Proc. of EuroDAC/EuroVHDL, Hamburg, 1993

Features Supporting System-Level Specification in HDLs

Sanjiv Narayan and Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA, 92717, USA

Abstract

As synthesis tools become more advanced and reliable, the entry point for the designer in the design process is moving towards higher levels of specification. In this paper, issues related to the specification of embedded systems are discussed. We compare VHDL with five other specification languages: HardwareC, SDL (Specification and Description Language), Statecharts, SpecCharts, and CSP (Communicating Sequential Processes). The capabilities of these languages with respect to specifying designs at the system-level are highlighted. We conclude by presenting a list of features which are desirable in a language to be used for specifying systems.

1 Introduction

Many tools and environments have been developed to aid system designers achieve the implementation of a design. As synthesis tools become more advanced and reliable, the entry point for the design tools in the design process is moving to higher levels of specification. Before invoking traditional high-level synthesis tasks (such as scheduling and allocation), it is often necessary to *refine* a system's abstract specification into a set of one or more communicating behavioral descriptions representing chips or other physical modules of the system. We refer to this task as *System Level Synthesis*.

By system-level designs, we refer mainly to embedded systems i.e. systems whose behavior is defined by its interaction with its environment. An embedded system usually will sequence between a set of *modes*, where each mode may represent a state of being or some computation. Examples of such systems are computer and telecommunication systems.

Any specification language represents a tradeoff between the *degree of expressivity* and the *ease of implementation*. For example, if a system is described using a structural netlist of components, the designer is limited to specifying his/her design with a fixed set of components and their interconnections, but there exist a lot of tools which can generate a layout from a given netlist. On the other hand, a designer specifying a design in English has virtually unlimited power of expression, but it would be impossible to synthesize a design automatically from such a description.

As specification languages progress towards higher levels of abstraction, the question that arises is what features are desirable in a specification language at the system-level ? The primary aim of a system-level specification language is to be able to capture the designer's conceptual view of the system with minimum designer effort. This conceptual view differs from application to application. For example, reactive systems (i.e. event-driven, control-dominated systems such as telephone and avionics systems) might be more easily visualized as a state transition diagram, while an algorithm performing some computation is more easily conceptualized as a programming language code segment. The specification language must possess appropriate abstractions or constructs to which the designers can directly and easily translate their conceptual view. Second, describing entire systems might require some mechanisms (such as hierarchy and concurrency) for structuring the specification so as to be able to enter, maintain and modify it in a reasonable manner. In addition, since most systems typically consist of several concurrent activities or *processes* which interact with each other, appropriate communication and synchronization constructs may be required.

By examining the constructs (and their intended use) in several existing hardware description languages, this survey identifies the features that would be useful in specifying system-level designs.

2 System Specification Languages

The languages that we examine encompass a broad range of description styles and application domains, and are fairly representative of design specification methods currently in use. Since the purpose of this paper is mainly to highlight the different features of the languages, the details of the languages themselves are not presented. We examined six languages: VHDL, HardwareC, SDL (Specification and Description Language), Statecharts, SpecCharts and CSP (Communicating Sequential Processes).

VHDL [1, 2] is an IEEE standard intended to be used for the development, synthesis, verification and documentation of hardware designs. It has the capability of specifying a system in terms of *behavior*, *dataflow* or *structure* or any combination thereof.

HardwareC [3, 4] was designed expressly for synthesis. Based on the C Programming language [5],

it has well-defined semantics and constructs for hardware description.

SDL [6], standardized by the CCITT, has been mainly used in the telecommunication field. An SDL specification can be summarized as consisting of hierarchical dataflow diagrams with state transition diagrams at the leaf level.

Statecharts [7, 8] was designed primarily for specifying reactive systems. Statecharts provide an elegant extension to traditional FSM's by adding hierarchy, concurrency and broadcast communication.

The SpecCharts language [9], developed for system specification capture, combines the three aspects of system specification - control, behavior, and structure into a single, unified concept. It is essentially a combination of hierarchical/concurrent state diagrams [7] and VHDL sequential statements in leaf-level states.

The CSP [10] language has been recently used for hardware system specification [11]. A CSP specification consists of a set of concurrent processes with constructs to specify communication and synchronization between processes.

3 Evaluating Specification Languages

Having introduced the specification languages, we now examine some of their features/constructs with respect to system specification.

3.1 Hierarchy

Hierarchy in specifications can be *behavioral* or *structural*. Each of these are examined separately.

3.1.1 Behavioral Hierarchy

We define a behavior to be a set of actions and a set of conditions describing when each action is to occur. Actions are anything that can change a system value (e.g. a variable's value or the system's state) or an algorithm. We define a language supporting *behavioral hierarchy* as one which permits the following:

1. *Behavioral Decomposition* – the ability to specify a behavior's actions as sub-behaviors, either concurrent or sequential.

2. *Hierarchical Activation/Deactivation* – the ability of a behavior to (de)activate a sub-behavior at *any* time.

3. *Behavioral Completion* – the ability of a behavior to indicate completion and the ability of other behaviors to detect this completion. This may be used for sequencing from one sub-behavior to another.

The underlying model of SpecCharts supports the concept of behavioral hierarchy. A system is captured as a set of hierarchical/concurrent behaviors. The TOC (transition on completion) arc is used to indicate behavioral completion, while a TI (transition immediately) arc specifies hierarchical deactivation.

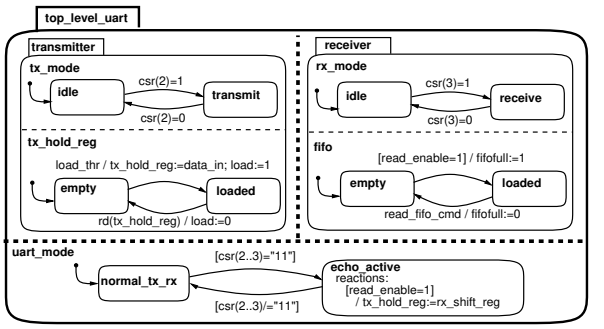

Figure 1: Hierarchy/concurrency in the UART Statechart

True behavioral decomposition is not supported in VHDL, HardwareC, and SDL. All these languages provide only a single level of concurrency (processes) followed by one level of sequentiality (sequential statements in the process). In addition, there is no support for deactivation of processes.

The hierarchy of states and transition arcs in Statecharts capture behavioral decomposition and deactivation respectively, as shown in the Universal Asynchronous Receiver Transmitter (UART) [14] in Figure 1. However, behavior completion is not supported. While behavioral decomposition in CSP can be specified using the process activation statement, process deactivation is not supported.

Hardware systems requiring behavioral hierarchy are common [9]. The Intel 8237 DMA Controller [12] and the Pipelined SISC processor [13] that we modeled had 4 levels of behavioral hierarchy. Behavioral hierarchy is essential for two reasons: (1) It enables the designer to capture the design in manner which is closer to his/her conceptual view of the system. For example, to specify an embedded system in HardwareC, the designer has to coerce the design specification into a model of a single-level of concurrent processes, which may not always be possible, (2) It avoids specification modification errors that inadvertently emerge when modelers modify the behavior to fit the language's constructs rather than write a correct model (which would become difficult to understand otherwise). The most common example is that of an asynchronous system reset, which is often converted to a synchronous reset checked at the end of an execution cycle (e.g. SISC processor description in [13]).

3.1.2 Structural Hierarchy

A structural hierarchy allows system specifications to be represented as a set of interconnected components. Each component's internal structure is specified with a set of lower-level interconnected components. Structural hierarchy enables the designer to represent new components using pre-existing components. Second, it allows specification of the internal

structure of the system as new details are added as a result of design activity. For example, a VHDL process may get refined after synthesis into a set of interconnected RT-level components which implement the behavior specified by that process.

VHDL, HardwareC and SDL provide a *declarative hierarchy* consisting of nested blocks with the capability of specifying and interconnecting ports (signals and channels in SDL) at the block boundaries. In addition, the hierarchy of blocks also define the scope for all symbols declared in a block at any level. SpecCharts allow the designer to specify ports, channels and their interconnections for a behavior at any level of the hierarchy. Statecharts and CSP do not have the capability of specifying any structural information.

3.2 Concurrency

Hardware systems are often easily thought of as a set of concurrent communicating modules. Consequently, a specification language must have constructs to represent concurrency in system specifications.

3.2.1 Process-level parallelism

Process-level parallelism refers to the capability of specifying the system as a set of "chunks of computations" or *processes*, all of which execute concurrently with each other. VHDL, SDL and HardwareC have a single level of processes. SpecCharts and Statecharts can specify concurrency at any level of the hierarchy. For example, in the UART Statechart of Figure 1, the topmost state *top_level_uart* consists of three concurrent substates: *transmitter, receiver* and *uart_mode*. CSP and SDL have the capability of creating new processes dynamically.

3.2.2 Statement-level parallelism

Statement-level parallelism refers to the specification of concurrency at the statement level. For example, HardwareC has the parallel compound statement in which each of the enclosed statements are executed in parallel as shown below :

```
< A = B ;   B = A ; >
```

As a result, the values of A and B will get swapped. Similarly, the signal assignments semantics of VHDL (and consequently SpecCharts) allow for specification of fine-grain parallelism.

3.3 Timing

The ability to incorporate timing information in a design specification is important since it can capture the behavior intended by the designer more accurately. In addition timing specifications in some cases may also serve as constraints for subsequent synthesis tools.

In VHDL and SpecCharts, timing specification takes one of two forms:

1. *After clauses* which specify the time into the future when the value of the signal is to be updated with the new value (e.g., `A <= 2 after 20 ns`).
2. *Timeout clauses* which specify the maximum time to be spent at the wait statement (e.g.,`wait on A for 100 ns`)

Statecharts and SpecCharts have the *timeout arc* which causes a transition from a behavior on the expiry of the time-period associated with it. The designer can specify the minimum and maximum time spent in any state in Statecharts. While HardwareC has no explicit timing constructs, the language allows the designer to specify timing constraints between pairs of statements. SDL has a *timer* object which can indicate a timeout by generating a signal on the expiry of the associated time period.

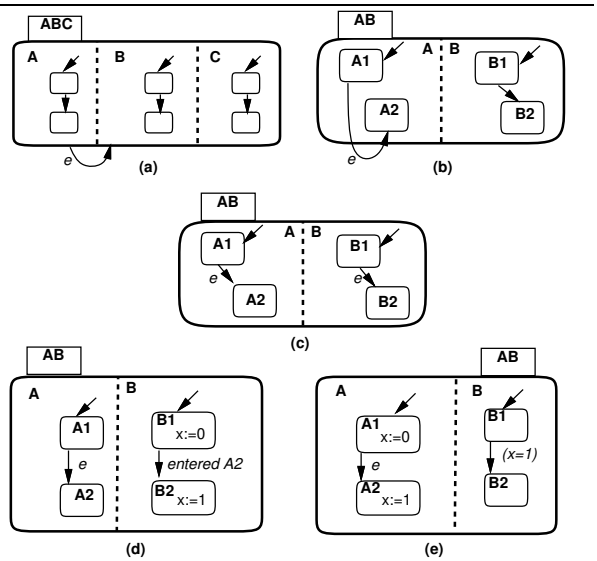

Figure 2: Synchronization methods in Statecharts

3.4 Synchronization

In a system specification with several concurrent processes, the need for synchronization may arise when two processes exchange data or when certain actions must be performed by different processes at the same time. Synchronization methods can be classified into the following categories and are illustrated with the Statecharts of Figure 2.

3.4.1 Synchronization by Initialization

In Figure 2(a), event e synchronizes all the orthogonal states `A, B` and `C` into their default substates. SpecCharts also support this kind of synchronization. Languages like VHDL and HardwareC have their processes synchronized in this manner *only when the system is first initialized.* In Figure 2(b), event e causes `B` to move to its default substate `B1` while at the same time moving `A` from `A1` to `A2`. Synchronization by initialization may be used to simultaneously initialize the stages of a pipeline which has been modeled as a set of concurrent behaviors.

3.4.2 Synchronization by Common Event

In Figure 2(c), the event e synchronizes both the states `A` and `B` into substates `A2` and `B2` respectively. In VHDL and SpecCharts, *wait* statements in two processes with a common signal on the sensitivity list will synchronize the processes when an event occurs on the common signal. Similarly, two SDL processes waiting for an event to occur on a common input signal, will synchronize on the occurrence of such an event. An example of synchronization by common event is a global clock signal which can be used to synchronize the CPU and its peripheral devices.

3.4.3 Synchronization by State Detection

The transition from `A1` to `A2` due to event e causes `B` to transit from `B1` to `B2` as shown in Figure 2(d). This synchronization method is unique to Statecharts. It can be used to efficiently represent situations such as tristating the transmitter output lines whenever a UART enters the receive *mode* or state.

3.4.4 Synchronization by Common Variable

In Figure 2(e), the assignment to the variable x in state `A2` synchronizes `B` into state `B2`. In VHDL, SpecCharts and SDL, processes can check a global variable or signal and based on their value, carry out certain actions. Thus, the VHDL statement "`wait until X = 1`" will resume execution only when the signal X is assigned the value "1" by some process. Updating a status bit which can be read by other processes is an example of this kind of synchronization.

3.4.5 Synchronization by Communication

In interprocess communication, the process which is ready first usually waits for the other process to be ready for the communication. In HardwareC, the *msg-wait* command indicates whether the specified channel has any pending messages and can be used for synchronizing the sending and receiving processes. In CSP, the input/output command is delayed until the other process is ready with the corresponding input or output. In SpecCharts, the necessity of synchronization for communication depends on the protocol associated with the communication channel between two concurrent behaviors. No synchronization is needed if the communication between them is implemented by connecting a storage element directly to the output ports (as in a *hardwired protocol*). If a handshaking protocol is chosen, one of the concurrent behaviors will block until the other is ready to receive/transmit data.

3.5 Interprocess Communication

Interprocess communication is implemented in specification languages using the *shared memory* or *message passing* paradigms.

3.5.1 Shared Memory

In the shared memory model, the sending processes update a shared medium such as a global store or port which is then read by all the receiving processes. Communication in VHDL, HardwareC, Statecharts and SpecCharts can be implemented using this model. In VHDL and SpecCharts, global signals are declared which are accessible by concurrent processes and behaviors. Communication in Statecharts is implemented with global variables. In HardwareC, VHDL and SpecCharts, ports can be declared on the boundaries of blocks/processes, blocks/entities and behaviors respectively. In these cases, the shared memory is represented by the wire/global signal used to connect these ports. The protocol to implement the transfer, if any, has to be specified as a part of the description of the processes.

3.5.2 Message Passing

The message passing model abstracts out the communication by viewing the communicating processes as being connected with *channels* over which the data or *message* is sent. In each process, communication is represented by simply specifying the data that is to be sent/received over the channel and the identity of the destination/source process which can be specified explicitly (as in CSP) or can be inferred from the interconnection of processes with channels (as in SpecCharts and HardwareC).

In HardwareC, implementation of the communication channel is determined by the synthesis tools. SpecCharts make message passing more flexible in that it lets the designer associate a protocol with each communication channel. A protocol consists of a set of ports and a behavior over those ports. During synthesis, if it is decided that another protocol is to be used for communication (e.g. serial transfer instead of parallel), only the protocol associated with the channel need be changed. The processes remain unchanged.

3.6 Exception Handling

Exception handling refers to the ability to respond to external events by suspending or terminating the current set of actions and transferring control to another portion of the specification. Exception handling is a subset of hierarchical deactivation which was discussed briefly in Section 3.1.

SpecCharts and Statecharts can easily model exceptions by providing a transition arc to transfer control to another behavior. Since a SpecChart behavior could have programming statements at the leaf level, the semantics of the TI (transit immediately) arc will cause the programming statements to terminate immediately and complete the transition.

Terminating a SDL or VHDL process is relatively difficult. A VHDL process could be at a wait statement which is not sensitive to the event causing the exception. Such wait statements will have to be modified to respond to the event. Terminating an SDL process will require transitions labeled with the exception event from all states in the state-transition diagram to

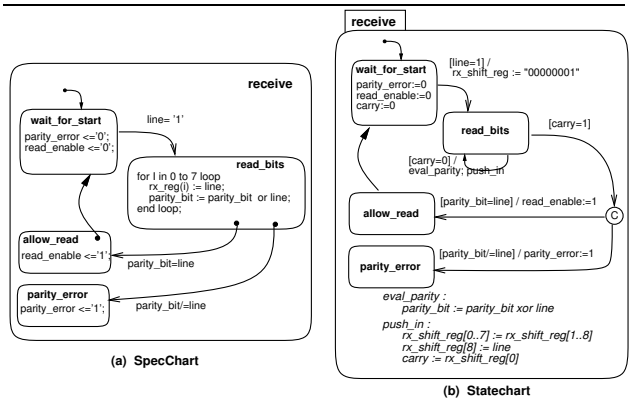

Figure 3: Specifying actions in the UART Receiver Process

the *stop* construct of the process. Such modifications can be cumbersome.

3.7 Specifying Actions using Programming Constructs

Using hierarchy, the designer can conceptually decompose the system into smaller and more manageable "chunks" of computation. This decomposition will reach a stage, where the functionality of each chunk or *actions* can be easily written using programming language constructs like loops, case statements etc. VHDL, SpecCharts, HardwareC and CSP have incorporated programming language constructs for specifying actions in processes.

Statecharts and SDL on the other hand allow only simple actions such as assignments in their state machines. Control constructs such as looping have to be implemented explicitly using state transitions. To see this, consider the UART receiver process in Figure 3 which detects a start bit (`line=1`) and then reads the next eight bits into a register. The reading of the eight bits is shown using the for loop in *read_bits* of the SpecChart of Figure 3(a). For Statecharts however, as shown in Figure 3(b), an 8-bit register is loaded with a '1' in its rightmost bit, '0's in the other bits. Each input bit is shifted into this register while the shift-out value (carry) is checked; a value of '1' signifies that all 8 bits have been shifted in. Such workarounds in the absence of programming constructs in Statecharts make specifications difficult to write and comprehend.

3.8 State-Based Specifications

Embedded systems often are inherently state-based (e.g. traffic light controller). These systems are more naturally described using a language like Statecharts, SDL or SpecCharts which have constructs to specify state transition diagrams. Modeling such systems with HardwareC or VHDL using processes or procedures can be cumbersome since the states and transitions have to be implemented explicitly, using some combination of control signals, state variables, case and loop statements.

3.9 Designer Interaction for Synthesis

Amongst the languages discussed in this paper, HardwareC has several features it offers to the designer for guiding the synthesis tools. Firstly, it has constructs that enable the designer to specify *timing constraints* and a *resource allocations*. Secondly, for any group of statements, the designer can specify their relative order of execution by using the serial, data-parallel and parallel compound statements. Such information can be used during the *scheduling* of the operations in a process. The designer can specify *binding information* by associating individual operations with specific instances of a component template. Finally, the designer can specify which variables are to be implemented as storage by declaring them to be static, while at the same time letting synthesis tools decide whether the other variables are to be implemented as wires or registers.

Languages like VHDL and SpecCharts support specification refinement which allows detail to be added to the specification as a result of synthesis tasks. The most significant advantage of this approach is that the refined specification containing the changes added during the previous synthesis step is presented to the designer in the same format as the original specification. A single language reflects the design throughout synthesis, as opposed to requiring other models (e.g. a control/dataflow graph), which a designer must then correlate with the initial specification.

4 Conclusion

In the previous sections, we have presented six system specification languages and examined some of their salient constructs. Figure 4 summarizes the various features of the languages we have discussed.

It must be emphasized here that each of the languages is ideally suited for their respective application domains and it is not our intention to recommend any one language over the others. Based on the discussion of the preceeding section, we conclude that a system-level specification must have the following features:

1. *Hierarchy* to manage the complexity in terms of specification size. *Behavioral hierarchy* is inherent in embedded systems and presents the most natural way to represent the designer's conceptual view of the system.
2. *Concurrency* to represent designs which often consist of several communicating, concurrent processes. Statement-level parallelism is required since, even within a process, several actions may take place at the same time.
3. *Programming Constructs* to specify the transformational aspect of the system - the computation

Language	Hierarchy	Concurrency	Actions	Timing	Exception Handling	State–based Specification	Synchronization	Communication	Structural details
VHDL	Declarative Hierarchy	Single level of processes, Statement–level parallelism	Prog. constructs	AFTER clause, WAIT statement	Cumbersome to represent process termination	Cumbersome to represent with prog. constructs	Common Event. Global Signal	shared memory	Yes
HardwareC	Declarative Hierarchy	Single level of processes, Statement–level parallelism	Prog. constructs	Timing constrs. between statements	Cumbersome to represent process termination	Cumbersome to represent with prog. constructs	Global Ports, Message Wait	shared memory message passing	Yes
SDL	Declarative Hierarchy	Single level of processes	Assignments in state transition m/c.	Timeout signals using timers	Cumbersome to represent process termination	Hierarchical State Diagrams	Global Signals	message passing	Yes
Statecharts	Hierarchical Finite State Machines	AND decompostion at any level	Assignments in states and transitions	Timeout arcs, Min/Max time spent in state	Easily handled by transition arcs	Hierarchical State Diagrams	Initialization, Common State, Common Event, Global Variables	shared memory	No
SpecCharts	Behavioral Hierarchy	Concurent sub–behaviors at any level	Prog. constructs	AFTER clause, WAIT statement, Timeout arcs	Easily handled by TI (transition immediately) arcs	State Diagrams	Common Event, Global Signal, Protocol–based Blocking Channels	shared memory message passing	Yes
CSP	Hierarchy of Processes	Dynamic process activation	Prog. constructs	none	none	Cumbersome to represent with prog. constructs	Blocking channels	message passing	No

Figure 4: Evaluating Features of Hardware Description Languages

carried out by an algorithm. A state-transition diagram which supports only simple assignments is cumbersome to use in specifying computations involving statements like loops.

4. *Abstract Communication* constructs to enable the designer to avoid specifying the low-level details like port assignments and to make the specification readable. The *message-passing* model is ideally suited since it represents communication in its most abstract form. At the same time the designer should have the flexibility in deciding how the communication channel is to be implemented by allowing him to associate a protocol with the channel.
5. *Synchronization mechanisms* to allow several concurrent processes to rendezvous with each other over time. Synchronization by common event and common variable are essential for system-level specifications.
6. *Exception Handling* to be able to represent instantaneous response to an external event by terminating all current activities.
7. *Structural Representation* in order for the designer to be able to specify portions of the design using previously designed off-the-shelf components. In addition it gives the language capability of representing system refinement as a result of synthesis.
8. *State-based constructs,* since embedded systems are more easily conceptualized as state-transition diagrams.

We have in this report identified some of the features of specification languages which are well suited for the specification of embedded systems. Each feature recommended above attempts to reduce the amount of detail that a designer must be concerned with during specification and must be an integral part of any system specification language.

5 Acknowledgements

We would like to thank Frank Vahid for his valuable assistance and suggestions. This work was supported by the Semiconductor Research Corporation (grant #91-DJ-146). We are grateful for their support.

References

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

[2] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL : Hardware Description and Design*. Kluwer Academic Publishers, 1989.

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

[4] G. Micheli and D. Ku, "HERCULES - A System for High-Level Synthesis," in *Proceedings of the DAC*, 1988.

[5] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

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

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

[8] D. Drusinsky and D. Harel, "Using Statecharts for Hardware Description and Synthesis," in *IEEE Transactions on Computer-Aided Design*, 1989.

[9] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proceedings of thge ICCAD*, 1991.

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

[11] R. Auletta, "CSP Specified Digital Repeater and Translation to Synchronous Logic." Technical Report, Dept. ECE, George Mason University, 1991.

[12] *Intel Peripheral Design Handbook*, 1981.

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

[14] I-Logix Corporation, "ExpressVHDL: UART Example." VHDL International Users' Forum, 1991.