the desired system behavior formally using a simulatable program-like language, from which one then derives a design implementation. The behavioral specification is free of any implementation decisions such as the division of the system into chip modules. Such approaches are replacing design-and-capture approaches in which a design is derived from an informal specification and then captured as schematics and simulated.

Specify-and-design approaches provide many advantages through a design's lifecycle. First, by creating a test-bench early in the design process and simulating the behavior, functional errors and omissions are detected early and easily corrected. Similar corrections can be extremely difficult to make late in the design cycle. Second, by defining module behavior completely, fewer integration problems are likely to occur after concurrent design of each of the modules. Third, by using a machine readable language, automated estimators and synthesis tools can be applied to reduce the design time or to rapidly evaluate alternative implementations. Finally, by writing a behavioral specification independent of any implementation information, redesign is greatly simplified. Little or no reverse engineering is required to determine an existing design's behavior before modifying it for another application.

A variety of languages have been proposed for behavioral specification, such as VHDL [1], Verilog [2], HardwareC [3], CSP [4], and Statecharts [5]. A good language should support a conceptual model useful for the particular system to be specified. For example, C++ supports an object-oriented conceptual model which has proven useful for many large software systems. A language should also be able to represent a design through several steps of refinement, in which one successively adds implementation detail to the behavior, such as module partitions and communication protocols.

Many systems can be classified as *embedded systems*, i.e. systems whose behavior is defined by its interaction with its environment. The behavior describes the appropriate outputs as a function of the inputs to the system and its current mode or state. Existing languages support conceptual models (such as finite-state machines or communicating processes) which are inadequate for behavioral specification of embedded systems. This deficiency forces designers to either (1) informally describe the system using a good conceptual model, but then spend a great deal of time and effort coercing the description into the language constructs, or (2) describe the system using a less appropriate conceptual model which maps directly into language constructs. Both cases result in large specification times, more errors, and specifications that are difficult to read and maintain.

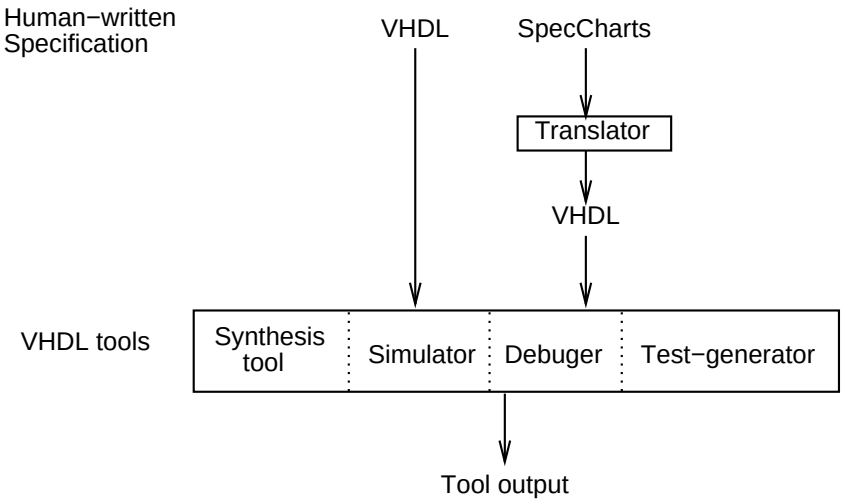
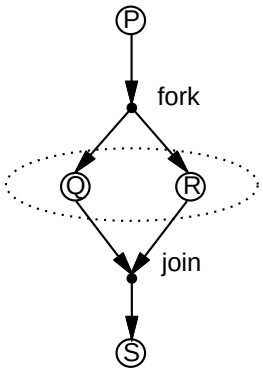


Figure 1: SpecCharts as a front-end language in VHDL design environment

To overcome these limitations, we developed an extended version of VHDL called SpecCharts, specifically intended to support a conceptual model useful for embedded systems. SpecCharts combines hierarchical/concurrent finite-state machines with the programming language paradigm, resulting in significantly reduced specification times, fewer errors, and more readable documentation. The SpecCharts extensions to VHDL can be applied to other program-like languages such as other HDLs or even C.

The SpecCharts language is intended to be used in a VHDL design environment. While VHDL is quite general as a specification language, it does not appear that any single language can easily capture all possible classes of system behavior. Using front-end languages such as SpecCharts can help remedy this problem. One may specify portions of a system with SpecCharts. Other portions may use other front-ends, such as timing diagrams for interfaces, with VHDL being used for all other portions. Because SpecCharts is an extension of VHDL, a designer familiar with VHDL can learn the language in only a few hours. A SpecCharts file is translated to VHDL and then treated as any other VHDL file, as illustrated in Figure 1. This use of SpecCharts is analogous to that of YACC for specifying parsers in a C programming environment.

This paper is organized as follows. In Section 2, we highlight the essential characteristics of embedded systems and demonstrate the inability of VHDL, as well as other languages, to specify such systems. In Section 3, we introduce a new conceptual model, refered to as *Program-State Machines*, that can easily represent embedded systems. The SpecCharts language is introduced and we show how its constructs allow easy capture of the Program State Machine model. In Section 5, we describe an algorithm for translating SpecCharts to VHDL. In Section 6, we describe several experiments performed that demonstrate the advantages of using SpecCharts for the specification of embedded



(a)

```
signal QR_activate, Q_complete, R_complete;

Main: process
  P();
  QR_activate <= true;
  wait until Q_complete
    and R_complete;
  QR_activate <= false;
  S();
end process;

Q_PROC: process
  wait until QR_activate;
  Q();
  Q_complete <= true;
  wait until QR_activate=false;
  Q_complete <= false;
end process;

R_PROC : process
  wait until QR_activate;
  R();
  R_complete <= true;
  wait until QR_activate=false;
  R_complete <= false;
end process;
```

(b)

Figure 2: Describing decomposable activities using VHDL processes: (a) desired functionality (b) VHDL description

systems. In Section 7, we present the current language and supporting tool status and plans for future work. In Section 8, we discuss several conclusions.

2 Existing HDL Limitations for Embedded Systems

As a result of attempting to specify several examples of embedded systems, including an ethernet coprocessor, an aircraft collision avoidance system, an interactive television processor, a telephone answering machine, a bladder-volume monitor, and a microwave-transmitter controller, we determined three basic characteristics of a good conceptual model for embedded systems which are not easily supported by VHDL: decomposable activities, state-transitions, and immediate mode changes. We now discuss each of these in detail and describe the difficulties in capturing these using VHDL.

2.1 Sequentially and Concurrently Decomposable activities

System behavior can be thought of as being comprised of a set of activities. An *activity* represents a particular mode of system behavior. It may be a computation, which is possibly complex or time-consuming, or it may be recursively-defined as a composition of sub-activities, where the sub-activities may be *sequential* or *concurrent* to one another. For example, consider the desired behavior shown in Figure 2(a). The system behavior is conceptualized as four activities, P, Q, R and S, each of which may be an arbitrarily complex computation. First P is executed, followed by a concurrent execution of Q and R, followed by S. Concurrent decomposition such as that of activities Q and R is often called a *fork*.

VHDL does not support concurrent decomposition of an activity. It supports system description as a set of concurrently executing sequential programs, called processes. The program statements support description of an activity's computation, while procedures support sequential decomposition. However, a process or procedure cannot be decomposed into sub-processes, thus limiting support of concurrent decomposition only to the top-level activity in the hierarchy only. The concurrent process model is appropriate for modeling existing hardware, where each hardware module can be mapped to a process. Behavior, on the other hand, represents a system's functionality and often does not map directly to a set of concurrent processes. Hence, VHDL is limited in its ability to capture system behavior.

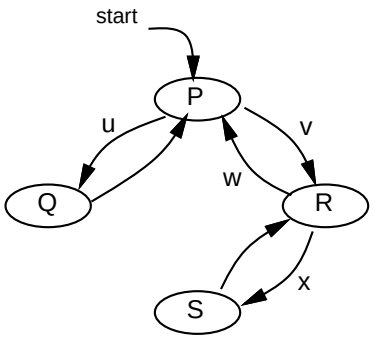
We can coerce description of a fork in VHDL using extra processes and signals as shown in Figure 2(b). A signal is created for each fork. When control reaches a point in the VHDL sequential statements when the fork should occur, the signal is asserted. Each concurrent subactivity of the fork is mapped to a process which remains idle until the signal is asserted. To implement a *join*, additional control signals are added which are asserted at the end of each fork process. In the figure, the *Main* process first calls procedure P which returns when it has completed its actions. *Main* then asserts the signal *QR_activate* which causes processes *Q_PROC* and *R_PROC* to begin executing their actions. When *Q_PROC* and *R_PROC* have completed their actions, they assert signals *Q_complete* and *R_complete*, respectively. Upon assertion of both completion signals, *Main* proceeds to call procedure S.

It should be noted that a person attempting to comprehend the VHDL description will initially be misled into believing that the system is composed of three concurrent activities, *Main*, Q and R. Only after mentally executing the code and tracing the effects of the signal assertions does he/she discover that *Q_PROC* and *R_PROC* are in fact subactivities of *Main*; that *Q_PROC* and *R_PROC* never actually execute concurrently with *Main*.

2.2 State-transitions

Embedded systems often contain many modes, or states, of behavior. The system may transition between modes in an unstructured manner. For example, Figure 3(a) depicts a system which transitions between modes P, Q, R and S based on some conditions.

Such an unstructured jumping from one mode to another is not easily described in VHDL. VHDL is a structured programming language that supports only regular branching, i.e. there is no GOTO statement.



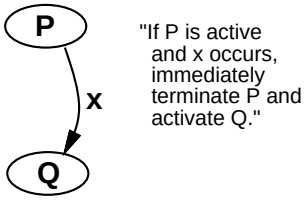
(a)

```
type STATES is (P, Q, R, S);
variable  state_var : STATES;

state_var := P;
loop
  case  state_var  is
    when  P  =>
        P();
        if (not (u or v)) then
          wait until (u or v);
        end if;
        if (u) then
          state_var  :=  Q;
        elsif (v) then
          state_var  :=  R;
        end if;
    when  Q  =>
        Q();
        state_var := P;
    when  R  =>
        R();
        if (not (w or x)) then
          wait until (w or x);
        end if;
        if (w) then
          state_var := P;
        elsif (x) then
          state_var := S;
        end if;
    when  S  =>
        S();
        state_var := R;
  end case;
end loop;
```

(b)

Figure 3: Describing state-transitions in VHDL: (a) desired functionality (b) VHDL description

To coerce the desired behavior into VHDL, one must describe a state-machine using sequential program constructs. A common technique involves the use of a state variable, case statements, and an infinite loop. Figure 3(b) demonstrates how the state-machine of the given example may be specified in VHDL. A state variable *state_var* is declared as an enumerated type with four possible states P, Q, R, and S. The state variable is initialized to the initial state P. A case statement enclosed in an infinite loop decodes the state variable and executes the appropriate branch. Each branch first calls a procedure containing the actions of the current state. For example, branch P first calls procedure P. Upon completion of the procedure, a wait statement waits until one of the arcs leaving this state can be traversed. This is followed by an if-then-else statement that sets the state variable to the next state as determined by the arc that is to be traversed. Control then loops back to the case statement where the state variable is once again decoded and the appropriate next state's branch entered, and so on.



(a)

```
P begin
      stmt1;
      wait until c1
.....................................
      stmt2;
      stmt3;
      wait on c2
.....................................
      stmt4;
end P;
```

(b)

```
P_loop : loop
      stmt1;
      wait until c1   or x;
      if (x) then
        exit P_loop;
      end if;

      stmt2;
      stmt3;
      wait on c2 , x;
      if (x) then
        exit P_loop;
      end if;

      stmt4;
      exit P_loop;
end loop P_loop;
<Q statements>
```

(c)

```
state_var := s1;
P_loop :  while  (not x)  loop
    case (state_var) is
        when s1 =>
            stmt1;
            wait until c1 or x;
            state_var := s2;
        when s2 =>
            stmt2;
            stmt3;
            wait on c2, x;
        when s3 =>
            stmt4;
    end case;
end loop P_loop;

<Q statements>
```

(d)

Figure 4: Describing immediate mode change using VHDL sequential statements: (a) desired behavior (b) sequential statements in P (c) modified P code to represent immediate mode change due to event x (d) grouping sequential statements of P to reduce number of checks for external event

2.3 Immediate mode change

Often an embedded system must react immediately to an external event, such as an interrupt or a reset. It must exit the current mode, even if it is in the middle of a computation, and enter the appropriate next mode. Figure 4(a) depicts an example in which mode P must be terminated immediately upon occurrence of event x, and mode Q must then be activated. To demonstrate that mode P can be a complex computation, we show a set of VHDL sequential statements describing P in Figure 4(b).

In VHDL, there is no construct that deactivates a process or procedure based on some event.

To ensure that a VHDL process or procedure behavior terminates when an event occurs, we must poll for the event throughout the behavior's statements. A behavior could be at a wait statement not sensitive to the event. The event therefore must be added to the sensitivity list of all the wait statements in the behavior. In addition, after every wait statement one must check if the wait statement terminated due to the occurrence of the external event, in which case control must jump to the end of the behavior. Since there is no goto statement in VHDL, the jump is achieved by enclosing the behavior's statements in a loop and using an exit statement. Figure 4(c) provides an example of how P's statements must be modified to terminate upon event x.

The modifications may clutter the behavior's statements, making the functionality difficult to comprehend. To reduce the cluttering of the VHDL code, many experienced modelers create a loop

with the event as the termination condition. The behavior's statements are then separated into groups, with each group inserted in the branch of a case statement contained in the loop. A state variable indicates the next group to be executed. The technique is illustrated in Figure 4(d), where the set of statements in Figure 4(b) have been grouped together into branches of a single case statement, resulting in fewer checks for the event x. While this technique reduces the static number of event checks in the statements to one, a person attempting to comprehend the code will likely confuse this technique with the above technique for capturing state-transitions. Also, it is often difficult to create a group of statements without a wait statement in the middle of the statements; hence, even this technique often results in incorrect functionality in which an event is missed or responded to very slowly.

2.4 Other embedded system characteristics

Two additional characteristics of embedded systems *are* supported by VHDL. The first involves *sequential algorithms*. Many modes of computation in an embedded system are easily conceptualized as a sequential algorithm, i.e. a series of sequential steps, some of which are performed conditionally and others of which are iterated. VHDL sequential statements such as the *if, case, loop* and *assignment statements* easily capture such algorithms. In addition, the wait statement is especially useful when the algorithm interacts with other behaviors via shared signals.

The second characteristic involves *activity completion*. On executing, many activities reach a point when they are complete; they do not repeat infinitely nor are they terminated by external events alone. Defined completion of an activity is important because upon such completion, we may wish to initiate another activity. VHDL procedures are considered complete upon their return, and therefore procedures easily support this concept. However, note that VHDL processes repeat infinitely so do not have a defined completion.

2.5 Limitations of other specification languages

Several other languages have been developed for behavior specification, but none support all five of the embedded-system characteristics outlined above. HardwareC [3] extends the sequential programming paradigm with concurrent processes, so it shares the same problems as VHDL. Verilog [2] extends the sequential programming paradigm with two additional constructs, fork/join and behavior disable, which support activity decomposition and immediate mode change. However Verilog does not support state-transitions. Esterel [6] makes similar extensions, but also does not support state-transitions.

Communicating Sequential Processes, or CSP [4], has fork/join constructs but no disable construct, so it does not support immediate mode change. In addition, CSP does not support state-transitions.

Statecharts [5] is based on a hierarchical finite-state machine model. Its major drawback for embedded systems is the lack of programming constructs, since without such constructs leaf activities must be described using finite-state machines, which can be tedious to write and difficult to comprehend. The Argos language [6], which attempts to improve Statecharts by making it more modular and eliminating non-determinism, also shares this drawback.

SDL (Specification and Description Language) [7] system specifications consist of hierarchical dataflow diagrams with a state machine at the leaf level. SDL supports state-based specifications inside processes and activity completion. For embedded systems, immediate mode change is very cumbersome to represent in SDL. There is no support for programming language constructs and activity decomposition is limited to a single level of processes.

In summary, because VHDL constructs do not easily support three of the five embedded system characteristics discussed above, specifying embedded systems with VHDL can be extremely tedious, time-consuming, and error-prone, and the resulting description may be difficult to comprehend. The same conclusion can be made in the case of other existing languages. Many of the advantages of the specify-and-design approach are therefore lost due to shortcomings in the language used.

3 Program-State Machines and SpecCharts

We shall now introduce a new conceptual design model, called *Program-State Machines*, that supports all the essential characteristics of embedded systems outlined in the previous section. We also describe the SpecCharts language whose constructs are intended to support that conceptual model.

Program-State Machines (PSM) are essentially a combination of the hierarchical finite-state machine and programming language paradigms. A system is specified as a hierarchy of *program-states*, where each program-state represents a mode of computation. At any given time only a subset of program-states are active, i.e. are actively carrying out their computations. A single root program-state represents the entire system and is always active.

A program-state may either be a *composite* program-state or a *leaf* program-state. A **composite** program-state may be hierarchically decomposed either into a set of *concurrent program-substates* (all program-substates are active when the program-state is active), or into a set of *sequential program-substates* (only one of the program-substates is active at a time when the program-state is active). A

sequentially decomposed program-state contains a set of transition arcs to represent the sequencing between the program-substates. There are two types of transition arcs. A *transition-on-completion arc (TOC)* is traversed when the source program-substate has completed its computation and the associated arc condition evaluates to true. A *transition-immediately arc (TI)* is traversed immediately when the arc condition becomes true, regardless of whether or not the source program-substate has completed its computation. A **leaf** program-state is at the bottom of the behavioral hierarchy and has its computation described using programming language statements.

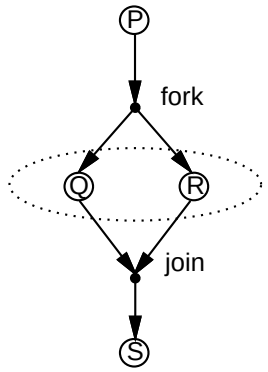
We can see that all the three embedded system characteristics outlined earlier are easily handled by the PSM model. Decomposition is directly supported in that a program-state can consist of sequential or concurrent program-substates. State-transitions are handled by the TOC and TI arcs which sequence sequential program-states. Immediate mode changes can be specified by using TI arcs.

The SpecCharts language follows the PSM model closely. The "process" and "block" VHDL constructs are now replaced by a "behavior" construct. A SpecCharts behavior corresponds directly to a program-state in the PSM model. A system is captured in SpecCharts as a set of hierarchical behaviors. A behavior can contain VHDL declarations, such as types, signals, variables, and procedures, whose scope extends to all subbehaviors. Subbehavior descriptions are nested within the parent behavior. Constructs exist to textually represent the TI and TOC arcs for sequencing between subbehaviors. Leaf behaviors are described using VHDL sequential statements such as loop, if-then-else, variable/signal assignment, procedure call, wait, etc.

For a leaf behavior, completion is defined as execution of the last sequential statement in the behavior. A sequentially decomposed behavior is defined to be complete when a transition to a special predefined "complete" subbehavior is made. A concurrently decomposed behavior is complete when all (or a selected subset) of the subbehaviors have completed.

The execution semantics of SpecCharts are similar to VHDL. The active behaviors in a SpecCharts are treated just as VHDL processes, except there is no implicit loop enclosing a behavior. For example, they execute and are suspended at wait statements, there is no delay between two successive wait statements, and all signals are updated in delta-time. Inactive behaviors are ignored; i.e. they are treated as suspended processes without a sensitivity list and with all signal drivers shut off.

If a sequential subbehavior completes but no TOC arc condition is true, there is an implicit wait until a condition becomes true. Arcs leaving a subbehavior are ordered by priority. TI arcs have priority over TOC arcs, and TI arcs highest in the hierarchy have a higher priority. For further details on SpecCharts syntax and semantics, the reader is refered to [8].

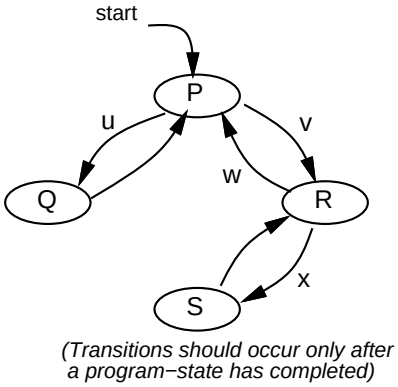


(a)

```
behavior main type sequential subbehaviors is
begin
    P :  (TOC, true, Q_R);
    Q_R  :  (TOC, true, S);
    S : ;

    behavior P .....

    behavior  Q_R  type concurrent subbehaviors is
    begin
        Q : (TOC, true, complete);
        R : (TOC, true, complete);

        behavior Q ....
```

(b)

Figure 5: Describing decomposable activities in SpecCharts: (a) desired functionality (b) SpecCharts description

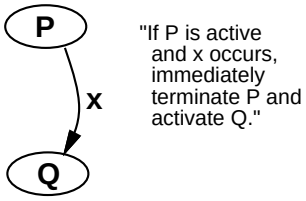


(a)

```
behavior main type sequential subbehaviors is
begin
    P  :  (TOC, u, Q), (TOC, v, R);
    Q  :  (TOC, true, P);
    R  :  (TOC, w, P), (TOC, x, S);
    S  :  (TOC, true, R);

    behavior P .....
```

(b)

Figure 6: Describing state-transitions in SpecCharts: (a) desired functionality (b) SpecCharts description



(a)

```
behavior A  type sequential subbehaviors is
begin
    P  :  (TI, x, Q);
    Q  :  ;

    behavior  P  type leaf is
    begin
        stmt1;
        wait until c1;
        stmt2;
        stmt3;
        wait on c2;
        stmt4;
    end P;

    behavior Q type ....
```

(b)

Figure 7: Describing behavior deactivation in SpecCharts: (a) desired functionality (b) SpecCharts description

Embedded System Characteristic	SpecCharts	VHDL	Verilog	Esterel	Statecharts (Argos)	CSP	SDL
Seq/Conc activity decomposition	yes	partial	yes	yes	yes	yes	partial
State transitions	yes	no	no	no	yes	no	yes
Immediate mode change	yes	no	yes	yes	yes	no	no
Programming constructs	yes	yes	yes	partial	no	yes	no
Activity completion	yes	procedures only	yes	yes	no	yes	no

Figure 8: Comparison of SpecCharts features with other languages

Figures 5, 6 and 7 show SpecChart descriptions of activity decomposition, state-transitions, and behavior deactivation for the examples of Figures 2, 3 and 4. Note the straightforward correspondence between the conceptual models and the SpecCharts descriptions, as opposed to the very indirect correspondence of the conceptual models with the VHDL descriptions.

Figure 8 summarizes the embedded system characteristics supported by SpecCharts, VHDL, Verilog, Esterel, Statecharts, Argos, CSP, and SDL.

4 An Example

For every system that has to be designed, there exists a conceptual view of the system. This conceptual view may be in the form of an English language specification or a loose understanding of the system residing in the designer's mind. We now demonstrate how the SpecCharts language can capture the conceptual view in a straightforward manner using the example of a telephone answering machine [8].

Consider the English description for the machine as given in Figure 9. Two things are evident from this description. First, we want to be able to specify distinct actions which constitute the behavior of the answering machine such as playing the outgoing announcement, recording the caller's message and hanging up. Second, we want to be able to specify some sort of ordering or sequencing between these actions. Thus, we must first play the announcement, then record the caller's message and finally hang up the phone. The corresponding SpecCharts description uses state transitions using TOC arcs to sequence between the three sequential behaviors *play_announcement*, *record_message*, and *hangup*. Note that TOC arcs are graphically shown as originating from a dot within a box representing a behavior.

When the phone rings,

- play the announcement ,
- record the caller's message ,
- hangup

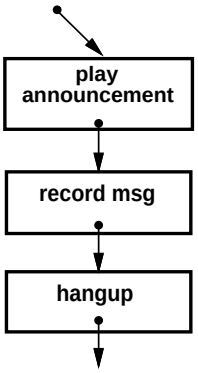


Figure 9: Answering machine: Capturing sequencing between distinct activities

We might need to group behaviors which have similar properties. For example, as shown in the English specification of Figure 10, if we are playing the announcement or recording a message, and any of the machine buttons are pressed, we should terminate the current behavior and start responding to the machine button pressed. Thus, behaviors *play_announcement*, *record_message* and *hangup* are grouped together into a single behavior, *answer*. Since SpecCharts can specify immediate mode change using a TI arc, a TI arc labeled *"machine_button_pushed"* causes *answer* to terminate and start the behavior *respond_to_machine_button*. Similarly, we may want to decompose behaviors into sub-behaviors. For example, the specification of *respond_to_machine_button* is decomposed into several distinct subbehaviors (such as *handle_play_pushed* and *handle_fwd_pushed*) which are executed depending on which machine button was pressed.

If any of the machine buttons are pushed while the machine is responding to a call playing an announcement or recording a message, immediately respond to the machine button.

If the play button was pressed, play any previously recorded messages.

If the forward button was pressed, forward the tape until the button is released.

If the rewind button is pressed, rewind until the beginning of the tape

.....

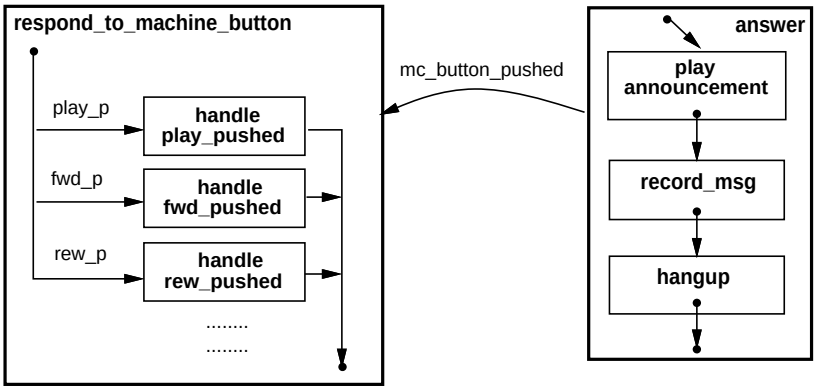


Figure 10: Answering machine: Behavior decomposition and immediate mode changes

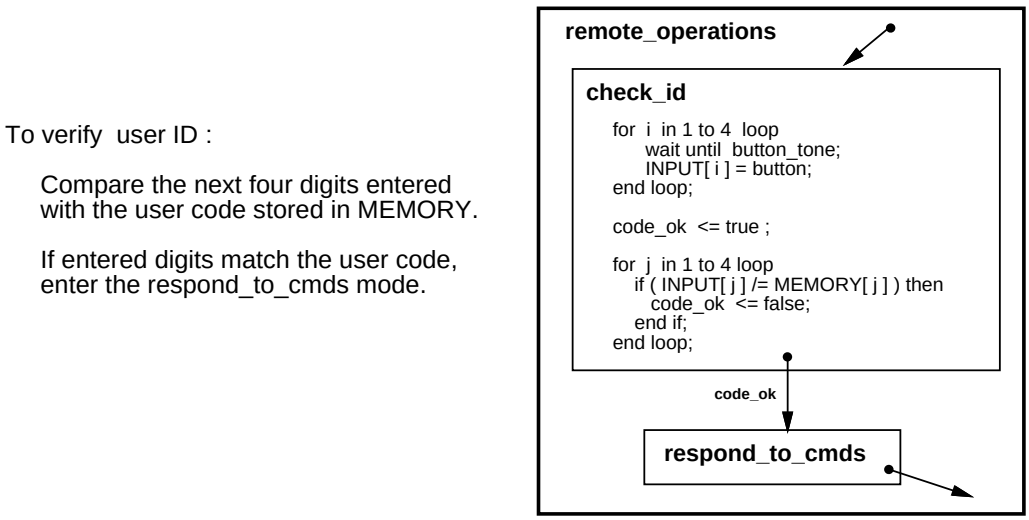


Figure 11: Answering machine: Specifying actions using programming statements

As we gradually decompose a design's behavior into hierarchical sub-behaviors, we might reach a stage where we want to specify the functionality of a behavior using programming constructs like for/while loops and case statements. SpecCharts support VHDL sequential statements to describe leaf-level behaviors. For example, in Figure 11, the description of the behavior *check_id* has been captured directly using programming constructs.

Figure 12 shows the SpecCharts specification for the telephone answering machine, excluding the leaf behaviors' program statements. There are eight levels of hierarchy in the SpecCharts resulting from successive behavior decompositions. On examining Figure 12 two things become evident. First, we can see how SpecCharts permits a very natural decomposition and consequent capture of the system's conceptual view. For example, when the phone rings the answering machine in state *answer* first plays the announcement (sequential subbehavior *play_announcement*), then records the caller's message (subbehavior *record_msg*) and finally hangs up (subbehavior *hangup*). Second, SpecCharts is able to concisely capture the immediate mode changes at various levels of the hierarchy using TI (transition immediately) arcs. Thus, the transition labeled *"system_on_p = 0"* from *system_on* to *system_off* specifies an immediate mode change for *all* the subbehaviors of *system_on*.

5 Translation Algorithm

The SpecCharts language is intended to enhance, not replace, existing VHDL design environments. Rather than developing tools that accept SpecCharts as input, we have concentrated on automated translation of SpecCharts to VHDL. While tools that accept SpecCharts as input could be developed, such tools would not have the reliability and speed of existing, widely used VHDL tools. In addition,

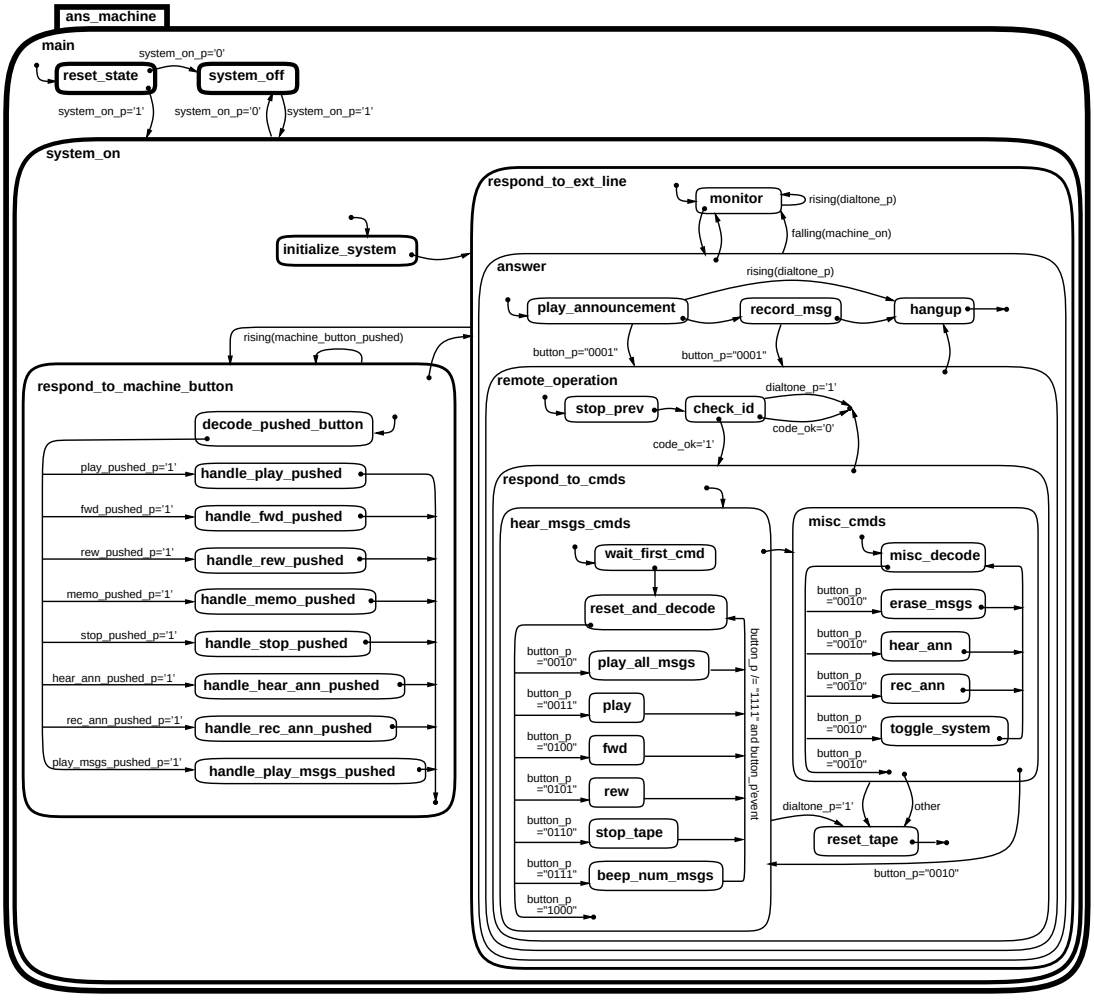


Figure 12: Answering machine: SpecCharts specification with eight levels of hierarchical behaviors

they would require designers to learn how to use them, even though they perform the same tasks as existing VHDL tools (e.g. simulation, synthesis). In this section, we present the translation scheme from SpecCharts to functionally equivalent VHDL. Although the generated VHDL will be slightly less readable than the original SpecCharts (as demonstrated in the previous sections), the scheme attempts to keep the VHDL as readable as possible and ensures that each portion of the generated VHDL is correlatable with the SpecChart.

Several translation schemes have been published for translating a variety of specification languages to VHDL [9, 10, 11, 12, 13, 14, 15]. These languages support a very different model than SpecCharts or support only a subset of the embedded-system characteristics outlined in Section 2. As a result, the translation schemes do not address the requirements for translation of SpecCharts to VHDL.

In our translation scheme, each behavior is always considered to be *inactive, active-executing*, or *active-complete*. Each behavior in the SpecCharts description is mapped to its own process with

these three distinct sections. A process representing a composite behavior, when *active-executing*, asserts and deasserts control signals in order to activate or deactivate the processes of descendant subbehaviors. Nested blocks are used to maintain the hierarchy of behavior.

Figure 13 outlines a recursive algorithm for translating a hierarchical behavior to equivalent VHDL constructs. For a more detailed algorithm, refer to [16]. The input is the root behavior from the SpecCharts file, and all output is written to the VHDL file. Procedure *CreateCompletionHandshakeStmts(B)* creates a set of statements which first indicates completion of behavior B to its parent behavior by asserting the *B_complete* signal, waits until the parent deactivates B through deassertion of the *B_active* signal, and then deasserts *B_complete*. Procedure *CreateWaitOnArcsStmts(arcs)* creates a wait statement which determines if an arc transition should occur, i.e. a subbehavior S has completed and a TOC arc condition from S is true, or a subbehavior S is active and a TI arc condition from S is true. Procedure *CreateIfStmtsForArcs(arcs)* creates an if-then-else statement with a branch for each arc. The statements in each branch deactivate the current subbehavior and activate the arc's destination subbehavior. Procedure *InsertPolling(stmts, active_sig)* inserts polling code into statements *stmts* causing a jump to the end of the statements if *active_sig* is deasserted, as discussed in Section 2.3. Procedure *CreateSignalShutoffStmts(stmts)* creates statements that shut off the drivers for all signals written in *stmts*. This is necessary because processes for inactive behaviors must be completely ignored, so they should not drive a value for a signal. See [1] for more information on signal drivers. Procedure *Append(l,m)* appends list m to the end of list l.

Starting with the topmost (root) behavior in the hierarchy, the algorithm traverses the behavioral hierarchy in depth-first order, outputing VHDL as each behavior is visited. The VHDL code for each behavior is enclosed in a block, so nested blocks maintain the hierarchy of the SpecCharts and the correct scoping of declarations.

A process is included in the block corresponding with each behavior, containing three sections:

- *Inactive*: In this section, the behavior is waiting to be activated via assertion of a control signal by the process corresponding to the parent behavior.
- *Active-executing*: Composite behaviors in this section are activating/deactivating appropriate subbehaviors via control signals, while leaf behaviors are executing their VHDL code.
- *Active-complete*: After indicating completion to the parent behavior via a control signal, the behavior is waiting to be deactivated via deassertion of a control signal by its parent.

Note that the algorithm modifies a process' statements to jump to their end, and hence reenter the inactive section, if deactivation is detected via a control signal.

```
BehaviorToVhdl ( B )

    Output block start with label B'block and declarations B.declarations

    If IsCompositeBehavior(B)
       For each S in B.subbehaviors
           Output boolean signal declarations S'active, S'complete
           BehaviorToVhdl(S) /* recursive call */

    /* Create the behavior's process statements
    */

    /* The INACTIVE section */
    stmts = NULL;
    Append(stmts, "wait until B'active;")

    /* The ACTIVE-EXECUTING section */
    If IsCompositeBehavior(B)
       Append(stmts, "loop")
       Append(stmts,      CreateWaitOnArcsStmts(B.arcs)
       Append(stmts,      CreateIfStmtsForArcs(B.arcs))
       Append(stmts, "end loop;")
    Else if IsLeafBehavior(B)
       Append(stmts, B.statements)

    /* The ACTIVE-COMPLETE section */
    Append(stmts, CreateCompletionHandshakeStmts(B))

    InsertPolling(stmts, B'active)
    Append (stmts, CreateSignalShutoffStmts(stmts))

    Output process with label B and statements stmts
    Output the end of B'block
```

Figure 13: Behavior to VHDL translation algorithm

The translation scheme presented above may result in incorrect simulations due to the creation of VHDL processes communicating via control signals. In VHDL, a signal is updated with a new value only after an infinitely small unit of time called a *delta*. The front-end language (e.g. SpecCharts) may also possess data items which are updated in infinitely-small time or even zero time; these items must be translated to VHDL signals if the translation scheme requires that they be shared by concurrent behaviors. Now suppose a particular behavior B is to be deactivated due to some event. B's parent behavior will detect the event and deassert a control signal, causing B to terminate its

actions. However, the control signal is updated only after a delta, or more if there are several levels of behavioral hierarchy, during which time a SpecCharts signal may be updated when it shouldn't have. In more general terms, the delta-time required for process control signals to be updated may interfere with the delta-time updates of signals in the SpecCharts. The simple solution to this problem is to shift the time in which SpecCharts signals are updated to a larger unit than the time in which the control signals are updated. Therefore, updates for SpecCharts signals are shifted to the femtosecond time scale; likewise other time scales are shifted to the next higher time scale if necessary. An inverse time shift of the simulation output may also be required. See [17] for details of this approach. The same problem and solution applies to many of the above referenced translation schemes.

The translation algorithm presented in this section has several advantages. First, the VHDL is functionally equivalent to the SpecCharts for all cases, not just for a subset as in previous schemes. Second, the behavioral hierarchy of the SpecCharts is maintained in the VHDL, and there is a one-to-one mapping between SpecCharts behaviors and VHDL processes, making correlation easy between the SpecCharts and VHDL (and vice-versa). Finally, the recursive aspect of the scheme makes it easy to implement.

The VHDL generated by the translation scheme may present inefficient hardware when VHDL synthesis tools are used. The primary reason is that current synthesis tools assume that one controller and one datapath are required to implement each VHDL process. Since the generated VHDL contains one process per behavior, synthesis from the VHDL may result in an excessive number of controllers and inefficiently used datapaths. The simple solution to this problem is to flatten the hierarchical behaviors into sequential statements of one leaf behavior before translation; this task is easily automated.

6 Results

We performed several experiments to demonstrate the advantages and practicality of using SpecCharts as compared to VHDL for embedded systems.

6.1 SpecCharts vs. VHDL in the specification capture experiment

The goal of this experiment was to demonstrate that using SpecCharts for specification capture reduces the specification time and the number of errors in the specification. Modelers were given an English description of an example system; half of the modelers were asked to specify it in VHDL, and the other half in SpecCharts. The specification time and errors between the two groups were then compared.

Specification information	VHDL	SpecCharts
Average specification−time (minutes)	40	16
Number of modelers	3	3
Number of correct specifications	1	3
Number of incorrect specifications	2	0
Number of incorrect specifications after second iteration	1	0

Figure 14: Capturing specifications with SpecCharts vs. VHDL

The example chosen was an aircraft traffic-alert/collision-avoidance system [18]. This system was considered since it is an excellent example of an existing embedded system and because documentation was available from an outside source (thus reducing the possibility of experimenter bias). Because of time limitations, a subset of the system's functionality was selected for specification. Three modelers specified this subset in VHDL, and three in SpecCharts.

The results of the specification capture by the two groups of modelers are summarized in Figure 14. The VHDL modelers required an average of 2.5 times longer to capture the specification of the system. In addition, two of the VHDL specifications possessed a major control error. Any implementation derived from those erroneous specifications would respond unacceptably slowly if a pilot attempted to override the collision-avoidance system in an emergency situation. This was pointed out to the VHDL modelers who then attempted to fix their specifications. Only one modeler was able to remedy the problem in the allotted time.

6.2 SpecCharts vs. VHDL in the comprehension experiment

The goal of this experiment was to show that a SpecCharts specification is easier to comprehend than a corresponding VHDL specification. One group of modelers was given the VHDL specification of a system, another group given the SpecCharts specification, and each group was asked several questions. The number of correct answers and the time required by each group to understand the system functionality were then compared.

The example chosen was a portion of an Ethernet coprocessor [19], for which an HDL specification was available from an outside source. We manually created a functionally equivalent SpecCharts specification. Three modelers were given the VHDL description, and three were given the SpecCharts specification. Each person was asked to measure the time to "understand the general behavior of the system". Also, fourteen questions were asked about the system, such as "What happens when the Enable signal goes low?", "How many preamble bytes are transmitted for any given data?", and

"What is the purpose of variable v?".

The modelers who were given the VHDL specification required three times longer to understand the general behavior. In addition, they averaged two incorrect answers to the questions, whereas the persons given SpecCharts description answered all questions correctly.

6.3 Specification quantification experiment

To quantify the several differences between a SpecChart, VHDL and Statecharts specification, we specified a single system in all three languages, and then measured several characteristics of each specification.

The example chosen was that of a telephone answering machine. An English specification was described using a PSM model, which was then converted to SpecCharts, VHDL, and Statecharts. Two VHDL versions were created. One maintained the hierarchy by using nested blocks and processes communicating via control signals, as discussed in Section 5. The other flattened the hierarchy into a single state machine (where each state represented a behavior) described as a single process.

Figure 15 shows the results of this experiment. The flat VHDL description has fewer program-states since only the leaf program-states need to be specified, but this reduction is at the expense of almost four times more arcs. The large increase in the number of transition arcs occurs because each arc high in the hierarchy must be replicated to point from *each* of its descendant leaf states. In addition, immediate-transitions require polling to be performed throughout the code, as described earlier. Finally, arcs must be represented using sequential statements. These three factors result in over four times more words in the flat VHDL description than in the SpecCharts. The hierarchical VHDL does not require increasing the number of program-states or arcs, but does require addition of 84 control signals, two per program-state, for interprocess control. The writing and reading of these signals, along with the polling required for immediate transitions and the representation of arcs with sequential statements, results in almost four times more words than in a SpecChart. Clearly, the higher the number of lines and words in a specification, the greater is the specification time, comprehension time, and occurrence of errors. With regards to leaf program-states, both VHDL versions require about four times more statements per leaf than in SpecCharts. The increase is significant because it impairs the readability of the leaf program-state, hence defeating the leaf's purpose of modularizing the functionality into easy-to-comprehend portions.

The consequence of the lack of programming constructs in Statecharts can be clearly seen in this example. Because the leaf behaviors must also be described using states and arcs, the Statecharts

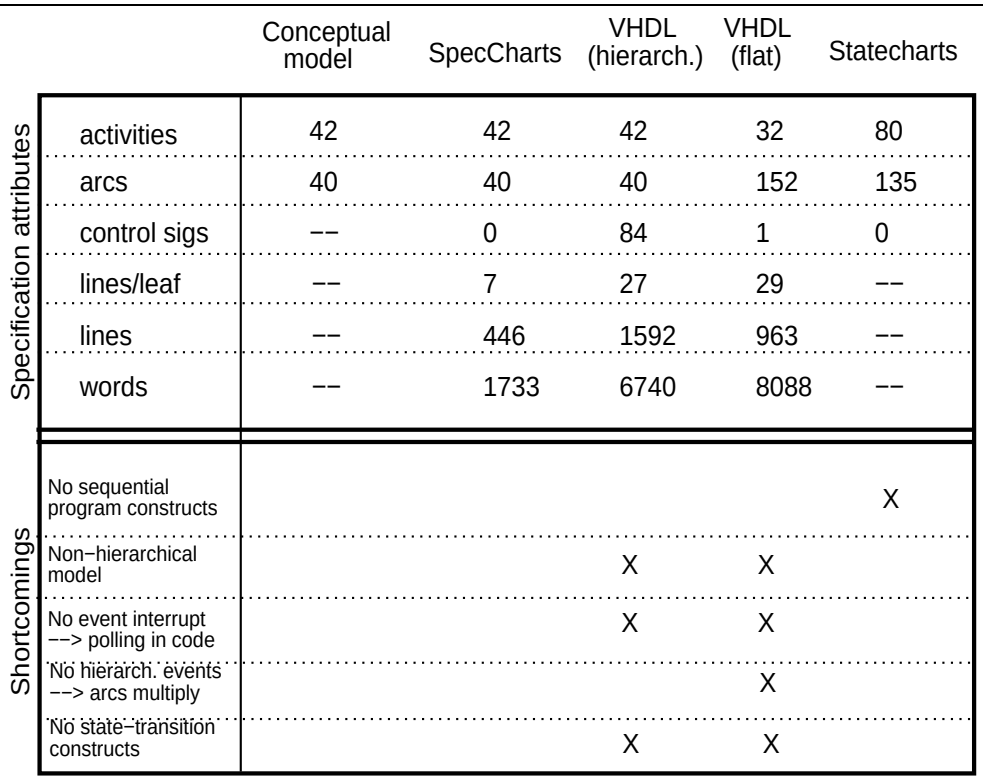


Figure 15: Specification comparisons of SpecCharts, VHDL and Statecharts

contains almost double the number of states and over thrice as many arcs as the SpecCharts. Using states and arcs to describe the leaf behaviors (such as a loop) can be quite tedious and unnatural, as compared to using sequential program constructs. Note that since Statecharts are only defined graphically, lines and words are undefined.

6.4 SpecCharts vs. manual design experiment

The goal of this experiment was to demonstrate that using SpecCharts does not sacrifice design quality as compared with manual design. We compared number of transistors of a manual design with that resulting from using a specify-and-design approach with SpecCharts as the input language.

The example chosen was the telephone answering machine. An English specification was given to two designers. One designer took a common manual approach to design. The datapath was first created by hand and the controlling state machine described using Berkeley's KISS format, on which the KISS synthesis tool was applied to generate the controller logic. The other designer specified the system with SpecCharts, flattened the hierarchy automatically and translated the flattened SpecCharts to VHDL automatically. From the VHDL the designer then created a datapath and controlling state machine manually (but automatable with existing commercial synthesis tools), and applied the KISS

Design attribute	Designed from English	Designed from SpecCharts
control transistors	3130	2630
datapath transistors	2277	2251
total transistors	5407	4881
total pins	38	38

Figure 16: Design quality from SpecCharts vs. English specifications

synthesis tool to generate controller logic.

The results of creating the design using the two approaches is shown in Figure 16. Design time for each person was roughly the same, about 30 man-hours. The important thing to note is that the number of transistors in the final design obtained by the SpecCharts designer is not greater than that obtained by the manual designer. In this case, the number is actually fewer, resulting from fewer control states. The manual designer had to keep the state machine readable since it was the functional specification of the system and hence had to be modified as functional errors and omissions were detected. Keeping the state machine readable prevented him from sharing many states. Conversely, SpecCharts was the functional specification for the other designer, so readability of the state machine was not an issue. The designer used an algorithm to convert SpecCharts to a state machine that resulted in many shared states.

7 Status and Future Work

A parser and VHDL translator have been implemented for SpecCharts. The implementation consists of 16,000 lines of C code. The implementation includes several automated SpecCharts transformations, such as flattening the hierarchy and procedure inlining, as well as the time-shift transformation discussed in Section 5 . Figure 17 summarizes translation time and resulting VHDL lines of code for numerous examples; descriptions of all of the examples can be found in [8]. The variation in the ratio of SpecCharts lines to VHDL lines across examples results from variations in the number of levels of hierarchy and the number of arcs in the examples. The greater the number of levels of hierarchy or number of arcs that exist, the larger the generated VHDL will be relative to the SpecCharts.

A concern often expressed is that simulation efficiency will be very poor for the generated VHDL, which would imply that a SpecCharts simulator should be written. To test the simulation efficiency, we obtained two VHDL models of the DRACO peripheral interface example manually specified by an external source. The handwritten VHDL models contained 388 and 531 lines respectively, while the VHDL model generated from SpecCharts contained 432 lines. We simulated all three models

Example	SpecChart lines	Generated VHDL lines	Translation time
am2910	359	408	3.6
ans	526	1667	1.4
cc	114	615	3.7
draco	253	432	3.5
ether	1003	1120	27.1
mwt	708	1785	9.7
uart	262	364	1.3

Figure 17: Translation results

using the industry test vectors, consisting of 23,000 lines of VHDL code; simulation times (on Zycad's VHDL simulator version 1.0 running on a Sparc2) were 220 seconds, 250 seconds, and 550 seconds respectively. While simulation of the generated VHDL is slower, it is not by an order of magnitude which might then require a SpecCharts simulator.

SpecCharts is currently being used as the input language (along with VHDL) to a set of system-design tools which we are implementing, including a partitioner [20], estimators [21], and an interface synthesis tool. It has also been used in an industry design of a controller in which the behavior was specified in SpecCharts, which was then manually translated to C and then compiled for execution on a microcontroller. The designer estimated a 20% reduction in design time over writing C initially, which is the standard approach, and estimates a 40% reduction if an automated translator from SpecCharts to C is implemented.

We plan to upgrade the current SpecCharts tool set as follows. First, since current synthesis tools place restrictions on acceptable VHDL input, we plan to write translators that output VHDL amenable to various synthesis tools. Second, because portions of a system are often implemented as a program running on a processor or microcontroller, we plan to implement a translator to the C language so that existing C compilers can be used. Finally, since current synthesis tools and compilers do not modify the process-level parallelism of a program, we plan to include automated transformations to convert sequential behaviors to concurrent ones, and vice-versa.

Other future work may include a graphical capture tool for SpecCharts. While text is adequate, there may be some cases where graphics are preferred. In addition, a graphical simulation tool which highlights active behaviors might prove beneficial.

8 Conclusions

Increased system complexity requires new solutions to system design. Specifying systems on too low of an abstraction level increases the number of functional errors and incompleteness, increases the

number of design iterations required, slows down integration of system modules, delays introduction of a product on the market, and increases the cost of product support and enhancement over its lifetime. Thus there is a need for specification and modeling on a high-level. We have introduced the SpecCharts language based on a new conceptual model (Program-State Machines) that satisfies embedded system needs, and have described a simple translation scheme to VHDL. The language allows designers to easily capture and comprehend embedded system functionality, as was demonstrated by several experiments. Our approach also fits well into present methodologies.

9 Acknowledgements

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #92-DJ-146). We are grateful for their support. We would also like to thank Bob Larsen and Rockwell International for their help and suggestions.

References

[1] *IEEE Standard VHDL Language Reference Manual*, 1988.

[2] D.E. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[3] D. Ku and G. Micheli, "HardwareC - A Language for Hardware Design." Stanford University, Technical ReportCSL-TR-90-419, 1988.

[4] C. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, August 1978.

[5] D. Harel, "Statecharts : A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, 1987.

[6] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[7] D. H. F. Belina and A. Sarma, *SDL with Applications from Protocol Specifications*. Prentice Hall, 1991.

[8] S. Narayan, F. Vahid, and D. Gajski, "Modeling with SpecCharts." UC Irvine, Dept. of ICS, Technical Report 90-20,1990.

[9] N. Dutt, J. Cho, and T. Hadley, "A User Interface for VHDL Behavioral Modeling," in *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, 1991.

[10] O. Pulkkinen and K. Kronlof, "Integration of SDL and VHDL for High-Level Digital Design," in *Proceedings of the European Design Automation Conference*, 1992.

[11] B. Lutter, W. Glunz, and F. Rammig, "Using VHDL for Simulation of SDL Specifications," in *Proceedings of the European Design Automation Conference*, 1992.

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

[13] T. Tikanen, T. Leppanen, and J. Kivela, "Structured Analysis and VHDL in Embedded ASIC Design and Verification," in *Proceedings of the European Conference on Design Automation*, 1990.

[14] A. Arsenault, J. Wong, and M. Cohen, "VHDL Transition from System to Detailed Design," in *VHDL Users' Group*, April 1990.

[15] A. Jerraya, P. Paulin, and D. Agnew, "Facilities for Controllers Modeling and Synthesis in VHDL," in *VHDL Users' Group*, April 1991.

[16] S. Narayan, F. Vahid, and D. Gajski, "Translating System Specifications to VHDL," in *Proceedings of the European Conference on Design Automation*, 1991.

[17] F. Vahid and D. Gajski, "Obtaining Functionally Equivalent Simulations Using VHDL and a Time-shift Transformation," in *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[18] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese, "Requirements Specification for Process-Control Systems." UC Irvine, Dept. of ICS, Technical Report 92-106,1992.

[19] R. Gupta and G. D. Micheli, "System-level Synthesis using Re-programmable Components," in *Proceedings of the European Conference on Design Automation*, 1992.

[20] F. Vahid and D. Gajski, "Specification Partitioning for System Design," in *Proceedings of the Design Automation Conference*, 1992.

[21] S. Narayan and D. Gajski, "System Clock Estimation based on Clock Slack Minimization," in *Proceedings of the European Design Automation Conference*, 1992.