Syntax and Semantics of the SpecC+ Language

Jianwen Zhu, Rainer Dömer, Daniel D. Gajski

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

Abstract

In this paper, we describe the goals for the development of an executable modeling language in the context of a homogeneous codesign methodology featuring the synthesis, reuse and validation flow. A C based language called SpecC+ is proposed as an attempt to achieve these goals. The syntax and semantics of the language is presented and compared with existing HDLs and we conclude it is conceptually more abstract, syntactically simpler, and semantically richer.

1 Introduction

To tackle the complexities involved in systems-on-a-chip design, designers often follow two strategies. In favor of synthesis, the top-down strategy starts from an abstract specification of the systems functionality, and then performs stepwise refinement into implementations with the help of various synthesis tools. In favor of reuse, the bottom-up approach tries to implement the systems functionality by integrating existing components. In reality, these two approaches have to be combined, because for the top-down approach, it is often unaffordable for new designs to start from scratch, whereas for the bottom-up approach, it is rarely possible to build systems completely from existing components without adding new ones.

New codesign methodologies have to be developed to support the seamless mixture of both strategies. One such methodology is illustrated in Figure 1(a), where boxes represent **design tasks**, and ellipses represent design descriptions, or their abstract form, the **design models**, which are shown in detail in Figure 1(b).

For the synthesis flow, designers may start from the specification stage, where the intended functionality of the system is specified in terms of a set of behaviors communicating via a set of channels. Note that at this stage, the designers may already decide to use some components from the reuse library, so these components should also be able to be instantiated in the specification.

At the second stage, designers may partition the behaviors onto ASICs and programmable processors. The design model generated at this stage should carry these design decisions.

At the communication and synchronization refinement stage, the abstract channels among behaviors have to be resolved into something implementable, such as behaviors transferring data over a bus according to a predefined protocol. The design model generated at this stage is simply the replacement of the abstract channels into detailed ones.

The final stage might be the compilation of software into binary code and high level synthesis of ASIC behavior into register transfer level designs. The model generated at this stage might be a software behavior (in the form of an instruction stream) communicating with a hardware behavior (in the form of a RTL netlist) via a bus functional model, which translates each instruction into a series of activities on the bus wires.

For the validation flow, the design model generated at each stage should be simulated before proceeding to the next synthesis step. For the reuse flow, there exists a library of reusable components, which as a repository of design models, can be "downloaded" or "uploaded" by synthesis tools or designers.

For this methodology, it is desirable that we

Objective 1: capture the design model at each design stage in a **single language**;

Objective 2: make the language **executable**.

The meeting of Objective 1 will greatly simplify the codesign methodology since synthesis tasks are now merely transformations from one program into another using the same language, and validation tasks are just the program executions. This **homogeneous** approach is in contrast with traditional approaches where systems functionality is specified using

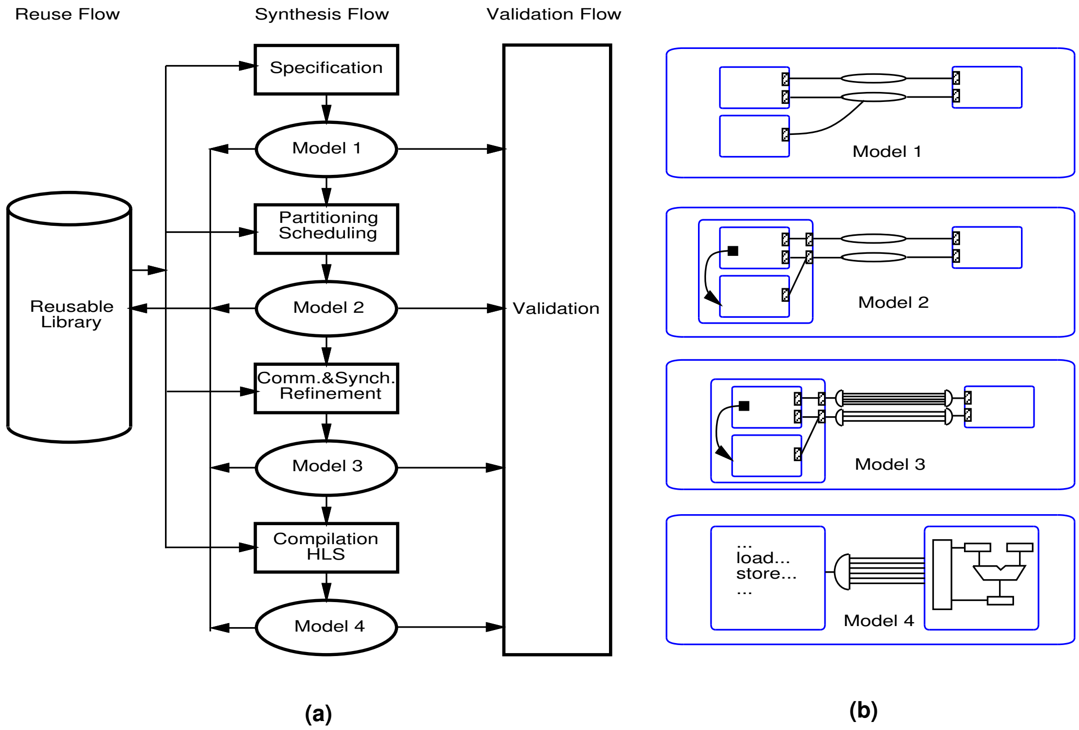


Figure 1: A homogeneous codesign methodology

one specification language, or a mixture of several, and is then opaquely synthesized into design descriptions represented by different languages, for example, C for software, HDLs for hardware.

The executable requirement (Objective 2) of the language is also of crucial importance for validation. For example, the specification needs to be validated to make sure that the functionality captured is what the designer wants. Validation is even necessary for the intermediate design models, for example those after the communication refinement stage, where the interaction between hardware and software becomes very tricky and error-prone.

It is also desirable for the language to

Objective 3: encourage **reuse**;

Objective 4: facilitate **synthesis**.

The design models stored in the library can be checked in and out easily if they are captured in the same language as the design. Furthermore, the language should be designed in such a way that a design model of one component is encouraged to be decoupled from those of other components. Components captured in this way tend to be more reusable.

Since we use the same language to model the design generated by each synthesis task, a design task can be considered as a refinement of one program into another program. It is desirable for the change made by the transformation to be incremental, meaning that it will only work on a local part of the program without affecting other parts. For example, the communication refinement task should only replace abstract channels into more detailed ones without changing the model of the behaviors which use this channel. The locality of changes makes the synthesizer easier to write and the results generated more understandable.

In this paper, we attempt to achieve the objectives above by proposing a new C based language called SpecC+. For SpecC+ to be able to model designs of mixed abstraction levels, it has to be able to capture concepts commonly found in embedded systems, such as concurrency, state transitions, structural and behavioral hierarchy, exception handling, timing, communication and synchronization, as discussed in [GVNG94]. These concepts have to be organized in an orthogonal way so that the language can be minimal. Section 2 presents the set of constructs in SpecC+ that support these concepts. While Section 2 gives an idea that SpecC+ is conceptually more abstract and syntactically simpler than VHDL, Section 3 describes why it is semantically richer. Section 4 shows how the objectives defined above are achieved by presenting an example system. Finally Section 5 concludes this paper with comparisons to traditional HDLs.

2 Syntax

In this section, we give an introduction of the constructs provided by SpecC+.

2.1 Basic structure

The SpecC+ view of the world is a hierarchical network of **actors** interconnected by **channels**.

Each actor possesses

- a set of ports, such as *p1* and *p2* of actor Z in Figure 2;
- a set of state variables;
- a set of channels, such as *c1* and *c2* in actor Z in Figure 2;
- and a behavior, which contains computations updating the state variables and communications via its ports connected to the channels.

The actor may be a composite actor which contains a set of child actors, in which case the behavior of the composite actor is specified by composing the behavior of its child actors, for example, the behavior of actor Z is composed of the behaviors of X and Y in Figure 2.

SpecC+ provides the *actor* construct to capture all the information for an actor. This *actor* construct looks like a C++ class which exports a *main* method, for example, the actor X in Figure 2(b). Ports are declared in the parameter list. State variable, channels and child actor instances are declared as typed variables, and the behavior is specified by the methods, or functions starting from *main*. The *actor* construct can be used as a type to instantiate actor instances.

A channel is an entity responsible for communication. It can be a primitive channel in the form of built-in data types such as *int, char* and *float*, or it can be a complex channel whose specification is split into two constructs: the *interface* construct declares what kind of communications a channel can perform; the *channel* construct provides the implementation of how the communication is performed. The *interface* is usually used as a type to declare ports, whereas the *channel* is used as a type to instantiate channels. Section 2.3 explains in detail how complex channels are specified and why they are useful.

In summary, the SpecC+ program consists of a list of specifications of actors and channels.

2.2 Hierarchy

The *actor* construct can orthogonally capture both **structural** hierarchy and **behavioral** hierarchy.

Structural hierarchy means that composite actors can be decomposed into child actors interconnected by channels. Channels define the set of paths through which the child actors communicate, and when they communicate, how the communication is performed. However, when the communication is performed is determined by the behavior of the child actors. Structural information can be captured by actor instantiation where its ports are mapped to channels or the ports of the parent actor.

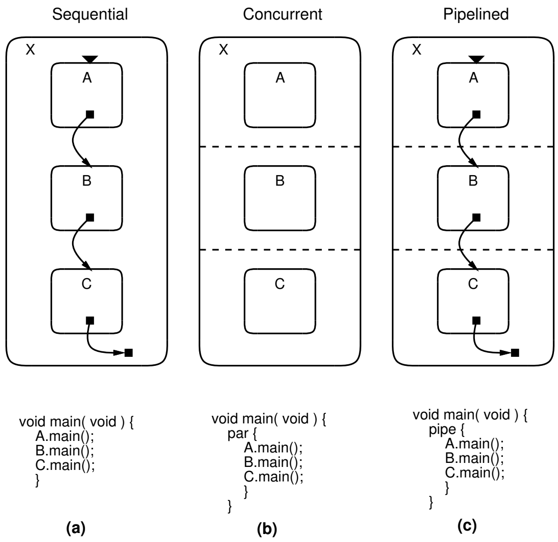
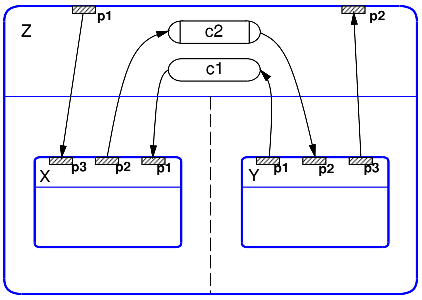


Figure 3: Behavioral hierarchy

Behavioral hierarchy means that the behavior of a composite actor is composed of behaviors of the child actors. There are constructs to specify how behaviors of child actors are composed in time into more complex behavior of the composite actor. For example, we can specify a behavior being the sequential composition of the behaviors of its child actors using sequential statements, as shown in Figure 3(a), where X finishes when the last actor C finishes. Second, we can use the parallel composition using the *par* construct, as shown in Figure 3(b), where X finishes when all its child actors A, B and C are finished. Also, the pipelined composition is possible using the *pipe* construct, as shown in Figure 3(c), where X starts again when the slowest actor finishes.

In summary, the structural hierarchy is captured by the tree of actor instantiations, whereas the behavioral hierarchy is captured by the tree of function calls to the actor *main* methods.



(a)

```
interface ILeft( void ) {
        void    write( int val );
        };

interface IRight( void ) {
        int     read( void );
        };

channel CSharedI( void )
                implements ILeft, IRight {
        int     storage;
        bool    valid;

        void    write( int val ) {
                storage = val;
                valid = 1;
                }
        int     read( void ) {
                while( !valid );
                return storage;
                }
        };


actor   Y( out int p1, IRight p2, out int p3 ) {
        int     local;

        void    write( void ) {
                ...
                local = p2.read();
                ...
                }
        };
```

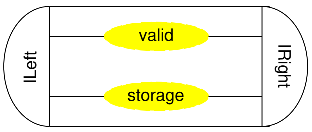
```
actor   X( in int p1, ILeft p2, in int p3 ) {
        int     local;

        void    main( void ) {
                ....
                p2.write( local );
                ...
                }
        };


actor   Z( in int p1, out int p2 ) {
        int             c1;
        CShared     c2;

        X       x( c1, c2, p1 );
        Y       y( c1, c2, p2 );

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

(b)

Figure 2: Basic structure of a SpecC+ program

2.3 Communication

We have mentioned in Section 2.1 that the channel concept is used to model communication and the specification of a complex channel is split into the *interface* and *channel* constructs. The *interface* construct encapsulates a set of method prototype declarations, which specify what kind of communications a channel can perform. The channel encapsulates a set of media in the form of variables, for example a set of storages or a set of wires, and the set of method implementations which specify how the communications are performed.



(a)

```
interface ILeft( void ) {
        void    write( int val );
        };
interface IRight( void ) {
        int     read( void );
        };
channel CShared( void )
                implements ILeft, IRight {
        int     storage;
        bool    valid;
        void    write( int val ) {
                storage = val;
                valid = 1;
                }
        int     read( void ) {
                while( !valid );
                return storage;
                }
        };
```

(b)

Figure 4: Shared memory channel

Figure 4 shows a shared variable channel which can be accessed by concurrent actors. The communications that can be performed are *read* and *write*, as declared in the *ILeft* interface and *IRight* interface respectively.

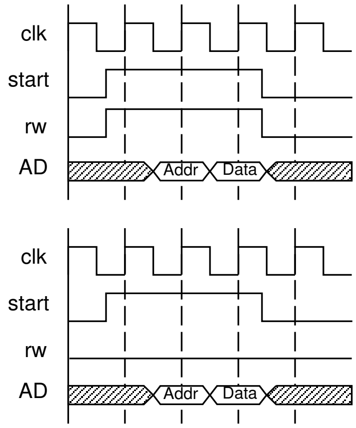
The channel *CShared* encapsulates the variable *storage* being shared, a *valid* bit, which the *write* operation has to set and the *read* operation has to spinwait for, as well as definitions of the *read* and *write* methods.

The channel is related to the interfaces by the *implements* keyword followed by the list of interfaces, which implies that it is mandatory for the channel to implement the methods declared in the interfaces.

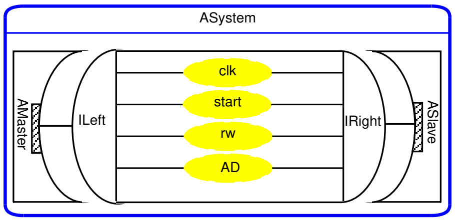
A more complex example is shown in Figure 5, which describes a bus channel with a synchronous protocol, which is illustrated in Figure 5(a). Two interfaces are declared: *ILeft* specifies what a master of the channel can perform, in this case the *read_word* and *write_word* methods; *IRight*, specifies what a slave of the channel can perform, in this case the method *monitor*.

The channel *CBus* in Figure 5(c) encapsulates the set of wires consisting of *clk, start, rw* and *AD*, as well as the method bodies. For example, the *read_word* method called by master actors will initiate a bus cycle by asserting the *start* signal, and then raise the *rw* signal, sample the *data* and finally deassert the *start* signal. As another example, the *monitor* method called by slave actors will watch the activities on the wires and detect if a read cycle or write cycle is ongoing, and then in turn perform the appropriate operations.

The users of this channel, in Figure 5(c) actors *AMaster* and *ASlave*, will use the *ILeft* and *IRight*



(a)



(b)

```
interface ILeft( void ) {
    void   read_word( word addr, word *d );
    void   write_word( word addr, word d );
    };

interface IRight( void ) {
    void   monitor(
          void   (*grab_data)( word addr, word *d ),
          void   (*deliver_data)( word addr, word d )
          );
    };

channel CBus( void ) implements ILeft, IRight {
    clock          clk;
    signal<bit>    start;
    signal<bit>    rw;
    signal<word>   AD;

    void   read_word( word addr, word *d ) {
          start = 1, rw = 1,        clk.tick();
          AD = addr,                clk.tick();
          *d = AD, start = 0, rw = 0, clk.tick();
          }
    void   write_word( word addr, word d ) {
          start = 1, rw = 0,     clk.tick();
          AD = addr,             clk.tick();
          AD = d, start = 0,     clk.tick();
          }

    void   monitor(
          void   (*grab_data)( word addr, word *d ),
          void   (*deliver_data)( word addr, word d )
          ) {
          word   a, d;

          while( start == 0 ) clk.tick();
          if( rw == 1 ) {              // read cycle
                clk.tick();
                a = AD, clk.tick();
                (*grab_data)( a, &d ), AD = d, clk.tick();
                }
          else {
                clk.tick();
                a = AD, clk.tick();
                d = AD, (*deliver_data)( a, d ), clk.tick();
                }
          }
    };
```

```
actor   AMaster( ILeft bus ) {
   word   d;

   void   main( void ) {
      ...
      bus.read( 0x10, &d );
      d ++; bus.write( 0x10, d );
      ...
      }
   };

actor   ASlave( IRight bus ) {
   word   storage[0x100];

   void   my_grab_data( word addr, word *d ) {
      *d = storage[addr]; }
   void   my_deliver_data( word addr, word d ) {
      storage[addr] = d; }

   void   main( void ) {
      for( ; ; ) {
         bus.monitor( my_grap_data, my_deliver_data );
         }
      }
   };

actor   ASystem( void ) {
   CBus       bus;
   AMaster      master( bus );
   ASlave      slave( bus );

   csp   main( void ) {
      par { master.main(); slave.main(); }
      }
   };
```

(c)

Figure 5: Synchronous bus channel

interfaces as their ports, which will later be mapped to the *CBus* channel when they are instantiated. Note that it is prohibited by the language for a port to be mapped to a channel which does not implement the interface type of the port.

The difference between methods in a channel and methods in an actor has to be emphasized. While methods of an actor specify the behavior of itself, methods of a channel specify behavior of the caller, in other words, they will get **inlined** into connected actors when the system is finally implemented. When a channel is inlined, the encapsulated media are exposed, the methods are moved to the caller and the interface port is flattened into ports connected to the exposed media.

The fact that a port of an interface type can be resolved to a real channel at the time of its instantiation is called **late binding**. Such a late binding mechanism makes it possible for an actor to perform function calls via the interface port without knowing what kind of channel it will be eventually mapped to. In this way, any channel can be plugged in as long as it conforms to this interface.

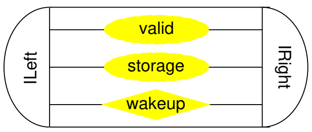
Such a "plug-and-play" feature is essential to both reuse and incremental refinement. For reuse, for example, it is possible to replace the channel *CBus* in Figure 5(c) with another bus channel that uses an asynchronous bus protocol without affecting the description of *AMaster* and *ASlave* at all, as long as that channel implements both interfaces *ILeft* and *IRight*. It is obvious that the models *AMaster* and *ASlave* are highly reusable, since they can be adapted to different buses with different wires and protocols. For incremental refinement, for example, the channel *CBus* can be replaced without affecting the connected actors by another more detailed channel with the same interface, which might contain a memory actor, the wires which access this memory, as well as the protocols used.

2.4 Synchronization

Concurrent actors often need to be synchronized to be cooperative. In SpecC+, there is a built-in type *event* which can serve as the basic unit of synchronization.

An event can be considered as a channel with a set of methods including *wait* and *notify*. A *wait* call on an event will suspend the caller. A *notify* call on an event will resume all actors that are suspended due to a wait on this event .

The shared memory example introduced in Figure 4



(a)

```
interface ILeft( void ) {
        void    write( int val );
        };
interface IRight( void ) {
        int     read( void );
        };
channel CShared( void )
                implements ILeft, IRight {
        int     storage;
        bool    valid;
        event  wakeup;
        void    write( int val ) {
                storage = val;
                valid = 1;
                notify( wakeup );
                }
        int     read( void ) {
                if ( !valid )
                        wait( wakeup );
                return storage;
                }
        };
```

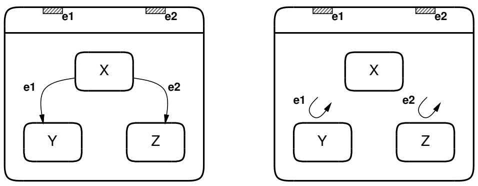
(b)

Figure 6: Event and shared memory channel

can be rewritten using the event mechanism as shown in Figure 6. In Figure 4(b), the *read* method has to poll the valid bit constantly, which is unefficient. A better way, as shown in Figure 6(b), is to use an event called *wakeup*, so that a call to the *read* method can suspend itself when *valid* is false, and will be resumed later when a write operation notifies the *wakeup* event.

2.5 Exceptions

In order to model exceptions in an embedded system SpecC+ supports two concepts, namely **abortion** and **interrupt**, as shown in Figure 7.




```
void    main( void ) {
    try { x.main(); }
    trap( e1 ) { y.main(); }
    trap( e2 ) { z.main(); }
    }
```

(a)

```
void    main( void ) {
    try { x.main(); }
    interrupt( e1 ) { y.main(); }
    interrupt( e2 ) { z.main(); }
    }
```

(b)

Figure 7: Exception handling: (a) abortion, (b) interrupt.

The *try-trap* construct shown in Figure 7(a) aborts actor x immediately when one of the events $e1$, $e2$ occurs. The execution of actor x (and all its child actors) is terminated without completing its computation and control is transferred to actor y in case of $e1$, to actor z in case of $e2$. This type of exception usually is used to model the reset of a system.

On the other hand the *try-interrupt* construct, as shown in Figure 7(b), can be used to model interrupts. Here again execution of actor x is stopped immediately for events $e1$ and $e2$, and actor y or z, respectively, is started to service the interrupt. After completion of interrupt handlers y and z control is transferred back to actor x and execution is resumed right at the point where it was stopped.

For both types of exceptions, in case two or more events happen at the same time, priority is given to the first listed event.

2.6 Timing

In the design of embedded systems the notion of real time is an important issue. However, in traditional imperative languages such as C, only the ordering among statements is specified, the exact information on when these statements are executed, is irrelevant. While these languages are suitable for specifying functionality, they are unsufficient in modeling embedded systems because of the lack of timing information. Hardware description languages such as VHDL overcome this problem by introducing the notion of time: statements are executed at discrete points in time and their execution delay is zero. While VHDL gives an exact definition of timing for each statement, such a treatment often leads to **over-specification**.

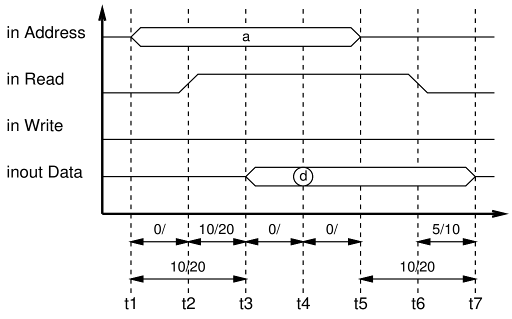
One obvious over-specification is the case when VHDL is used to specify functional behavior. The timing of functional behaviors is unknown until they are synthesized. The assumption of zero execution time is too optimistic and there are chances to miss design errors during specification validation.

SpecC+ overcomes this problem by differentiating between **timed behavior**, which executes in zero time, and **untimed behavior**, on which no assumption of timing can be made. Syntactically an *untimed* modifier type will make the behavior contained in a function untimed behavior. The execution semantics of SpecC+ as described in Section 3 will allow the synchronized execution of timed and untimed behavior.

Other cases of over-specification are timing constraints and timing delays, where events have to happen, or, are guaranteed to happen in a **time range**, instead of at a fixed point in time, as restricted by VHDL.

SpecC+ supports the specification of timing explicitly and distinguishes two types of timing specifications, namely **constraints** and **delays**. 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.

Consider, for example, the timing diagram of the read protocol for a static RAM, as shown in Figure 8(a). In order to read a word from the SRAM, the address of the data is supplied at the *address* port



(a)

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

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

channel C_SRAM( void )
   implements I_SRAM {
     signal<word> Address, Data;
     signal<bit>    Read, Write;
     A_SRAM sram(
                  Address, Data,
                  Read, Write );

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

(b)

```
void  read_word(
      word a, word *d ) {
   do {
      t1 : { Address = a; }
      t2 : { Read = 1; }
      t3 : {  }
      t4 : { *d = Data; }
      t5 : { Address.disconnect(); }
      t6 : { Read = 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 );
      }
   }
```

(c)

```
void  read_word(
      word a, word *d ) {
   Address = a;
   Read = 1;
   waitfor(10);
   *d = Data;
   Address.disconnect( );
   Read = 0;
   waitfor(10);
   }
```

(d)

Figure 8: Read protocol of a static RAM: (a) timing diagram, (b) SRAM channel, (c) timing at specification level, (d) timing at implementation level.

and the read operation is selected by assigning 1 to the *read* and 0 to the *write* port. The selected word then can be accessed at the *data* port. The diagram in Figure 8(a) explicitly specifies all timing constraints that have to be satisfied during this read access. These constraints are specified as arcs between pairs of events annotated with x/y, where x specifies the minimum and y the maximum time between the value changes of the signals. The times are measured in real time units such as nanoseconds.

Figure 8(b) shows the SpecC+ source code of a SRAM channel *C_SRAM*, which instantiates the actor *A_SRAM*, and the signals, which are mapped to the ports of the SRAM. Access to the memory is provided by the *read_word* method, which encapsulates the read protocol explained above (due to space constraints write access is ignored).

Figure 8(c) shows the source code of the *read_word* method at the specification level. The *do-timing* construct used here effectively describes all information contained in the timing diagram. The first part of the construct lists all the events of the diagram, which are specified as a label and its associated piece of code, which describes the changes of signal values. The second part is a list of *range* statements, which specify the timing constraints or timing delays using 4-tuples $T = (e1, e2, min, max)$, where $e1$ and $e2$ are event labels and min and max specify the minimum and maximum time, respectively, between these events.

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. Figure 8(d) shows the implementation of the *read_word* method after an ASAP scheduling is performed. All timing constraints are replaced by delays, which are specified using the *waitfor* construct.

3 Execution semantics

In this section, we give a formal definition of the execution semantics of a SpecC+ program. We use a graph-based notation similar to [Fr95].

3.1 Graph-based representation

In its most primitive form with all the high level constructs flattened and syntactical sugar stripped, the SpecC+ program can be represented by a directed graph. Figure 9 shows an example of such a graph.

The flattened SpecC+ program can be broken into blocks of statement sequences, which become vertices of the graph. Four types of constructs, which become edges of the graph, break the program into blocks:

Type 1: branches, including *if-else*, *while*, *for* loops etc,

Type 2: delay (*waitfor*) statements,

Type 3: *wait* statements,

Type 4: *notify* statements.

Note that while Type 1 constructs break the program into basic blocks as in traditional control flow analysis, Type 2-4 constructs break the basic blocks further.

A vertex is called a **timed** vertex, if the execution of the vertex takes zero time, as determined by the syntax of the language. It is referred to as an **untimed** vertex otherwise. Another way of classifying the vertices is to call a vertex a **reactive** vertex if it is preceded by a *wait* statement. Otherwise, it is called a **non-reactive** vertex.

Furthermore, there are special vertices used to represent high level language constructs such as *par, cpar* and *pipe*, namely, *fork* and *join* vertices. For the simplicity of the presentation, we ignore the constructs *try-trap* and *try-interrupt* in this discussion.

There are two types of edges. The **sensitizing** edges, represented by dotted arrows in Figure 9(b), are derived from the sequencing information of the program where the control flows to reactive vertices. The execution of the source vertex of a sensitizing edge will make the sink vertex sensitive to the events that it waits for. Each sensitizing edge is associated with a *condition*.

The **triggering** edges are represented by solid arrows in Figure 9(b) and are annotated by *condition/delay* pairs. The execution of the source vertex of a triggering edge will cause the sink vertex to execute at *delay* time steps later if the *condition* is true. If the sink vertex is a reactive vertex, it also has to be sensitive in order to be executed. There are three sources of program information for the triggering edges: (1) sequencing information where control flows to non-reactive vertices; (2) delay information associated with *waitfor* statements; (3) synchronization information associated with *wait-notify* pairs.

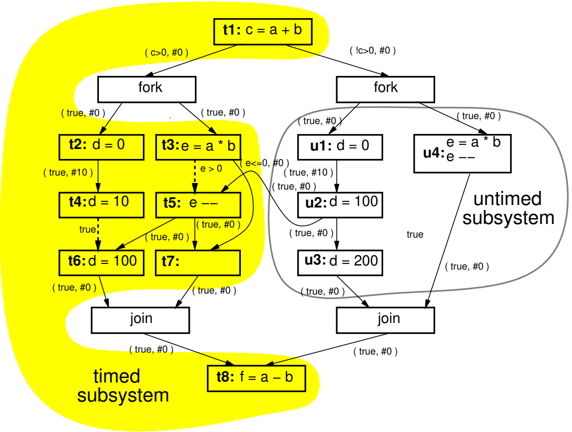
The subgraph spanned by the set of timed vertices is called the **timed subsystem**. The subgraph spanned by the untimed vertex set is called the **untimed subsystem**. The timed subsystem is intended to model the synthesized hardware using discrete event semantics equivalent to VHDL. The untimed subsystem is intended to model unsynthesized behavior or software, whose timing is unknown.

More formally, a SpecC+ program can be represented as a graph $G = \langle V, v_0, E_s, E_t \rangle$, where

- $V = T \cup U \cup Fork \cup Join$, where T is the set of timed vertices, U is the set of untimed vertices, $Fork$ is the set of *fork* vertices, $Join$ is the set of *join* vertices;
- $v_0 \in V$ is the start vertex;
- $E_s \subseteq V \times V \times B$ represents the conditional sensitizing edges, where B is the set of boolean expressions;
- $E_t \subseteq V \times V \times B \times Z^+$ represents the conditional triggering edges with delays, where Z^+ is the set of positive integers.

```
timed { c = a + b; }
if( c > 0 ) {
   par {
      timed { d = 0;   waitfor(10); d = 10;   wait(e1); d = 100; }
      timed { e = a * b; if( e > 0 ) { wait(e2); e --; notify(e1); } }
      }
   }
else {
   par {
      untimed { d = 0; waitfor(10); d = 100; notify(e2); d = 200; }
      untimed { e = a * b; e --;}
      }
   }
timed { f = a – b; }
```

(a)



(b)

Figure 9: Execution semantics: (a) pseudo SpecC+ code, (b) graph-based representation.

For convenience of notation, we define the set of reactive vertices to be

$$W = \{v | \exists v' \in V, b \in B, \langle v', v, b \rangle \in Es\}.$$

3.2 Semantics

The state of the computation can be represented by $s = \langle N, R, D, M, C \rangle$, where

- $N \subseteq V$ is the set of sensitive vertices,
- $R \subseteq V$ is the set of vertices ready to be executed,
- $D \subseteq V \times Z^+$ is the set of all pairs $\langle v, d \rangle$, such that vertex v has d time steps left before it can execute,
- M is the memory store that maps each variable to its current value,
- C is the memory store which holds the execution context of all the vertices.

We assume the existence of the following operations:

- $v = \mathit{SelectOneOf}(R)$ deterministically chooses one vertex v from the set of ready vertices R .
- $M' = \mathit{ExecTimed}(v, M)$ applies the code associated with timed vertex v to the memory store and returns a new memory store M' .
- $\langle t, M', C' \rangle = \mathit{ExecUntimed}(v, M, C)$ applies the code associated with untimed vertex v under a context C to the memory store for some host time. It will return a triple $\langle t, M', C' \rangle$, where $t \in Z^+$ represents the time steps elapsed during the execution, M' represents the updated memory store and C' represents the new context. Note that the granularity for each execution of the untimed vertex, that is, how much code it executes and how many time steps it will take, can be specified by the user.
- $\mathit{Finished}(v, C)$ returns whether vertex v has finished its execution, where C is the context.
- $\mathit{Eval}(b, M)$ evaluates the boolean expression b using the memory store M and returns TRUE or FALSE.

We define $s' = \mathit{Next}(\langle N, R, D, M, C \rangle)$ as $\langle N', R', D', M', C' \rangle$, where

- **Type I Transition** represents the **execution** of a vertex at the current time step:

  if $R \neq \emptyset$, then Let $v = \mathit{SelectOneOf}(R)$

  if $v \in T$, then

$$
\begin{aligned}
M' &= \mathit{ExecTimed}(v, M) \\
C' &= C \\
N' &= N - \{v\} \cup \{v' | \langle v, v', b \rangle \in Es \\
&\quad \wedge \mathit{Eval}(b, M')\} \\
R' &= R - \{v\} \cup \{v' | \langle v, v', b, d \rangle \in Et \\
&\quad \wedge \mathit{Eval}(b, M') \ \wedge \ d = 0 \\
&\quad \wedge (v \in W \ \wedge \ v \in N' \ \vee \ v \notin W)\} \\
D' &= D \cup \{\langle v', d \rangle | \langle v, v', b, d \rangle \in Et \\
&\quad \wedge \mathit{Eval}(b, M') \ \wedge \ d > 0\}
\end{aligned}
$$

  else if $v \in \mathit{Fork}$, or if $v \in \mathit{Join} \wedge$

$$\forall v' \in \{v' | \langle v', v, b \rangle \in Et\}, \ \mathit{Finished}(v', C)$$

  then

$$
\begin{aligned}
M' &= M \\
C' &= C \\
N' &= N - \{v\} \cup \{v' | \langle v, v', b \rangle \in Es \\
&\quad \wedge \mathit{Eval}(b, M')\} \\
R' &= R - \{v\} \cup \{v' | \langle v, v', b, d \rangle \in Et \\
&\quad \wedge \mathit{Eval}(b, M') \ \wedge \ d = 0 \\
&\quad \wedge (v \in W \ \wedge \ v \in N' \vee v \notin W)\} \\
D' &= D \cup \{\langle v', d \rangle | \langle v, v', b, d \rangle \in Et \\
&\quad \wedge \mathit{Eval}(b, M') \ \wedge \ d > 0\}
\end{aligned}
$$

  else if $v \in \mathit{Join} \wedge$

$$\exists v' \in \{v' | \langle v', v, b \rangle \in Et\}, \ \neg \mathit{Finished}(v', C)$$

  then

$$
\begin{aligned}
M' &= M, \\
C' &= C \\
N' &= N \\
R' &= R - \{v\} \\
D' &= D
\end{aligned}
$$

  else if $v \in U$, then

$$\langle t, M', C' \rangle = \mathit{ExecUntimed}(v, M, C)$$

  if $\mathit{Finished}(v, C')$ then

$$
\begin{aligned}
N' &= N - \{v\} \cup \{v' | \langle v, v', b \rangle \in Es \\
&\quad \wedge \mathit{Eval}(b, M')\} \\
R' &= R - \{v\} \cup \{v' | \langle v, v', b, d \rangle \in Et \\
&\quad \wedge \mathit{Eval}(b, M') \ \wedge \ d = 0 \\
&\quad \wedge (v \in W \ \wedge \ v \in N' \vee v \notin W)\} \\
D' &= D \cup \{\langle v', d + t \rangle | \langle v, v', b, d \rangle \in Et \\
&\quad \wedge \mathit{Eval}(b, M') \ \wedge \ d > 0\}
\end{aligned}
$$

  else

$$
\begin{aligned}
N' &= N \\
R' &= R - \{v\} \\
D' &= D \cup \{\langle v, t \rangle\}
\end{aligned}
$$

- **Type II Transition** represents the **advance of time** to the next step when a vertex can execute.

  if $R = \emptyset$ and $D \neq \emptyset$, then let

$$
\begin{aligned}
d_0 &= min_{\langle v, d \rangle \in D} d \\
A &= \{\langle v, d \rangle | \langle v, d \rangle \in D \ \wedge \ d = d_0\}
\end{aligned}
$$

then

$$\begin{aligned}
M' &= M \\
N' &= N \\
R' &= \{v | \langle v, d\rangle \in A \\
&\quad \wedge (v \in W \;\wedge\; v \in N' \vee v \notin W)\} \\
D' &= \{\langle v, d\rangle | \langle v, d + d_0\rangle \in D - A\} \\
C' &= C
\end{aligned}$$

- **Type III Transition** represents the **end of execution**: if $R = \emptyset$ and $D = \emptyset$, then $s = \perp$, denoting no next state.

The execution of graph G is a sequence of states $[s_0, s_1, \ldots, \perp]$, where

- $s_0 = \langle \emptyset, \{v_0\}, \emptyset, M_0, C_0\rangle$, where M_0 represents the initial memory store that maps every variable to its initial value, and C_0 represents the initial context which sets all the untimed vertices to its first statement.

- $s_{i+1} = \mathit{Next}(s_i, G)$

In summary, we present the semantics of the SpecC+ language in a formal graph-based notation. The semantics is rich in the sense that it covers the semantics of many other languages. For example, if we constrain all the triggering edges to a delay of zero (disallow *waitfor* statements), then the triggering edges induce a partial ordering on the vertices, and there is no notion of the passage of time. Such a system is equivalent to the systems captured by concurrent languages. If we further constrain that no *fork* and *join* vertices are allowed, such a system is equivalent to those captured by sequential imperative languages such as C. On the other hand, if we exclude the untimed subsystem, the language is semantically equivalent to VHDL.

4 Meeting the objectives

In this section, we briefly review how the SpecC+ language addresses the design objectives discussed in Section 1.

Objective 1 requires SpecC+ to be capable of modeling designs at different abstraction levels, or mixed levels of abstractions. In the codesign domain, a **computation** at a high abstraction level may be a behavior with only partial ordering of operations specified, but exact timing missing. On the other hand, a computation with lower abstraction level may be a behavior with the exact information on when each operation is performed. Similarly, a **communication** at a high abstraction level may be a shared variable accessable by concurrent processes. On the other hand, at a lower abstraction level, these shared variables may be distributed over different processing elements, while accesses to them may involve consistency protocols and complex transactions over system buses.

For computation, SpecC+ allows the simultaneous specification of an untimed system, which is primarily used to model unsynthesized behavior whose timing is not resolved yet, and a timed system, which is primarily used to model synthesized behavior whose timing is known. This flexibility makes it possible to describe a variety of system modeling configurations. For example, Figure 10 shows a typical microprocessor system with two IO devices. Software behavior, which is not yet compiled and bound to any processor, can be modeled as untimed behavior, such as actor *AProgram* in Figure 10. So does a hardware behavior that is to be synthesized, such as actor *ADevice2*. Compiled software behavior can be modeled as timed behavior, such as channels *CDriver1* and *CDriver2* representing the device drivers for actors *ADevice1* and *ADevice2*. A processor model *CProcModel* is modeled as a channel which exports its instruction set in the form of methods. Each instruction is modeled as timed behavior, which operates on the processor bus signals. Hardware components from the library, that is, synthesized hardware behaviors, can be modeled as timed behavior, for example, actor *ADevice2* in Figure 10.

For communication, SpecC+ allows the simultaneous specification of communication using primitive channels, which is used to model abstract communication via implicit read and write operations over typed variables, and complex channels, which is primarily used to model communication at the implementation level, for example, the transfer of a block of data over a standard bus.

The execution semantics of SpecC+ laid the basis to realize Objective 2. The benefit of being able to execute models at mixed abstraction levels is two fold: first, intermediate design models can be validated before the next synthesis step; second, designs can be validated with appropriate speed-accuracy trade-off. For example, in Figure 10, actor *AProgram* is modeled as untimed behavior, while channels *CDriver1* and *CDriver2* are modeled as timed behavior. The rationale behind this configuration is that behaviors in *CDriver1* and *CDriver2* contain IO instructions which interact intensively with the hardware such as *ADevice* and *ATransducer*, and it is this type of behavior that is error-prone and should receive our attention. On the other hand, the behavior contained in actor

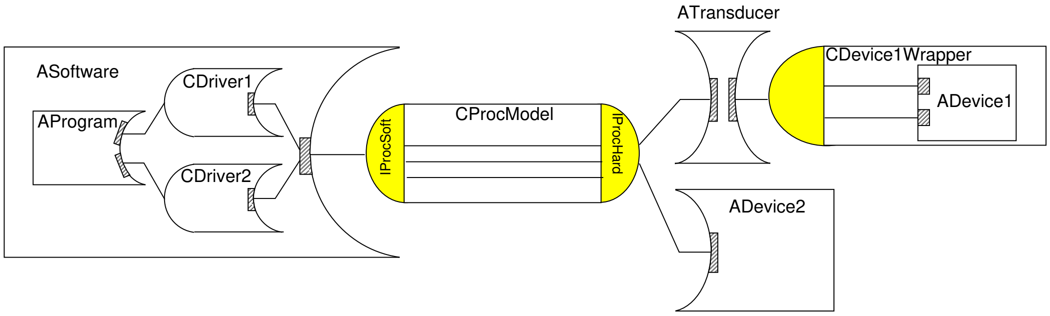


Figure 10: Model of a microprocessor system with IO devices.

AProgram contains just normal operations of the processor and need not to be verified at such a detailed level.

The ability of SpecC+ to model designs at different abstraction levels makes a large category of design artifacts eligible to be modeled and entered in a reuse library. For example, a hardware component can be modeled as an actor, such as *ADevice1* in Figure 10, and can be stored in a library. The protocol on how to communicate with a hardware component, which before is documented by a data sheet, can now be modeled as a wrapper, such as *ADevice1Wrapper* in Figure 10, and stored in a library. The bus protocols, including standard system buses such as *PCI bus, PIBus*, or *VMEBus*, and processor bus models, such as *Pentium* bus and *PowerPC* bus, can also be stored in the library in the form of channels.

The abstraction of communication into a set of functions in the *channel* construct and the abstraction of the channel implementation into a set of function prototypes in the *interface* construct makes it possible to decouple the computational aspect of an actor from the communication. This feature is helpful for realizing both Objective 3 and Objective 4. For reuse, the actors described in this way can be used without modification in different situations. For synthesis and refinement, abstract channels can be replaced by detailed channels without affecting the connected actors.

5 Conclusion

In conclusion, we proposed SpecC+ as a modeling language for codesign, which supports a homogeneous codesign methodology.

SpecC+ can be considered as an improvement over traditional HDLs such as VHDL.

Semantically, SpecC+ allows the specification of behavior with exact timing as well as unknown timing, whereas VHDL only allows specification of behavior with exact timing, which often leads to overspecification.

Conceptually, SpecC+ raises the abstraction level, while reorganizing important concepts. For example, concurrency is decoupled from structure, synchronization and timing are decoupled from interconnections (for example, in VHDL signals are used both for synchronization, interconnection, and even timing), function interfaces are decoupled from function implementations, ports are generalized into interfaces, wires are generalized into channels.

Syntactically, SpecC+ is based on C, which allows the inheritance of a large archive of existing code, and makes it easy for an implementation of the language to leverage traditional C compilers.

Philosophically, SpecC+ is intended to be a codesign modeling language (CML) with single semantics, while VHDL is a hardware description language (HDL) with different semantics in simulation and synthesis.

6 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.

[Fr95] R. French, M. Lam, J. Levitt, K. Olukotun "A General Method for Compiling Event-Driven Simulation"; *Proceedings of 32th Design Automation Conference*, 6, 1995.

[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.

[OMG95] *Common Object Request Broker: Architecture and Specification*; http:// www.omg.org/corbask.htm.

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