System Specification and Synthesis with the SpecCharts Language

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

Abstract

There is a need for capturing behavioral specifications of entire systems and obtaining multi-chip designs from those specifications. We discuss system level specification and synthesis issues, along with the unique requirements they place on a specification language. Since no current language meets those requirements, the SpecCharts language was created on top of VHDL. Its model and constructs permit capturing system specifications simply and precisely, while aiding synthesis.

1 Introduction

Traditional high-level synthesis involves tasks such as scheduling and allocation of functional units to convert a behavioral specification to hardware. Before invoking such synthesis, it is often necessary to divide a system's abstract specification into a set of one or more *interconnected chip behavioral descriptions*. We refer to the automating of this task as *System Level Synthesis* (Figure 1).

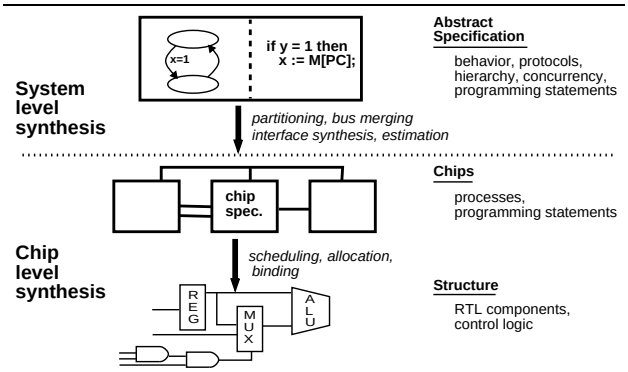

Figure 1: Domain of System Level Synthesis

Most current approaches synthesize logic or structure before chip partitioning, or partition at the control/dataflow graph (CDFG) level. However, as chip capacities as well as system sizes grow, there are several advantages to be gained if partitioning is done using original specification objects (e.g. processes and storage structures like memories). These advantages include making important architectural decisions (bus merges and bus arbiter insertion, parallel/serial communication tradeoffs, memory selection), writing functional test patterns for each chip yielding separately testable chips, limiting the effect of future specification changes, and enabling designer control over all the synthesis tasks.

Current specification languages are unable to support system capture or synthesis satisfactorily. Most hardware description languages [1, 2] contain an underlying model of concurrent processes, which is not sufficient for inherently state-based or behaviorally hierarchical systems. Other languages [3, 4, 5], are either useful only in a very specific domain or require the designer to mentally resolve several views (functional, behavioral and structural) of the same design. None possess appropriate abstract and extensible communication constructs, or provide for designer interaction during synthesis.

In this paper we present the requirements of system level specification languages and discuss the essential synthesis tasks. In the course of modeling several existing chips, we identified several useful constructs and abstractions, which led to the development of the *SpecCharts* language. The language possesses an underlying model which supports behavioral hierarchy and appropriate abstractions suitable for precisely and easily specifying a design through the system and chip levels of synthesis. We demonstrate its usefulness through real examples.

2 Synthesis Requirements and Tasks

The black-box approach to synthesis is unacceptable at the system level since designers are not willing to relegate certain important decisions to a synthesis tool. Thus the results of many tasks (e.g. partitioning) must be fed back to the designer. Our methodology is to reflect the results as a new specification as shown in Figure 2. This approach allows close monitoring and interaction by the designer. It also facilitates partial design – if the language is capable of representing the added detail, the initial specification could have already contained this detail. Finally, 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.

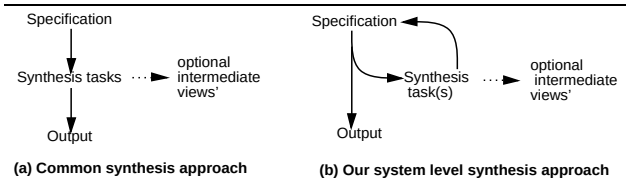

Figure 2: System Level Synthesis Approach

Several tasks must be performed to transform a behavioral system specification to multiple chip specifications. *Partitioning:* distributing specifications among chips to satisfy constraints on design parameters such as chip area, pins, or performance. *Estimation:* providing the partitioner with estimates of the above design parameters. *Interface Synthesis:* maintaining or modifying interchip communication by selecting appropriate protocols, making

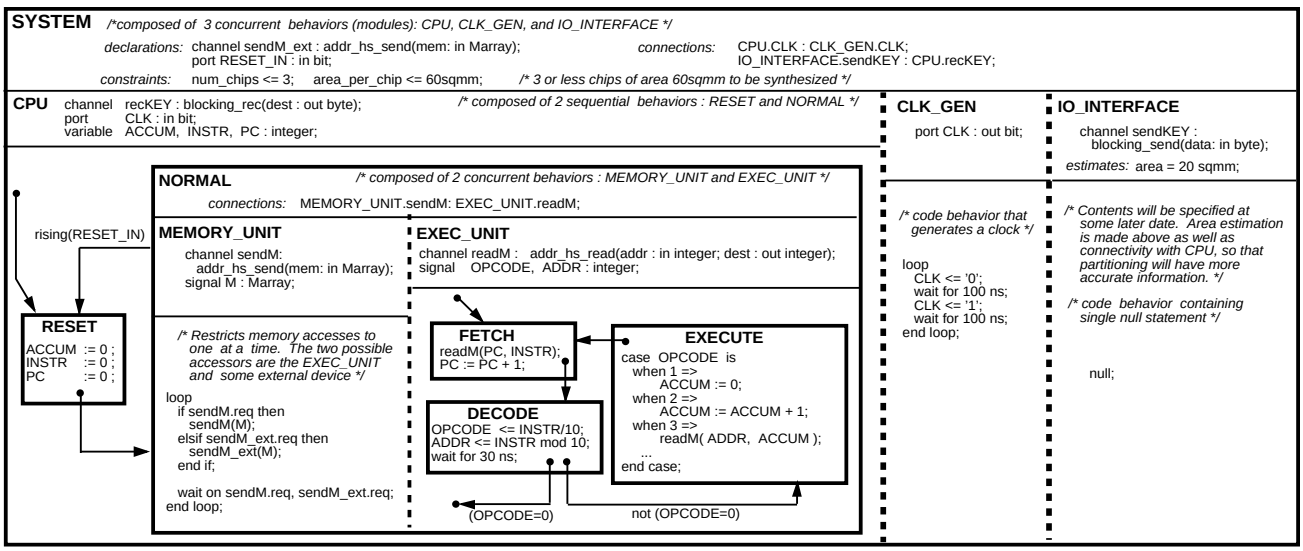

Figure 3: Graphical SpecChart of a simple computer system. Normally, all hierarchical levels would not be viewed simultaneously.

serial/parallel data transfer tradeoffs, or merging buses. *Arbiter Synthesis:* adding arbiters to resolve bus contention issues created by interface synthesis. *Simulation:* verifying proper functionality at any stage of system synthesis. *Providing a path to chip synthesis:* translating to a language (in our case, VHDL) in a manner acceptable to a high-level synthesis tool to synthesize structure for each chip.

The above methodology and tasks place unique requirements on the specification language used. (1) It must be easy to understand, not only to simplify the initial specification, but also so that modifications can be easily understood. This requires: functional and structural hierarchy, abstract communication constructs, and centralized information, as opposed to multiple 'views' of a design. (2) It must be able to represent partial structural information (e.g. chip distribution, buses and bus merges). (3) It must permit partial specification, so that synthesis can still be performed without knowing the details of portions of the specification. (4) It must be executable.

Ideally, the language would capture a designer's conceptualization of a system with minimal designer effort, i.e. capture the system using constructs a system modeler uses, such as flowcharts, transition arcs and programming constructs. Detail is added to the specification at each synthesis step, until enough detail exists to invoke a high-level synthesis tool for each chip.

3 The SpecCharts Language

The SpecCharts language [6], created to satisfy the requirements outlined earlier, can be summarized as a combination of hierarchical/concurrent state diagrams [7] and VHDL. The language, introduced with the example of Figure 3, attempts to exploit the advantages of VHDL (wide acceptability and availability of simulators) by developing a representation on top of it, *not as a replacement for it.*

The basic object in SpecCharts is a *behavior.* A behavior can be expressed in one of three ways:

- As **concurrent sub-behaviors**. In Figure 3, SYSTEM is composed of three concurrent sub-behaviors – CPU, CLK_GEN, and IO_INTERFACE. Concurrent behaviors are often called processes.
- As **sequential sub-behaviors**. This corresponds somewhat to traditional state diagrams. CPU is composed of 2 sequential sub-behaviors, RESET and NORMAL, sequenced by arcs. When the CPU is active, it is always in one of these two *substates.*
- As **VHDL sequential statements**. CLK_GEN and FETCH are **leaf** behaviors since they are not further decomposed into sub-behaviors. When activated, they begin executing their VHDL code.

Graphical conventions used are similar to those presented in [7]. Briefly, behaviors are represented as boxes, sequential behaviors are sequenced by transition arcs and concurrent behaviors are separated by dotted lines.

At any time a behavior is either **active** or **inactive**. An active behavior is either:

- *Executing* – if it is a leaf behavior, it is executing its code; if it is a non-leaf, it is activating/deactivating the appropriate sub-behaviors.
- *Complete* – the behavior has completed execution but has not yet been deactivated by its parent. If it is a leaf, the end of the code has been reached AND all signals assigned in this behavior have received their new values. If it is a non-leaf, all sub-behaviors are inactive and control has flowed to a special 'stop' dot from where no sub-behavior can be reactivated.

Transition arcs are used for sequencing from one behavior (*source*) to another. A **TOC** (Transition-On-Completion) arc is traversed only if the source behavior has *completed* execution and the arc condition is true. A **TI** (Transition-Immediately) arc is traversed the instant its associated condition becomes true, regardless of the execution status of the source behavior and any of its descendant behaviors. In Figure 3, TOC arcs (originating from bold dots within a state) sequence the FETCH, DECODE and EXECUTE behaviors, whereas the TI arc (originating from the state perimeter) labeled "rising(RESET_IN)" puts the CPU in the RESET behavior.

Each behavior may contain declarations (registers, memories, types, procedures etc.) identical to those in VHDL, whose scope is the behavior and any descendant behaviors. The CPU declares the variables ACCUM, INSTR and PC. Interface information may also be declared, such as channel recKEY and port CLK in CPU. Connections between sub-behavior ports/channels can be specified. SYSTEM has a connection specified between the CLK ports on CLK_GEN and CPU. Each behavior may possess predefined attributes. For example, *is_chip* indicates that the behavior describes a single chip. Synthesis constraints can be specified, such as area, pin count, or expected execution time for a behavior. User estimates of design parameters such as area or execution time can be provided for a behavior, thus allowing synthesis tasks such as partitioning to be performed even when the internal details for a portion of the design are not yet available.

```
protocol addr_hs_read ( addr : in integer;
                        dest : out integer)
  /* definition of an addressed handshake
     read protocol */

  port A :    out integer;  -- address bus
  port D :    in integer;   -- data bus
  port REQ : out bit;       -- request line
  port RDY : in bit;        -- ready line

  begin
    A   <= addr ;
    REQ <= '1';
    wait until RDY='1';
    dest <= D;
    REQ <= '0';
  end;
```

```
protocol addr_hs_send ( mem : in Marray)

    /* sender protocol, corresponding to
       addressed handshake read */

  port A :    in integer;   -- address bus
  port D :    out integer;  -- data bus
  port REQ : in bit;        -- request line
  port RDY : out bit;       -- ready line

  begin
    if (REQ='1') then
      D   <= mem(A);
      RDY <= '1';
      wait until REQ='0';
      RDY <= '0';
    end if;
  end;
```

Figure 4: Definition of the two complementary halves of a Memory Read protocol

Concurrent objects such as modules or processes in HDL's usually communicate through shared data (e.g. global signals in VHDL) or ports. While procedures can be used to hide the behavioral aspect of communication from an object, the structural information such as the number of ports and buses for implementing the communication must be explicitly specified and passed as procedure parameters. For example, specifying an off-chip memory read access would require specifying the address/data buses and control signals explicitly. Specifying the structural details detracts from the actual function, i.e. the memory is being read. Using built in constructs such as send/receive [8] neglects the structural aspect entirely and limits the protocols that can be used.

Our solution is to add two constructs, the *channel* and the *protocol*, to the language. Two concurrent behaviors can communicate over a channel. A protocol definition is identical to a procedure except that ports are also declared, and the procedure body describes a behavior over those ports as shown in Figure 4. One then declares a channel at the behavior boundary and associates a protocol with it. In Figure 3, EXEC_UNIT and MEMORY_UNIT both have channels (readM and sendM respectively) declared for communication. The two channels are connected in parent NORMAL's connections just like ports and have associated protocols. Whenever the memory has to be accessed in the EXEC_UNIT (as in sub-behavior FETCH), the readM channel is used as a procedure call with the registers involved in the transfer specified as parameters. The constructs have semantics identical to declaring and passing the ports explicitly.

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 needs to be changed. These simple constructs successfully abstract communication, while being general enough to specify any protocol and structural interconnection. This greatly aids both system specification and synthesis.

4 Underlying Model of SpecChart Specifications

We now discuss the underlying model and constructs of SpecCharts, contrasting them with design representations in VHDL [1] and Statemate [3] (see Figure 5).

We define a system's 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). Conceptually, a behavior may take zero time (e.g. a VHDL variable assignment) or non-zero time (e.g. the FETCH state of a CPU, or the CPU itself). For the latter case, the behavior is active while it is occuring. An active behavior which can no longer change a system value is said to be complete.

We define a language which supports **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 activate/inactivate a sub-behavior at *any* time.
3. *Behavioral Completion* – the ability of a behavior to indicate completion and that of other behaviors to detect this completion. This may be used for sequencing from one sub-behavior to another.

(1) is not supported in VHDL. It provides only one level of concurrency (processes) followed by one level of sequentiality (sequential statements). This is true for most other HDL's and communicating FSM models. (2) is also not provided since there is no construct to reset a process regardless of its execution status, when an event occurs. Note that nested blocks merely provide declarative hierarchy (symbol scoping rules), not a behavioral hierarchy.

In the Statemate environment, three languages are used to represent behavioral, functional, and structural views respectively. The first two combined partially provide for (1). There is limited support for (2), while (3) is unsupported. Also, resolving the three system views into one can be quite awkward.

It is also interesting to consider Communicating Sequential Processes [9], which provides for (1) and (3), but not for (2). This is because the only method of behavioral decomposition is by sequential statements. Process activation is one such statement, but this statement does not possess any semantics for deactivation.

SpecCharts use transition arcs to achieve deactivation of behaviors. These arcs permit a *state-based* description, which is quite common and useful in hardware system specifications. For example the functionality of the DRACO chip [10] was divided into POWER_ON, RESET, WAIT, ADDRESS, READ and WRITE states with appropriate transition arcs between them. Implementing this in an HDL using processes or procedures proved cumbersome since the control had to be implemented explicitly, using some combination of control signals, state registers, case and loop statements.

Hardware systems requiring behavioral hierarchy are abundant (see Figure 5). For example, the Intel 8203 RAM controller consists of several sequential modes, one of which (REFRESH) contains two concurrent behaviors (perform refresh, latch a read request). If the first concurrent behavior completes, the second is deactivated and the next mode is started. This requires (1), (2), and (3). Since behavioral hierarchy is unsupported by most HDL's, modelers often modify the behavior rather than write a correct model (which would become difficult to understand). 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 in [2]). SpecCharts was created to support behavioral hierarchy.

Turning from initial specification to synthesis, all features in Figure 5 have proven useful, except *history* (ability to enter the most recently visited state amongst a group of states) and *multiple behavioral entry points* (ability to enter a state from any level of the hierarchy) . Behavioral hierarchy and state-based specifications were discussed above. A unified view of structure, function and control is essential to describe chips and their interconnections (structure) along with the individual chip behaviors. *Abstract communication* constructs (e.g. channels/protocols in SpecCharts) are useful in system level synthesis as in the case of interfacing a memory and a processor which have been partitioned into separate chips, or when merging communication buses of a channel. The specification is kept readable, and easily modifiable. Concurrent signal assignments are useful to specify hardwirings of registers to ports; the absence of these from SpecCharts makes description of such behavior a bit cumbersome.

	Features	Statemate	VHDL	Spec Charts	*Models for which feature would be useful in the INITIAL specification*
Behavioral Hierarchy	Behavioral Decomposition	✓		✓	b, d, e, f
	Hierarchical Activation/Deactivation	limited support		✓	b, d, e, f
	Behavioral Completion (TOC arcs)			✓	a, b, d, e, f
	Unified view of Structure, Function, & Control	3 languages	✓	✓	c
	Multiple Behavioral Entry points	✓			-
	State based Specification	✓		✓	a, b, d, e, f
	Declarative Hierarchy		✓	✓	a, b, c, e, f
	Structural Hierarchy	✓	✓	✓	-
	Ability to specify Abstract Communication			✓	-
	Concurrent Signal Assignments		✓		b, c, d
	History	✓			-

(a) DRACO - 1781 Discrete I/O Backplane Custom IC
(b) Intel 8237 - Programmable DMA Controller
(c) Intel 8251 - Programmable Communication Interface
(d) Intel 8203 - RAM Controller
(e) SISC Processor
(f) SISC Pipelined Processor

Figure 5: Comparing SpecCharts with Statemate & VHDL

5 Status and Future Work

The SpecCharts language has been implemented. This includes an X widget based graphical interface which outputs a textual SpecChart. It also includes a parser and a large set of query and manipulation routines for use by synthesis tools. Simulation is via automatic translation to VHDL [11], which has also been implemented (translation time is of the order of one second).

Using this platform, we have modeled and simulated several examples, including the Rockwell DRACO I/O chip, SISC processors and several Intel chips (including I8203 Dynamic RAM Controller, I8237 Programmable DMA Controller, and I8251 Programmable Peripheral Interface). Details of these chips can be found in [2, 10, 12].

We have implemented system synthesis algorithms as tools in our system level synthesis environment, including area, pin and clock estimators, and a partitioner which currently permits application of numerous algorithms and manual partitioning, along with simple synthesis of interfaces. Future work includes the addition of performance based partitioning algorithms, and developing interface and arbiter synthesis tools. Also, we will soon have an interface to a VHDL synthesis system [13].

6 Conclusions

The SpecCharts language was the result of an effort to overcome the shortcomings of existing specification methods when applied to large systems. SpecCharts permit concise, understandable, and accurate specification of systems while supporting the concept of behavioral hierarchy, which considerably aided the specification of hardware systems that we modeled. Its constructs aid system level synthesis tasks such as partitioning, estimation, interface synthesis, and bus merging by permitting high level communication, and maintaining information and permitting modification at the level most modelers think at.

7 Acknowledgements

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #90-DJ-146). We are grateful for their support. We would also like to thank Tedd Hadley, Bob Larsen and Rockwell International for their help and suggestions.

References

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

[2] E. Sternheim, R. Singh and Y. Trivedi, "Hardware Modeling with Verilog HDL." Automata Publishing Company, 1990.

[3] D. Harel, et al, "STATEMATE : A Working Environment for the Development of Complex Reactive Systems," in *Proceedings of the Intl. Conf. on Software Engineering*, 1988.

[4] E. Lee, et al, "Gabriel: A Design Environment for Programmable DSP's," in *Proceedings of the 26th Design Automation Conference*, 1989.

[5] W. Birmingham, A. Gupta and D. Siewiorek, "The MICON System for Computer Design," in *Proc. of the 26th Design Automation Conference*, 1989.

[6] F. Vahid, S. Narayan, D. Gajski, "SpecCharts : A Language for System Level Synthesis," in *Proc. of the 10th Intl. Symposium on Computer Hardware Decription Languages*, 1991.

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

[8] G. Micheli and D. Ku, "HERCULES - A System for High-Level Synthesis," in *Proceedings of the 25th Design Automation Conference*, 1988.

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

[10] S. Narayan, F. Vahid, and D. Gajski, "Modeling With SpecCharts." UC Irvine, TR 90-20, July 1990.

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

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

[13] J. Lis and D. Gajski, "Synthesis from VHDL," in *Proceedings of the International Conference on Computer Design*, 1988.