

UC Irvine

ICS Technical Reports

Title

A survey of system-level specification languages

Permalink

<https://escholarship.org/uc/item/24g9x7rd>

Author

Narayan, Sanjiv

Publication Date

1992-10-09

Peer reviewed

ARCHIVES

Z

699

C3

no. 92-100

A Survey of System-Level Specification Languages

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Sanjiv Narayan

Technical Report #92-100

October 9, 1992

Dept. of Information and Computer Science

University of California, Irvine

Irvine, CA 92717

(714) 856-8059

narayan@ics.uci.edu

Abstract

As synthesis tools become more advanced and reliable, the entry point for the design tools in the design process is moving towards higher levels of specification. In this report, issues related to the specification of reactive, transformational systems are discussed. Several existing system specification languages are examined and their capabilities with respect to specifying designs at the system-level are compared.

A Survey of System-Level Specification Languages

Sanjiv Narayan

Technical Report #92-100

October 9, 1992

Dept. of Information and Computer Science

University of California, Irvine

Irvine, CA 92717

(714) 856-8059

narayan@ics.uci.edu

Contents

1	Introduction	3
1.1	System-Level Designs	3
1.2	Motivation	4
1.2.1	Capturing Designer's Conceptual View	5
1.2.2	Managing Design Complexity	5
1.2.3	Representing Interprocess Communication	5
1.2.4	Specification refinement / Designer interaction	6
2	Specification Languages	7
2.1	HardwareC	7
2.2	VHDL	10
2.3	Statecharts	14
2.4	SpecCharts	15
2.5	SDL	21
2.6	CSP	23
2.7	Silage	25
3	Evaluating Specification Languages	26
3.1	Hierarchy	26
3.1.1	Behavioral Hierarchy	27
3.1.2	Structural Hierarchy	29
3.2	Concurrency	29
3.2.1	Process-level parallelism	29
3.2.2	Statement-level parallelism	30
3.3	Timing	31
3.4	Synchronization	32

3.4.1	Synchronization by Initialization	32
3.4.2	Synchronization by Common Event	33
3.4.3	Synchronization by State Detection	34
3.4.4	Synchronization by Common Variable	34
3.4.5	Synchronization by Communication	34
3.5	Interprocess Communication	35
3.5.1	Shared Memory	35
3.5.2	Message Passing	36
3.6	Exception Handling	37
3.7	Specifying Actions using Programming Constructs	38
3.8	Other Features	39
3.8.1	State-Based Specifications	39
3.8.2	Dataflow Semantics	39
3.8.3	Designer Interaction for Synthesis	40
4	Conclusion	40
5	Acknowledgements	44
6	References	44

List of Figures

1	Domain of System Level Synthesis	3
2	Tradeoff between Degree of Expressivity and Ease of Implementation	4
3	Specification refinement during synthesis	6
4	HardwareC : Communication between processes	8

5	HardwareC : Compound statement types, (a) Serial compound statement (b) Data-parallel compound statement and (c) Parallel compound statement	9
6	VHDL : Design Entity Block structure	11
7	VHDL : Description styles, (a) Behavioral, (b) Dataflow and (c) Structural.	12
8	Statecharts : Partial specification of the UART	14
9	Block diagram of the answering machine phone.	16
10	SpecCharts : Partial specification of the answering machine phone. Not all the hierarchical levels are shown.	17
11	SpecChart Leaf behavior MONITOR of the answering machine	18
12	SpecCharts : Definition of the two complementary halves of a Memory Read protocol	19
13	SDL : Hierarchical decomposition of specifications	21
14	SDL : Partial specification of the answering machine	22
15	Concurrency specification in Statechart for the UART	30
16	Synchronization methods in Statecharts	33
17	Synchronization by Common Event in VHDL	34
18	Specifying actions in the Receiver Process of the UART	38
19	Evaluating Features of Hardware Description Languages	41

1 Introduction

Many tools and environments have been developed to aid system designers achieve the implementation of a design. Most are intended to implement the lower level details of the design, such as layout (placement, routing, transistor sizing, partitioning). Tools have also been developed to reduce design effort at the logic and microarchitectural levels, including optimizers and schematic synthesizers. At an even higher level, tools and environments exist to aid in the design of structure from a register transfer level description (such as one described using a hardware description language).

1.1 System-Level Designs

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. Synthesis tasks such as scheduling and allocation of functional units to convert a behavioral specification to hardware are commonly referred to as high-level synthesis. 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).

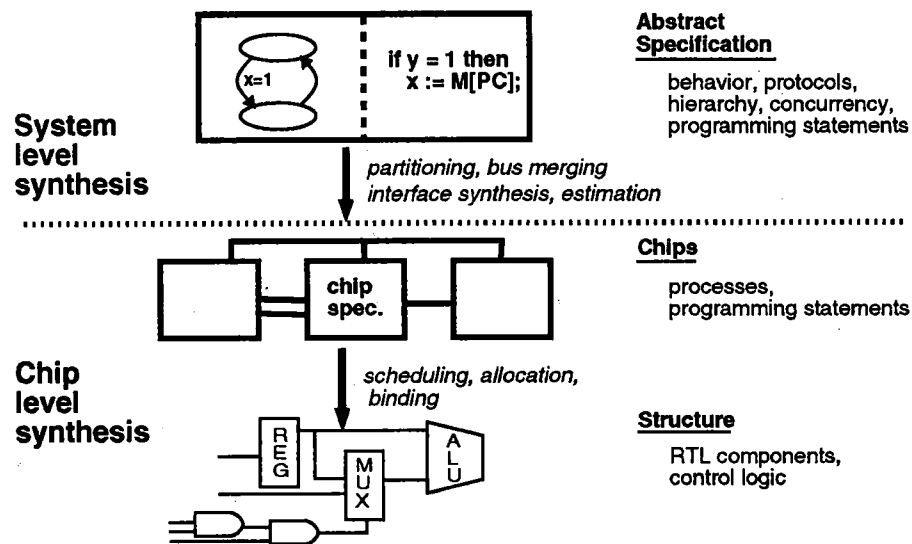


Figure 1: Domain of System Level Synthesis

By system-level designs, we are referring to *reactive, transformational* systems. A reactive system [1] is one which is essentially event-driven and has to respond continuously to external and internal

stimuli. A transformational system is one in which an algorithm performs operations on a set of input data items and produces output data. Examples of such systems are computer systems and telecommunication systems.

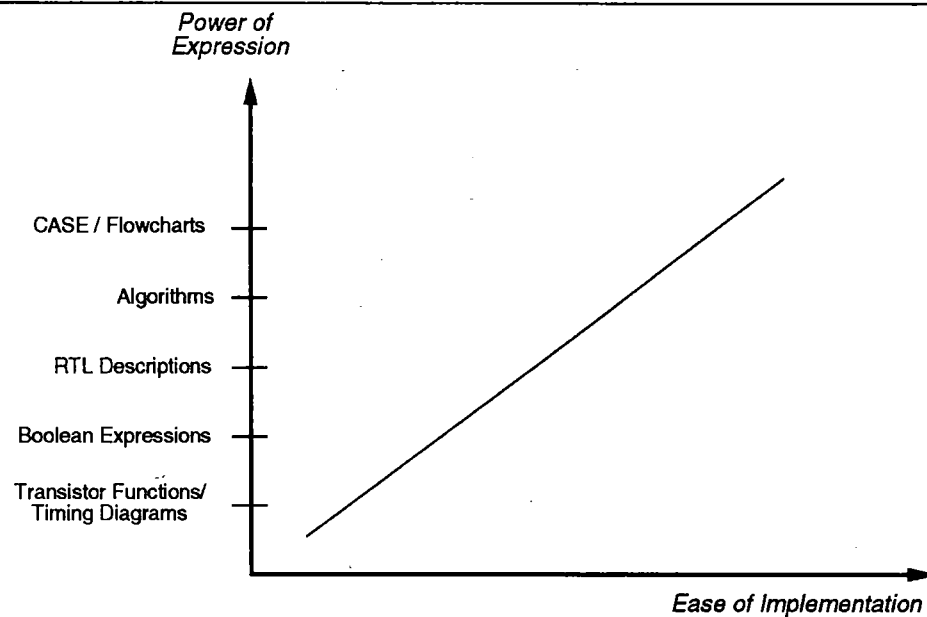


Figure 2: Tradeoff between Degree of Expressivity and Ease of Implementation

1.2 Motivation

Any specification language is essentially a tradeoff between the *degree of expressivity* and the *ease of implementation* (Figure 2). For example, a system can be described using a structural netlist of components. While the designer is limited to specifying his design with a fixed set of components and their interconnections, there are a lot of tools available which can generate a layout from a given netlist. On the other hand, if the designer specifies the design in English, he has virtually unlimited power of expression, but it would be impossible to synthesize a design automatically from such a description. Figure 2 illustrates this tradeoff between the power of expression and ease of implementation.

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. To be useful, a feature of the language should greatly reduce the effort required by the designer to capture a particular aspect of the design (e.g., communication, timing, etc.), while *not* sacrificing the implementability of a design specified in that language. Addressing this question is one of the main goals of this report.

In order to specify designs at the system-level, the specification language must have the capability to address several issues, which are discussed individually below.

1.2.1 Capturing Designer's Conceptual View

The primary aim of a system-level 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, a reactive system 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 designer can directly and easily translate his conceptual view. This not only aids the initial specification task, but also makes the description more readable during subsequent stages of the system design process.

1.2.2 Managing Design Complexity

Describing entire systems will require some mechanism for structuring the specification so as to be able to enter, maintain and modify it in a reasonable manner. Some form of a design *hierarchy* which enables successive decomposition of a system into smaller and smaller units of specification is essential. Design complexities can also be contained by permitting specification of *concurrency*. This not only allows a more natural representation for a hardware system which has several actions occurring in parallel, but also prevents an exponential increase in the specification size if the designer was restricted to using a language without constructs for concurrency.

1.2.3 Representing Interprocess Communication

Most systems typically consists of several activities occurring concurrently. These "chunks of actions" or *processes* may need to transfer data between themselves. In addition, two or more processes often need to *synchronize* with each other (as when exchanging data amongst them). The specification language must provide appropriate communication constructs and synchronization mechanisms.

1.2.4 Specification refinement / Designer interaction

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 synthesis tasks (e.g. partitioning the specification into chips) must be fed back to the designer. The language must have the capability of representing the details added at each synthesis step. An example of such a detail is the representation of structural information which results from design activity. 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, as shown in Figure 3. 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. Finally, since the language is capable of representing the added detail, the designer could have directly included the details in the initial specification, thus supporting partial design.

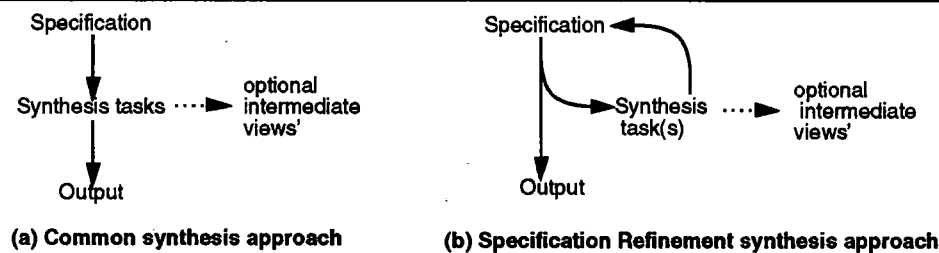


Figure 3: Specification refinement during synthesis

This survey examines several of the specification languages, comparing their capabilities with respect to specifying designs at the system-level. The languages that we will examine encompass a broad range of description styles and application domains, and are fairly representative of design specification methods currently in use. These languages are HardwareC [2, 3], VHSIC Hardware Description Language (VHDL) [4, 5], Statecharts [1], Specification and Description Language (SDL) [6], Silage [7], SpecCharts [8, 9] and CSP [10]. In the next section, we will briefly introduce each of the above specification languages and present their major constructs. In Section 3, we will evaluate some of the features of these languages and the applicability of the languages for system-level specifications. Our conclusions are presented in Section 4.

2 Specification Languages

2.1 HardwareC

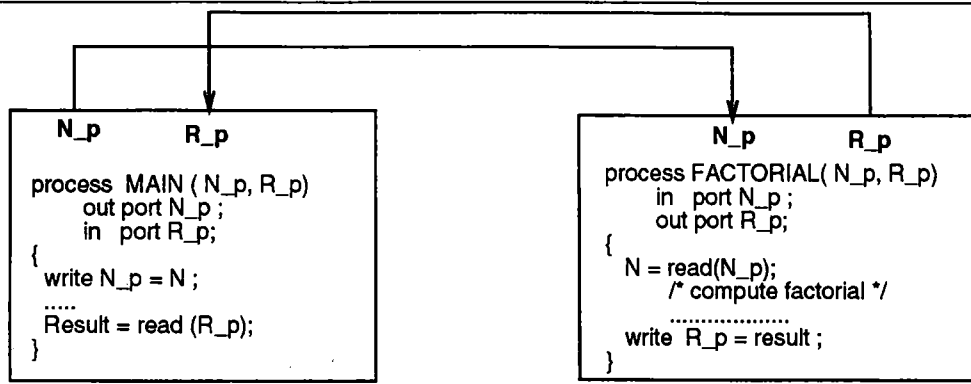
HardwareC [2, 3] was designed expressly to be a *synthesis-oriented* hardware description language. While HardwareC is based on the C Programming language [11], it has its own well-defined semantics and constructs for hardware description.

A HardwareC specification of a system consists of a collection of concurrent processes which communicate with each other. Processes can be enclosed within a hierarchy of blocks. Blocks can be used to define structural relationships between the processes. A process specifies an algorithm as a set of sequential operations described using a subset of programming constructs of the C language. Each process restarts itself upon completion.

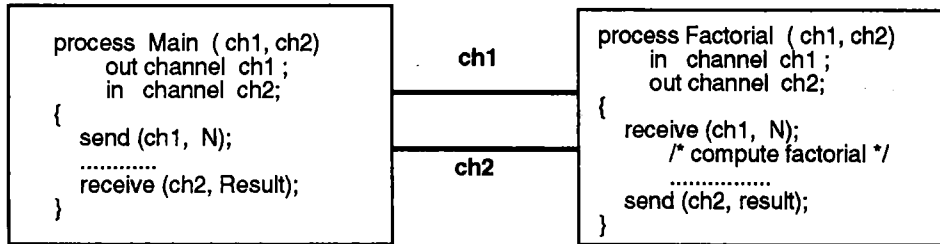
There are three main variable types in HardwareC - *integer*, *boolean* and *static*. Integer variables must be scalar and may be used in expressions, loop indices and accessing slices of a boolean vector. Integer variables are valid only when their values can be resolved at compile time, i.e. their values should be statically determinable. Thus, integers in HardwareC are very similar to constants. Boolean variables (e.g. bit vectors) are used to hold results of computations performed by the algorithm. A boolean variable may be implemented as a wire or a register, based on decisions made by the synthesis tools. The value of a boolean variable is not retained across procedure invocations. Static variables are identical to boolean variables except that they are implemented as registers in the resulting hardware. Using static variables, the designer has the flexibility of specifying explicitly to the synthesis tools what is to be implemented as storage, while at the same time leaving the implementation of boolean variables to the synthesis tool.

All operations in a process are assumed to be synchronized by a single-phase system clock and take an integral number of clock cycles to execute. All I/O operations, message passing and register loading is performed synchronously.

Communication between processes can be specified in one of two ways as shown in Figure 4. In *port passing*, the designer defines global ports on the boundaries of the communicating processes or the enclosing blocks. The protocol which governs the communication is specified as part of the process



(a) Port passing



(b) Message Passing over channels

Figure 4: HardwareC : Communication between processes

description. In Figure 4(a), process **MAIN** provides the value **N** to process **FACTORIAL** over port **N_p** and receives the result over port **R_p**. Explicit commands exist in HardwareC to read, write and tristate a global port. Port passing assumes the existence of a shared medium such as wires or a memory that provides the interconnection between the declared ports. Local ports (similar to the parameters of a procedure) can also be declared for communicating across the calling hierarchy involving procedures and function calls.

In *message passing*, explicit *send/receive* constructs are used for data transfer and synchronization. This is achieved by declaring communication channels between the communicating processes/blocks. Communication over channels is synchronous and blocking. The designer need only specify the data that is to be transferred across the channel - the handshaking protocol and the corresponding hardware is automatically synthesized by the synthesis tools. HardwareC provides constructs for detecting pending messages on channels and specifying interconnections between channels. In Figure 4(b), process **MAIN** uses channel **ch1** to send the value **N** and receives the result over channel **ch2**.

Since the language was targeted primarily towards synthesis, it provides the designer flexibility

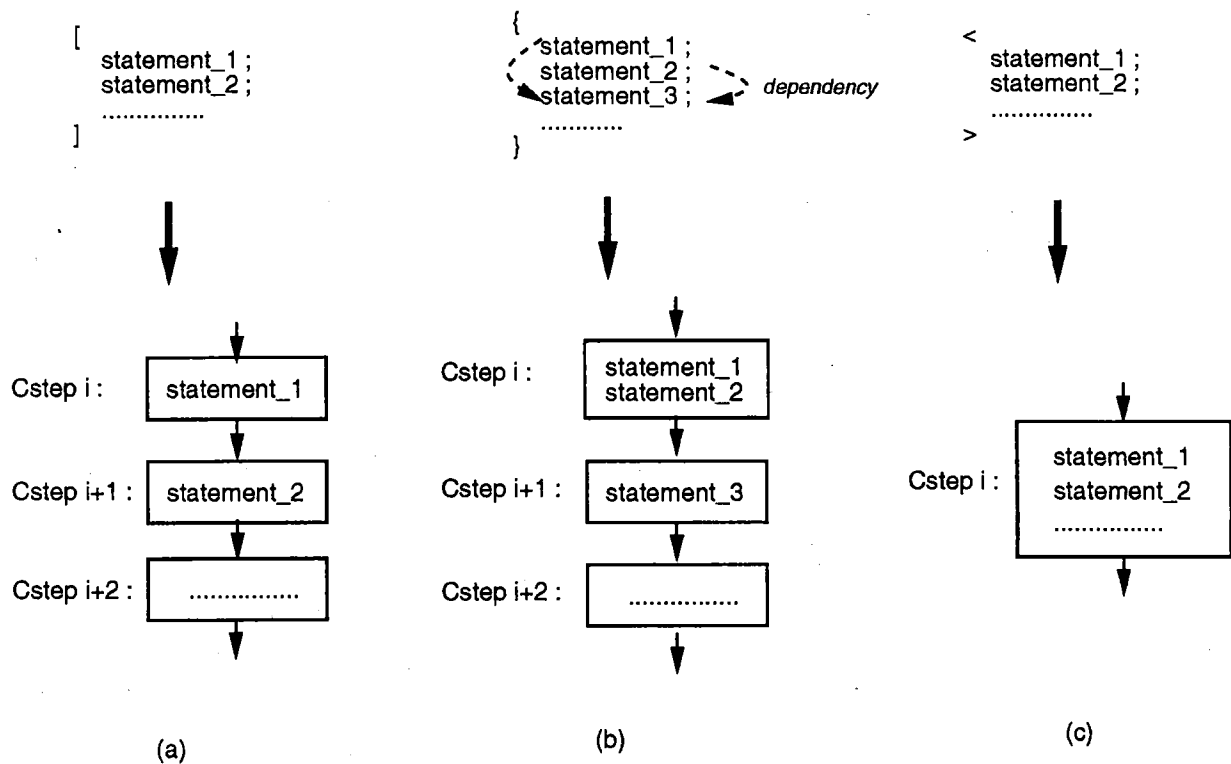


Figure 5: HardwareC : Compound statement types, (a) Serial compound statement (b) Data-parallel compound statement and (c) Parallel compound statement

with respect to specifying the relative occurrences of operations in a process. The designer can group statements into a compound statement in one of three ways :

1. *Serial Compound Statement* : All statements within a serial compound statement are guaranteed to executed in serial order, regardless of the data-dependencies between them.
2. *Data-parallel Compound Statement* : All statements within a data-parallel compound statement can possibly execute concurrently, subject to the data-dependencies between them.
3. *Parallel Compound Statement*: All statements within a parallel compound statement are guaranteed to executed concurrently.

Figure 5 shows the order of execution for statements in the three types of compound statements. For the sake of simplicity it is assumed that each of the statements requires exactly one clock cycle (or control-step) to execute. In Figure 5(b), there is a dependency between *statement_1* and *statement_3* and between *statement_2* and *statement_3*. Thus, *statement_3* is executed in the next control-step

after execution of *statement_1* and *statement_2*.

Another interesting feature of HardwareC is the capability to specify parameterized descriptions or *templates* for blocks, processes, procedure and functions. A template can be instantiated by providing integer values for its formal parameters. Templates can be used to describe library components such as adders and multipliers with formal parameters that may relate to the bit width, number of inputs, etc. The designer can also specify binding information which can be used by synthesis tools. This is achieved by instantiating a template and invoking a *specific* instance of it. For example, a particular add operation could be bound to a specific instance of an adder template. Thus, the designer has the flexibility to specify resource sharing at the description level.

HardwareC also allows the designer to specify timing and resource constraints. Timing constraints are represented as minimum and maximum delays between two statements. Resource constraints are specified by indicating the maximum the number of instances of a particular resource that can be used.

2.2 VHDL

The VHSIC Hardware Description Language (VHDL) [4, 5] is intended to be a standard language for the development, synthesis, verification and documentation of hardware designs. Adopted as an IEEE standard, it has a wide range of tools available for design tasks such as simulation and synthesis.

The primary hardware abstraction in VHDL is a *design entity*, which represents a portion of the design which performs a specific functions and possesses well-defined inputs and outputs. A design entity may represent any abstraction level from a logic gate to a complete system. A design entity can be described in terms of a hierarchy of blocks, each of which represents a portion of the whole design. The top-level block is the design entity itself. The design entity can also be thought of as an *external* block which can reside in a library and may be instantiated as a component in other entities. Nested blocks in the hierarchy are *internal* blocks - defined by block, process or component instantiation statements. This design hierarchy is shown in Figure 6 [12].

The relationship between the inputs and outputs of a design entity is specified in an associated

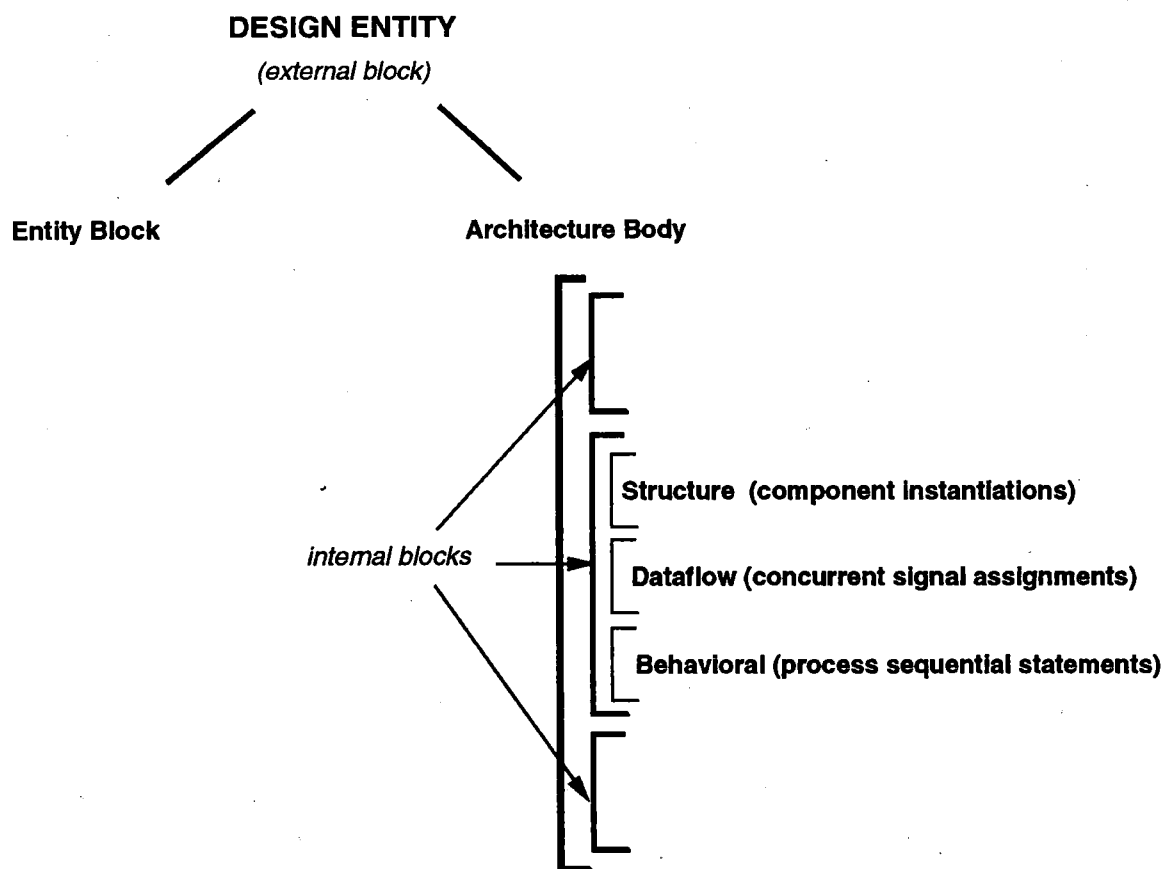


Figure 6: VHDL : Design Entity Block structure

architecture body. This relationship may be described in terms of *behavior*, *dataflow* or *structure* or any combination thereof.

Behavioral Description

Behavioral descriptions have an *imperative semantics*, i.e., they simply specify the relationship between the input and output values using some abstract algorithm. System behavior can be described with VHDL *processes* and *concurrent procedure calls*. Each process can be viewed as a program, with sequentially ordered statements found commonly in programming languages such as Pascal or C. A concurrent procedure call is equivalent to a process statement containing the corresponding sequential procedure call.

All processes in a model execute concurrently. Processes communicate with each other through *signals*. Signals are different from variables in that they have their value defined over time. Thus, in addition to their current value, they have an associated projected output waveform. If several

```

entity COUNTER_E is
  port (CLK : in bit ; CNT : out integer);
end COUNTER_E;

architecture COUNTER_BEH of COUNTER_E is
begin
  process
  begin
    wait until (CLK='1') and not(CLK'stable);
    if (CNT = 9) then
      CNT <= 0;
    else
      CNT <= CNT + 1;
    end if;
  end process;
end COUNTER_BEH ;

```

(a)

```

entity COUNTER_E is
  port (CLK : in bit ; CNT : out integer);
end COUNTER_E;

architecture COUNTER_DFLOW of COUNTER_E is
begin
  block ( (CLK='1') and not(CLK'stable) )
  begin
    CNT <= guarded
      0 when (CNT=9) else
      CNT + 1 ;
  end block ;
end COUNTER_DFLOW ;

```

(b)

```

entity COUNTER_E is
  port (CLK : in bit ; CNT : out integer);
end COUNTER_E;

architecture COUNTER_STRUCT of COUNTER_E is

  component REG_C
    port ( D : in integer ; CLK : bit ; Q : out integer ) ;
  end component ;

  component ADD_C
    port ( A, B : in integer ; O : out integer ) ;
  end component ;

  component MUX_C
    port ( I0, I1 : in integer ; SEL : in bit ; O : out integer ) ;
  end component ;

  signal ONE : integer := 1 ;
  signal ZERO : integer := 0 ;
  signal CNT_IN, CNT_OUT, ADD_OUT : integer ;
  .....
begin
  CONREG : REG_C
    port map ( CNT_IN, CLK, CNT_OUT ) ;
  ADDER : ADD_C
    port map ( CNT_OUT, ONE, ADD_OUT ) ;
  MULTIPLEXER : MUX_C
    port map ( ZERO, ADD_OUT, COMP_OUT, CNT_IN ) ;
  .....
  CNT <= CNT_OUT ;
end COUNTER_STRUCT ;

```

(c)

Figure 7: VHDL : Description styles, (a) Behavioral, (b) Dataflow and (c) Structural.

processes assign a value to the same signal, a *resolution function* is needed to resolve the multiple values into a single value. Signals can be read by any number of processes.

A process can either be 'active' when it is executing one the above statements or it can be 'suspended' by execution of the "wait" statement. The wait statement can be used in several ways:

1. *Delay specification*, as in wait for 20 ns;.
2. *Synchronization*, as in wait until (RESET = '1'), or wait on START.

The behavioral description of a simple modulo-10 counter is shown in Figure 7(a). The counter has two external ports - *CLK* and *CNT*. The counter counts up at every rising edge of the clock signal, *CLK*, resetting to 0 whenever the count reaches 9. The architecture body is described using a process.

Dataflow Description

The dataflow description style allows for the specification of concurrent events under the control of some synchronous/asynchronous signals. A dataflow description can be specified in VHDL using a *concurrent signal assignment* which assigns a waveform to a signal based on a set of conditions or set of control expressions. For every concurrent signal assignment, there exists an equivalent process which assigns different values to signals under certain conditions as specified in the right hand side of (i.e. the transform) of the concurrent signal assignment statement. A concurrent signal assignment is sensitive to the signals in the transform. The *dataflow semantics* of the VHDL concurrent assignment statement essentially defines the paths taken by input values and the intermediate results without indicating any sequencing or ordering between the computations.

An example of a dataflow style description of the simple counter is given in Figure 7(b). A 'guarded' concurrent signal assignment is used to update the value of CNT appropriately only on the rising edge of the *CLK*.

Structural Description

The behavioral and dataflow styles do not specify any structural information about the implementation of the design. The *declarative semantics* (i.e. the specification consists of a set of interconnected components) of VHDL allows a structural specification through the use of *component instantiations* and *generate* statements. The designer can instantiate components which represent previously defined design entities and specify their interconnections by associating ports on the components to the same signal. The generate statement provides a mechanism for iterative elaboration of portion is of the description which can viewed as multiple instances of the same sub-component.

The structural description of the counter is given in Figure 7(c). The components which will be used are declared with the formal port parameters in the architecture declaration portion (e.g., *REG_C*, *ADD_C* and *MUX_C*). The components are instantiated in the architecture body (e.g. *CONREG* and *ADDER*). Association of component ports with signals is used to specify the interconnection between the components. For example, the signal *CNT_OUT* is mapped to the formal ports *Q* and *A* of the components *CONREG* and *ADDER* respectively, thus specifying that the output of the register *CONREG* is connected to one of the inputs of *ADDER*.

2.3 Statecharts

Statecharts [1, 13] was designed primarily for specifying *reactive systems*, which are essentially event-driven, control-dominated systems. Examples of reactive systems are telephones, avionics systems and communication networks. The Statecharts language is introduced with the example of Figure 8 which represents a universal asynchronous receiver-transmitter (UART) [9].

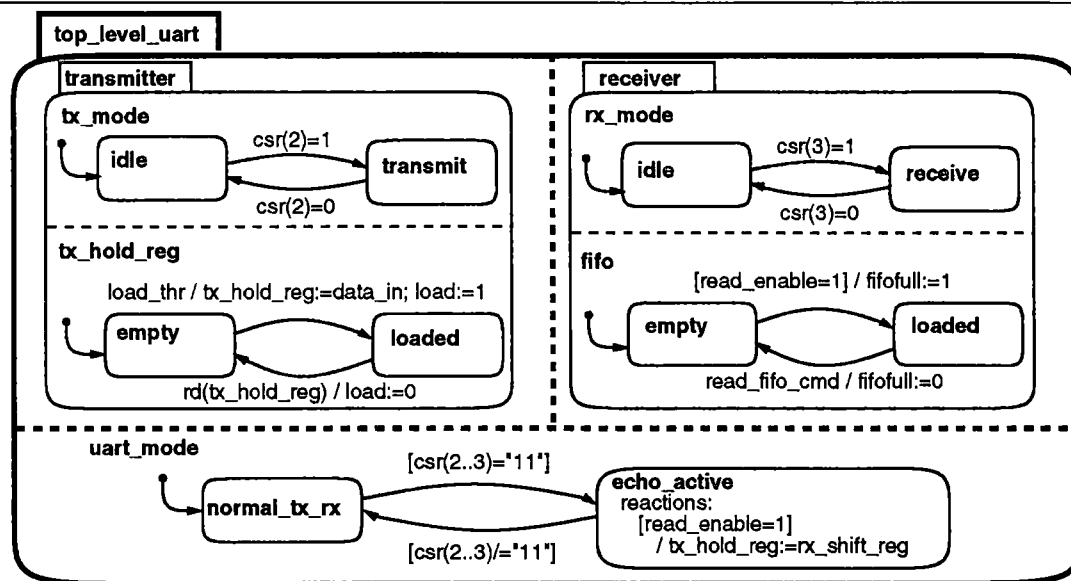


Figure 8: Statecharts : Partial specification of the UART

Finite state machines (FSM) are inherently flat and sequential which limit their application to complex systems. Statecharts provide an elegant representation which extends traditional FSM's by adding three elements - hierarchy, concurrency and communication. The basic object in Statecharts is a *state*. Transitions between states occur based on a combination of events and conditions.

Statecharts allow any specification to be decomposed into a hierarchy of states in one of two ways :

- *OR-decomposition* : A state can be composed of a state machine with states and arcs. In Figure 8, state **tx_mode** consists of two sequential substates **idle** and **transmit**. Being in state **tx_mode** means being in only one of the two substates - **idle** or **transmit**. In addition, **idle** is specified as the default initial state which is entered whenever parent state **tx_mode** is entered.
- *AND-decomposition* : A state can be composed of orthogonal states, where being in the state means being in all the orthogonal states at the same time. Orthogonal or *concurrent* states are

separated by dotted lines. In Figure 8, state `top_level_uart` consists of three concurrent states : `transmitter`, `receiver` and `uart_mode`. Whenever the state `top_level_uart` is entered, all the three concurrent substates are also entered.

By providing for decomposition of a state into sequential and concurrent substates, Statecharts avoids the exponential blowup of states in conventional FSM's, which are inherently flat and sequential. This allows complex reactive systems to be specified in a concise and easy to understand manner.

Transitions in Statecharts are not level-restricted in that transitions between states at different hierarchical levels can be specified. A special transition arc, the *timeout* transition, allows the specification of lower and upper bounds on the time that the system can spend in the desired state.

Along with expressions and conditions, actions can be associated with arcs. These actions represent zero-delay computations. In Figure 8, the transition in state `fifo` from substate `empty` to `loaded` sets the `fifo_full` signal. Actions may also be performed when entering or exiting a state, such as the assignment to `tx_hold_reg` in state `echo_active`.

Broadcast communication is achieved by permitting events, conditions, and actions that can check the status of other states and variables or can start/stop other states. An interesting feature of the Statecharts language is the variety of methods available for synchronizing concurrently executing state machines. These methods are discussed in detail in Section 3.4.

Statecharts also support the concept of a system's *history* as a way of entering a group of states. 'Enter-by-history' enters a group of states at that substate which was most recently visited.

2.4 SpecCharts

The SpecCharts language [14, 8], developed for system specification capture, combines the three aspects of system specification (control, behavior, and structure) into a single, unified concept. The language possesses abstractions which enable the designer to easily and concisely represent his/her conceptual view of the design. It can be summarized as a combination of hierarchical/concurrent state diagrams [1] and VHDL.

We shall introduce the SpecCharts language by means of an answering machine phone example [9]. The block diagram of the answering machine is given in Figure 9 and a partial SpecChart is given in Figure 10.

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

- As **concurrent sub-behaviors**. In Figure 10, the behavior `ans_mc_phone` is composed of three concurrent sub-behaviors : `ans_machine`, `auto_dialer` and `phone_memory`. All the three sub-behaviors are activated simultaneously when `ans_mc_phone` is activated. Concurrent behaviors can exist at any level of the hierarchy and are often called processes.
- As **sequential sub-behaviors**. This corresponds somewhat to traditional state diagrams. The behavior `answer` is composed of five sequential sub-behaviors: `play_announcement`, `record_msg`, `hangup`, `fwd_message` and `remote_operation`, sequenced by arcs. When the behavior `answer` is active, it is always in *one* of these five *substates*.
- As **program code (leaf behaviors)** using VHDL sequential statements. `Reset_state` and `monitor` are leaf behaviors since they are not further decomposed into sub-behaviors. When activated, they begin executing their VHDL code.

In SpecCharts, each behavior may contain declarations (signals, variables arrays, types, procedures, etc.) identical to those in VHDL, whose scope is the behavior and any descendant behaviors. Interface information such as external ports and communication channels between concurrent sub-behaviors may

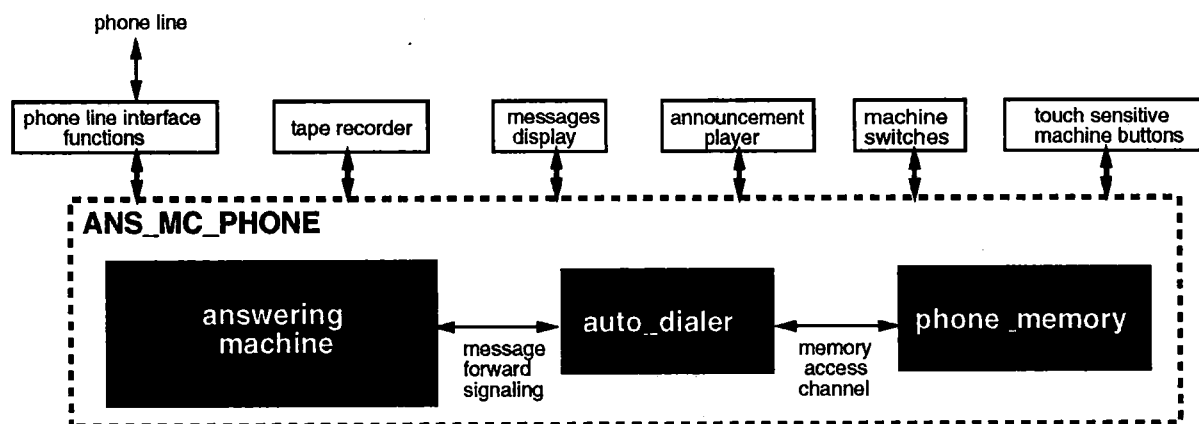


Figure 9: Block diagram of the answering machine phone.

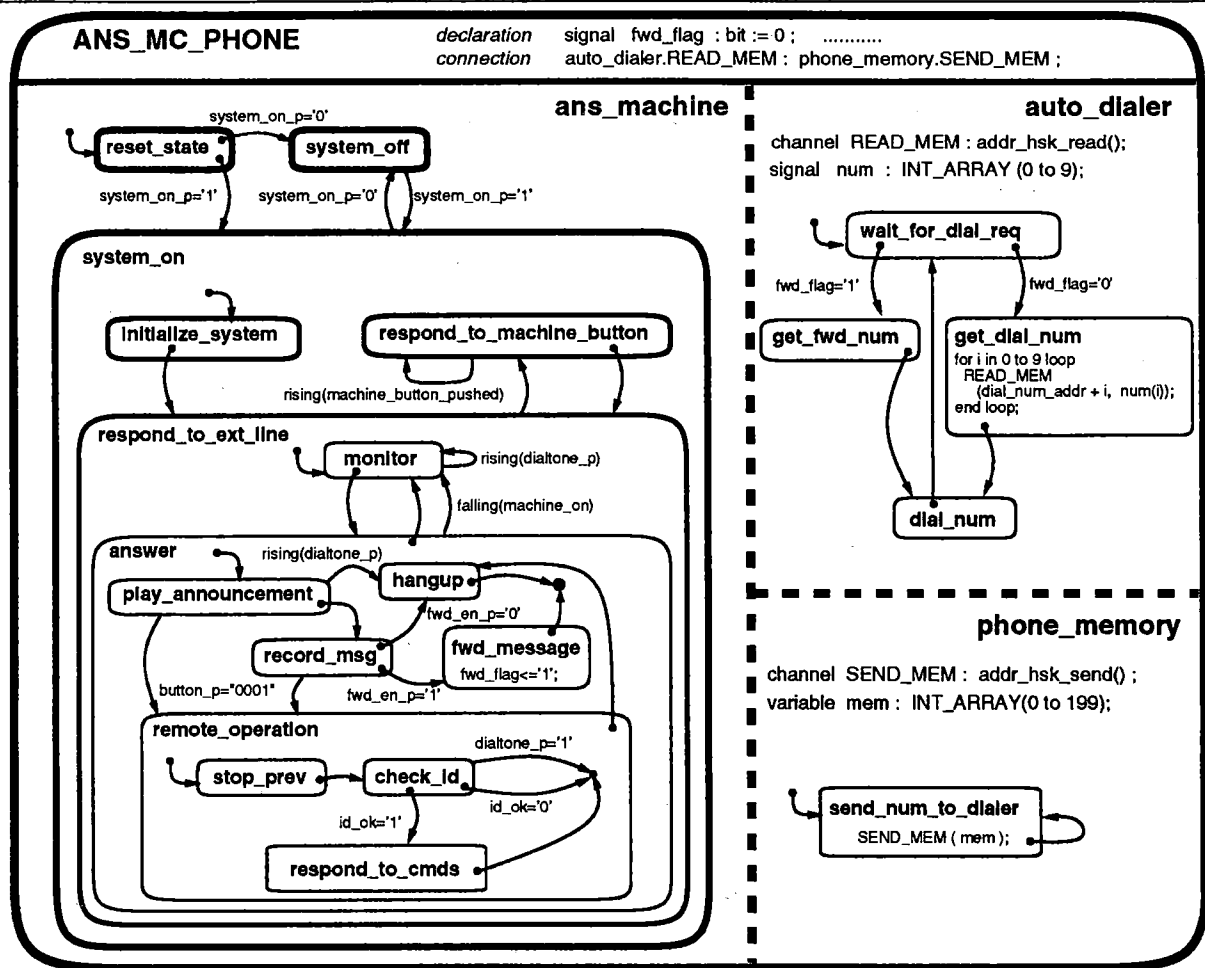


Figure 10: SpecCharts : Partial specification of the answering machine phone. Not all the hierarchical levels are shown.

also be specified. For example, the behavior *auto_dialer* has declarations for signal *num* and channel *READ_MEM*.

Each behavior may possess predefined attributes. For example, the “chip” attribute in a behavior indicates that the behavior and all its descendant sub-behaviors describe a single chip.

In Figure 10, most declarations and details of leaf behaviors have been omitted for reasons of clarity. Figure 11 shows the details of the *monitor* leaf behavior, which monitors the phone line and “answers” only when the required number of rings are detected. We have labeled all external ports with the suffix “_p”. Graphical conventions used are similar to those presented in [1]. Briefly, behaviors are represented as boxes, sequential behaviors are sequenced by transition arcs and concurrent behaviors are separated by dotted lines.

```

name {MONITOR}
declarations
{
    variable num_rings: integer range 1 to 20;
    variable i: integer range 0 to 20;
    procedure determine_num_rings( num_msgs : in integer ; tollsaver, machine_on : in bit ;
                                   variable num_rings : out integer ) is
    begin
        if ((num_msgs > 0) and (tollsaver = '1') and (machine_on = '1')) then
            num_rings := 2;                                -- toll saver mode
        elsif (machine_on = '1') then
            num_rings := 4;                                -- standard machine "on" mode
        else
            num_rings := 15;                               -- override in case machine not turned "on"
        end if;
    end;
}

code
{
    determine_num_rings(num_msgs, tollsaver_p, machine_on, num_rings);

    i := 0;
    while (i < num_rings) loop
        wait on ring_p, machine_on, tollsaver_p;
        if (ring_p = '1' and ring_p'event) then            -- count rings
            i := i + 1;
        elsif (machine_on'event or tollsaver_p'event)      -- machine is switched on / tollsaver enabled
                                                         -- recompute number of rings to be counted
        then
            determine_num_rings(num_msgs, tollsaver_p, machine_on, num_rings);
        end if;
    end loop ;

    offhook_p <= '1';
}

```

Figure 11: SpecChart Leaf behavior MONITOR of the answering machine

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 behavior, 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 10, a **TOC** arc (originating from bold dots within a state) in behavior **system_on** causes a transition from **initialize_system** to **respond_to_ext_line**, whereas the **TI** arc (originating from the state perimeter) with the associated condition “*system_on_p = '0'*” changes the **answering_machine** from the **system_on** to the **system_off** behavior. Another **TI** arc with condition “*rising(dialtone_p)*” causes the **monitor** behavior to restart execution from the beginning of its code. A special **TI** arc called the *timeout arc* causes a transition on the expiry of the associated time period. We have found that the ability to represent completion and termination is very useful to achieve simple system specifications.

<pre> protocol ADDR_HSK_READ (addr : in integer; dest : out integer) /* definition of an handshake read protocol involving an address */ 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; </pre>	<pre> protocol ADDR_HSK_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 wait until (REQ='1'); D <= mem(A); RDY <= '1'; wait until REQ='0'; RDY <= '0'; end; </pre>
--	--

Figure 12: SpecCharts : Definition of the two complementary halves of a Memory Read protocol

Communication between concurrent behaviors in the SpecCharts language is achieved by two constructs, the *channel* and the *protocol*, to the language. Two concurrent behaviors can communicate over a channel which has an associated protocol. 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 12. To specify communication, we declare a channel at the behavior boundary and associate a protocol with it. For two behaviors to communicate, the appropriate channels on both behaviors are connected together. In Figure 10, the concurrent behaviors **auto_dialer** and **phone_memory** both have channels (**READ_MEM** and **SEND_MEM** respectively) declared for communication. The two channels are connected in parent **ans_mc_phone** just like ports and have associated protocols.

Whenever the `auto_dialer` needs to access the memory to fetch a previously stored number, as in sub-behavior `get_num_from_mem`, the `READ_MEM` channel is used as a procedure call with the appropriate data involved in the transfer specified as parameters. The constructs have semantics identical to declaring and passing the ports explicitly.

It must be mentioned at this stage that a global signal can also be used for communicating between concurrent behaviors. In Figure 10, the signal `fwd_flag` (set by behavior `ans_machine`) indicates to the `auto_dialer` that the recorded message has to be forwarded to a telephone number stored in the memory.

Comparing SpecCharts with Statecharts

From the graphical conventions used, the SpecCharts language seems similar to the Statecharts language [1], but they differ in many significant aspects. Statecharts provide an elegant representation for hierarchical finite state machines wherein a state is essentially a condition or stage of being. SpecCharts on the other hand blend sequential programming constructs with state-transition diagrams. Thus, a "state" in SpecCharts (i.e., a behavior) has an associated functionality which can be expressed either as substates or as sequential VHDL statements. Consequently, SpecCharts have two distinct types of transition arcs which are required to indicate completion (TOC arc) or termination (TI arc) of the actions associated with states. SpecCharts are highly modular in that they only allow transitions between "states" (behaviors) which are at the same hierarchical level and share the same parent state. Statecharts permit state transitions across hierarchical levels, a situation, which if allowed in SpecCharts, is akin to the use of *goto* statements in programming. In Statecharts, communication between concurrent states is through the use of global variables. Communication in SpecCharts can be specified either by using global signals or by using abstractions like channels and protocols. SpecCharts allow representation of structural information as interconnections between concurrent behaviors. SpecCharts semantics are identical to VHDL which is an IEEE standard and widely used as a hardware description language. To summarize, SpecCharts can be thought of as a combination of behavioral VHDL and the sequentially/concurrently decomposable state-based language of Statecharts.

2.5 SDL

SDL (Specification and Description Language) [6] is a language for specification and description of systems and was developed and standardized by the CCITT. While SDL has been used mainly in the telecommunication field, it is well suited for specifying all real-time and interactive systems. SDL essentially specifies the behavior of the system, its interaction with its environment. It has the capability of representing the internal structure of the system, so that the system can be understood and developed one part at a time.

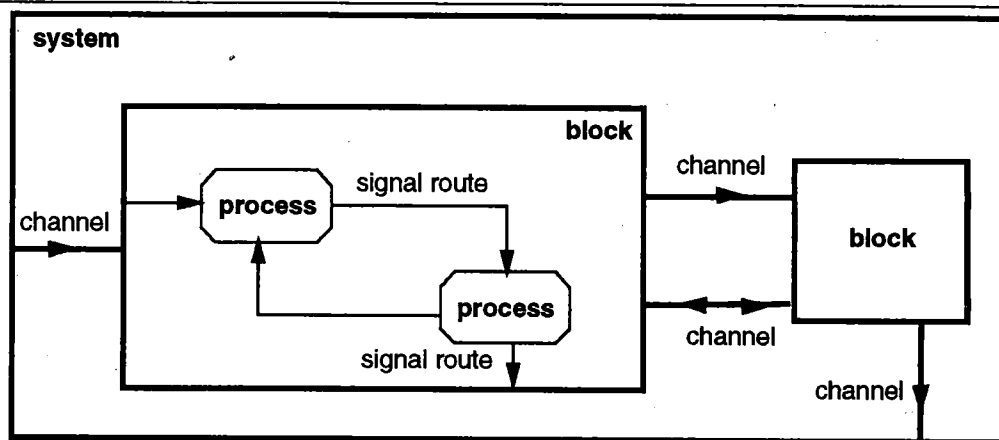


Figure 13: SDL : Hierarchical decomposition of specifications

An SDL specification can be summarized as consisting of hierarchical dataflow diagrams with a state machine at the leaf level. The target object for specification in SDL is a *system* (Figure 13). Everything that is part of the system is called the *environment*. A system can be specified with a hierarchy of *blocks*, which are the main structuring concept in SDL. A system contains one or more blocks connected with each other and with the boundary of the system by *channels*. Channels are essentially the pathways over which the signals associated with them are conveyed. Repeated decomposition of blocks into other blocks results in a block tree structure with the system at the root. All *leaf* blocks have one or more *processes*. A process is essentially a state machine which works concurrently with other processes.

A partial SDL representation of the answering machine phone introduced in Figure 10 of Section 2.4 is shown in Figure 14. The system consist of three blocks - **Answering Machine**, **Auto_dialer** and **Phone_memory**. The blocks are connected to each other and to the environment by the channels **Phone_Line**, **Tape**, **Display**, **Forward**, **Commands**, **Machine** and **Memory**. For example, the chan-

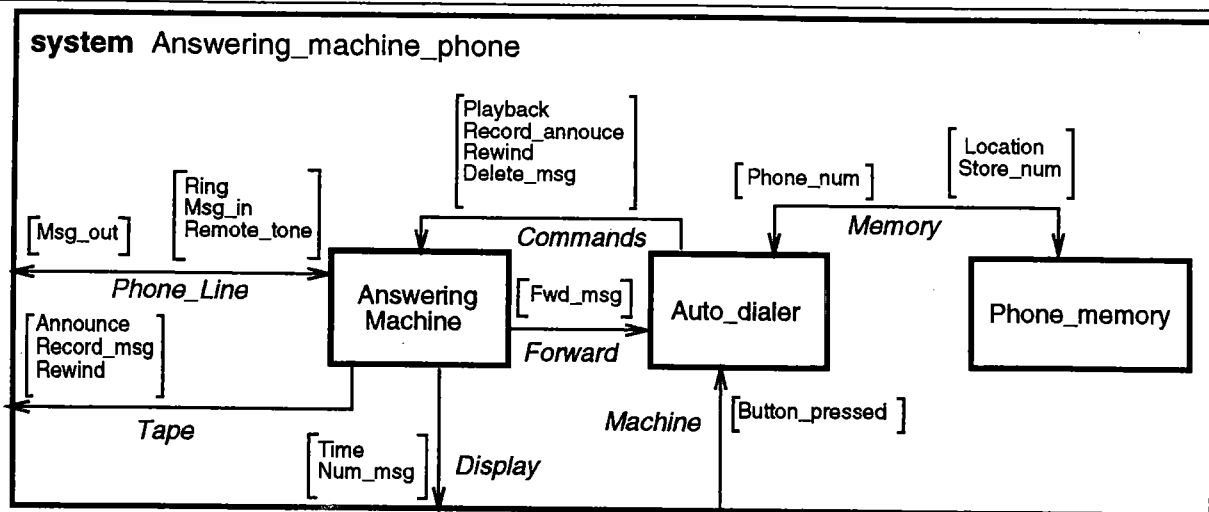


Figure 14: SDL : Partial specification of the answering machine

nel Display conveys the signals *Time* and *Num_msg* to the external LCD display unit.

Since SDL is targeted towards specifying entire systems, it has several constructs which enable the user reduce the overall complexity and enhance the readability of the specification. Thus, *block partitioning* allows a block to be decomposed into sub-blocks, *channel partitioning* allows a channel to be represented as lower-level blocks and channels, and *signal refinement* which enable the decomposition of a signal into a set of sub-signals.

Identical portions of a SDL process body which appear in more than one place can be represented by procedures. Also, processes can be decomposed into *services* interacting with each other by sending signals over *service signal routes*.

Processes communicate asynchronously with each other using signals conveyed over *signal routes*. Signal routes connect processes with each other and to the block boundary. Variables can be declared and assigned to in processes. Processes can either be in a *state* or performing transitions between states. During a transition, the process can manipulate variables, make decisions, create new process instances, send signals to other processes and activate or reset timers used to generate timing signals. In addition, there is an input queue associated with each process, which buffers each incoming signal. The signals are removed in a FIFO order from the queue as they are consumed by the process.

Processes in SDL can be created either at the time of system initialization or during the lifetime of the system by another process within the same block. Thus there may exist an arbitrary number of

processes of any type at a given time. The language allows the user to specify a limit on the number of processes instances of a given type that can be created. Once created, a process may only terminate when the *stop* construct in it is reached.

Processes in SDL have access to the current global time using the predefined expression **now** . In addition, an SDL process may declare a *timer* which generates a *timer signal* to be generated on expiry of a preset interval.

2.6 CSP

The CSP (Communicating Sequential Processes) [10] language was developed to overcome the limitations of traditional programming languages with respect to programs running on multi-processor machines. As the name implies, CSP allows the specification of a program as a set of concurrent processes with constructs to simplify the specification of communication and synchronization between the processes. In addition to its use as a programming language, CSP has been used to specify hardware systems [15].

A CSP program consists of a list of *commands*. A command specifies the behavior of a device executing the command. The command list specifies a sequential execution ordering of the commands in the list. There are two classes of commands:

1. *Simple Commands* : If successful, a simple command can

- alter the internal state of the executing device as in the assignment statement $A := B + C$,
- affect its external environment as with output command $p!x$ which sends the variable x to process p , or
- affect both the internal state as well the external environment as in the input command, $p?y$, which receives the value y from process p (enabling process p to continue if was blocked, waiting for the input command to execute).

2. *Structured Commands* : Structured commands (e.g., *alternative*, *parallel* and *repetitive* commands) involve the execution of all their constituent commands, and will be successful if and

only if all the constituent commands execute successfully. The alternate command is used for decision making, the parallel command is used to create new processes, and the repetitive command can be used for implementing iterative behavior.

Control constructs in CSP are implemented using the *guarded* command. A guarded commands consists of a list of guards or conditions and a command list which is executed only when all the conditions in the guard list evaluate to true. Alternative commands specify the execution of exactly one of its constituent guarded commands. Thus, the C language "if" statement:

```

if (A ≥ B)
    { max = A ; }
else
    { max = B ; }

```

can be represented using the following alternative command in CSP:

$$[A \geq B \rightarrow \max := A \mid A < B \rightarrow \max := B]$$

In an alternative command, it may be the case that several of the constituent guarded commands are successful. Under such circumstances, an arbitrary guard is selected and executed. This gives rise to *non-deterministic* behavior. The capability of the CSP language to express non-deterministic behavior is one of the salient differences between CSP and the other languages discussed so far.

An interesting feature of CSP is its ability to spawn off new processes through the *parallel command*. Processes created by the parallel command must use variables which are not shared across processes, i.e. the processes may *not* communicate with each other using global variables. All processes in the parallel command are executed sequentially and the parallel command terminates only when all its processes have terminated. Each process has a label which identifies it and is by itself a command list which could possibly have parallel commands in it. A CSP program is "static" in that the maximum number of processes that will be active concurrently is determined by the program text.

Communication between concurrent processes is simply specified with explicit *input* and *output* commands. Communication occurs between two processes if *all* of the following hold:

1. The output command on one process specifies the other process as the destination of the data to be sent.
2. The input command on the other process specifies the first process as the source of the data to be received.
3. The type of the target in the input command matches the expression of the output command.

Communication through the use of input/output commands in *blocking*, i.e. to synchronize the input and output commands, the one that is ready first is delayed till the other command also is ready. This eliminates the need for any buffering of data that is to be exchanged between processes.

Subroutines in CSP are implemented as *coroutines*. i.e. the subroutine is implemented as a process which executes concurrently with the calling process. Recursive subroutines can be simulated by using an array of processes, each element of which represents one level of recursion.

2.7 Silage

The Silage language [7, 16] was developed to address issues related to the specification of DSP (digital signal processing) systems. DSP systems are easily conceived as data-flow graphs, where a set of data values enter at the input nodes, computations are performed on them and result values are delivered to the output node in the graph. Silage is essentially an *applicative* language in that it only specifies application of functions to manipulate a set of data values without having any variables or the assignment operator.

Silage expressions represent *streams of values*. Thus in an expression $(A + B)$, A and B represent a stream of numbers as opposed to representing variables or array elements in conventional programming languages. A Silage program receives the set of input values arriving in a synchronized manner. Results of Silage expressions also yield a stream of data values. A Silage program consists of a set of *definitions* which define new values as a function of other values. The order of definitions is not significant since they do not represent assignments to variables (which would introduce dependencies between statements). Recurrences which any element of a stream depends on previous values of the stream are allowed. The delay operator “@” is used to denote previous values in a stream. For example,

consider the following statement:

```
D = D01 + 1 ;
```

The above statement would mean that each value in the stream represented by D would be one higher than the previous value.

Silage has *array constructors* which can concisely represent specific elements of a vector. Reduction operators like *sum* and *max* are used to operate on entire arrays. Conditional expressions select one of the expressions from a set of expressions, based on the condition of a guard. Stream manipulation operators such as the *decimate* and *interpolate* can be used to reduce and increase respectively the sampling rate of a signal by the desired amount. Functions in Silage represent a grouping of definitions and are implemented as macro expansion. Silage does not permit recursion or iterations with non-static bounds.

3 Evaluating Specification Languages

Having discussed some of the specification languages in the previous section, we will now evaluate the languages with respect to certain features which are desirable in the specification of computer systems.

3.1 Hierarchy

Hierarchical specification becomes vital as design complexity grows. The capability to represent a design hierarchically has two main benefits:

1. Make the system specification more manageable by structuring it to have several levels of abstraction (e.g., the hierarchy in the answering machine phone of Figure 10).
2. Specify the internal structure of the system as new details are added as a result of design activity.

For example, abstract communication channels in SpecCharts may get refined after synthesis into

a set of interconnected ports, or a entity in VHDL with a process may get refined into a set of interconnected RT-level components which implement the behavior specified by that process.

Hierarchy in specification languages can be of two types: *behavioral* and *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). We define a language which supports a *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/deactivate a sub-behavior at *any* time. Hierarchical deactivation is necessary for a language to support *exception handling*.
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.

Hardware systems requiring behavioral hierarchy are abundant [8]. In the answering machine phone, the event *rising(machine_button_pushed)* deactivates the behavior *respond_to_ext_line* and all its *descendent behaviors* and activates *respond_to_machine_button*. Similarly, a rising *rising(dialtone_p)* event deactivates the behavior *play_announcement* and activates the behavior *hangup*. The complete SpecChart of the answering machine phone of Figure 10 has ten levels of behavioral hierarchy which requires the capability of expressing behavioral decomposition, hierarchical deactivation and behavioral completion. Similarly, the Intel 8237 DMA Controller [9] and the Pipelined SISC processor [9] that we modeled had 4 levels of behavioral hierarchy.

The capability of specifying behavioral hierarchy is essential for two reasons:

1. It enables the designer to capture his design in manner which is closer to his conceptual view of the system. For example, to specify a system in a language like VHDL, the designer has to coerce his 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 specifying a system using a language which does not support behavioral hierarchy. Since behavioral hierarchy is unsupported by most HDL's, modelers are often forced modify the behavior to fit the language 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 in [17]).

The underlying model of SpecCharts supports the concept of a **behavioral hierarchy**. The system is captured as a set of hierarchical/concurrent behaviors. SpecCharts use transition arcs to achieve deactivation of behaviors. The TOC (transition on completion) arc is used to indicate behavioral completion.

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). Hierarchical (de)activation is also not provided since there is no construct to reset a process regardless of its execution status, when an event such as a system reset occurs. Note that nested blocks merely provide declarative hierarchy (i.e. symbol scoping rules).

In the Statemate environment [18], the Statechart language is used to represent the behavioral (*Statecharts*), functional (*Activity charts*), and structural (*Module charts*) views of a system. The first two combined partially provide for behavioral decomposition. There is limited support for hierarchical (de)activation and behavioral completion. However, mentally resolving the three system views into one can be quite awkward especially since each synthesis step could possibly result in modifications to all the three views.

CSP provides for behavioral decomposition and behavioral completion but not for hierarchical (de)activation. 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.

3.1.2 Structural Hierarchy

A structural hierarchy is one in which a system specification is represented set of interconnected components. Each component has its own internal structure specified with a set of a lower-level interconnected components and so on. The capability of a language to represent a structural hierarchy enables the designer to construct new components from a set of already existing components. In addition, it allows the designer to specify portions of the design using previously designed off-the-shelf components. For example, given gate-level components, the designer can construct a 2-to-1 multiplexer by interconnecting 2 *AND* gates and 1 *OR* gate. RT-level components like muxes, registers and ALU's can be connected to form structural specifications of entire systems such as processors or memories. VHDL, HardwareC, SDL and SpecCharts support structural decomposition.

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. SDL is different from VHDL and HardwareC in that it allows a process to be split up into a set of *services* and specify *service signal routes* to communicate signals between the services.

SpecCharts allow the designer to specify ports and channels for a behavior at any level of the hierarchy. These ports and channels can be connected together at the next higher level of the hierarchy.

Statecharts, CSP and Silage do not have the capability of specifying any structural information.

3.2 Concurrency

Hardware systems are 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 Statechart of the UART of Figure 15, the topmost state `top_level_uart` consists of three concurrent substates: `transmitter`, `receiver` and `uart_mode`. The receiver in turn consists of two concurrent substates `rx_mode` and `fifo`. CSP and SDL have the capability of creating new processes dynamically.

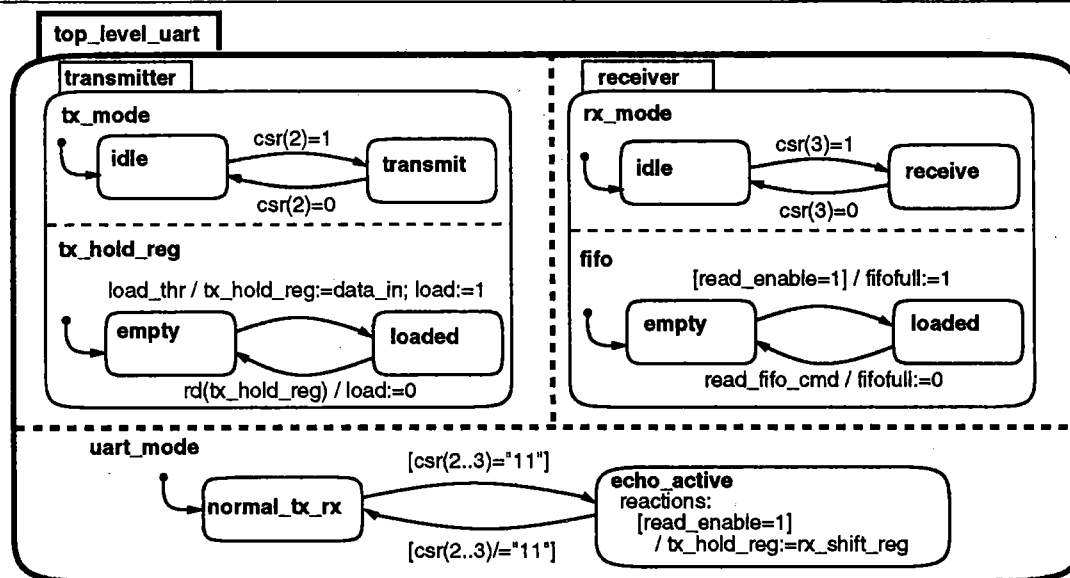


Figure 15: Concurrency specification in Statechart for the UART

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 in *A* and *B* will get swapped. Similarly the signal assignments semantics of VHDL (and consequently SpecCharts) allow for specification of fine-grain parallelism. For example, consider the following VHDL process code segment:

```

A <= 2 * B ;

```

```

B <= 2 * A ;
wait on A, B ;

```

The values of A and B used in the right-hand side of the assignments are the old values of the two signals. Thus, each of the two signals are updated simultaneously with twice the original value of the other signal.

Silage, being an applicative language, consists of a set of definitions applied to streams of values. Since they represent dataflow semantics, the definitions in a Silage program have implicit parallelism. For example, consider the following Silage definitions:

```

X = A + B ;
Y = C - X@1 ;

```

The computation of X and Y in the two definitions is carried out concurrently. The result would not be affected if the order of the two definitions were changed.

3.3 Timing

The ability to incorporate timing information in a design specification is important since it reflects the behavior intended by the designer more accurately. For example, the VHDL statement `A <= 2 after 20 ns` gives us more information on the designers intention (i.e. that A gets its new value 20 ns after the current time) than the statement `A <= 2`;. Apart from being able to specify behavior more accurately, timing specifications in some cases may also serve as constraints for synthesis tools. Thus, the *20ns* in the above statement represents to the synthesis tools the delay after which the assignment can take effect.

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

1. *Timeout clause* which specifies the maximum time that is to be spent at a wait statement (e.g., `wait on A for 100 ns`)
2. *After clauses* which specifies 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`).

In addition, SpecCharts have the *timeout arc* which causes a transition from a behavior on the expiry of the time-period associated with it.

In Statecharts, all operation are executed in zero-time. The designer can specify the minimum and maximum time spent in any state. SDL has a *timer* object which can be used to generate a signal, on the expiry of the associated time period, to the process which owns it. In addition, SDL and VHDL have access to the global time using the predefined expression **now**. Silage provides the delay operator, `,` which can be used to reference previous values of a data stream. Thus, `A1`, refers to the previous value in the data stream represented by `A`.

In HardwareC, a single phase clock is assumed to synchronize all operations in the specifications. While it has no explicit timing constructs, the language allows the designer to specify timing constraints between two statements.

3.4 Synchronization

In a system specification with several concurrent processes, synchronizing the processes will often be required. The need for synchronization may arise from the desire of two processes to exchange data or due to the fact that certain actions must be performed by different processes at the same time. The Silage language has no explicit constructs for synchronization as it is assumed that the input data streams arrive in step with each other. Synchronization methods in the other languages can be classified into the following categories and are illustrated with the Statecharts of Figure 16.

3.4.1 Synchronization by Initialization

In Figure 16(a), event *e* synchronizes all the orthogonal states *A*, *B* and *C* into their default substates. In Figure 16(b), event *e* causes *B* to move to its default substate *B1* while at the same time moving *A* from *A1* to *A2*. 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*. Synchronization by initialization may be used to simultaneously initialize the stages of a pipeline which has been modeled as a set of concurrent behaviors.

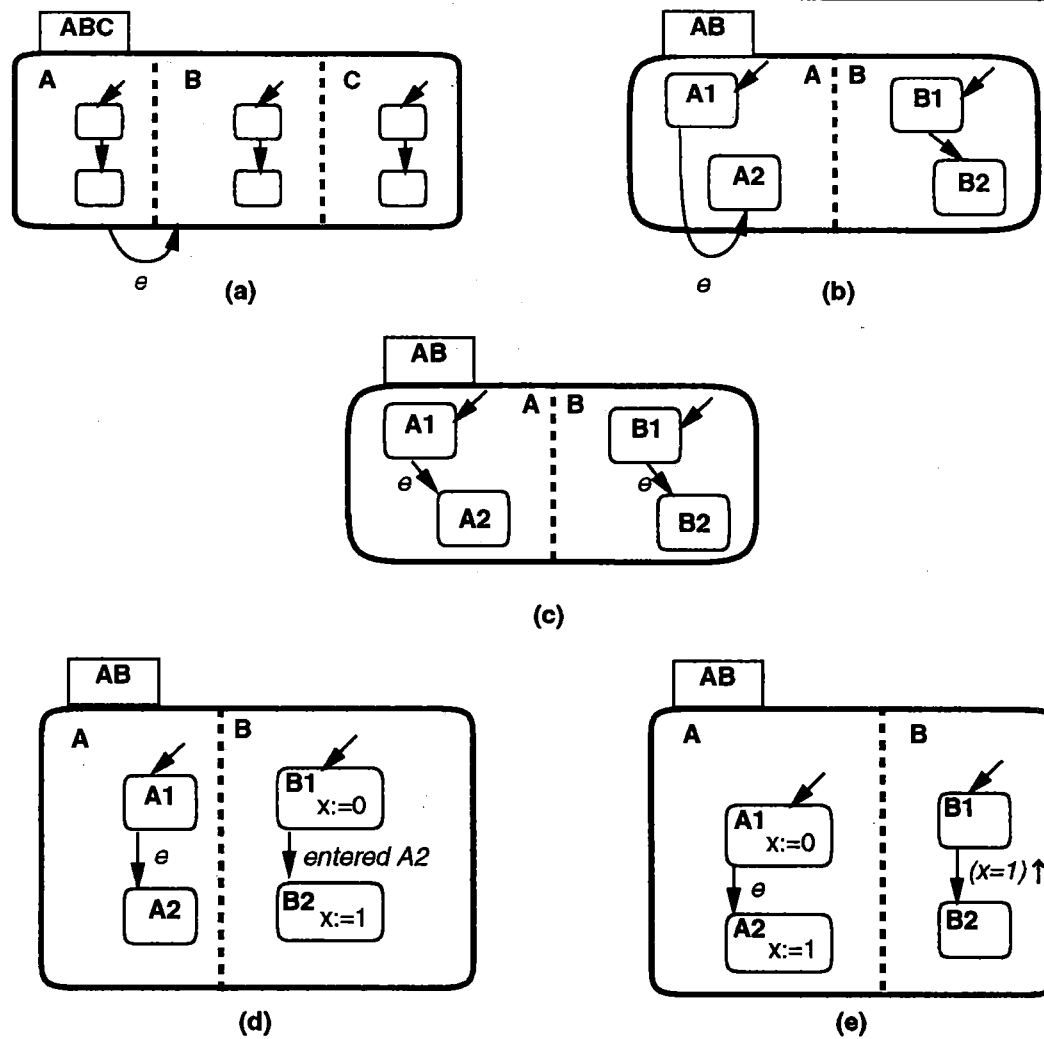


Figure 16: Synchronization methods in Statecharts

3.4.2 Synchronization by Common Event

In Figure 16(c), the event e synchronizes both the states A and B into substates A2 and B2 respectively. Apart from Statecharts, the languages SpecCharts, VHDL and SDL also have the capability of synchronizing processes based on a common event. *Wait* statements of VHDL (and consequently SpecCharts) in two processes with a common signal will synchronize the two processes when an event occurs on the common signal(s). Similarly, two SDL processes waiting for an event to occur on a common input signal, will synchronize on the happening of such an event, leading them both to the next states in their respective state-transition diagrams. In addition, any event on a signal which occurs in the sensitivity list of two or more VHDL processes will start the processes simultaneously. The various ways of synchronization by common event in VHDL is shown in Figure 17. A rising RESET

event will synchronize the assignments to *A* (in a wait statement), *B* (in a process with a sensitivity list) and *C* (in a concurrent assignment in block with a guard). An example of synchronization by common event is a global clock signal which can be used to synchronize the CPU and its peripheral devices.

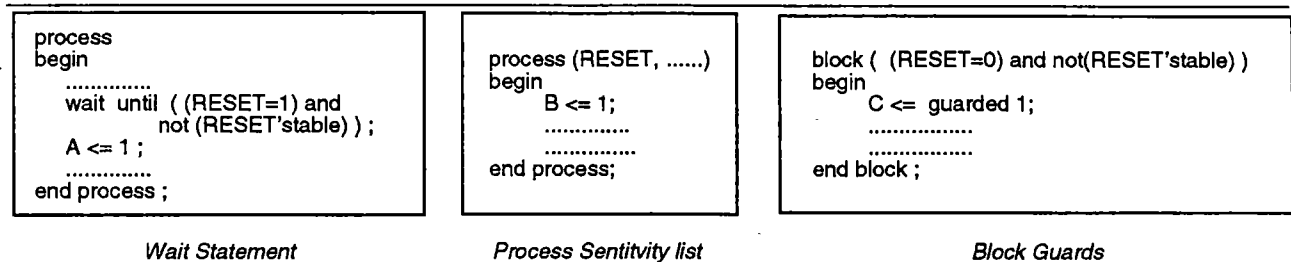


Figure 17: Synchronization by Common Event in VHDL

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 16(d). This method of synchronization is a unique feature of the Statechart language. 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 16(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 other process. Updating a status register which can be read by other processes is an example of this kind of synchronization.

3.4.5 Synchronization by Communication

In interprocess communication, it is often the case when either the sender or receiver process is ready for effecting the data transfer, the other is not. Under such cases, the process which is ready first

usually waits for the other process to be ready for the communication. In HardwareC, the *msgwait* command returns a flag indicating whether the specified channel has any pending messages. This can be used to synchronize by the receiving processes to check if the sender process has sent data on the channel data. 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*). However, 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. We will discuss each of them individually.

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. No synchronization is required to implement communication using a shared memory. Communication in VHDL, HardwareC, Statecharts, SDL and SpecCharts can be implemented using this model. In VHDL and SpecCharts, global signals are declared which are accessible by concurrent processes and behaviors. In Statecharts, communication 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 shared memory communication model requires the designer to specify communication details such as ports and/or specify the protocol for communication in each process. While procedures can be used to hide the behavioral aspect of communication from an object, the structural information such as the number and width 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 intended functionality of the communication, i.e. the memory is being read.

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 of the processes, communication is represented by simply specifying the data that is to be communicated from the sender process to the receiver process. The identity of the destination process can be specified explicitly (as is the case with CSP) or it can be inferred from the interconnection specification for the channels (as in the case of SpecCharts and HardwareC).

CSP and HardwareC use an unbuffered, synchronized, unidirectional model of communication using channels.

In HardwareC, the communication over the channel is specified using *send/receive* constructs. The decision on how the communication is to be implemented is left to the synthesis tools. The designer, having no way of specifying the details of the communication, has little control over how communication is implemented. This severely limits the protocols that can be used to implement the communication.

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 needs to be changed. If two communication channels are merged as a result of interface tradeoffs to meet pin constraints, the channels are updated with a common protocol representing the behavior of the resulting bus.

These simple constructs successfully abstract out both the behavioral and structural aspects of the communication from the main behavior, while being general enough to specify any protocol and structural interconnection. This greatly simplifies both system specification and synthesis.

It must be mentioned at this point, that while the SDL language has channels and signal routes which can convey signals from one process to another, they are closer to representing the paths for the flow of data between processes and hence do not represent a message passing model.

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 3.1.1.

SpecCharts and Statecharts can easily model exceptions by providing a transition arc (TI arc for SpecChart) 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 not so easy. A VHDL process could be at a wait statement not sensitive to the event causing the exception. Firstly, this event will have to be added to the sensitivity list of all the wait statements in a process which is cumbersome. In addition, after every wait statement we will have to check if the wait statement terminated due to the occurrence of the exception event (in which case control must somehow be transferred to the end of the process). Terminating an SDL process will require transitions labeled with the exception event from all states in the state-machine diagram of the *stop* construct of the process. This requires a lot of effort by the designer.

3.7 Specifying Actions using Programming Constructs

Using hierarchy, the designer can conceptually decompose his 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. SpecCharts, HardwareC, CSP and VHDL have incorporated programming language constructs for specifying actions in processes. The ability of the specification language to represent a design using such constructs greatly reduces the burden on the designer.

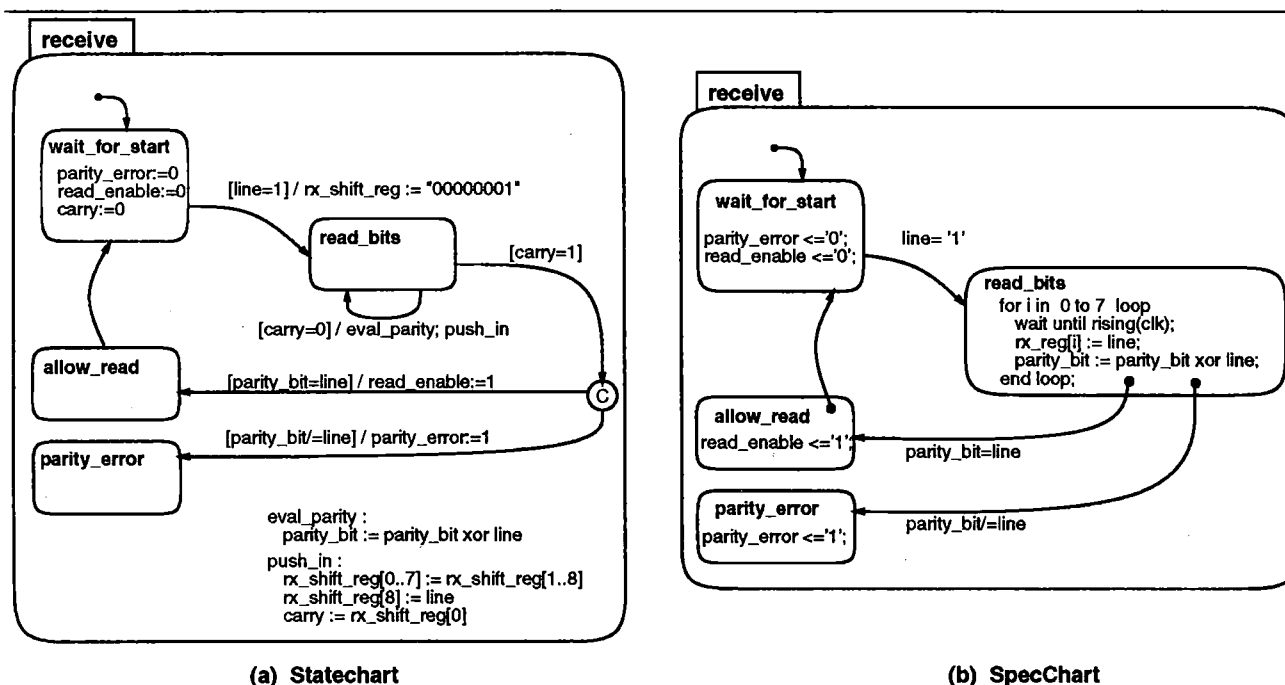


Figure 18: Specifying actions in the Receiver Process of the UART

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 appreciate the usefulness of programming constructs in specification of actions, consider the UART of Figure 8. The Statechart and the SpecChart for the receiver process of the UART are shown in Figure 18.

Figure 18 demonstrate major differences in the languages due to the lack of programming constructs in Statecharts. For example, the *receive* behavior detects a start bit ($line=1$) and then reads the next eight bits into a register. The reading of the eight bits is clearly shown using the for loop in *read_bits* of the SpecChart of Figure 18. Note that synthesis will likely unroll the for loop, so no index counter

is needed in the implementation.

In the Statecharts language, no such for loop is possible since only FSMs can be described. One possible specification approach is to mimic a *for* loop by introducing a new variable i initialized to 0. An arc with the condition $i < 8$ and action $i := i + 1$ is added that loops back on *read_bits*. Another arc with condition $i = 8$ points to the C fork. Not only is this less readable than a for loop, “unrolling” such a configuration during synthesis is difficult. Hence it is likely that a register and incrementer will be synthesized for i , which is inefficient. A second specification possibility uses a trick in which the 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. After each shift, the shift-out value (carry) is checked; a value of ‘1’ signifies that all 8 bits have been shifted in. This is how the loop is implemented in the Statecharts model. Such tricks are made necessary when trying to model behavior using FSMs; the result is a specification that is very difficult to comprehend.

3.8 Other Features

3.8.1 State-Based Specifications

Hardware designs often are inherently state-based such as a traffic light controller or a processor (with the fetch, decode and execute states). 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 an HDL (e.g., 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 registers, case and loop statements. Silage, being an applicative language, cannot specify a system as a state transition diagram.

3.8.2 Dataflow Semantics

In applications such as digital signal processing, algorithms are more naturally represented as dataflow graphs where node represent operations on data (e.g. addition) and the arcs represent the paths followed by the data items from the inputs to the outputs through the operation nodes. The Silage

language provides a textual representation for dataflow graphs with a wide range of data manipulation functions operating on data streams. This makes Silage most suitable for DSP applications. VHDL and HardwareC can also represent dataflow using the concurrent signal assignments and the parallel-compound statement respectively. Statecharts, SpecCharts, CSP and SDL are not well-suited for dataflow representations.

3.8.3 Designer Interaction for Synthesis

Amongst the languages discussed in this survey, HardwareC is unique in terms of the numerous 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 also specify *binding information* by associating individual operations with specific instances of a template of a component. Finally, the designer can specify which variables are to be implemented as storage by declaring them to be static, while at the same letting the synthesis tools decide whether boolean variables (e.g. bit vectors) are to be implemented as wires or registers. VHDL, SpecCharts, SDL and HardwareC have the capability of representing a design using mixed structure and behavior. An advantage of this is that the designer can specify portions of the design using specific off-the-shelf components while synthesis tools will synthesize the behavioral portions. All the above features enable the designer to use the specification language itself as a means of directing the synthesis tools.

4 Conclusion

In the previous sections, we have presented seven system specification languages and examined some of their salient constructs. Table 19 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. In this section we only examine the features that are necessary in a specification language for it to be able specify reactive, transformational systems.

<i>Language</i>	<i>Hierarchy</i>	<i>Concurrency</i>	<i>Actions</i>	<i>Timing</i>	<i>Exception Handling</i>
HardwareC	Declarative Hierarchy	Single level of processes Statement-level parallelism	Programming constructs	Assumes global single phase clock, Timing constraints between statements	Cumbersome to represent process termination due to external event
VHDL	Declarative Hierarchy	Single level of processes Statement-level parallelism	Programming constructs, Concurrent assignments	Delay specification using AFTER and FOR clauses	Cumbersome to represent process termination due to external event
Statecharts	Hierarchical Finite State Machines	AND decomposition at any level	Simple Actions (e.g. assignments) associated with transitions/states	zero-time computation, specification of min/max time spent in state	Easily handled by transition arcs
SpecCharts	Behavioral Hierarchy	Concurrent sub-behaviors at any level	Programming constructs	Timeout arcs, Delay specification with AFTER and FOR clauses	Easily handled by TI (transition immediately) arcs
SDL	Declarative Hierarchy	Single level of processes	Assignments in state transition machine of process	Generation of timeout signals using timers	Cumbersome to represent process termination.
Silage	none	Data-flow semantics.	Definitions manipulating streams of data values	Delay Operator @ to access previous values in data stream	none
CSP	Hierarchy of Processes	Dynamic process activation	Programming constructs	none	none

<i>Language</i>	<i>Structural Information</i>	<i>State-based Specification</i>	<i>Synchronization</i>	<i>Communication</i>	<i>Other Features</i>
HardwareC	Yes	Cumbersome to represent using CASE, LOOPS and state variables	Global Ports, Communication based (message wait)	shared memory message passing	Specification of constraints for synthesis
VHDL	Yes	Cumbersome to represent using CASE, LOOPS and state variables	Common Event, Global Signal	shared memory	-
Statecharts	No	Hierarchical State Diagrams	Initialization, Common State, Common Event, Global Variables	shared memory	History, Multi-level Transitions
SpecCharts	Yes	Hierarchical State Diagrams	Common Event, Global Signal, Communication based (blocking channels)	shared memory message passing	-
SDL	Yes	State Diagrams	Global Signals	message passing	-
Silage	No	No	Implicit - Data streams arrive in step (frames)	-	-
CSP	No	Cumbersome to represent	Communication based (blocking channels)	message passing	Non-determinism

Figure 19: Evaluating Features of Hardware Description Languages

What features are essential in a language to specify reactive, transformational systems ?. Based on the discussion of the preceeding sections, we conclude that a system-level specification must have the following features:

1. *Structural hierarchy* is clearly a necessity for system-level specifications due to the complexity in terms of specification size. *Behavioral hierarchy* is inherent in reactive systems and presents the most natural way to represent the designer's conceptual view of the system.
2. *Concurrency* is essential since a system-level design consists 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* are required to specify the transformational aspect of the system - the computation carried out by an algorithm. A state-transition diagram with simple assignments is difficult to use for system-level designs. In the absence of programming constructs, implementing even a simple looping mechanism will require a complicated set of state-transitions, making the design specification cumbersome and less readable.
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. This is another application of hierarchical deactivation which is a part of behavioral hierarchy.
7. *Structural Representation* is necessary 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* are required since reactive systems are often easily conceptualized as state-transition diagrams.

The following features are believed to be *non-essential* for the system-level specifications:

1. *Dataflow* representation, while it may prove useful in describing the transformational part of the system, it is not capable of specification of reactive behavior at all (e.g., Silage).
2. *Design constraints*, as can be specified in HardwareC, are meant for solely for the synthesis tools. They do not contribute to the behavior of the system being specified. Design constraints can be provided to the synthesis tools, separate from the design specification in the form of annotations and comments.
3. *History* is meaningful in Statecharts since all the actions (simple assignments) associated with a state or transition take place in zero time. On re-entering a state, it is possible to resume execution from the last state visited. On the other hand in a transformational system where we specify computations using programming constructs, it is difficult determine the point at which execution of the programming language code segment was terminated. In those cases where it is necessary to have the concept of history, a variable may be used to achieve the same effect.
4. *Multi-level transitions* across the hierarchy such as those found in Statecharts are similar to the situation where a "goto" statement is used to jump to the middle of a procedure from the main program. In the course of specifying several designs, we did not require multi-level transitions at all.

We have in this report examined the features of specification language which are well suited for the specification of reactive, transformational systems. Each feature recommended above attempts to reduce the amount of detail that a designer must be concerned with.

The SpecChart language, designed for system-level specification, has all the features recommended above. We have modeled and simulated several example systems including the Rockwell DRACO I/O chip, SISC processors, several Intel chips [19] (including 8203 Dynamic RAM Controller, 8237 Programmable DMA Controller, and 8251 Programmable Peripheral Interface), an answering machine, a RISC signal processor, a multi-tasking RISC process scheduler, a microwave transmitter controller,

and an Ethernet network co-processor. Details of these examples, along with a detailed comparison of SpecCharts with Statecharts, VHDL, and other HDL's, can be found in [9]. The ease with which SpecCharts was able to specify the above designs indicates that the constructs recommended above are well-suited for system-level specification.

5 Acknowledgements

I would like to thank Prof. Daniel Gajski for the numerous discussions that we had on the specification issues/languages and his helpful suggestions. My thanks also go to Frank Vahid for his valuable assistance, especially for designing from scratch the answering machine phone example which clearly demonstrates the expressive power of the SpecCharts language. This work was supported by the Semiconductor Research Corporation (grant #91-DJ-146). I am grateful for their support.

6 References

- [1] D. Harel, "Statecharts : A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987.
- [2] D. Ku and G. Micheli, "HardwareC - A Language for Hardware Design." Stanford University, Report CSL-TR-90-419, 1998.
- [3] G. Micheli and D. Ku, "HERCULES - A System for High-Level Synthesis," in *Proc. 25th DAC*, 1988.
- [4] *IEEE Standard VHDL Language Reference Manual*, 1988.
- [5] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL : Hardware Description and Design*. Kluwer Academic Publishers, 1989.
- [6] D. H. F. Belina and A. Sarma, *SDL with Applications from Protocol Specifications*. Prentice Hall, 1991.
- [7] P. Hilfinger and J. Rabey, *Anatomy of a Silicon Compiler*. Kluwer Academy Publishers, 1992.
- [8] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proc. of the International Conference on Computer-Aided Design*, 1991.
- [9] S. Narayan, F. Vahid, and D. Gajski, "Modeling with SpecCharts." UC Irvine, TR ICS 90-20, 1990.
- [10] C. Hoare, "Communicating Sequential Processes," *Comm. of the ACM*, August 1978.
- [11] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
- [12] J. Lis and D. Gajski, "Structure Modeling for VHDL Synthesis." UC Irvine, TR ICS 89-14, 1989.

- [13] D. Drusinsky and D. Harel, "Using Statecharts for Hardware Description and Synthesis," in *IEEE Transactions on Computer-Aided Design*, 1989.
- [14] F. Vahid, S. Narayan, and D. Gajski, "SpecCharts: A Language for System Level Synthesis," in *Proc. of the International Symposium on Computer Hardware Description Languages and their Applications*, 1991.
- [15] R. J. Auletta, "CSP Specified Digital Repeater and Translation to Synchronous Logic." Report, Dept. ECE, George Mason University, 1991.
- [16] P. Hilfinger, *A High-Level Language and Silicon Compiler for Digital Signal Processing*. Proceedings of the CICC, 1985.
- [17] D.E. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.
- [18] D. Harel, et al, "STATEMATE : A Working Environment for the Development of Complex Reactive Systems," in *Proc. of the Intl. Conf. on Software Engineering*, 1988.
- [19] *Intel Peripheral Design Handbook*, 1981.