SpecCharts: A VHDL Front-End for Embedded Systems

Frank Vahid, *Member, IEEE,* Sanjiv Narayan, *Member, IEEE,* and Daniel D. Gajski, *Fellow, IEEE*

Abstract— **VHDL and other hardware description languages are commonly used as specification languages during system design. However, the underlying model of those languages does not directly support the specification of embedded systems, making the task of specifying such systems tedious and error-prone. We introduce a new conceptual model, called Program-State Machines (PSM), that caters to embedded systems. We describe SpecCharts, a VHDL extension that supports capture of the PSM model. The extensions we describe can also be applied to other languages. SpecCharts can be easily incorporated into a VHDL design environment using automatic translation to VHDL. We highlight several experiments that demonstrate the advantages of significantly reduced specification time, fewer errors, and improved specification readability.**

I. Introduction

LANGUAGE based methodologies are gaining popularity in embedded hardware and software design. In such approaches, one first precisely specifies a system's desired functionality using a program-like simulation language, from which one then derives an implementation. The functional specification is free of any implementation decisions, such as the division of the system among hardware or software modules, the scheduling of concurrent processes into a single execution thread, or the mapping of operations to register-transfer components. Such approaches are replacing implementation-based approaches, in which one first derives an implementation (such as a gate-level netlist or C code) from an informal specification, and then one simulates or executes that implementation to verify its functionality.

Language-based approaches provide many advantages through a design's lifecycle. First, by creating a test-bench early in the design process and simulating the functional specification, one can detect and easily correct functional errors. Such corrections would be extremely difficult to make later in the design process. Second, by precisely defining functionality, one can expect fewer difficulties when integrating the system with other, concurrently-designed systems. Third, by using a machine readable language, one can apply automated estimation and synthesis tools to reduce the design time or to rapidly evaluate alternative implementations. Finally, by writing a functional specification void of implementation details, one can easily redesign the system for another application, without reverse engineering from an existing low-level design implementation.

Many languages have been proposed for functional specification, including VHDL [1], Verilog [2], CSP [3], and Statecharts [4]. The best language to use for a particular system depends largely on how well the language supports capture of a good conceptual model for that system. A well-known example of this relationship between a language and a model is seen with C++ and the object-oriented model: C++ supports the capture of the object-oriented model, which in turn has proven useful for many large software applications. A good language should also be able to represent the system through several stages of refinement, such that implementation details can be successively captured in the language. Such implementation details may include communication protocols or a partitioning of the design among hardware and software components.

Many systems are of a type currently referred to as **embedded systems**. Although there is no widely-accepted definition of an embedded system, we note that such a system is typically designed to perform one particular function as part of a larger system. A large part of an embedded system's functionality consists of continually responding to external events and conditions in real time. Embedded systems often consist of software running on a standard processor, custom hardware implemented on an ASIC, or a combination thereof. Examples include automobile cruise controllers, fuel-injection systems, aircraft autopilots, network switches, video focusing units, Ethernet coprocessors, aircraft collision avoidance systems, interactive television processors, telephone answering machines, volume-measuring medical instruments, microwave-transmitter controllers, fuzzy-logic controllers, image-processing systems, MPEG decoders, bus controllers, and robotic arm-tracking algorithms. After examining instances of many of the above examples (the last eleven, to be precise), we determined five characteristics common to embedded systems: **sequential and concurrent behavior decomposition, state transitions, exceptions, sequential algorithms,** and **behavior completion**.

Unfortunately, no existing language supports all five characteristics. We say a language "supports" a characteristic if there is a simple, direct mapping of the characteristic to a language construct. A lack of support does not mean that

Manuscript received April 22, 1994; revised December 2, 1994. This work was supported by the National Science Foundation Grant MIP-8922851 and the Semiconductor Research Corporation Grant 92-DJ-146. This paper was recommended by Associate Editor M. McFarland.

F. Vahid is with the Department of Computer Science, University of California, Riverside, CA 92521 USA.

S. Narayan is with the R&D group at Viewlogic Systems Inc., Marlboro, MA 01752 USA.

D. D. Gajski is with the Department of Information and Computer Science, University of California, Irvine, CA 92717 USA.

IEEE Log Number 9411258.

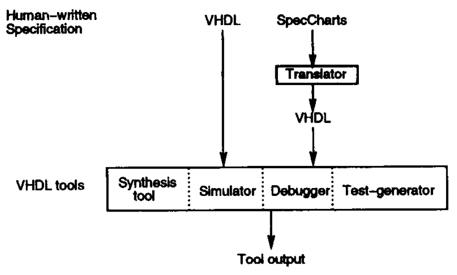
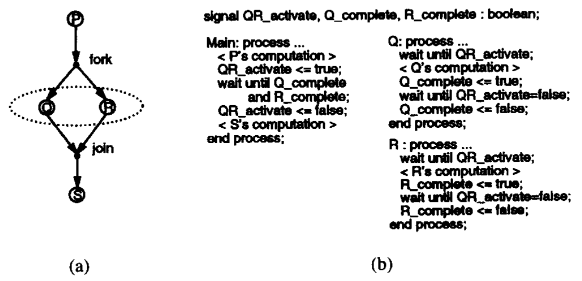


Fig. 1. SpecCharts as a front-end language in a VHDL design environment.



```
signal QR_activate, Q_complete, R_complete : boolean;

Main: process ...
  < P's computation >
  QR_activate <= true;
  wait until Q_complete
        and R_complete;
  QR_activate <= false;
  < S's computation >
end process;

Q: process ...
  wait until QR_activate;
  < Q's computation >
  Q_complete <= true;
  wait until QR_activate=false;
  Q_complete <= false;
end process;

R : process ...
  wait until QR_activate;
  < R's computation >
  R_complete <= true;
  wait until QR_activate=false;
  R_complete <= false;
end process;
```

Fig. 2. Describing concurrent decomposition in VHDL. (a) Desired functionality. (b) VHDL description.

the characteristic cannot be described, but instead means that the characteristic cannot be described *easily*. For example, consider the program characteristic of recursion. We say that the C language supports recursion, while assembly language does not, even though C is mapped to assembly language during compilation and hence, recursion can be described in either language.

To overcome the lack of support of those characteristics, we first developed a new conceptual model, called Program-State Machines (PSM). The PSM model elegantly combines the hierarchical/concurrent Finite-State Machine (FSM) model with the programming language paradigm, to easily support all embedded system characteristics. We then defined the SpecCharts language to directly capture the PSM model. Because we defined SpecCharts as an extension of the widely-used VHDL language, one can easily integrate the language into an existing VHDL environment (other languages, like Verilog or C, can be similarly extended to support the PSM model). It is easy to integrate because designers who are familiar with VHDL can learn SpecCharts in just a few hours, and because we can easily translate SpecCharts to VHDL so that existing simulation and synthesis tools can be applied. For example, Fig. 1 demonstrates that a system's SpecCharts specification can be translated to VHDL, after which we could apply a VHDL simulator to see the system's input/output relationships, we could apply a VHDL debugger to step through simulation cycles and examine intermediate signal and variable values, we could apply a VHDL synthesis tool to generate a custom hardware implementation, or we could apply a VHDL to C translator to generate an embedded software implementation.

This paper is organized as follows. In Section II, we demonstrate the difficulty of specifying embedded system characteristics using VHDL. In Section III, we introduce the new PSM model, which easily represents embedded system characteristics. We then introduce the SpecCharts language and we show how its constructs easily capture a system described as a PSM model. In Section IV, we provide an example in SpecCharts. In Section V, we describe an algorithm for translating SpecCharts to VHDL. In Section VI, we highlight several experiments that demonstrate the advantages of reduced specification time and fewer errors achieved when using SpecCharts for embedded system specification. In Section VII, we discuss the status of the language and supporting tools, and plans for future work. In Section VIII, we provide conclusions.

II. Existing VHDL Limitations for Embedded Systems

In this section, we will describe the five characteristics of embedded systems. We will also demonstrate the difficulties of capturing three of those five characteristics in VHDL.

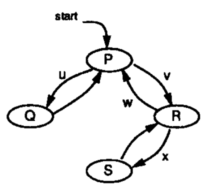
A. Sequential and Concurrent Behavior Decomposition

To cope with the complexity of system functionality, we usually need to hierarchically decompose functionality into simpler pieces, or behaviors. Such behaviors may be either sequential or concurrent to one another, and may themselves be further decomposed. For example, Fig. 2(a) illustrates the decomposition of system functionality into four behaviors, P, Q, R, and S, each of which may be some arbitrarily complex computation. First P is executed, followed by a concurrent execution of Q and R, followed by S.

VHDL supports sequential decomposition using procedures, but it only partially supports concurrent decomposition. In particular, VHDL supports concurrent decomposition of a system's top-level functionality using processes, but does not support concurrent decomposition of a process or procedure, i.e., forking is not directly supported.

We can coerce description of a fork in VHDL by introducing extra processes and extra signals, as shown in Fig. 2(b). We create a signal for the fork *QR_activate*, and we assert the signal when control reaches the point in the VHDL sequential statements where the fork should occur. We create top-level processes, Q and R, for each of the fork's concurrent behaviors; each fork process executes only when the fork signal is asserted. To implement a join, we add additional control signals, *Q_complete* and *R_complete*, and we assert them at the end of each fork process. Execution of the functionality proceeds as follows. The process *Main* first calls procedure P, which returns when complete. *Main* then asserts the signal *QR_activate*, which causes processes Q and R to execute. When Q and R have completed their computations, they assert the signals *Q_complete* and *R_complete*, after which *Main* proceeds to call procedure S.

It should be noted that a person reading the VHDL description will initially be misled into believing that the system is composed of three concurrent behaviors, *Main*, Q, and R. Only after mentally executing the code and tracing the effects of the signal assertions does that person discover that Q and R are in fact forked subbehaviors of *Main*, so that Q and R never actually execute concurrently with procedure P or S in *Main*.

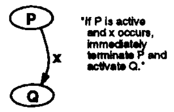


(a)

```
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)

Fig. 3. Describing state-transitions in VHDL. (a) Desired functionality. (b) VHDL description.


P
Q
x
"If P is active and x occurs, immediately terminate P and activate Q."

(a)

```
stmt1;
wait until c1;
stmt2;
stmt3;
wait on c2;
stmt4;
```

(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;
         state_var := s3;
      when s3 =>
         stmt4;
         exit P_loop;
   end case;
end loop P_loop;

<Q statements>
```

(d)

Fig. 4. Describing exceptions in VHDL. (a) Desired functionality. (b) VHDL sequential statements for P. (c) P modified for immediate reaction to event x. (d) P rewritten as a state-machine to reduce polling clutter.

B. State Transitions

Embedded systems often contain many modes of functionality, referred to as states. The system may transition between states in an unstructured manner. For example, Fig. 3(a) illustrates a system that transitions between states P, Q, R, and S based on some conditions.

VHDL does not support state transitions. In fact, it does not support any unstructured jumping, since it is a structured programming language that does not include explicit *goto* statements.

One way to coerce the description of state transitions in VHDL by using sequential program constructs is shown in Fig. 3(b). We declare a state variable *state_var* as an enumerated type with four possible values: P, Q, R, and S. We create a case statement that decodes the state variable and executes the appropriate branch. Each branch executes the appropriate behavior, and then sets the state variable to the appropriate next state based on the transition arc conditions (if no arc condition evaluates to true, we wait until at least one arc condition becomes true). The case statement is enclosed in an infinite loop, so the statements repeat the following activity forever: decode the current state, execute the current state's behavior, set the current state to the next state based on the arc conditions, and again decode the current state.

Once again, note that a person reading the VHDL description will have to mentally execute the code to discern the state-machine from the VHDL.

C. Exceptions

An embedded system often must react immediately to an external event, such as an interrupt or a reset; such an event is often called an exception. The exception requires termination of the current behavior, even if the behavior is in the middle of a computation, and requires execution of the appropriate next behavior. For example, Fig. 4(a) illustrates a behavior P that must be terminated immediately upon occurrence of event x, after which behavior Q must be activated. To demonstrate that behavior P can be a complex computation, we show a set of VHDL sequential statements describing P in Fig. 4(b).

VHDL does not possess a construct to immediately deactivate a process or procedure upon the occurrence of an event.

We can coerce exception handling into VHDL by polling for the exception throughout the behavior's sequential statements, and then jumping to the end of the statements if the exception is detected through such polling. To accomplish such polling, we modify the sensitivity list of all the wait statements in the behavior. We follow each wait statement by a check to see if the wait terminated due to the exception, in which case we 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. For example, Fig. 4(c) illustrates how P's statements can be modified to terminate upon event x.

Some expert VHDL writers reduce the clutter resulting from polling by first dividing the behavior's statements into groups, and treating each group as a "state." Hence, the behavior is described as an FSM, as in the previous section, with the enclosing loop modified to terminate on occurence of the exception, as shown in Fig. 4(d).

Note that, in both approaches, a single exception requires a major modification of the VHDL code. The modification is greatly increased when there are multiple exceptions, possibly with different priorities.

D. Sequential Algorithms

Many computations in an embedded system are easily conceptualized as a sequential algorithm, i.e., as a sequence of steps, some of which are performed conditionally and others which are iterated. Sequential program statements, such as *if, case, loop, procedure,* and *assignment statements*, easily capture such algorithms.

VHDL fully supports sequential algorithms through its extensive set of sequential statements.

E. Behavior Completion

A behavior may execute to a point where its computation is complete, rather than repeating infinitely or being terminated by an external event. After a behavior completes, we may then want to execute another behavior. For example, one behavior may apply some function to each element of an array, after which another behavior should transmit this array over an external port. In this example, the transition from the first behavior to the second was dependent solely on the completion of the first behavior, and not on some external event.

VHDL partially supports behavior completion. VHDL procedures are complete when a return statement is reached. However, VHDL processes do not complete. We can coerce description of process completion in VHDL by adding a new signal, which is asserted when the end of the process is reached, and can be monitored by other processes.

F. Limitations of Other Specification Languages

Several other languages have been developed for functional specification, but none support all five embedded system characteristics. Verilog [2] extends the sequential programming paradigm with two additional constructs, fork/join and behavior disable, which in turn support forking and exceptions. However, Verilog does not support state transitions. Esterel [5] makes similar extensions, but also does not support state transitions. Communicating Sequential Processes, or CSP [3], extends the programming paradigm with fork/join constructs, but it does not support state transitions or exceptions. Statecharts [4] permits capture of a hierarchical/concurrent FSM model. However, it does not support sequential algorithms since it does not support general programming constructs. Argos [5] shares these features and limitations of Statecharts. SDL (Specification and Description Language) [6] permits description of hierarchical dataflow diagrams with an FSM at the leaf level. It supports state transitions inside processes and behavior completion. However, it does not support exceptions or sequential algorithms, and it does not support forking.

In summary, VHDL constructs do not easily support three of the five embedded system characteristics discussed above, so we must coerce those characteristics into existing language constructs, as was shown above. Such coercion can mean that specifying embedded systems with VHDL may 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.

III. PROGRAM-STATE MACHINES AND SPECCHARTS

In this section, we introduce a new conceptual model, called Program-State Machines, that supports embedded systems characteristics. We also describe the SpecCharts language, whose constructs support PSM capture.

A. PSM

The Program-State Machine (PSM) model is a combination of the hierarchical/concurrent FSM model and the programming language model. Briefly, the PSM model is an FSM model where each leaf state may be described as an arbitrarily complex program. As such, PSM subsumes the FSM model and the programming language model. In other words, we can describe any FSM as a PSM, simply by restricting the leaf state programs to trivial assignments. Alternatively, we can describe any program as a PSM, simply by creating only one state that contains the program. Most importantly, we can describe an infinite number of combinations of FSM's and programs. Therefore, the PSM model can be thought of as the next step in the evolution of computation models, combining the model traditionally used to describe hardware with the model traditionally used for describing software, addressing the fact that the border between hardware and software is rapidly becoming blurred.

We define the PSM model more precisely as follows. A PSM is a pair $\langle I, P_{\text{root}} \rangle$, where I is the set of input/output ports, and P_{root} is a program-state at which the hierarchy of program-states comprising a PSM is rooted.

A program-state P is a three-tuple $\langle$*decls, status, comp*$\rangle$. *Decls* consists of any program declarations, such as variables and procedures, whose scope is P and any descendants. *Status* is the current status of the program-state, where *status* $\in$ $\{$*inactive, executing, complete*$\}$. The third and most important part of a program-state is its computation *comp*, where *comp* $\in$ $\{$*leaf, concurrent, sequential*$\}$. Specifically, there are three types of program-states.

1) **Leaf** program-state: A leaf program-state's computation is described as an arbitrarily-complex sequence of programming language statements.
2) **Concurrently**-composed program-state: A concurrently-composed program-state's computation is simply described as a set of program-substates $\text{PSS}_{\text{conc}} = \{P_1, P_2, \cdots\}$, where all the program-substates execute concurrently with one another.
3) **Sequentially**-composed program-state: A sequentially-composed program-state's computation is also described as a set of program-substates $\text{PSS}_{\text{seq}} = \{P_1, P_2, \cdots\}$, along with a list of transition arcs, $T = \{t_1, t_2, \cdots\}$, which determines the single program-substate that should be executing at any given time. One of the program-substates in PSS_{seq} is denoted as an initial state (P_{init}), to which control is transferred when the parent state is first activated. A transition arc t_i is a four-tuple $\langle$*src, cond, dest, type*$\rangle$. The source program-state, or origin, of the transition arc is represented by *src*, where *src* $\in \text{PSS}_{\text{seq}}$. The condition under which the transition is effected is represented as a boolean expression, *cond*. The destination program-state to which control is transferred by the transition is denoted as dest, where dest $\in \{\text{PSS}_{\text{seq}} \cup$ *complete*$\}$. *Complete* refers to a special program-state that is akin to a final state in a traditional finite-state machine. A transition arc's *type* is either TOC (transition-on-completion) or

TI (transition-immediately). A TOC arc is traversed if and only if the source program-state has finished its computation and the arc condition is true. A TI arc is traversed whenever the arc condition evaluates to true, regardless of whether the source program-state has actually finished its computations.

We now informally define the execution semantics of a PSM model. We initially activate the root program-state by setting $P_{\text{root}}.\mathit{status}$ to *executing*. Whenever a program-state is first activated, we do one of the following.

- If the program-state is concurrently-composed, then we activate all its program-substates, i.e., we set $P_i.\mathit{status} = \mathit{executing}, \forall P_i \in \text{PSS}_{\text{conc}}$.
- If the program-state is sequentially-composed, then we activate its first program-substate, i.e., we set $(P_{\text{init}}.\mathit{status} = \mathit{executing})\ P_{\text{init}} \in \text{PSS}_{\text{seq}}$.
- If the program-state is a leaf, then we begin executing its statements.

Execution proceeds for all executing program-states, causing status changes of various program-states. There are two causes of status changes: *completions* and *exceptions*.

Completion means that a program-state P has finished its computation. A leaf program-state completes when its last statement has been executed. A sequentially-composed program-state completes when an arc transition to the special *complete* program-substate occurs. A concurrently-composed program-state completes when all its program-substates are complete. In all three cases, completion for a program-state P means that $P.\mathit{status}$ changes from *executing* to *complete*. The change of a program-state's status to *complete* may in turn cause a TOC arc transition in its parent program-state. A TOC arc is traversed if and only if $\mathit{src.status} = \mathit{complete}$ and $\mathit{cond} = \mathit{true}$, and the transition is achieved by setting $\mathit{src.status} = \mathit{inactive}$ and $\mathit{dest.status} = \mathit{executing}$.

Exceptions occur whenever a TI arc condition becomes true and the arc's source program-state is active, i.e., $\mathit{src.status} = \mathit{executing} \mid \mathit{complete}$ and $\mathit{cond} = \mathit{true}$. As with TOC arcs, the TI transition is achieved by setting $\mathit{src.status} = \mathit{inactive}$ and $\mathit{dest.status} = \mathit{executing}$. The source program-state computation terminates immediately when the source is deactivated in this manner.

The PSM model is made fully deterministic by imposing the following semantics. If more than one TOC arc pointing from a particular program-substate could be traversed at a given time, then the one closest to the front of the arc list is traversed; likewise for TI arcs. TI arcs have priority over TOC arcs, and TI arcs higher in the hierarchy have priority over TI arcs at lower levels.

Note that the PSM model is modular. In particular, each level of hierarchy can be developed without any knowledge of higher or lower levels. Such modularity is made possible due to the consistent definition of completion for any type of program-state, and to the prohibition of arcs that cross hierarchical levels (neither of which is true for Statecharts).

We see that the PSM model supports description of all five of the embedded system characteristics discussed earlier. Behavior decomposition, including forking, is directly supported since any program-state can consist of sequential or concurrent program-substates. State transitions are supported by the TOC and TI arcs. Exceptions are easily supported by the TI arcs. Sequential program algorithms are supported by the sequential statements in a leaf program-state. Behavior completion is supported by the TOC arcs and by the definition of program-state completion for all three types of program-states.

B. SpecCharts

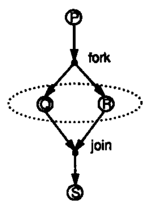
The textual-version of the SpecCharts language is almost identical to VHDL, with a few additional constructs. As in VHDL, an entity specifies the system's interface, an architecture specifies the system's contents, and "use" clauses can incorporate VHDL packages. However, we replace the "process" and "block" VHDL constructs by a "behavior" construct, which corresponds directly to a PSM program-state. A **behavior** has the following syntax.

behavior	::= **behavior** identifier **type**
	behavior_type **is**
	[declarations]
	begin
	computation
	end behavior
	[*behavior*_identifier];
behavior_type	::= **leaf** | **sequential subbehaviors** |
	concurrent subbehaviors
*leaf*_computation	::= statements
*concurrent*_computation	::= behaviors
*sequential*_computation	::= subbehavior_definitions
	behaviors

The three *behavior_type* possibilities correspond to the three types of PSM program-states. The *declarations* in a behavior is an optional list of VHDL declarations, such as types, subtypes, procedures, functions, variables, signals, etc., whose scope covers the start to end of this behavior (including descendant subbehaviors). If the *behavior_type* is *leaf*, then the behavior's computation will consist of VHDL sequential statements, such as loops, assignments, procedure calls, wait statements, etc. If the *behavior_type* is *concurrent subbehaviors*, then the behavior's computation will consist of a set of concurrent behaviors, where each behavior in *behaviors* is (recursively) defined as earlier. If the *behavior_type* is *sequential subbehaviors*, then the behavior's computation will consist of the *subbehavior_definitions* and *behaviors*, where:

subbehavior_definitions	::= {subbehavior_definition}
subbehavior_definition	::= *subbehavior*_identifier [:arcs];
arcs	::= {arc}
arc	::= (arctype, condition,
	*next*_identifier)
arctype	::= **TOC** | **TI**
behaviors	::= {behavior}

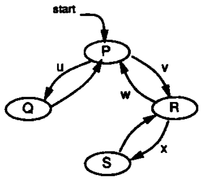
The *subbehavior_definitions* simply lists the subbehaviors of the current behavior and any transitions between those subbehaviors. Each subbehavior definition has its own arcs, each of which has that subbehavior as its source. Each arc has a type (either TOC or TI), a condition, and the identifier of the next subbehavior to which the arc points (or *complete*). The



```
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 ;
        R ;

        behavior Q ....
```

(a) (b)

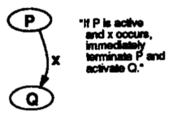
Fig. 5. Describing concurrent decomposition in SpecCharts. (a) Desired functionality. (b) SpecCharts description.



```
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 .....
```

(a) (b)

Fig. 6. Describing state-transitions in SpecCharts. (a) Desired functionality. (b) SpecCharts description.


P
Q
x
"If P is active and x occurs, immediately terminate P and activate Q."

```
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 ....
```

(a) (b)

Fig. 7. Describing exceptions in SpecCharts. (a) Desired functionality. (b) SpecCharts description.

Embedded system characteristic	SpecCharts	VHDL	Verilog	Esterel	Statecharts (Argos)	CSP	SDL
Behavior decomposition	yes	partial	yes	yes	yes	yes	partial
State transitions	yes	no	no	no	yes	no	yes
Exceptions	yes	no	yes	yes	yes	no	no
Sequential algorithms	yes	yes	yes	partial	no	yes	no
Behavior completion	yes	procedures only	yes	yes	no	yes	no

Fig. 8. Comparison of SpecCharts features with other languages.

subbehavior definitions are followed by the list of behaviors, recursively defined using the earlier behavior definition, where there must be exactly one behavior for each subbehavior definition.

We have also defined a graphical syntax for the language, since state-transitions are often more easily visualized graphically. Each behavior is drawn as a rectangle with rounded corners. Subbehaviors are drawn inside their parent's rectangle. Concurrent subbehaviors are separated by dotted lines, while sequential subbehaviors are connected with transition arcs. A TI arc is drawn from a source subbehavior's perimeter, while a TOC arc is drawn from a small square within the perimeter. Priority of arcs is ordered clockwise, starting from the top center of the subbehavior's rectangle. The special *complete* sequential subbehavior is represented by a small square. Finally, the initial sequential subbehavior is pointed to by a small triangle. Examples of the graphical SpecCharts syntax are found in the next section. Note that there is a one-to-one mapping between textual and graphical constructs, so the only difference between the two forms is aesthetic. We have found that experienced SpecCharts writers prefer a textual description during development of a specification (perhaps using informal graphical sketches before creating the text), so as to minimize the amount of "sizing-and-placing-and-routing" necessary when creating a graphical description. Once the specification is finalized, time can then be spent to create an equivalent graphical description for documentation purposes. However, the preference of text versus graphics will vary greatly from designer to designer.

The above completely defines a textual and graphical syntax for the extensions to VHDL made by the SpecCharts language. Note that the above syntax is the only syntax that a VHDL designer must learn in order to start capturing specifications with textual SpecCharts; in other words, only a few simple syntactical extensions lead to a substantial increase in descriptive abilities. In particular, Figs. 5, 6, and 7 show how easily we can capture behavior decomposition, state-transitions, and exceptions with SpecCharts, for the examples of Figs. 2, 3, and 4. Note the straightforward correspondence between these characteristics and the SpecCharts descriptions, as opposed to the very indirect correspondence of the characteristics with the earlier VHDL descriptions. Fig. 8 summarizes the embedded system characteristics supported by SpecCharts, VHDL, Verilog, Esterel, Statecharts, Argos, CSP, and SDL.

IV. EXAMPLE

In this section, we use an example to demonstrate the ease with which a complex embedded system can be captured using SpecCharts. We also show how several precise functionality issues are discovered and described during the specification process. The example is a part of the telephone answering machine controller found in [7]. We show how an informal English specification of the controller system's functionality is straightforwardly mapped to a precise SpecCharts specification.

Part of the English specification for the answering machine controller indicates that, when the machine is responding to the telephone line, it performs one of three tasks: "Monitoring the line for rings," where rings are counted until the required number are detected; "Normal answering activity," where the announcement is played and the message is recorded; and "Remote-operation answering activity," where the caller (assumed to be the machine's owner) can listen to the recorded messages by pressing a sequence of buttons on a remote phone. We refer to this line response behavior as *RespondToLine*, and we decide that *RespondToLine* is best understood by decomposing it into two major subbehaviors called *Monitor* and *Answer*, as shown in Fig. 9. The normal versus remote-operation distinction will be specified later in the *Answer* behavior.

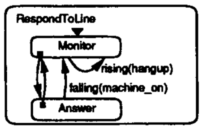


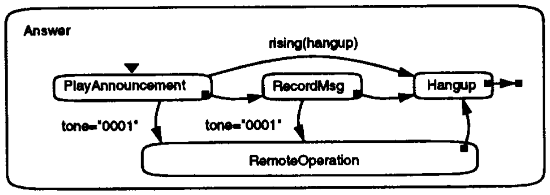
Fig. 9. The RespondToLine behavior in SpecCharts.

When the appropriate number of rings is detected, *Monitor* completes and *Answer* is activated. Thus, we add a TOC arc from *Monitor* to *Answer*. The English description states that after the message is recorded, "The machine hangs up and again monitors for rings." Thus we add a TOC arc from *Answer* to *Monitor*. The English specification also mentions the following: "If 'on/off' is pressed after the machine has answered, any current activity is terminated and the machine monitors the phone line. Such functionality is useful for screening calls, since one can listen to a message and then pick up the phone and press 'on/off' to turn the machine 'off' and begin speaking with the caller." This functionality is captured as a TI arc from *Answer* to *Monitor* with the condition *falling(machine_on)*. The *machine_on* signal introduced into the specification is "true" if the current state of the machine is "on."

We decide that if the phone begins ringing but the caller hangs up before the call is answered, the behavior *Monitor* should start over again. We capture this behavior as a TI arc from *Monitor* pointing back to itself, which is transitioned if a hangup is detected, causing *Monitor* to start again. Equivalently, we could have specified this hangup behavior as part of *Monitor* itself. The TI arc solution results in a simpler *Monitor* behavior.

The functionality of the answering activity is described as follows: "Once the machine has answered the line, it plays the announcement. When the announcement is complete, a beep is produced and the message on the phone line is recorded until a hangup is detected, or until a maximum message time expires. The machine hangs up and again monitors for rings. If a hangup is detected while playing the announcement, the machine immediately hangs up, and does not proceed to record a message. If button-tone '1' is detected, either while playing the announcement or while recording a message, the machine immediately enters remote-operation mode."

We thus decompose *Answer* into four subbehaviors: *PlayAnnouncement*, *RecordMsg*, *Hangup*, and *RemoteOperation*, as shown in Fig. 10(a). As long as no exceptions occur, we perform the first three in order, transitioning on completion using TOC arcs. One exception that can occur is a hangup during *PlayAnnouncement*. In such a case, we transition immediately to *Hangup* using a TI arc. A hangup occurring during *RecordMsg* is not considered an exception, but a normal completion. Another exception is the occurrence of *tone* = "0001" during *PlayAnnouncement* or *RecordMsg*. Such an occurrence requires an immediate transition to *RemoteOperation*, as indicated by TI arcs from each of *PlayAnnouncement* and *RecordMsg*. After *RemoteOperation* is complete, we transition to *Hangup*. Completion of *Hangup* is always followed by transition to behavior *Answer's* completion point.



(a)

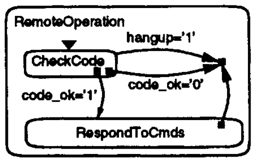
```
behavior PlayAnnouncement type leaf is
begin
   ann_play <= '1';
   wait until ann_done = '1';
   ann_play <= '0';
end;
```

(b)

```
behavior RecordMsg type leaf is
begin
   ProduceBeep(1 s);
   if (hangup = '0') then
      tape_rec <= '1';
      wait until hangup='1' for 100 s;
      ProduceBeep(1 s);
      num_msgs <= num_msgs + 1;
      tape_rec <= '0';
   end if;
end;
```

(c)

Fig. 10. The Answer behavior in SpecCharts.



(a)

```
behavior CheckCode type leaf is
begin
   code_ok <= true;
   for (i in 1 to 4) loop
      wait until tone /= "1111" and tone'event;
      if (tone /= user_code(i)) then
         code_ok <= false;
      end if;
   end loop;
end;
```

(b)

Fig. 11. The RemoteOperation behavior in SpecCharts.

Playing the announcement consists of three simple steps, which are captured as the three sequential statements shown in Fig. 10(b). They control the operation of the recording device of the machine.

Recording a message is also very simple. The steps required are captured as sequential statements as shown in Fig 10(c). After a one-second beep, the message is recorded until a hangup occurs or until the 100 s message limit elapses. A second beep is produced to indicate the end of the message and the number of messages is incremented.

Note that the description accounts for the possibility that the caller may hang up during the one second beep. If the caller does hang up, then *hangup* will be '1,' so the behavior completes without executing the statements that record and increment the number of messages.

The description of remote operation begins as follows: "The first step in the remote-operation mode is to check a user-identification number. The next four button-tone numbers that are pushed are compared to four numbers stored internally. If they do not match, the machine hangs up the phone. If they do match, the machine enters the *basic-commands mode*, in which it can be instructed to perform any of several basic commands."

We thus decompose *RemoteOperation* into two sequential subbehaviors, *CheckCode* and *RespondToCmds*, as shown in Fig. 11(a). After checking the entered four-digit code with the stored user identification number, we transition to *RespondToCmds* only if the code was correct. We introduce a boolean signal *code_ok*, which behavior *CheckCode* will set to "true" only if the code was correct. Two TOC arcs are used. One

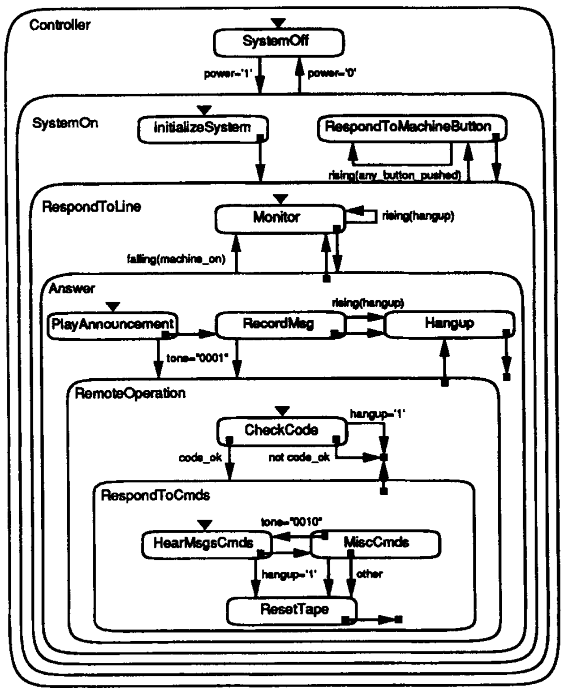


Fig. 12. The answering machine controller specification.

causes transition from *CheckCode* to *RespondToUserCommands* only if *code_ok* is "true." The other causes transition from *CheckCode* to *RemoteOperation*'s completion point if *code_ok* is "false." If a hangup occurs during *CheckCode*, a TI arc transitions to *RemoteOperation*'s completion point.

The behavior for *CheckCode* can be described using the statements in Fig. 11(b). From the program we see that, if the next four button tones match those stored in *user_code*, *code_ok* will be "true," otherwise it will be "false." Note that even if an incorrect tone is detected, the algorithm continues until all four tones sound. This continuation prevents the machine from hanging up immediately after an incorrect button is pressed, which would inform an unauthorized user which button tone was incorrect.

We omit the details of specifying the entire system for brevity. Fig. 12 shows the SpecCharts specification for the complete telephone answering machine controller, excluding the leaf behaviors' program statements. Note the easy capture of the five characteristics of embedded systems. Sequential decomposition is abundant, both through the use of sequential subbehaviors and through procedures. Concurrent decomposition, though not shown above, is also easily handled; in fact, the *Monitor* behavior is composed of two concurrent subbehaviors in the complete example. State-transitions are captured directly, such as when describing the four states and multiple arcs of the *Answer* behavior. Exceptions are extremely simple to specify using TI arcs, such as the *hangup* exception in the *Answer* behavior. Sequential algorithms are described straightforwardly, such as in the *CheckCode* behavior. Finally, completion is fully supported, such as the activation of *RecordMsg* following the completion of *PlayAnnouncement*.

V. TRANSLATION TO VHDL

In this section, we describe translation of SpecCharts to VHDL. We discuss the requirements of such translation, detail our algorithm, and discuss simulation and synthesis of the generated VHDL.

A. Requirements

We have chosen to translate SpecCharts to VHDL, rather than developing a suite of tools that directly supports SpecCharts, for several reasons. First, VHDL simulators are widely used, so they have become fast and reliable, and designers have much experience with them. Second, VHDL is required documentation for many projects, in which case using SpecCharts and translating can yield well-structured, error-free VHDL code more quickly than writing VHDL by hand. Third, several other powerful tools have evolved that take VHDL as input, such as synthesis and verification tools. In summary, the SpecCharts language is intended to enhance, not replace, a VHDL design environment; translating to VHDL achieves this goal.

Our requirements for translation include readability, simulation efficiency, and synthesizability. Readability is important because the VHDL will likely be read by humans, when debugging the specification during simulation, when designing an implementation manually or through synthesis, and when fulfilling VHDL documentation requirements. Simulation efficiency must be considered, since very slow simulations of the generated VHDL would probably lead designers to hand-write the VHDL. Synthesizability is important because we want to be able to generate small and fast hardware using synthesis tools.

B. VHDL Translation Algorithm

Although the generated VHDL will be less readable than the original SpecCharts (as demonstrated in the previous sections), the translation scheme presented in this section keeps the VHDL as readable as possible and ensures that each portion of the generated VHDL can be correlated with the SpecChart.

Several translation schemes have been published for translating a variety of specification languages to VHDL [8]–[14]. These languages support a very different model than SpecCharts or support only a subset of the embedded-system characteristics outlined in Section II. 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 either *inactive, executing*, or *complete*. We map each SpecCharts behavior to its own process with these three distinct sections. For every subbehavior S of a composite (sequential or concurrent) behavior B, two control signals *S_active* and *S_complete* are defined. These control signals are asserted and deasserted by the process representing the composite behavior in order to activate or deactivate the processes of its descendant subbehaviors.

Fig. 13 outlines a recursive algorithm for translating a hierarchical behavior to equivalent VHDL constructs. For a more detailed algorithm, refer to [15]. The input is the root

```
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 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 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
```

Fig. 13. Behavior to VHDL translation algorithm.

behavior from the SpecCharts file, and all output is written to a VHDL file. Procedure *CreateWaitOnArcsStmts*(*arcs*) creates a wait statement that determines if an arc transition should occur, i.e., if a subbehavior S has completed and a TOC arc condition from S is true, or if 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. (For concurrent subbehaviors, all subbehaviors are activated, and transition to complete occurs when all have completed.) Procedure *CreateCompletionHandshakeStmts*(B) creates a set of statements that 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 *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 II-C. 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. More information on VHDL signal drivers can be found in [1]. 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 behavior 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. The algorithm generates a process in each block, containing three sections.

- *Inactive* In this section, the behavior is waiting to be activated via assertion of a control signal by the parent behavior.
- *Executing* In this section, a composite behavior is activating/deactivating appropriate subbehaviors via control signals, while a leaf behavior is executing its VHDL code.
- *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 inserts polling code that causes the process to jump to its end whenever the behavior is deactivated.

The above algorithm satisfies two of the translation requirements specified earlier. First, the VHDL is very readable (relative to other possible VHDL descriptions of the same functionality). It is readable because the PSM model's program states can still be discerned since each is now represented as its own process, the PSM model's arcs are kept separate from computations, and hierarchy is maintained. An added advantage is that we can easily find the relevant VHDL process for a given SpecCharts behavior, and vice versa. Second, the VHDL simulates efficiently; although we have found that the two signals added per behavior does tend to slow the simulation, this slowing is negligible for most cases. As an added advantage, the recursive aspect of the translation scheme makes it easy to implement. However, we need to modify the translation scheme to get good results from synthesis, as will be discussed shortly.

C. Simulation of the Generated VHDL

The generated VHDL consists of an entity and architecture for the system. Thus, we can use the same techniques for simulating the generated VHDL as we would any other entity. For example, we can create another VHDL file that instantiates the entity as a component, assigns values to the component's inputs, and monitors its outputs. In addition, we can monitor the *B_active* and *B_complete* signals, generated during translation, to examine the dynamic transitioning between PSM program-states.

The translation technique must be modified for some examples that rely on the delta-delay semantics of a signal. Specifically, note that SpecCharts supports VHDL declarations, so that we can declare a signal in SpecCharts. Signals differ from variables in that they not only possess a value, but they possess that value at a particular simulation time. When we assign a value to a signal using an assignment statement, we must specify the time in the future at which the signal should get the new value; the smallest such time is an infinitely small unit of time, called a delta. However, note that our translation scheme introduces new control signals *B_active* and *B_complete*, which themselves are updated in delta time. For correct functionality, all updates of control signals must by completed *before* any updates of regular signals, since the control signals determine which behaviors should be active; otherwise, a regular signal might get updated in a behavior B, but then after the control signals are all updated we might find that the behavior B should have been deactivated and so the regular signal update should not have occurred.

In more general terms, the delta-time required for the control signals introduced during translation may interfere with the delta-time of regular signals declared by the designer. The simple solution to this problem is to shift the time in which regular signals are updated to a larger unit of time. Therefore, delta-time updates for regular signals are shifted to a higher time scale, which is smaller than other time scales used in the specification and simulation results are then shifted back. See [16] for details of this approach. The same problem and solution applies to many of the above referenced translation schemes.

D. Synthesis from the Generated VHDL

The VHDL generated by the translation scheme may present inefficient hardware when VHDL synthesis tools are used, because synthesis tools typically 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 datapaths. The simple solution to this problem is to automatically flatten the hierarchical behaviors into sequential statements of one leaf behavior before applying the above translation algorithm. For sequential behaviors, such flattening is performed in two steps. First, we eliminate all program-states, except for leafs, by replicating each arc at higher hierarchical levels to depart from each descendant leaf behavior. There is thus an increase in the number of arcs roughly equal to the number arcs times the number of leafs; note that this is not an exponential increase. Second, we convert the resulting state-transitions to sequential programming constructs, as described in Section II-B. Flattening sequential behaviors before synthesis is very commonly done, since in such cases the designer usually uses hierarchy only for ease of description, but really wanted a single controller/datapath implementation for the design. Flattening improves synthesizability at the cost of less readability (due to loss of hierarchy); however, the resulting VHDL is still well-structured, so probably more readable than handwritten VHDL.

We usually do not want to flatten concurrent behaviors, since we actually want a separate controller for each concurrent behavior. Flattening concurrent behaviors leads to an exponential increase in behavior size, so we must reserve such flattening for very small behaviors.

VI. RESULTS

The earlier sections should have provided some intuitive sense of the benefits of using SpecCharts to capture embedded system specifications. Without SpecCharts, specification capture time may be longer, comprehension of the system's functionality may be reduced, and functional errors may be more abundant. In this section, we describe several experiments that demonstrate these issues quantitatively. The experiments compare the use of SpecCharts to VHDL for the specification of embedded systems.

A. Specification Capture

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. Two groups of modelers were given an English description of an example system. One group was asked to specify the system in VHDL, and the other in SpecCharts. The specification time required and the number of errors in the specifications of these two groups were then compared.

The example was an aircraft traffic-alert and collision-avoidance system [17]. This system was chosen since it represents an existing embedded system, and secondly because its documentation was available from an outside source, thus reducing the possibility of experimenter bias. Because of time limitations, only a subset of the system's functionality was selected for specification. Three modelers specified the selected subset in VHDL, and three in SpecCharts.

The VHDL modelers required an average of 2.5 times longer than SpecCharts modelers to capture the specification of the system. In addition, two of the VHDL specifications possessed a major control error, resulting in very slow system reactions to external events. This problem 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. SpecCharts proved to be more effective because of its support of state transitions and exceptions.

B. Specification Comprehension

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 and another the SpecCharts specification; each group was asked several questions about the system's functionality. 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 [18], 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. The time each person took to understand the specification was measured. After the specification was understood, 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?" "What is the purpose of variable v?"

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

C. Specification Quantification

To quantify the several differences between SpecCharts, VHDL, and Statecharts specifications, a single system was specified in all three languages. Several different characteristics of each specification were then measured. The example chosen was a telephone answering machine. An English description was captured in SpecCharts, VHDL, and Statecharts. Two VHDL versions were created. One maintained the

Specification attributes	Conceptual model	SpecCharts	VHDL (hierarch.)	VHDL (flat)	Statecharts
Program–states	42	42	42	32	80
Arcs	40	40	40	152	135
Control signals	—	0	84	1	0
Lines/leaf	—	7	27	29	—
Lines	—	446	1592	963	—
Words	—	1733	6740	8068	—

Fig. 14. Comparison of SpecCharts, VHDL, and Statecharts.

Design attribute	Designed from English	Designed from SpecCharts
Control transistors	3130	2630
Datapath transistors	2277	2251
Total transistors	5407	4881
Total pins	38	38

Fig. 15. Design quality from SpecCharts versus English specifications.

hierarchy by using nested blocks and processes communicating via control signals, as discussed in Section V-B. The other flattened the hierarchy into a single program-state machine, as discussed in Section V-D, which was then described as a single process.

Fig. 14 shows the results of this experiment. The flat VHDL has fewer program-states since only leaf program-states exist. This reduction is achieved at the expense of almost four times as many arcs, an increase required for the following reasons. In the hierarchical model, an arc at a higher level in the hierarchy can describe concisely a transition to another state, regardless of which descendent leaf state the system is in. In the flattened model, such an arc must be replicated to point explicitly from *each* leaf state to the correct next state. Immediate transitions, moreover, require polling throughout the code, as described earlier. Furthermore, arcs are represented using sequential statements. These three reasons result in over four times as many words in the flat VHDL as in SpecCharts.

The hierarchical VHDL does not require any additional arcs, but does require adding 84 control signals (two per program-state) for implementing control among the many processes. Writing and reading these signals, along with the polling required for immediate transitions and the representation of arcs with sequential statements, result in almost four times as many words as in SpecCharts. Clearly, the higher the number of lines and words in a specification, the greater the specification time, comprehension time, and occurrence of errors. With regard to leaf program-states, both VHDL versions require about four times as many statements per leaf program-state as SpecCharts. The increase is significant because it impairs the readability of the leaf program-state, defeating the leaf's purpose of modularizing the functionality into easily understood portions.

The consequence of the lack of programming constructs in Statecharts can be clearly seen in this example. Because the programming constructs in the leaf behaviors must be described using states and arcs, the Statecharts description contains almost twice the number of states and three times as many arcs as the SpecCharts description. Using states and arcs to describe the programming constructs can be quite tedious and unnatural, compared to using sequential program constructs. For example, a simple *for loop* must be described using several states and arcs. Note that, since Statecharts is only defined graphically, lines and words are undefined.

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
mwt	708	1785	9.7
uart	262	364	1.3

Fig. 16. Translation results.

D. Design Quality

The goal of this experiment was to demonstrate that designing from SpecCharts does not produce a lower design quality than designing from English. We compared the number of transistors in a design derived from an English specification with the number derived from a SpecCharts specification.

The example chosen was the answering machine described in this chapter. An English specification was given to two designers. One designer generated a datapath and controller directly from this specification. The other designer first specified the system with SpecCharts, flattened the hierarchy automatically, and then generated a datapath and controller from the flattened SpecCharts specification. In both cases, a synthesis tool was used to synthesize controller logic from an FSM description of the controller.

Results are shown in Fig. 15. Design time for each person was roughly the same, about 30 person-hours. Note that the number of transistors in the final design obtained from SpecCharts is not greater than that obtained from English. In this case, the number is actually smaller, since fewer control states are used in the design. The reason for the reduction in states is as follows. The English-specification designer captured the functionality using an FSM. The FSM served as the only precise specification of functionality. The designer had to keep this FSM readable in order to mentally verify the correctness of the machine's functionality. This readability requirement prevented him from grouping many states into a single state, since such a grouping would have made mental verification more difficult. On the other hand, the SpecCharts designer verified the functionality using the SpecCharts specification. When translating to an FSM, readability of the FSM was not an issue. States were grouped during this translation, resulting in less control logic (the grouped states were not actually fully equivalent, so a synthesis tool could not have made the same grouping).

E. Translation

Finally, we conducted experiments to ensure that SpecCharts to VHDL translation times, generated lines of VHDL, and simulation efficiency were within reason for practical use. Fig. 16 summarizes translation time and resulting VHDL lines of code for numerous examples; descriptions of all of the examples can be found in [19]. 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.

To test the simulation efficiency, we obtained two VHDL models of a peripheral interface example manually specified by an industry 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 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 s, 250 s, and 550 s, respectively. While the simulation of the VHDL generated from SpecCharts is slower, it is not slower by an order of magnitude, which in turn might have required the development of a custom SpecCharts simulator.

VII. Status and Future Work

A parser and VHDL translator have been implemented for the textual SpecCharts language. The implementation consists of 20 000 lines of *C* code, and includes several automated transformations, such as flattening the hierarchy, procedure inlining, and the time-shift discussed in Section V. It also includes a graphical tree display of the hierarchy (i.e., it displays the behaviors graphically, but not the arcs), such that the designer can click on any subbehavior's node to popup a text editor for that subbehavior. Several queries are also supported, such as indicating which nodes access which symbols. The tool has been released to over 20 companies and universities, and is currently being used in several industry and university projects. SpecCharts also serves as input, along with VHDL, to the SpecSyn system-design tool [20], [21].

There are some language extensions that could be made straightforwardly. One extension is the creation of a fourth program-state type, a concurrent leaf, which would contain VHDL concurrent signal assignments only, without any sequential program constructs. Adding such a type would enable direct support for the dataflow characteristic, which is not found in PSM, but is still useful for some embedded systems and especially for signal-processing systems. Another extension is the refinement of the concept of a concurrently-composed behavior's completion. Rather than defining its completion as the completion of all subbehaviors (a standard join), we can define the behavior's completion as the completion of a subset of those subbehaviors (a selective join); we have found that selective joins would be useful for several examples. In fact, we can define the behavior's completion as the completion of all subbehaviors in the subset (a selective AND-join), or as the completion of at least one subbehavior in the subset (a selective OR-join). A third extension could easily be made to the allowable condition on a TI arc. Note that a TI arc with condition "cond" is equivalent to a VHDL "wait until cond" statement being fired off at the same time the arc's source substate is activated; when the wait statement's condition is satisfied, the arc is traversed. Because of this equivalence to a wait statement, we can associate a timeout clause ("for T") and a sensitivity clause ("on $s1, s2, \cdots$") with each arc (see [1] for details of these clauses), just as allowed in a VHDL wait statement, thus permitting powerful yet concise specifications of exception conditions.

We plan to enhance the SpecCharts tool set as follows. First, since current synthesis tools place restrictions on acceptable VHDL input, we plan to develop translators for various synthesis tools. Second, since current synthesis tools and compilers do not modify the the overall organization of a specification, such as the process-level parallelism, we plan to include a suite of automated specification transformations, such as those that convert sequential behaviors to concurrent ones, and vice versa. Other future work may include a graphical capture tool for SpecCharts, including a graphical simulation tool that highlights active behaviors.

VIII. Conclusion

Increasing system complexity and reduced time-to-market requires new solutions to system design problems, especially the problem of functional specification. Existing languages don't support direct specification of common embedded system characteristics, so we developed a new conceptual computation model, PSM, and a language based on that model, SpecCharts. PSM combines the common hardware models and software models into one, so it addresses the fact that today's embedded systems include both software and custom hardware. SpecCharts supports this model by building on a popular standard language, VHDL, so it fits in well with current methodologies. Using SpecCharts to specify embedded systems can lead to fewer functional errors and easier integration of system modules, which in turn result in fewer design iterations, faster time-to-market, and improved product support and enhancement over a product's lifetime.

References

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

[2] D. Thomas and P. Moorby, *The Verilog Hardware Description Language.* Norwell, MA: Kluwer, 1991.

[3] C. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[4] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programming 8*, 1987, pp. 231–274.

[5] N. Halbwachs, *Synchronous Programming of Reactive Systems.* Norwell, MA: Kluwer, 1993.

[6] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specifications.* Englewood Cliffs, NJ: Prentice–Hall, 1991.

[7] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1994.

[8] N. Dutt, J. Cho, and T. Hadley, "A user interface for VHDL behavioral modeling," in *Proc. Int. Symp. Comput. Hardware Description Languages and Their Applicat.*, 1991, pp. 375–393.

[9] O. Pulkkinen and K. Kronlof, "Integration of SDL and VHDL for high-level digital design," in *Proc. European Design Automation Conf. (EuroDAC)*, 1992, pp. 624–629.

[10] B. Lutter, W. Glunz, and F. Rammig, "Using VHDL for simulation of SDL specifications," in *Proc. European Design Automation Conf. (EuroDAC)*, 1992, pp. 630–635.

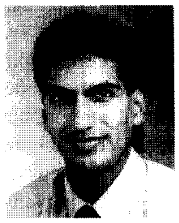
[11] R. MacDonald and R. Waxman, "Operational specification of the SINCGARS radio in VHDL," in *AFCEA-IEEE Tactical Commun. Conf.*, 1990, pp. 1–17.

[12] T. Tikanen, T. Leppanen, and J. Kivela, "Structured analysis and VHDL in embedded ASIC design and verification," in *Proc. European Conf. Design Automation (EDAC)*, 1990, pp. 107–111.

[13] A. Arsenault, J. Wong, and M. Cohen, "VHDL transition from system to detailed design," in *VHDL Users' Group*, Apr. 1990.

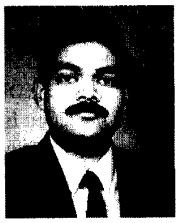
[14] A. Jerraya, P. Paulin, and D. Agnew, "Facilities for controllers modeling and synthesis in VHDL," in *VHDL Users' Group*, Apr. 1991.

[15] S. Narayan, F. Vahid, and D. Gajski, "Translating system specifications to VHDL," in *Proc. European Conf. Design Automation (EDAC)*, 1991, pp. 391–394.
[16] F. Vahid and D. Gajski, "Obtaining functionally equivalent simulations using VHDL and a time-shift transformation," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 362–365.
[17] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese, "Requirements specification for process-control systems," Dept. of ICS, Univ. of California, Irvine, Tech. Rep. 92-106, 1992.
[18] R. Gupta and G. DeMicheli, "System-level synthesis using re-programmable components," in *Proc. European Conf. Design Automation (EDAC)*, 1992, pp. 2–7.
[19] S. Narayan, F. Vahid, and D. Gajski, "Modeling with SpecCharts," Dept. of ICS, Univ. of California, Irvine, Tech. Rep., pp. 90–20, 1990.
[20] D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: Executable-specification refinement," in *Proc. European Conf. Design Automation (EDAC)*, 1994, pp. 458–463.
[21] D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, and P. Fung, "System design methodologies: Aiming at the 100-hour design cycle," to appear in *IEEE Trans. VLSI Syst.*, 1995.



Frank Vahid (M'89) received the B.S. degree in electrical engineering from the University of Illinios at Urbana-Champaign, in 1988. He received the M.S. and Ph.D. degrees in computer science from the University of California, Irvine, in 1990 and 1994, respectively, where he was an SRC Fellow.

He is currently with the Department of Computer Science at the University of California, Riverside as an Assistant Professor. His research interests include embedded-system specification and design, hardware/software co-design, functional partitioning, and behavioral synthesis.



Sanjiv Narayan (M'89) received the B.S. degree in computer science in 1988 from the Indian Institute of Technology, New Delhi and the M.S. and Ph.D. degrees in information and computer science in 1994 from the University of California at Irvine.

He is a Member of the R&D Group at View-logic Systems, Inc. in Marlboro, MA. His research interests include system specification and design, hardware description languages, and behavioral synthesis.

Dr. Narayan was the recipient of the Director's Gold Medal at the Indian Institute of Technology and was a Chancellor's Fellow at UC-Irvine.

Daniel D. Gajski (M'77–SM'83–F'94) received the Dipl. Ing. and M.S. degrees in electrical engineering from the University of Zagreb and the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia.

After ten years of industrial experience in digital circuits, telecommunications, supercomputer design, and VLSI structures, he joined academia as a Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, and was the founder of the lab for supercomputer research. Presently, he is with the Department of Information and Computer Science at the University of California, Irvine as a Professor. His interests are in CAD environments, ASIC and system design methodology, high-level synthesis, and hardware-software co-design.