The SpecC+ Language

Daniel D. Gajski
Jianwen Zhu
Rainer Dömer

Technical Report ICS-97-15
April 15, 1997

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(714) 824-8059

gajski@ics.uci.edu
jzhu@ics.uci.edu
doemer@ics.uci.edu

Abstract

In this report, we discuss the characteristics necessary for specifying embedded hardware-software systems. We describe the constructs needed to capture these characteristics and propose a new C based specification language to describe heterogeneous embedded systems.

Contents

List of Figures

The SpecC+ Language

Daniel D. Gajski, Jianwen Zhu, Rainer Dömer

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

In this report, we discuss the characteristics necessary for specifying embedded hardware-software systems. We describe the constructs needed to capture these characteristics and propose a new C based specification language to describe heterogeneous embedded systems.

1 Introduction

A system can be described at any one of several distinct levels of abstraction, each of which serves a particular purpose. By describing a system at the logic level, for example, designers can verify detailed timing as well as functionality. Alternatively, at the architectural level, the complex interaction among system components such as processors, memories, and ASICs can be verified. Finally, at the conceptual level, it is possible to describe the system's functionality without any notion of its components. Descriptions at such level can serve as the specification of the system for designers to work on. Increasingly, designers need to conceptualize the system using an **executable specification** language, which is capable of capturing the functionality of the system in a machine-readable and simulatable form.

Such an approach has several advantages. First, simulating an executable specification allows the designer to verify the correctness of the system's intended functionality. In the traditional approach, which started with a natural-language specification, such verification would not be possible until enough of the design had been completed to obtain a simulatable system description (usually gate-level schematics). The second advantage of this approach is that the specification can serve as an input to codesign and synthesis tools, which, in turn, can be used to obtain an implementation of the system, ultimately reducing design times by a significant amount. Third, such a specification can serve as comprehensive documentation, providing an unambiguous description of the system's intended functionality. Finally, it also serves as a good medium for the exchange of design information among various users and tools. As a result, some of the problems associated with system integration can be minimized, since this approach would emphasize well-defined system components that could be designed independently by different designers.

The increasing design complexity associated with systems-on-a-chip also makes an **executable modeling** language extremely desirable where an intermediate implementation can be represented and validated before proceeding to the next synthesis step. For the same reason, we need such a modeling language to be able to describe design artifacts from previous designs and intellectual properties (IP) provided by other sources.

Since different conceptual models possess different characteristics, any given specification language can be well or poorly suited for that model, depending on whether it supports all or just a few of the model's characteristics. To find the language that can capture a given conceptual model directly, we would need to establish a one-to-one correlation between the characteristics of the conceptual model and the constructs in the language.

In this report, we will begin by describing some of the characteristics commonly found in design models. We will then introduce the SpecC+ language and demonstrate how well it supports the modeling of embedded systems.

2 Characteristics of conceptual models

In this section, we will present some of the characteristics most commonly found in conceptual models used by designers. In presenting these characteristics, part of our goal will be to assess how useful each characteristic is in capturing one or more types of system behavior.

2.1 Concurrency

Any system can be decomposed into chunks of functionality called **behaviors**, each of which can be described in several ways, using the concepts of processes, procedures or state machines. In many cases, the functionality of a system is most easily conceptualized as a set of concurrent behaviors, simply because representing such systems using only sequential constructs would result in complex descriptions that can be difficult to comprehend. If we can find a way to capture concurrency, however, we can usually obtain a more natural representation of such systems. For example, consider a system with only two concurrent behaviors that can be individually represented by the finite-state machines F_1 and F_2. A sequential representation of the system would be a cross product of the two finite-state machines, $F_1 \times F_2$, potentially resulting in a large number of states. A more elegant solution, then, would be to use a conceptual model that has two or more concurrent finite-state machines, as do the Statecharts [Har87] and PSM [GVN93] models.

Concurrency representations can be classified into two groups, data-driven or control-driven, depending on how explicitly the concurrency is indicated. Furthermore, a special class of data-driven concurrency called pipelined concurrency is of particular importance to signal processing applications.

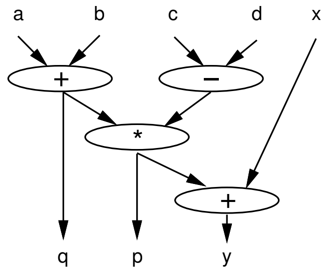
2.1.1 Data-driven concurrency

Some behaviors can be clearly described as sets of operations or statements without specifying any explicit ordering for their execution. In a case like this, the order of execution would be determined only by data dependencies between them. In other words, each operation will perform a computation on input data, and then output new data, which will, in turn, be input to other operations. Operation executions in such **dataflow** descriptions depend only upon the availability of data, rather than upon the physical location of the operation or statement in the specification. Dataflow representations can be easily described from programming languages using the **single assignment rule**, which means that each variable can appear exactly once on the left hand side of an assignment statement.

Consider, for example, the single assignment statements in Figure 1(a). As in any other data-driven ex-

1: q = a + b
2: y = p + x
3: p = (c – d) * q

(a)



(b)

Figure 1: Data-driven concurrency: (a) dataflow statements, (b) dataflow graph generated from (a).

ecution, it is of little consequence that the assignment to p follows the statement that uses the value of p to compute the value of y. Regardless of the sequence of the statements, the operations will be executed solely as determined by availability of data, as shown in the dataflow graph of Figure 1(b). Following this principle, we can see that, since a, b, c and d are primary inputs, the add and subtract operations in statements 1 and 3 will be carried out first. The results of these two computations will provide the data required for the multiplication in statement 3. Finally, the addition in statement 2 will be performed to compute y.

2.1.2 Pipelined concurrency

Dataflow description in the previous section can be viewed as a set of operations which consume data from their inputs and produce data on their outputs. Since the execution of each operation is determined by the availability of its input data, the degree of concurrency that can be exploited is limited by data dependencies. However, when the same dataflow operations are applied to a stream of data samples, we can use **pipelined** concurrency to improve the throughput, that is, the rate at which the system is able to process the data stream. Such throughput improvement is achieved by dividing operations into groups, called pipeline **stages**, which operate on different data sets in the stream. By operating on different data sets, pipeline stages can run concurrently. Note that each stage will take the same amount of time, called a **cycle**, to compute its results.

For example, Figure 2(a) shows a dataflow graph operating on the data set $a(n), b(n), c(n), d(n)$ and x(n), while producing the data set $q(n), p(n)$ and $y(n)$, where the index n indicates the nth data in the stream, called **data sample** n. Figure 2(a) can be converted into a pipeline by partitioning the graph into three

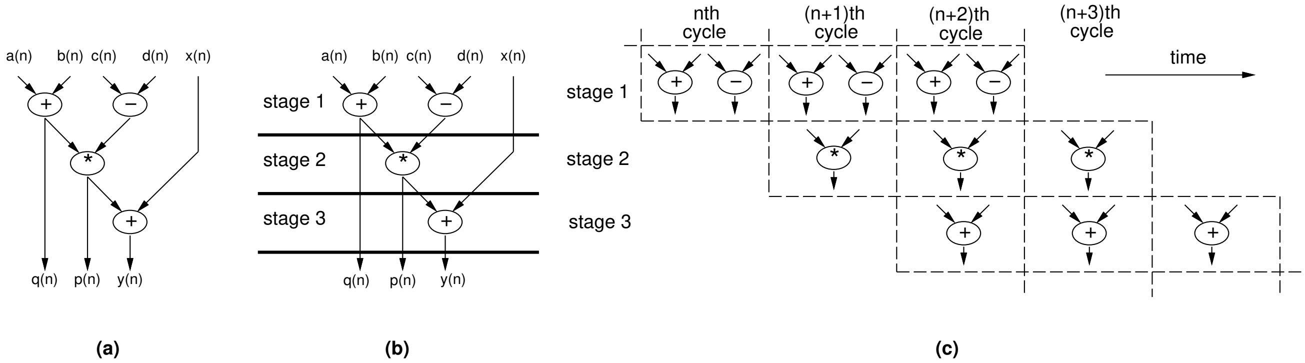


Figure 2: Pipelined concurrency: (a) original dataflow, (b) pipelined dataflow, (c) pipelined execution.

stages, as shown in Figure 2(b).

In order for the pipeline stages to execute concurrently, storage elements such as registers or FIFO queues have to be inserted between the stages (indicated by thick lines in Figure 2(b)). In this way, while the second stage is processing the results produced by the first stage at the previous cycle, the first stage can simultaneously process the next data sample in the stream. Figure 2(c) illustrates the pipelined execution of Figure 2(b), where each row represents a stage, each column represents a cycle. In the third column, for example, while the first stage is adding $a(n+2)$ and $b(n+2)$, and subtracting $c(n+2)$ and $d(n+2)$, the second stage is multiplying $(a(n+1)+b(n+1))$ and $a(n+1)-d(n+1)$, and the third stage is finishing the computation of the nth sample by adding $((a(n)+b(n))*(c(n)-d(n))$ to $x(n)$.

2.1.3 Control-driven concurrency

The key concept in control-driven concurrency is the control thread, which can be defined as a set of operations in the system that must be executed sequentially. As mentioned above, in data-driven concurrency, it is the dependencies between operations that determine the execution order. In control-driven concurrency, by contrast, it is the control thread or threads that determine the order of execution. In other words, control-driven concurrency is characterized by the use of explicit constructs that specify multiple threads of control, all of which execute in parallel.

Control-driven concurrency can be specified at the task level, where constructs such as fork-joins and processes can be used to specify concurrent execution of operations. Specifically, a *fork* statement creates a set of concurrent control threads, while a *join* statement waits for the previously forked control threads to terminate. The *fork* statement in Figure 3(a), for example, spawns three control threads *A, B* and *C*, all of which execute concurrently. The corresponding *join* statement must wait until all three threads have terminated, after which the statements in *R* can be executed. In Figure 3(b), we can see how *process* statements are used to specify concurrency. Note that, while a fork-join statement starts from a single control thread and splits it into several concurrent threads as shown in Figure 3(c), a *process* statement represents the behavior as a set of concurrent threads, as shown in Figure 3(d). For example, the **process** statements of Figure 3(b) create three processes *A, B* and *C*, each representing a different control thread. Both *fork-join* and *process* statements may be nested, and both approaches are equivalent to each other in the sense that

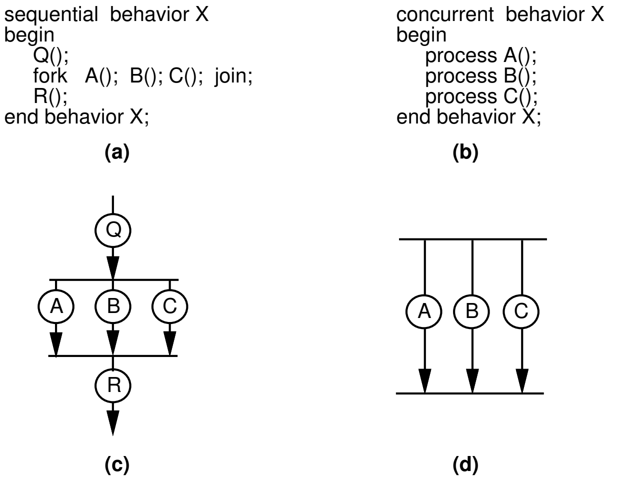


Figure 3: Control-driven concurrency: (a) *fork-join* statement, (b) *process* statement, (c) control threads for *fork-join* statements, (d) control threads for *process* statement.

a *fork-join* can be implemented using nested processes and vice versa.

2.2 State transitions

Systems are often best conceptualized as having various **modes**, or states, of behavior, as in the case of controllers and telecommunication systems. For example, a traffic-light controller [DH89] might incorporate different modes for day and night operation, for manual and automatic functioning, and for the status of the traffic light itself.

In systems with various modes, the transitions between these modes sometimes occur in an unstructured manner, as opposed to a linear sequencing through the modes. Such arbitrary transitions are akin to the use of *goto* statements in programming languages. For example, Figure 4 depicts a system that transitions between modes *P, Q, R, S* and *T*, the sequencing determined solely by certain conditions. Given a state machine with N states, there can be $N \times N$ possible transitions among them.

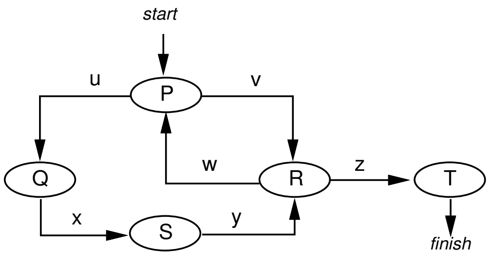


Figure 4: State transitions between arbitrarily complex behaviors.

In systems like this, transitions between modes can be triggered by the detection of certain events or certain conditions. For example, in Figure 4, the transition from state P to state Q will occur whenever event u happens while in P. In some systems, actions can be associated with each transition, and a particular mode or state can have an arbitrarily complex behavior or computation associated with it. In the case of the traffic-light controller, for example, in one state it may simply be sequencing between the red, yellow and green lights, while in another state it may be executing an algorithm to determine which lane of traffic has a higher priority based on the time of the day and the traffic density. In traditional and hierarchical finite-state machine conceptual models, simple assignment statements, such as $x = y + 1$, can be associated with a state. In the PSM [GVN93] model, any arbitrary algorithm with iteration and branching constructs can be associated with a state.

2.3 Hierarchy

One of the problems we encounter with large systems is that they can be too complex to be considered in their entirety. In such cases, we can see the advantage of hierarchical models. First, since hierarchical models allow a system to be conceptualized as a set of smaller subsystems, the system modeler is able to focus on one subsystem at a time. This kind of modular decomposition of the system greatly simplifies the development of a conceptual view of the system. Furthermore, once we arrive at an adequate conceptual view, the hierarchical model greatly facilitates our comprehension of the system's functionality. Finally, a hierarchical model provides a mechanism for scoping objects, such as declaration types, variables and subprogram names. Since a lack of hierarchy would make all such objects global, it would be difficult to relate them to their particular use in the model, and could hinder our efforts to reuse these names in different portions of the same model.

There are two distinct types of hierarchy – structural hierarchy and behavioral hierarchy – both of which are commonly found in conceptual views of systems.

2.3.1 Structural hierarchy

A structural hierarchy is one in which a system specification is represented as a set of interconnected components. Each of these components, in turn, can have its own internal structure, which is specified with a set of lower-level interconnected components, and so on. Each instance of an interconnection between components represents a set of communication channels connecting the components. The advantage of a model that can represent a structural hierarchy is that it can help the designer to conceptualize new components from a set of existing components.

This kind of structural hierarchy in systems can be specified at several different levels of abstraction. For example, a system can be decomposed into a set of chips/modules communicating over buses. Each of these chips may consist of several blocks, and each block, in turn, may consist of several RT components, such as registers, ALUs and multiplexers. Finally, each RT component can be further decomposed into a set of gates. In addition, we should note that different portions of the system can be conceptualized at different levels of abstraction, as in Figure 5, where the processor has been structurally decomposed into a datapath represented as a set of RT components, and into its corresponding control logic represented as a set of gates.

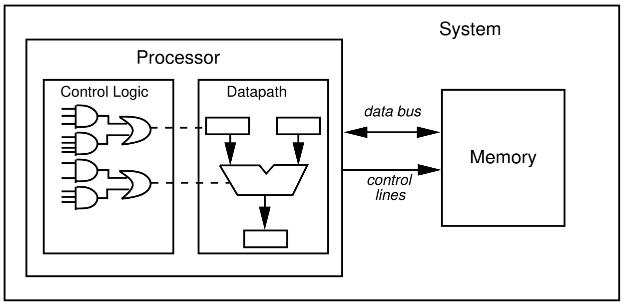


Figure 5: Structural hierarchy.

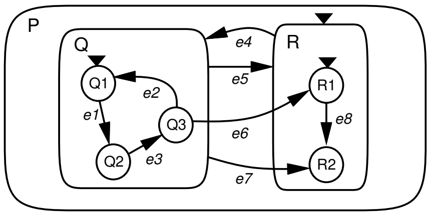
2.3.2 Behavioral hierarchy

The specification of a **behavioral hierarchy** is defined as the process of decomposing a behavior into distinct subbehaviors, which can be either sequential or concurrent.

The **sequential decomposition** of a behavior may be represented as either a set of procedures or a state machine. In the first case, a **procedural sequential decomposition** of a behavior is defined as the process of representing the behavior as a sequence of procedure calls. Even in the case of a behavior that consists of a single set of sequential statements, we can still think of that behavior as comprising a procedure which encapsulates those statements. A procedural sequential decomposition of behavior P is shown in Figure 6(a), where behavior P consists of a sequential execution of the subbehaviors represented by procedures Q and R. Behavioral hierarchy would be represented here by nested procedure calls. Recursion in procedures allows us to specify a dynamic behavioral hierarchy, which means that the depth of the hierarchy will be determined only at run time.

Figure 6(b) shows a **state-machine sequential decomposition** of behavior P. In this diagram, P is decomposed into two sequential subbehaviors Q and R, each of which is represented as a state in a state-machine. This state-machine representation conveys hierarchy by allowing a subbehavior to be represented as another state-machine itself. Thus, Q and R are state-machines, so they are decomposed further into sequential subbehaviors. The behaviors at the bottom

```
behavior P
   variable x, y;
begin
   Q(x) ;
   R(y) ;
end P;
```



(a) (b)

Figure 6: Sequential behavioral decomposition: (a) procedures, (b) state-machines.

level of the hierarchy, including *Q1, ...R2*, are called **leaf behaviors**.

In a sequentially decomposed behavior, the subbehaviors can be related through several types of transitions: simple transitions, group transitions and hierarchical transitions. A **simple transition** is similar to that which connects states in an FSM model in that it causes control to be transferred between two states that both occupy the same level of the behavioral hierarchy. In Figure 6(b), for example, the transition triggered by event *e1* transfers control from behavior *Q1* to *Q2*. **Group transitions** are those which can be specified for a group of states, as is the case when event *e5* causes a transition from *any* of the subbehaviors of Q to the behavior R. **Hierarchical transitions** are those (simple or group) transitions which span several levels of the behavioral hierarchy. For example, the transition labeled *e6* transfers control from behavior *Q3* to behavior *R1*, which means that it must span two hierarchical levels. Similarly, the transition labeled *e7* transfers control from Q to state *R2*, which is at a lower hierarchical level.

For a sequentially decomposed behavior, we must explicitly specify the **initial** subbehavior that will be activated whenever the behavior is activated. In Figure 6(b), for example, R is the first subbehavior that is active whenever its parent behavior P is activated, since a solid triangle points to this first subbehavior. Similarly, *Q1* and *R1* would be the initial subbehaviors of behaviors Q and R, respectively.

The **concurrent decomposition** of behaviors allows subbehaviors to run in parallel or in pipelined fashion.

Figure 7 shows a behavior X consisting of three subbehaviors A, B and C. In Figure 7(a) the subbehaviors are running sequentially, one at a time, in the order indicated by the arrows. In Figure 7(b), A, B and C run in parallel, which means that they will start when X starts, and when all of them finish, X will finish, just like the fork-join construct dis-

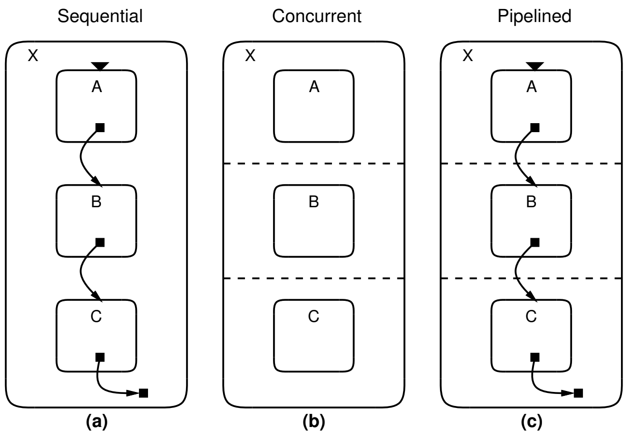


Figure 7: Behavioral decomposition types: (a) sequential, (b) parallel, (c) pipelined.

cussed in Section 2.1. In Figure 7(c), A, B and C run in pipelined mode, which means that they represent pipeline stages which run concurrently where A supplies data to B and B to C as discussed in Section 2.1.

2.4 Programming constructs

Many behaviors can best be described as sequential algorithms. Consider, for example, the case of a system intended to sort a set of numbers stored in an array, or one designed to generate a set of random numbers. In such cases, if the system designer manages to decompose the behavior hierarchically into smaller and smaller subbehaviors, he will eventually reach a stage where the functionality of a subbehavior can be most directly specified by means of an algorithm.

The advantage of using such programming constructs to specify a behavior is that they allow the system modeler to specify an explicit sequencing for the computations in the system. Several notations exist for describing algorithms, but programming language constructs are most commonly used. These constructs include assignment statements, branching statements, iteration statements and procedures. In addition, data types such as records, arrays and linked lists are usually helpful in modeling complex data structures.

The following code segment shows how we would use programming constructs to specify a behavior that sorts a set of ten integers into descending order. Note that the procedure *swap* exchanges the values of its two parameters.

```
int    buf[10], i, j;

for( i = 0; i < 10; i ++ )
    for( j = 0; j < i; j ++ )
        if( buf[i] > buf[j] )
            swap( &buf[i], &buf[j] );
```

Source Listing 1

2.5 Behavioral completion

Behavioral completion refers to a behavior's ability to indicate that it has completed, as well as to the ability of other behaviors to detect this completion. A behavior is said to have completed when all the computations in the behavior have been performed, and all the variables that have to be updated have had their new values written into them.

In the finite-state machine model, we usually designate an explicitly defined set of states as **final states**. This means that, for a state machine, completion will have occurred when control flows to one of these final states, as shown in Figure 8(a).

In cases where we use programming language constructs, a behavior will be considered complete when the last statement in the program has been executed. For example, whenever control flows to a return statement, or when the last statement in the procedure is executed, a procedure is said to be complete.

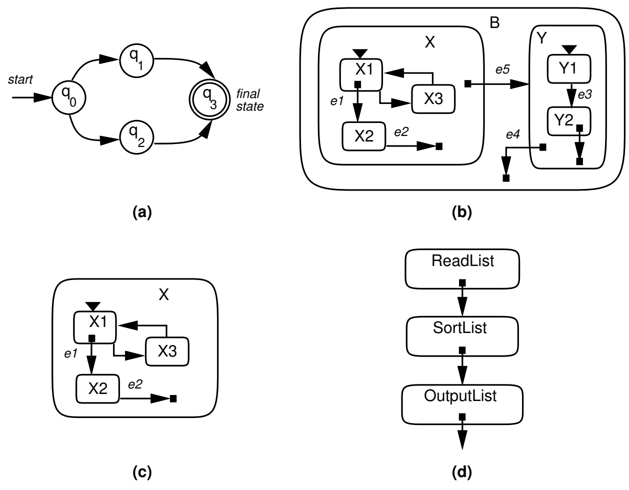


Figure 8: Behavioral completion: (a) finite-state machine, (b) program-state machine, (c) a single level view of the program-state X, (d) decomposition into sequential subbehaviors.

The PSM model denotes completion using a special predefined **completion point**. When control flows to this completion point, the program-state enclosing it is said to have completed, at which point the transition-on-completion (TOC) arc, which can be traversed only when the source program-state has completed, could now be traversed.

For example, consider the program-state machine in Figure 8(b). In this diagram, the behavior of leaf program-states such as *X1* have been described with programming constructs, which means that their completion will be defined in terms of their execution of the last statement. The completion point of the program-state machine for *X* has been represented as a bold square. When control flows to it from program-state *X2* (i.e., when the arc labeled by event *e2* is traversed), the program-state *X* will be said to have completed. Only then can event *e5* cause a TOC transition to program-state *Y*. Similarly, program-state *B* will be said to have completed whenever control flows along the TOC arc labeled *e4* from program-state *Y* to the completion point for *B*.

The specification of behavioral completion has two advantages. First, in hierarchical specifications, completion helps designers to conceptualize each hierarchical level, and to view it as an independent module, free from interference from inter-level transitions. Figure 8(c), for example, shows how the program-state *X* in Figure 8(b) would look by itself in isolation from the larger system. Having decomposed the functionality of *X* into the program-substates *X1*, *X2* and *X3*, the system modeler does not have to be concerned with the effects of the completion transition labeled by event *e5*. From this perspective, the designer can develop the program-state machine for *X* independently, with its own completion point (transition labeled *e2* from *X2*). The second advantage of specifying behavioral completion is that the concept allows the natural decomposition of a behavior into subbehaviors which are then sequenced by the "completion" transition arcs. For example, Figure 8(d) shows how we can split an application which sorts a list of numbers into three distinct, yet meaningful subbehaviors: *ReadList, SortList* and *OutputList*. Since TOC arcs sequence these behaviors, the system requires no additional events to trigger the transitions between them.

2.6 Exception handling

Often, the occurrence of a certain event can require that a behavior or mode be interrupted immediately, thus prohibiting the behavior from updating values further. Since the computations associated with any behavior can be complex, taking an indefinite amount of time, it is crucial that the occurrence of the event, or **exception**, should terminate the current behavior immediately rather than having to wait for the computation to complete. When such exceptions arise, the next behavior to which control will be transferred is indicated explicitly.

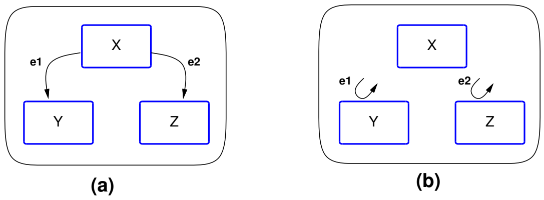


Figure 9: Exception handling: (a) abortion, (b) interrupt

Depending on if the control will be transferred back to the behavior that is interrupted, the exception can be further distinguished into two cases: Figure 9(a) shows the situation of **abortion**, where behavior *X* will be terminated by the occurrence of *e1* or *e2*. Figure 9(b) shows the case of **interrupt**, where control will transfer to *Y* or *Z* upon the occurrence of *e1* or *e2*, and it will return after they finish.

Examples of such exceptions include resets and interrupts in a computer system.

2.7 Timing

In system specifications, there may be components of the system where the notion of real time becomes relevant to ensure the correct implementation. In these situations, a component receives or generates events in specified time ranges. The time range is measured in real time units such as nanoseconds.

In general, a timing relation can be described by a 4-tuple $T =< e1, e2, min, max >$, where event $e1$ preceeds $e2$ by at least min time units and at most max time units. The timing relation which specifies what the component can ensure is called **timing delay**. The timing relation which specifies what the component has to satisfy is called **timing constraint**.

The added timing information is especially important for describing parts of the system which interact extensively with the environment according to a predefined **protocol**. The protocol defines the set of timing relations between signals, which both communicating parties have to respect.

A protocol is usually visualized by a **timing diagram**, such as the one shown in Figure 10 for the read cycle of a static RAM. Each row of the timing diagram shows a waveform of a signal, such as *Address, Read,*

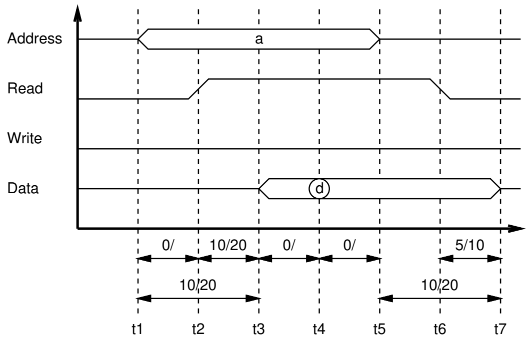


Figure 10: Timing diagram

Write and *Data* in Figure 10. Each dashed vertical line designates an occurrence of an event, such as *t1*, *t2* through *t7*. There may be timing delays or timing constraints associated with pairs of events, indicated by an arrow annotated by x/y, where x stands for the *min* time, y stands for the *max* time. For example, the arrow between *t1* and *t3* designates a timing delay, which says that *Data* will be valid at least 10, but no more than 20 nanoseconds after *Address* is valid.

The timing information is very important for the subset of embedded systems known as real time systems, whose performance is measured in terms of how well the implementation respects the timing constraints. A favorite example of such systems would be an aircraft controller, where failure to respond to an abnormal event in a predefined timing limit will lead to disaster.

2.8 Communication

In general, systems consist of several subbehaviors or processes which need to communicate with each other. This kind of communication between portions of the system is usually conceptualized in terms of shared memory or message passing paradigms. We will discuss each of these individually before we generalize them into the more flexible hierarchical communication model.

2.8.1 Shared-memory communication model

In a shared memory model, each sending process writes to a shared medium, such as a global variable, which can then be read by all receiving processes. If synchronization is required between the communicating processes, it must be specified explicitly. For example, the sending process could incorporate a special *valid* flag to indicate that the shared memory has been updated with a new value, which can then be read by the receiving processes. The shared memory model also includes the **broadcast** mechanism, which ensures that any value or event generated by one process or its environment will immediately be sensed by all the receiving processes.

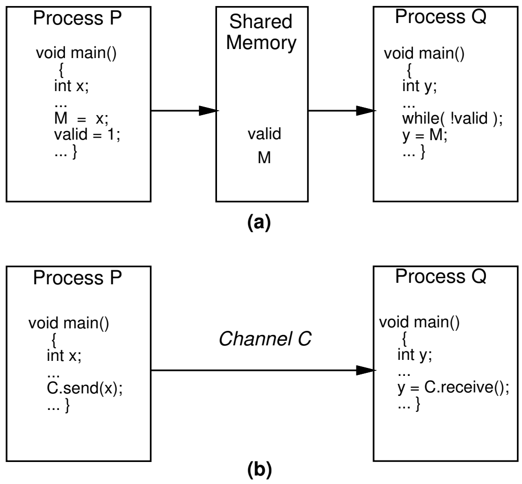


Figure 11: Inter-process communication paradigms: (a) shared memory, (b) message passing.

Figure 11(a) shows how the communication between processes P and Q could occur through the use of a shared memory M. To send data to process Q, process P simply updates the shared memory M and the *valid* flag, which is then read by process Q.

2.8.2 Message-passing communication model

In the message-passing model, the details of data transfers between processes are replaced by communication over an abstract medium called a **channel**, over which data or **messages** are sent. In each process, **send** and **receive** procedures would be used to transfer data over the channel.

Consider, for instance, Figure 11(b), which shows how the communication between processes P and Q could be achieved by using the message passing model. A channel C has been defined for the transfer of data from process P to process Q. The data, represented by the value of variable x in process P, is then transferred over the channel through the use of a *send* procedure. Finally, this data is received into the variable y in process Q by means of a *receive* function.

2.8.3 Hierarchical communication model

A useful generalization of shared memory and message passing paradigms is the hierarchical communication model, where a communication channel is specified as the encapsulator of a set of communication **media** in the form of variables, and a set of **methods** in the form of functions that operate on these variables. These functions specify how data is transferred over this channel. The accesses to the channel are restricted to these methods.

For example, the shared memory model in Figure 11(a) can be replaced by the following *integer* channel:

```
channel integer( void ) {
        bool  valid;
        int   M;

        int   read( void ) {
               while( valid == 0 );
               return M;
              }
        void   write( int a ) {
              M = a;
              valid = 1;
              }
        };
```

Source code 2

Here methods *read* and *write* provide the restricted accesses to the variables *M* and *valid*.

The adoption of this model can achieve information hiding, since the media and the way how the methods are implemented are hidden. In this way, the modeling complexity is reduced, since the users only need to make function calls to the methods. This model also encourages the separation of computation and communication, since the functionality responsible for communication can be confined in the channel specification and will not be mixed with descriptions used for computation.

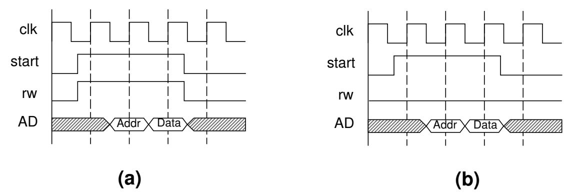


Figure 12: A simple synchronous bus protocol: (a) read cycle, (b) write cycle.

Consider, for example, a synchronous bus specification in Figure 12. A component using this bus can initiate a communication by asserting the *start* and *rw* signals in the first cycle, supplying the address in the second cycle, and then supplying data in the following cycles. The communication will terminate when the *start* signal is deasserted. This protocol description is shown in Source code 3, which encapsulates the communication media, in this case the signals *clock, start, rw* and *AD*, and a set of methods, in this case *read_cycle* and *write_cycle*, which implement the communication protocol for reading and writing the data as described.

```
channel  bus( void ) {
        signal<bit>     clk;
        signal<bit>     start;
        signal<bit>     rw;
        signal<word>    AD;

        void   read_cycle( word addr, word *d ) {
               start = 1, rw = 1,        clk.tick();
               AD = addr,                clk.tick();
               *d = AD,                  clk.tick();
               start = 0, rw = 0,        clk.tick();
               }
        void   write_cycle( word a, word d ) {
               start = 1, rw = 0,     clk.tick();
               AD = addr,             clk.tick();
               AD = d,                clk.tick();
               start = 0,             clk.tick();
               }
        };
```

Source code 3

2.9 Synchronization

In a system that is conceptualized as several concurrent processes, the processes are rarely completely independent of each other. Each process may generate data and events that need to be recognized by other processes. In cases like these, when the processes exchange data or when certain actions must be performed by different processes at the same time, we need to synchronize the processes in such a way that one process is suspended until the other reaches a certain point in its execution. Common synchronization methods fall into two classifications, namely control-dependent and data-dependent schemes.

2.9.1 Control-dependent synchronization

In control-dependent synchronization techniques, it is the control structure of the behavior that is responsible for synchronizing two processes in the system. For

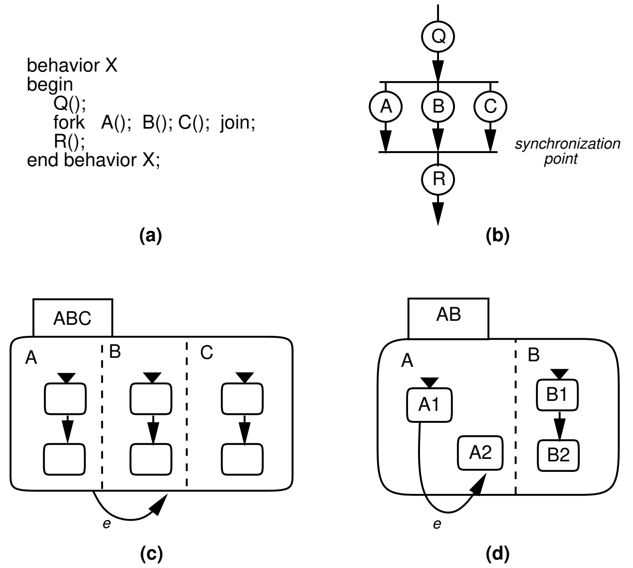


Figure 13: Control synchronization: (a) behavior X with a *fork-join*, (b) synchronization of execution streams by *join* statement, (c) and (d) synchronization by initialization in Statecharts.

example, the *fork-join* statement introduced in Section 2.3 is an instance of such a control construct. Figure 13(a) shows a behavior X which forks into three concurrent subprocesses, A, B and C. In Figure 13(b) we see how these distinct execution streams for the behavior X are synchronized by a *join* statement, which ensures that the three processes spawned by the fork statement are all complete before R is executed. Another example of control-dependent synchronization is the technique of **initialization**, in which processes are synchronized to their initial states either the first time the system is initialized, as is the case with most HDLs, or during the execution of the processes. In the Statechart [DH89] of Figure 13(c), we can see how the event e, associated with a transition arc that reenters the boundary of ABC, is designed to synchronize all the orthogonal states A, B and C into their default substates. Similarly, in Figure 13(d), event e causes B to initialize to its default substate *B1* (since AB is exited and then reentered), at the same time transitioning A from *A1* to *A2*.

2.9.2 Data-dependent synchronization

In addition to these techniques of control-dependent synchronization, processes may also be synchronized by means of one of the methods for interprocess communication: shared memory or message passing.

Shared-memory based synchronization works by making one of the processes suspend until the other process has updated the shared memory with an appropriate value. In such cases, the shared memory might represent an event, a data value or the status of another process in the system, as is illustrated in Figure 14 using the Statecharts language.

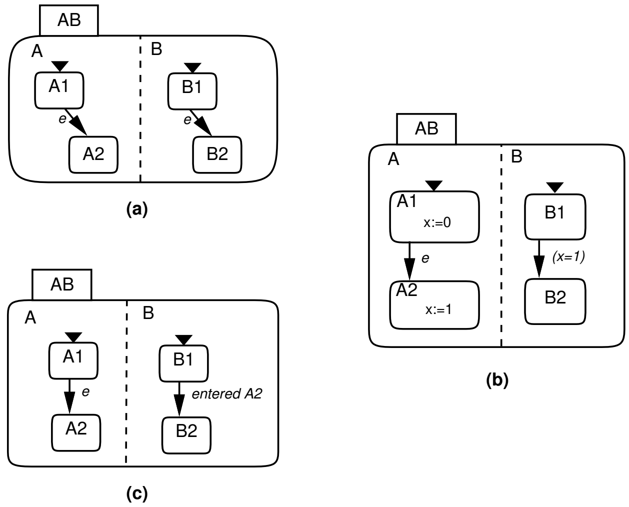


Figure 14: Data-dependent synchronization in Statecharts: (a) synchronization by common event, (b) synchronization by common data, (c) synchronization by status detection.

Synchronization by common event requires one process to wait for the occurrence of a specific event, which can be generated externally or by another process. In Figure 14(a), we can see how event e is used for synchronizing states A and B into substates *A2* and *B2*, respectively. Another method is that of **synchronization by common variable**, which requires one of the processes to update the variable with a suitable value. In Figure 14(b), B is synchronized into state *B2* when we assign the value "1" to variable x in state *A2*.

Still another method is **synchronization by status detection**, in which a process checks the status of other processes before resuming execution. In a case like this, the transition from *A1* to *A2* precipitated by event e, would cause B to transition from *B1* to *B2*, as shown in Figure 14(c).

3 SpecC+

In this section, we will present the SpecC+ language, which was specifically developed to capture directly a conceptual model possessing all the above discussed characteristics.

3.1 Language description

The SpecC+ view of the world is a hierarchical network of **actors**. Each actor possesses

- a set of ports through which the actor communicates with the environment;
- a set of state variables;
- a set of communication channels;
- a behavior which defines how the actor will change its state and perform communication through its ports when it is invoked.

Source code 4 shows the textual representation of the example in Figure 15, where an actor is represented by a rectangular box with curved corners.

```
typedef int       TData[16];

interface IData( void ) {
      TData   read( void );
      void    write( TData d );
      };

channel CData( void ) implements IData {
      bool          valid;
       event          s;
      TData          storage;

      TData   read( void ) {
             if( valid ) s.wait();
              return storage;
             }
      void    write( TData d ) {
              storage=d; valid = 1; s.notify();
             }
      };

actor X1( in TData i, out TData o ) { ... };
actor X2( in TData i, IData o ) {
       void main( void ) {
             ...
             o.write(...);
             }
      };

actor    X( in int a, IData c ) {
      TData   s;
      X1      x1( a, s );
      X2      x2( s, c );

      psm      main( void ) {
             x1 : ( TI, cond1, x2 );
             x2 : ( TOC, cond2, complete );
             }
      };

actor Y ( IData c, out int m ) {
      void      main( void ) {
             int      max, j;
             TData           array;

             array = c.read();
             max = 0;
              for( j = 0; j < 16; j ++ )
                     if( array[j] > max )
                           max = array[j];
             m = max;
             }
      };

actor B( in TData p, out int q ) {
      CData   ch;
      X       x( p, ch );
      Y       y( ch, q );

      csp      main( void ) {
             par { x.main(); y.main(); }
             }
      };
```

Source code 4

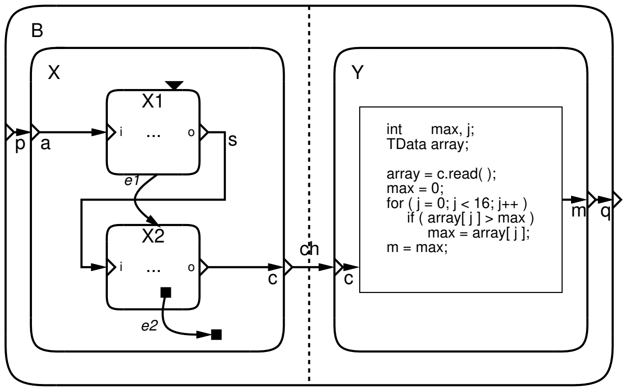


Figure 15: A sample SpecC+ specification.

There is an *actor* construct which capture all the information for an actor. An *actor* construct looks like a C++ class which exports an *main* method. The ports are declared in the parameter list. The state variable, channels and child actor instances are declared as typed variables, and the behavior is specified by the methods, or functions start from *main*. Actor construct can be used as a type to instantiate actor instances.

SpecC+ supports both **behavioral hierarchy** and **structural hierarchy** in the sense that it captures a system as a hierarchy of actors. Each actor is either a composite actor or a leaf actor.

Composite actors are decomposed hierarchically into a set of child actors. For structural hierarchy, the child actors are interconnected via the communication channels by child actor instantiation statements, similar to component instantiation in VHDL. For example, actor X is instantiated in line 57 of Source code 4 by mapping its port a and c to the ports (p) and communication channels (ch) defined in its parent actor B. For behavioral hierarchy, the child actors can either be concurrent, in which case all child actors are active whenever the parent actor is active, or can be sequential, in which case the child actors are only active one at a time. In Figure 15, actors B and X are composite actors. Note that while B consists of concurrent child actors X and Y, X consists of sequential child actors *X1* and *X2*.

Leaf actors are those that exist at the bottom of the hierarchy whose functionality is specified with imperative programming constructs. In Figure 15, for example, Y is a leaf actor.

SpecC+ also supports **state transitions**, in the sense that we can represent the sequencing between child actors by means of a set of transition arcs. In this language, an arc is represented as a 3-tuple $< T,\ C,\ N >$, where T represents the type of transition, C represents the condition triggering the transition, and N represents the next actor to which control is transferred by the transition. If no condition is associated with the transition, it is assumed to be "true" by default.

SpecC+ supports two types of transition arcs. A **transition-on-completion arc** (TOC) is traversed whenever the source actor has completed its computation and the associated condition evaluates as true. A leaf actor is said to have completed when its last statement has been executed. A sequentially decomposed actor is said to be complete only when it makes a transition to a special predefined completion point, indicated by the name *complete* in the next-actor field of a transition arc. In Figure 15, for example, we can see that actor X completes only when child actor *X2* completes and control flows from *X2* to the *complete* point when *cond2* is true (as specified by the arc $< TOC,\ cond2,\ complete >$ in line 36 of Source code 4). Finally, a concurrently decomposed actor is said to be completed when all of its child actors have completed. In Figure 15, for example, actor B completes when all the concurrent child actors X and Y have completed.

Unlike the TOC arc, a **transition-immediately arc** (TI) is traversed instantaneously whenever the associated condition becomes true, regardless of whether the source actor has or has not completed its computation. For example, in Figure 15, the arc $< TI,\ cond1,\ x2 >$ terminates *X1* whenever *cond1* is true and transfers control to actor *X2*. In other words, a TI arc effectively terminates all lower level child actors of the source actor.

Transitions are represented in Figure 15 with directed arrows. In the case of a sequentially-decomposed actor, an inverted bold triangle points to the first child actor. An example of such an initial child actor is *X1* of actor X. The completion of sequentially decomposed actors is indicated by a transition arc pointing to the completion point, represented as a bold square within the actor. Such a completion point is found in actor X (transition from *X2* labeled *e2*). TOC arcs originate from a bold square inside the source child actor, as does the arc labeled *e2*. TI arcs, in contrast, originate from the perimeter of the source child actor, as does the arc labeled *e1*.

SpecC+ supports both **data-dependent synchronization** and **control-dependent synchronization**. In the first method, actors can synchronize using common event. For example, in Figure 15, actor Y is the consumer of the data produced by actor X via channel c, which is of type *CData* in Source code 4. In the implementation of *CData* at line 8 of Source code 4, an event s is used to make sure Y can get valid data from X: the *wait* function over s will suspend Y if the data is not ready. In the second method, we could use a TI arc from actor B back to itself in order to synchronize all the concurrent child actors of B to their initial states. Furthermore, the fact that X and Y are concurrent actors enclosed in B automatically implements a barrier, since by semantics, B finishes when both the execution of X and Y are finished.

Communication in SpecC+ is achieved through the use of communication channels. Channels can be primitive channels such as variables and signals (like variable s of actor X in Figure 15), or complex channels such as object channels (like variable ch in Figure 15), which directly supports the hierarchical communication model discussed in Section 2.8.

The specification of an object channel is separated in the *interface* declaration and the *channel* definition, each of which can be used as data types for channel variables. The interface defines a set of function prototype declarations without the actual function body. For example, the interface *IData* in Figure 15 defines the function prototypes *read* and *write*. The channel encapsulates the communication media and provides a set of function implementations. For example, the channel *CData* encapsulates media s and *storage* and an implementation of methods *read* and *write*. The interface and the channel are related by the *implements* keyword. A channel related to an interface in this way is said to implement this interface, meaning the channel is obligated to implement the set of functions prescribed by the interface. For example, *CData* has to implement *read* and *write* since they appear in *IData*. It is possible that several channels can implement the same interface, which implies that they can provide different implementations of the same set of functions.

Interfaces are usually used as port data types in port declarations of an actor (as port c of actor Y at line 41 of Source code 4). A port of one interface type will be bound to a particular channel which implements such an interface during actor instantiation. For example, port c of actor Y is mapped to channel c of actor B, when actor Y is instantiated.

The fact that a port of interface type can be bound to a real channel until actor instantiation is called **late binding**. Such a late binding mechanism helps to improve the reusability of an actor description, since it is possible to plug in any channel as long as they implement the same interface.

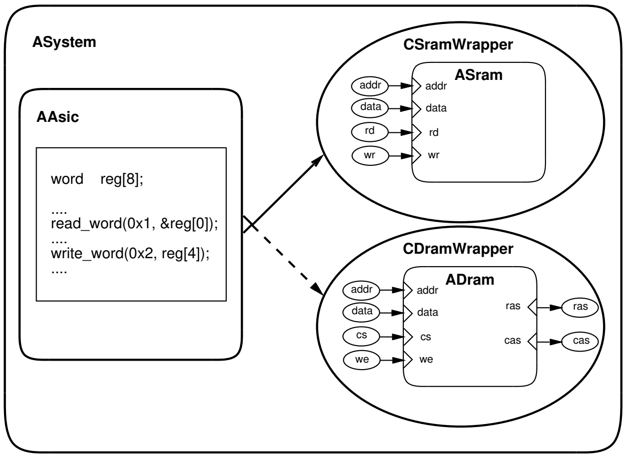


Figure 16: Component wrapper specification.

Consider, the example in Figure 16 as specified in Source code 5:

```
interface IRam( void ) {
      void      read_word( word a, word *d );
      void      write_word( word a, word d );
      };

actor AAsic( IRam ram ) {
      word    reg[8];

      void      main( void ) {
            ...
            ram.read_word( 0x0001, &reg[0] );
            ...
            ram.write_word( 0x0002, reg[4] );
            }
      };

actor ASram( in signal<word> addr,
      inout signal<word> data,
      in signal<bit> rd, in signal<bit> wr ) {
      ....
      };

actor ADram( in signal<word> addr,
      inout signal<word> data,
      in signal<bit> cs, in signal<bit> we,
      out signal<bit> ras, out signal<bit> cas ) {
      ....
      };

channel CSramWrapper( void ) implements IRAM {
      signal<word>  addr, data;   // address, data
      signal<bit>   rd, wr;       // read/write select
      ASram         sram( addr, data, rd, wr );

      void      read_word( word a, word *d ) { ... }
      void      write_word( word a, word d ) { ... }
      ...
      };

channel CDramWrapper( void ) implements IRam {
      signal<word>   addr, data; // address, data
      signal<bit>    cs, we;   // chip select, write enable
      signal<bit>    ras, cas; // row, col address strobe
      ADram          sram( addr, data, cs, we, ras, cas );

      void      read_word( word a, word *d ) { ... }
      void      write_word( word a, word d ) { ... }
      ...
      };

actor ASystem( void ) {
      CSramWrapper    ram;      // can be replaced by
      // CDramWrapper ram;      // this declaration
      AAsic           asic( ram );

      void     main( void ) { ... }
      };
```

Source code 5

The system described in this example contains an ASIC (actor *AAsic*) talking to a memory. The interface *IRam* specifies the possible transactions to access memories: read a word via *read_word* and write a word via *write_word*. The description of *AAsic* can use *IRam* as its port so that its behavior can make function calls to methods *read_word* and *write_word* without knowing how these methods are exactly implemented. There are two types of memories available in the library, represented by actors *ASram* and *ADram* respectively, the descriptions of which provide their behavioral models. Obviously, the static RAM

ASram and dynamic RAM *ADram* have different pins and timing protocols to access them, which can be encapsulated with the component actors themselves in channels called wrappers, as *CSramWrapper* and *CDramWrapper* in Figure 16. When the actor *AAsic* is instantiated in actor *ASystem* (lines 52 and 53 in Source code 5), the port *IRam* will be resolved to either *CSramWrapper* or *CDramWrapper*.

The improvement of reusability of this style of specification is two fold: first, the encapsulation of communication protocols into the channel specification make these channels highly reusable since they can be stored in the library and instantiated at will. If these channel descriptions are provided by component vendors, the error-prone effort spent on understanding the data sheets and interfacing the components can be greatly relieved. Secondly, actor descriptions such as *AAsic* can be stored in the library and easily reused without any change subject to the change of other components with which it interfaces.

It should be noted that while methods in an actor represent the behavior of itself, the methods of a channel represent the behavior of their callers. In other words, when the described system is implemented, the methods of the channels will be **inlined** into the connected actors. When a channel is inlined, the encapsulated media get exposed and its methods are moved to the caller. In the case of a wrapper, the encapsulated actors also get exposed.

Figure 17 shows some typical configurations. In Figure 17(a), two synthesizable components A and B (eg. actors to be implemented on an ASIC) are interconnected via a channel C, for example, a standard bus. Figure 17(b) shows the situation after inlining. The methods of the channel C are inserted into the actors and the bus wires are exposed. In Figure 17(c) a synthesizable component A communicates with a fixed component B (eg. an off-the-shelf component) through a wrapper W. When W is inlined, as shown in Figure 17(d), the fixed component B and the signals get exposed. In Figure 17(e) again a synthesizable component A communicates with a fixed component B using a predefined protocol, that is encapsulated in the channel C. However, B has its own built-in protocol, which is encapsulated in the wrapper W. A protocol transducer T has to be inserted between the channel C and the wrapper W in order to translate all transactions between the two protocols. Figure 17(f) shows the final situation, when both channels C and W are inlined.

SpecC+ supports the specification of **timing** explicitly and distinguishes two types of timing specifications, namely timing constraints and timing delays, as discussed in Section 2.7. At the specification level timing constraints are used to specify time limits that have to be satisfied. At the implementation level computational delays have to be noted.

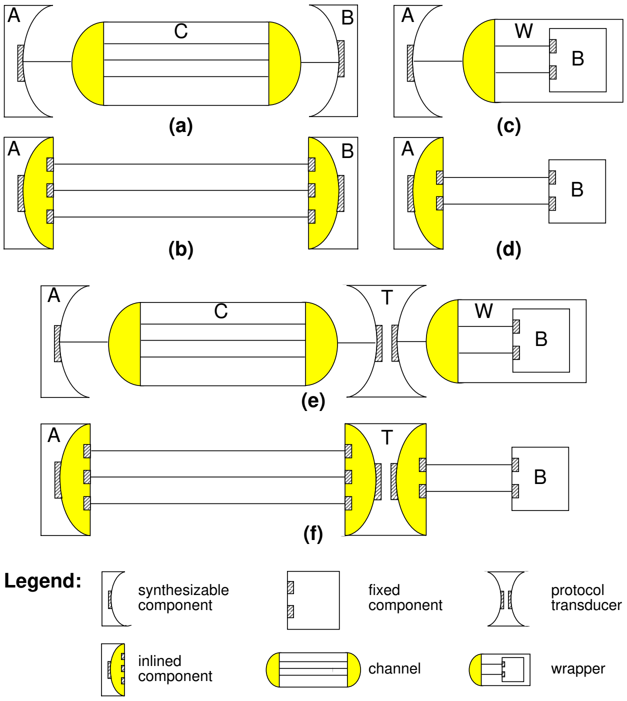


Figure 17: Common configurations before and after channel inlining: (a)/(b) two synthesizable actors connected by a channel, (c)/(d) synthesizable actor connected to a fixed component, (e)/(f) protocol transducer.

Consider, for example, the timing diagram of the read protocol for a SRAM, as shown earlier in Figure 10. The protocol visualized by the timing diagram can be used to define the *read_word* method of the SRAM channel above (line 35 in Source code 5). The following code segment shows the specification of the read access to the SRAM:

```
void read_word( word a, word *d ) {
    do {
        t1: { addr = a; }
        t2: { rd = 1; }
        t3: { }
        t4: { *d = data; }
        t5: { addr.disconnect(); }
        t6: { rd = 0; }
        t7: { break; }
    }
  timing {
```

```
        range( t1; t2; 0; );
        range( t1; t3; 10; 20 );
        range( t2; t3; 10; 20 );
        range( t3; t4; 0; );
        range( t4; t5; 0; );
        range( t5; t7; 10; 20 );
        range( t6; t7; 5; 10 );
    }
};
```

Source code 6

The *do-timing* statement effectively describes all information contained in the timing diagram. The first part lists all the events of the diagram. Events are specified as a label and its associated piece of code, which describes the change on signal values. The second part is a list of *range* statements, which specify the timing constraints or timing delays using the 4-tuples as described in Section 2.7.

This style of timing description is used at the specification level. In order to get an executable model of the protocol, scheduling has to be performed for each *do-timing* statement. Source code 7 shows the implementation of the *read_word* method which follows an ASAP scheduling, where all timing constraints are replaced by delays, which are specified using the *waitfor* function.

```
void read_word( word a, word *d ) {
    addr = a;
    rd = 1;
    waitfor( 10 );
    *d = data;
    addr.disconnect();
    rd = 0;
    waitfor( 10 );
};
```

Source code 7

4 Conclusion

The purpose of this report is to demonstrate the need for a direct mapping between the conceptual model characteristics and the language constructs we use to capture those characteristics. In the absence of such a one-to-one correspondence, the task of system specification can become cumbersome and the likelihood of errors or incompleteness in the specification is increased. We presented various characteristics of common conceptual models. For the particular case of embedded systems, we have shown that the ease with which the constructs in the SpecC+ language are able to capture conceptual model characteristics makes the language well-suited for embedded system specification. When there is a good match between a language and a model, we can expect shorter specification times and fewer errors, in addition to enhancing the comprehensibility and adaptability of the specifications themselves.

5 References

[Ag90] G. Agha; "The Structure and Semantics of Actor Languages"; *Lecture Notes in Computer Science, Foundation of Object-Oriented Languages*; Springer-Verlag, 1990.

[AG96] K. Arnold, J. Gosling; *The Java Programming Language*; Addison-Wesley, 1996.

[DH89] D. Drusinsky and D. Harel. "Using Statecharts for hardware description and synthesis". In *IEEE Transactions on Computer Aided Design*, 1989.

[GVN93] D.D. Gajski, F. Vahid, and S. Narayan. "SpecCharts: a VHDL front-end for embedded systems". UC Irvine, Dept. of ICS, Technical Report 93-31, 1993.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems.* New Jersey, Prentice Hall, 1994.

[Har87] D. Harel; "StateCharts: a Visual Formalism for Complex Systems"; *Science of Programming*, 8, 1987.

[LS96] E.A. Lee, A. Sangiovanni-Vincentelli; "Comparing Models of Computation"; *Proc. of ICCAD*; San Jose, CA, Nov. 10-14, 1996.

[St87] B. Stroustrup; *The C++ Programming Language*; Addison-Wesley, Reading, 1987.