Embedded Systems Conference San Francisco 2002

The SpecC System-Level Design Language and Methodology, Part 1

Class 309

Rainer Dömer

Center for Embedded Computer Systems
Universitiy of California, Irvine, USA

Abstract

A well-defined design methodology supported by a system-level design language (SLDL) is the key for managing the complexity of the design flow, especially at the system level. Only with well-defined and unambiguous models and transformations can we achieve productivity gains through synthesis, verification and tool interoperability. This paper presents the SpecC system design language. After a general overview of SLDL requirements, it describes the SpecC language as an example of a language specifically developed to support a formalized design flow.

This is the first paper in a two-part series. This part introduces the SpecC model and the SpecC language.

1 Introduction

Systems-on-Chip (SOCs) are faced with a huge increase of design complexity. A well-known solution in computer science for dealing with complexity is to exploit hierarchy. This effectively reduces the complexity in terms of the number of objects to be handled at one time.

Figure 1 illustrates this principle for digital systems. An embedded system, which at the lowest level consists of 10ths of millions of transistors, typically reduces to only thousands of components at the register-transfer level (RTL). Furthermore, RTL components are grouped together at the algorithm level. Finally, at the highest level, called *system level*, a single system is composed of only few components which include microprocessors, custom hardware units, memories, a variety of IPs, and busses.

Please note that the level of abstraction is a trade-off with the level of accuracy. The higher the abstraction level, the lower the accuracy, and vice versa.

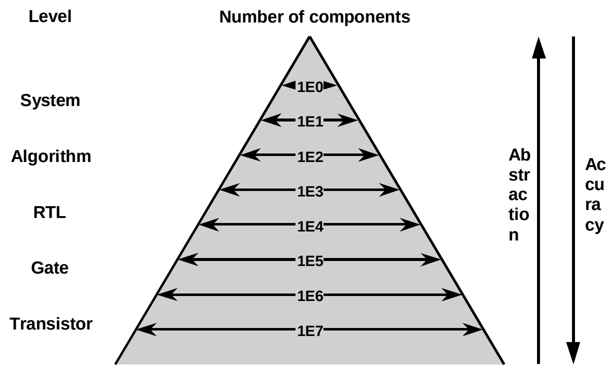


Figure 1: Abstraction Levels.

From Figure 1, it is obvious that the design of a complex embedded system is easier at the system level than at the detailed gate or transistor level. Thus, one solution to deal with the increased complexity is to move to a higher level of abstraction, called the system level.

From the system level, the SOC design methodology then works its way through several refinement steps down to the implementation. The design process of a new system usually starts from an abstract specification model and ends with a highly accurate implementation model that reflects the real system with all its details.

The advantage of such a top-down approach is that all necessary design decisions can be made at an abstraction level where all irrelevant details are left out of the model. This allows designers to work with a model of minimum complexity.

2 The SpecC Model

The following section takes a closer look at the SpecC model, including its components and its features.

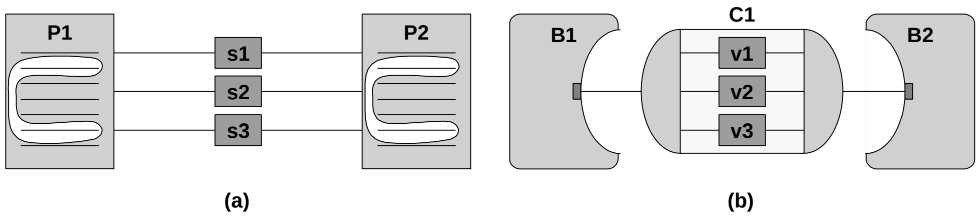


Figure 2: (a) Traditional Model, (b) SpecC Model.

2.1 Traditional Model

For the design of embedded systems, the key representation for any design is a block diagram. Block diagrams consist of a set of blocks and a set of interconnections between the blocks.

The blocks in the diagram represent components that perform a particular function or computation. Also, the blocks can communicate with each other through the interconnections. Here, it is important to note that there are two distinct actions performed by the blocks, namely *computation* and *communication*.

The traditional model of a block diagram, such as in VHDL or Verilog, is shown in Figure 2(a). Two processes, `P1` and `P2`, are communicating via variables `s1`, `s2` and `s3`. By assigning values to the signals, according to some predefined protocol, the processes can communicate and exchange data.

In this scenario, the processes `P1` and `P2` contain code for both communication and computation. The communication part is highlighted in the figure. Because communication and computation are freely intermixed in the code, they cannot be identified by any tool. As a result, it is not possible to automatically change the communication protocol when design constraints change. On the other hand, it is also impossible to automatically switch to a new algorithm to perform the computation.

2.2 SpecC Model

The SpecC model does not have this problem. In the SpecC model, communication and computation are clearly separated. Computation is encapsulated in *behaviors*, and communication is encapsulated in *channels*, as shown in Figure 1(b). Here, the computation is encapsulated in the behaviors `B1` and `B2`, and the communication is contained in the channel `C1`.

More specifically, the channel `C1` encapsulates the communication in form of functions, such as `send` and `receive`. These functions define the interfaces of the channel. The channel also encapsulates the communication media, i.e. the variables `v1`, `v2` and `v3`. On the other hand, the behaviors only contain computation. In order to communicate, the behaviors call the functions provided by the connected channel.

As a result of the separation of communication and computation, the SpecC model supports "plug-and-play". The communication protocol can be easily exchanged by use of another channel with compatible interfaces, whenever this is desirable in the design process. In the same manner, the behaviors can also be exchanged with others, without affecting the communication protocol.

2.3 Protocol Inlining

In SpecC, behaviors and channels are used throughout the design process, i.e. in the specification model and during design exploration and refinement. This way, the "plug-and-play" capability of the SpecC model can be exploited at any stage in the design flow, whenever it is useful.

However, in the final implementation model, channels are reduced to variables representing wires by a process called *protocol inlining*.

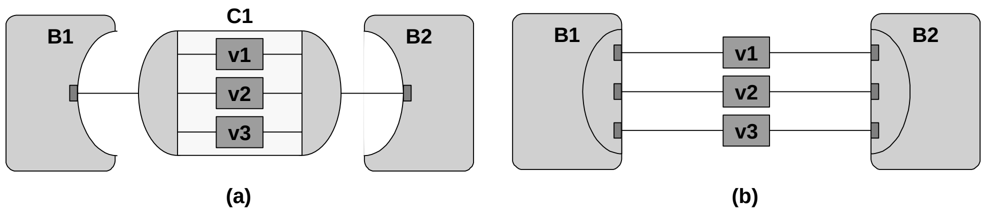


Figure 3: Protocol Inlining, (a) before, (b) after.

For the implementation of a channel, its functions are *inlined* into the connected behaviors and the encapsulated communication media are exposed. This is illustrated in Figure 3.

After the inlining process, the channel `C1` has disappeared. The encapsulated variables `v1`, `v2` and `v3` are exposed and the communication protocol is integrated into the behaviors `B1` and `B2`.

Please note that in this final implementation model communication and computation are no longer separated. Moreover, the resulting model is essentially the same as the traditional model discussed earlier. This allows a straightforward translation of the implementation model into the traditional VHDL or Verilog models. As a result, the SpecC design flow can be easily integrated with a conventional design process.

2.4 Plug-and-Play with Computation

Now that the basic "plug-and-play" feature of the SpecC model has been introduced, the reuse of intellectual property (IP) components will be addressed. First, the reuse of computation IP is shown by use of "plug-and-play" with the so-called *adapter* model.

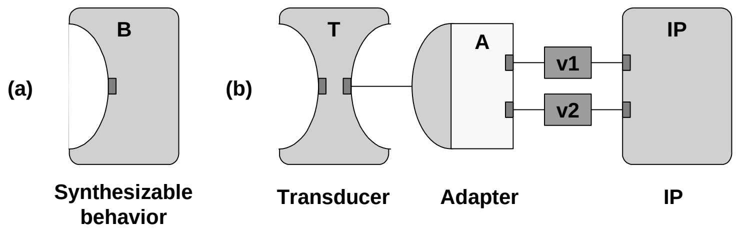


Figure 4: Computation IP, Adapter Model.

Figure 4(a) shows a synthesizable behavior `B`. Note that the behavior is still missing the code for a communication protocol, which will be inlined later from a connected channel. On the other hand, Figure 4(b) shows on the right a fixed IP core with a built-in communication protocol.

In order to allow "plug-and-play" with the IP core, it is connected to an adapter channel. An adapter is a channel that has ports just like a behavior. These ports are connected to the variables `v1` and `v2`, which in turn are mapped to the ports of the IP. Communication with the IP is established by use of a set of high-level communication functions provided by the adapter. These functions contain the detailed interface protocol to drive the wires connected to the IP. Thus, by using the adapter functions, other behaviors can easily communicate with the IP.

Further, a *transducer* `T` is needed to make the components on the right side equivalent to the synthesizable behavior `B` on the left. A transducer is a synthesizable behavior used to connect two channels. Later in the design process, the transducer `T` will incorporate two communication protocols.

The reason for the need of a transducer stems from the fact that two channels cannot be directly connected because they are passive components. In order to connect passive channels, an active transducer behavior is needed in the middle.

As a result, the synthesizable behavior `B` on the left can be replaced by the IP adapter model on the right at any time in the design process without affecting any other objects.

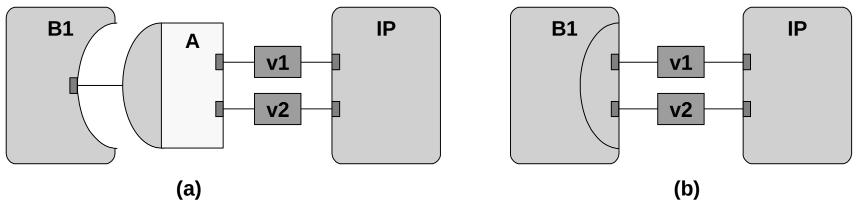


Figure 5: Protocol Inlining with Adapter, (a) before, (b) after.

The process of inlining with an adapter channel is shown in Figure 5. Before the protocol inlining, the IP adapter is connected to a synthesizable behavior `B1`, as shown on the left hand side.

On the right hand side, the same example is shown after protocol inlining. After the adapter has been inlined, the IP communication protocol has been integrated into the behavior `B1` and the variables `v1` and `v2` directly form the connecting wires to the IP. Note that the adapter `A` has completely disappeared.

2.5 Plug-and-Play with Communication

Similar to IP cores for computation, "plug-and-play" also works for communication IP in the SpecC model.

A proprietary implementation of a communication protocol is represented by an IP channel in SpecC, such as the one shown in Figure 6(b). With one exception, this channel is not different in its usage compared to the virtual channel on the left.

The only exception is that the IP channel typically needs to be wrapped by another channel for data type conversion. For example, the IP channel might provide native functions to send and receive blocks of 512 bytes of data. However, in order to use this channel in an application that needs to transfer pictures of 1024 by 768 pixels, a type conversion is required from the picture type into the transferable block type, and vice versa. This conversion can be easily performed by the wrapper channel `C2.`

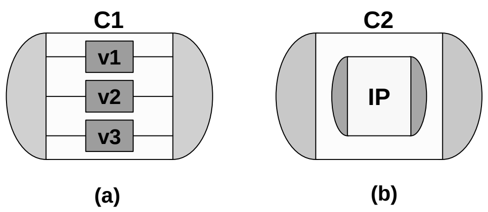


Figure 6: Communication IP, Channel with Wrapper.

Assuming that in an initial system specification the virtual channel `C1` is used to transfer the picture data, the channel `C2` can be used as an equivalent replacement at any time. Again, this change is only local and does not affect any other channels or behaviors in the system.

The process of protocol inlining for hierarchical channels, such as the IP channel with its wrapper, is essentially the same as for any other channel. The inlining process is just repeated several times, once for each level in the hierarchy.

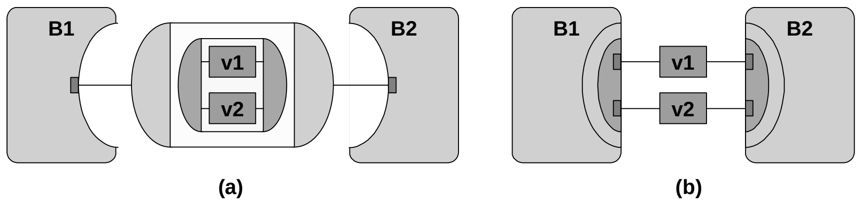


Figure 7: Protocol Inlining with Hierarchical Channel.

As shown in Figure 7, each protocol layer from the hierarchical channel on the left is incorporated into the behaviors `B1` and `B2`, and the communication media `v1` and `v2` from the innermost channel are exposed, forming the wires between the behaviors.

Note that the hierarchy of the protocol layers does not change during this process. The protocol hierarchy is reflected in the function calls inside the behaviors.

3 System-level Language Requirements

Before the concepts, features and syntactical details of the SpecC language are explained, the following sections briefly review the goals and requirements of system-level languages and motivate the design of the SpecC language.

3.1 Language Goals

Several years of research have been spent on defining the objectives and requirements for system-level languages. The following goals have been identified.

Executability is of crucial importance for simulation. Simulation enables validation of the system specification as well as verification of intermediate design models throughout the design process.

Synthesizability is important in order to obtain an implementation by use of tools. Ideally, every construct in the language should have at least one straightforward implementation, either in software, or in hardware. In other words, the complete language should be synthesizable, instead of only a subset. Otherwise the subset essentially becomes a new language. In addition, support for reuse of IP is also required.

Modularity is required in form of structural and behavioral hierarchy, allowing the hierarchical decomposition of a system. Moreover, modularity is needed to separate computation from communication.

Completeness implies that all concepts commonly found in embedded systems design need to be supported. These concepts include concurrency, hierarchy, synchronization, exception handling, timing, and explicit state transitions.

Orthogonality is desirable such that the orthogonal concepts found in embedded systems can be described by use of orthogonal, independent constructs in the language. Signals in VHDL can serve as a counter example, since they include the concepts of synchronization, data storage and timing at the same time.

Orthogonality also implies minimality of the constructs, which, together with *simplicity* makes the language easy to learn and easy to understand.

	C	C++	Java	VHDL	Verilog	HardwareC	Statecharts	SpecCharts	SpecC
Behavioral hierarchy	○	○	○	○	○	○	●	●	●
Structural hierarchy	○	○	○	●	●	●	○	○	●
Concurrency	○	○	◐	●	●	●	●	●	●
Synchronization	○	○	◐	●	●	●	●	●	●
Exception handling	◐	●	●	○	●	○	◐	●	●
Timing	○	○	○	●	●	◐	◐	◐	●
State transitions	○	○	○	○	○	○	●	●	●
Composite data types	●	●	●	●	◐	○	○	●	●

○ not supported ◐ partially supported ● supported

Figure 8: System-level Language Requirements and Coverage.

3.2 Language Requirements

Figure 8 compares some traditional languages against a set of language requirements. As mentioned before, the essential requirements are behavioral and structural hierarchy, concurrency, synchronization, exception handling, timing, and explicit state transitions. For software, composite data types are needed as well. In general, it can be expected that software languages are not suitable for describing hardware, and vice versa.

For each language, the table shows which requirements the language supports and which are missing or are only partially supported. Of course, this classification is only a rough characterization of each language. However, it indicates quite well which problems a language incorporates if it is used for system-level design.

As the table shows, all the traditional languages lack one or more of the requirements. Hence, these languages cannot be used without problems for modeling embedded systems. Because of this, a new language was developed, called SpecC.

The SpecC language, shown in the last column of this table, has been specifically designed to support all the required concepts. Moreover, SpecC precisely covers these requirements in an orthogonal manner.

4 The SpecC Language

The following sections cover the main part of this paper, namely the SpecC language. First, the foundation of the SpecC language is described. Then, for each of the identified concepts needed in system-level design, the supporting SpecC constructs are explained, both in syntax and in semantics. Also, for easy understanding, simple examples are used to highlight the use and benefits of each construct.

4.1 Foundation

Accepting the fact, that a new language needs to be developed, it has to be determined how the new language is being built. More specifically, the new language can either be developed from scratch, or can be built based upon an existing language. Obviously, the second approach can easily leverage knowledge that is already present in the given language, avoiding to ‘reinvent the wheel’.

Because of this, the SpecC language was based on C. C, or more precisely ANSI-C, was selected because of its maturity and its large amount of already existing code. Also, the C language is still the de-facto standard for software development. ANSI-C is well known, and tools and methodologies for C are well established and easily available.

With the selection of C, the requirements for software design are already satisfied. Furthermore, SpecC is a true super set of ANSI-C. In other words, every C program is also a SpecC program. In addition to the ANSI-C constructs, the SpecC language includes extensions for hardware design. These extensions are added as a minimal, orthogonal set of constructs, supporting orthogonal concepts.

Any ANSI-C program is composed of a set of functions. Each function describes the behavior of the program by use of assignments, control constructs, such as `if`, `while`, and `for`, and function calls. The execution of an ANSI-C program starts with the execution of the `main` function, which then calls the other functions in the program. As a very simple example, the famous `HelloWorld` program in ANSI-C is shown in Figure 9(a).

As already mentioned, the SpecC language is a true superset of ANSI-C. Therefore, the `HelloWorld` program in ANSI-C is already a valid SpecC description. However, the real `HelloWorld` program in SpecC looks a little different, as shown in Figure 9(b).

A SpecC program is a collection of classes. There are three types of classes, namely behaviors, channels, and interfaces. These directly reflect the structure of the SpecC model, as discussed before.

Syntactically, a SpecC behavior is specified by use of a `behavior` definition, such as the behavior `Main` in the example. In general, a `behavior` definition consists of a set of ports, a set of local variables, instantiations and methods, and a mandatory `main` method.

A SpecC program starts with the execution of the `main` method of the root behavior, which is identified by its name `Main`. Please note that `main` and `Main` are names that are recognized by automated tools, but these names are not keywords.

```
/* HelloWorld.c */

#include <stdio.h>

void main(void)
{
  printf(“Hello World!\n”);
}
```

```
// HelloWorld.sc

#include <stdio.h>

behavior Main
{
  void main(void)
  {
    printf(“Hello World!\n”);
  }
};
```

(a) **(b)**

Figure 9: "HelloWorld" Program, (a) ANSI-C, (b) SpecC.

In the SpecC version of the `HelloWorld` example, the `main` method is identical to the `main` function of the ANSI-C version. The only difference is that it is encapsulated in the `Main` behavior. In general, it is always the `main` method that is executed when an instantiated behavior is called. Also, the completion of the `main` method determines the termination of the execution of the behavior.

4.2 Types

The types and expressions supported by SpecC are mostly inherited from the C language. SpecC supports all the standard types, such as `int`, `float`, and `double`. It also supports composite types, such as pointers and arrays, as well as user-defined types, such as `struct`, `union` and `enum`. In addition, SpecC explicitly supports a boolean data type `bool` for the representation of truth values. As in C++, a boolean value has one of two values, `true` or `false`.

In order to model hardware, explicit support for bit vectors is required. SpecC provides a built-in bit vector type with arbitrary precision. Bit vectors are either `signed` or `unsigned`.

A bit vector can be thought of as a parameterized type whose bounds are defined with the name of the type. SpecC semantics require that the left and right bounds of any bit vector are constant expressions that can be evaluated statically. Hence, the length of any bit vector is known at compile time, which is a requirement for synthesis.

A bit vector can be used as any other integral type within expressions with the standard operators. Implicit conversion, i.e. extension or truncation, or promotion to integral types, such as `int`, `long`, or `double`, is automatically performed when necessary. In addition, a concatenation operation, denoted by `@`, and a bit slice operation, denoted by `[l:r]`, are supported in SpecC. Also, a bit access operation, denoted the same way as an array access `[b]`, is provided as a short-hand for accessing a single bit (`[b:b]`) in a bit vector. Bit vector constants are denoted as a sequence of zeros and ones, immediately followed by a suffix `b` or `ub` indicating `signed` or `unsigned` type.

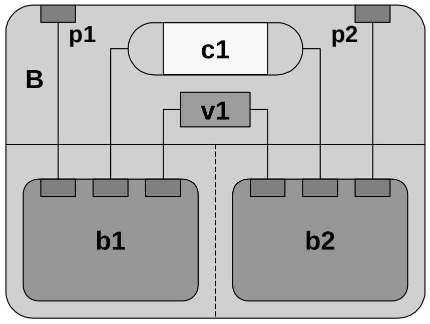
4.3 Structural Hierarchy

In SpecC, structural hierarchy is supported in the style of standard block diagrams. More specifically, structure is represented as a hierarchical network of behaviors and channels.

Figure 10 shows a behavior `B` with two ports, `p1` and `p2`, through which it can communicate with its environment. Internally, these ports are connected to two child behaviors, `b1` and `b2`. These child behaviors can communicate in two ways. First, both are connected to a shared variable `v1`, which is written by `b1` and then read by `b2`.

Second, `b1` and `b2` can communicate through the channel `c1`. For example, the behavior `b1` calls a function `Write` provided by the left interface of channel `c1`. Similar, behavior `b2` calls a `Read` function provided by the right interface.

Please note that Figure 10 only shows one level of the structural hierarchy. The child behaviors `b1` and `b2` can again consist of a network of behaviors and channels.



```
interface I1
{
  bit[63:0] Read(void);
  void Write(bit[63:0]);
};

channel C1 implements I1;

behavior B1(in int, I1, out int);

behavior B(in int p1, out int p2)
{
  int v1;
  C1  c1;
  B1  b1(p1, c1, v1),
      b2(v1, c1, p2);

  void main(void)
  { par { b1.main();
          b2.main();
         }
   }
};
```

Figure 10: Structural SpecC Example.

The diagram shown on the left can be described syntactically by a SpecC program as shown on the right. The program first defines the interface `I1`, which declares the communication methods `Read` and `Write` for the channel `C1`. Then, the channel `C1` is defined, which `implements` the interface `I1`. Even though the body of the channel is not defined, the `implements` keyword indicates that the methods `Read` and `Write` are available through this channel. Next, a behavior `B1` is declared. It has three ports, one input port, one port of type interface `I1`, and one output port.

Finally, the behavior B is specified with the two ports p1 and p2. Internally, B consists of the variable v1, the channel c1, and the two child behaviors b1 and b2. Note how the behaviors b1 and b2 are connected to the ports p1 and p2, the variable v1, and the channel c1.

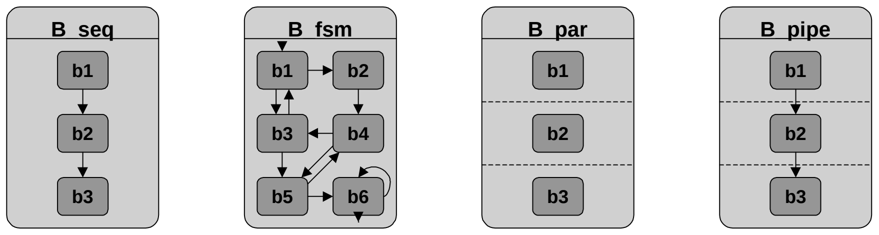
The child behaviors are executing concurrently, specified by the par statement in the main method of behavior B.

4.4 Behavioral Hierarchy

Behavioral hierarchy is the composition of child behaviors in time. In SpecC, child behaviors can either be executed sequentially or concurrently. Sequential execution, as shown in Figure 11(a) and (b), can be specified by standard sequential statements, or as a finite state machine (FSM) model with explicit state transitions. On the other hand, concurrent execution is either parallel (Figure 11(c)) or pipelined (Figure 11(d)).

With sequential execution, the child behaviors are running sequentially, one at a time, in the order indicated by the arrows. Syntactically, this is specified by calling the main methods of the instantiated behaviors in the desired order. The execution of the behavior B_seq starts with the execution of b1 and terminates when b3 has finished its execution.

The finite state machine execution, shown in Figure 11(b), is specified by a set of state transitions. Syntactically, the fsm construct is used for this, which will be explained later in detail.



```
behavior B_seq
{
 B b1, b2, b3;

 void main(void)
 { b1.main();
   b2.main();
   b3.main();
  }
};
```

(a)

```
behavior B_fsm
{
 B b1, b2, b3,
   b4, b5, b6;
 void main(void)
 { fsm { b1:{…}
         b2:{…}
         …}
  }
};
```

(b)

```
behavior B_par
{
 B b1, b2, b3;

 void main(void)
 { par{b1.main();
       b2.main();
       b3.main();
  }   }
};
```

(c)

```
behavior B_pipe
{
 B b1, b2, b3;

 void main(void)
 {pipe{b1.main();
       b2.main();
       b3.main();
  }   }
};
```

(d)

Figure 11: Behavioral Hierarchy, (a) Sequential, (b) FSM, (c) Concurrent, (d) Pipelined Execution.

Concurrent execution is shown in Figure 11(c), where b1, b2 and b3 run in parallel. They all start simultaneously when B_par starts. Once all of them have completed their execution, B_par will also finish. Syntactically, parallel execution is specified by use of the par construct.

Very similar to the par construct, the pipe construct allows execution in pipelined fashion. Again, this is explained later in more detail.

4.5 Finite State Machine Execution

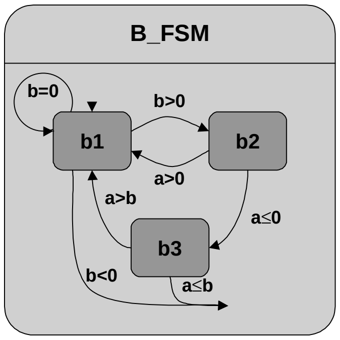
The SpecC language provides the fsm statement to specify finite state machines (FSMs) with explicit state transitions. Both Mealy and Moore type FSMs can be modeled with the fsm construct.

A state transition is defined as a triple of current state, an optional condition, and a next state. Syntactically, the names of child behaviors are used in form of labels to denote the current and the next state. The condition is a boolean expression which determines whether the transition is valid.

```
behavior B_FSM(in int a, in int b)
{
 B b1, b2, b3;

 void main(void)
 { fsm { b1:{ if (b<0) break;
              if (b==0) goto b1;
              if (b>0) goto b2; }
         b2:{ if (a>0) goto b1; }
         b3:{ if (a>b) goto b1; }
        }
 }
};
```

(a)



(b)

Figure 12: Finite State Machine Example.

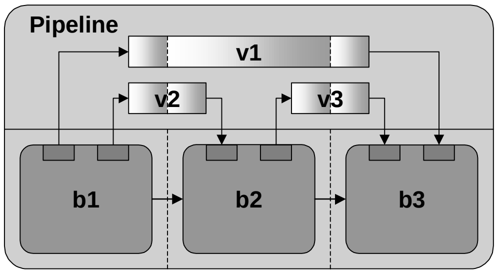
As shown in Figure 12, the execution of a `fsm` construct starts with the execution of the behavior that is listed first in the transition list, i.e. `b1`. Once this behavior has finished, its state transitions determine the next behavior to be executed. The conditions of the transitions are evaluated in the order they are specified, and, as soon as one condition is `true`, the behavior specified after the `goto` statement is started. As a special case, the `break` keyword indicates the completion of the FSM execution.

Please note that the `fsm` construct does not allow arbitrary statements. The SpecC syntax limits the state transitions to well-defined triples. This ensures that the `fsm` construct can be easily analyzed and synthesized by automated tools.

4.6 Pipeline Execution

The SpecC language provides explicit support for the specification of pipelines. Pipelined execution is a special form of concurrent execution. Syntactically, it is specified with the `pipe` construct. Each statement in the statement block after the `pipe` keyword forms a new thread of control. The set of control threads is then executed in a pipelined fashion.

In the example shown in Figure 13, the child behaviors `b1`, `b2` and `b3` form a three-stage pipeline of behaviors. When the pipeline is started, only `b1` is executed. When `b1` completes, the second iteration starts and `b1` and `b2` are executed in parallel. Finally, in the third and every following iteration, all three child behaviors are executed in parallel.



```
behavior Pipeline
{
 piped piped int v1;
 piped int v2;
 piped int v3;

 Stage1 b1(v1, v2);
 Stage2 b2(v2, v3);
 Stage3 b3(v3, v1);

 void main(void)
 {
   int i;
   pipe(i=0; i<10; i++)
       { b1.main();
         b2.main();
         b3.main();
        }
  }
};
```

Figure 13: Pipeline Example.

However, the `pipe` statement can take the same parameters as the `for` statement. This form allows to flush the pipeline as soon as the specified condition becomes true. More specifically, the arguments specify an initialization statement, a condition, and an iteration statement. For example, with the help of a local counter variable `i`, the pipeline can be flushed after 10 iterations, as shown in the example.

More specifically, this pipeline will start just as explained before. However, after 10 iterations, the pipeline will be flushed. Only `b2` and `b3` are executed in the 11th iteration, and only `b3` is run in the 12th and last iteration. As a result, each child behavior is executed exactly 10 times.

The `pipe` statement also supports automatic communication buffering between the pipeline stages. In other words, this mechanism allows to use automatic FIFOs instead of standard variables between the pipeline stages.

The example shown in Figure 13 includes variables for communication between the pipeline stages. Variable `v1` feeds data from `b1` to `b3`, whereas `v2` and `v3` send data from `b1` to `b2`, and `b2` to `b3`, respectively.

However, this will not work without special care. Since the pipeline stages are executed concurrently, `b1` may overwrite the data in `v1` and `v2` that is still needed by `b2` and `b3`. In other words, some sort of synchronization is needed to ensure that the data is correctly shifted from one stage to the next.

The SpecC keyword `piped` solves this problem. `piped` is a special storage class to be used for variables connecting two pipeline stages. A variable with a `piped` storage class can be thought of as a variable with two storages. A write access to such a variable always writes to the first storage. A read access, on the other hand, always reads from the second storage. In addition, the contents of the first storage are shifted to the second storage whenever the pipeline starts a new iteration.

In the code, the `piped` storage class is specified for the pipeline variables `v1`, `v2` and `v3`. Note that it is specified twice for the variable `v1`. This specifies two buffers for `v1`, instead of only one, such as for `v2` and `v3`. This ensures that `v1` transfers data correctly over two pipeline stages from `b1` to `b3`.

In other words, the data produced by `b1` will be accessable by `b2` only in the second iteration, and by `b3` only in the third iteration, as expected for a real pipeline. Thus, the data is automatically buffered and transferred synchronously with the pipeline.

4.7 Communication

The clear separation of communication from computation is one of the strengths of the SpecC language. Communication can be modeled by use of variables or channels between behaviors.

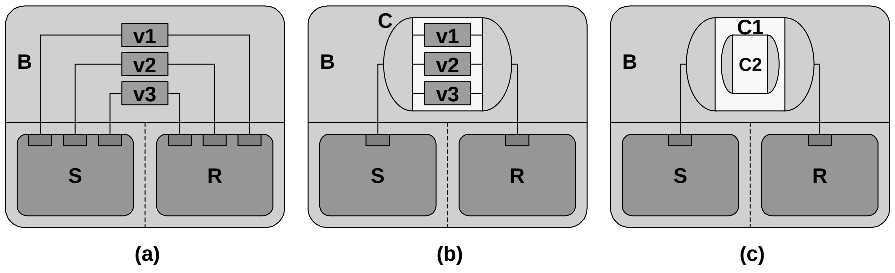


Figure 14: Communication Methods, (a) Shared Memory, (b) Message Passing, (c) Protocol Stack.

Variables are used to represent a shared memory communication model in SpecC. As shown in Figure 14(a), the communication variables `v1`, `v2` and `v3` are defined in the behavior B and are then connected to the ports of the child behaviors `S` and `R`. This way, the sender `S` can store data in the variables, which then can be read by the receiver `R`.

In other words, the variables represent wires for communication. Note that these wires hold their value over time, just as a memory. Access to these variables is established through the ports that are mapped to them.

In summary, with the shared memory model, child behaviors can communicate by assigning values to their output ports (send) and observing values at their input ports (receive).

While communication through shared variables is sufficient for simple cases, more complex communication protocols are needed in the general case. Typically, these protocols involve synchronization, timing, buffering, error correction, etc.

A message passing communication scheme can be specified by use of a virtual channel, as shown in Figure 14(b). The channel is called virtual, since it does not yet represent a real implementation, such as a PCI bus protocol. Virtual channels are typically implemented as basic, non-hierarchical channels, which are also called leaf channels.

The channel `C` shown here consists of the local variables `v1`, `v2` and `v3`, and a set of communication functions. The functions of the channel use the local variables to realize the communication.

The communication functions of the channel are made available to behaviors through the interfaces of the channel. In this figure, the sender `S` can access the left interface of `C`, whereas the receiver `R` can access the right interface of `C`.

As a third communication scheme, hierarchical channels can be used, as shown in Figure 14(c). A channel is called a *hierarchical channel* if it contains a child channel, such as `C2` in the figure.

A typical example for hierarchy in channels is a communication protocol stack. For example, a channel `C1` providing `send` and `receive` functions for large blocks of data might use an internal channel `C2` that provides functions to send and receive single bytes of data.

Of course, a communication protocol consisting of many protocol layers, such as a protocol following the seven layers of the ISO/OSI standard, can be specified by use of a hierarchy of multiple channels.

4.8 Synchronization

In order to allow cooperation and communication among concurrent executing behaviors, a synchronization mechanism is required. In SpecC, the built-in type `event` serves as the basic unit of synchronization. These events are used with the `wait`, `notify` and `notifyone` statements which all take a list of events as arguments.

A `wait` statement suspends the current behavior from execution until one of the events specified with the `wait` statement is triggered by another behavior. The execution of the waiting behavior then resumes.

The `notify` statement triggers all specified events so that all the behaviors waiting on one of these events can resume their execution. If no behavior is waiting on the triggered events at the time of the `notify` statement, the notification is ignored. The `notifyone` statement acts similar as the `notify` statement. However, `notifyone` allows only one out of all currently waiting behaviors to resume its execution.

```
behavior S(out event Req,
           out float Data,
           in  event Ack)
{
 float X;
 void main(void)
 { ...
   Data = X;
   notify Req;
   wait Ack;
   ...
 }
};
```

(a)

```
behavior R(in  event Req,
           in  float Data,
           out event Ack)
{
 float Y;
 void main(void)
 { ...
   wait Req;
   Y = Data;
   notify Ack;
   ...
 }
};
```

(b)

Figure 15: Synchronization, (a) Sender, (b) Receiver.

The example in Figure 15 shows a simple hand-shake between a sender `S` and a receiver `R`. The sender stores its information in the shared variable `Data`, notifies the event `Req`, and then waits for an acknowledge signal. The receiver first waits for a send request, then gets the data and finally sends a receipt. This way, no data is overwritten or lost.

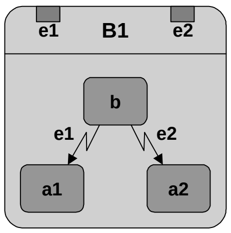
4.9 Exception Handling

The SpecC language provides explicit support for two types of exception handling, namely *abortion* and *interrupt*, as shown in Figure 16 (a) and (b), respectively.

The behavior B1 contains three child behaviors, b, a1, and a2. Normally, the execution of B1 will start the child b and will finish when b terminates. However, when the event e1 or e2 is notified during the execution of b, then an exception occurs. In case of e1, the execution of b will be aborted immediately and a1 is executed. Similar, when e2 occurs, a2 is executed. The completion of a1 or a2 will also terminate the execution of B1.

In other words, for abortion, the execution of the behavior b is aborted immediately and will not be resumed. Instead, an abortion behavior, such as a1 and a2, will take over and finish the execution.

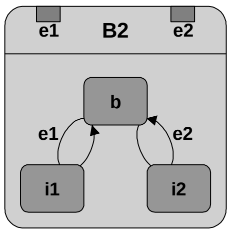
Syntactically, exceptions are implemented by use of the try construct that makes the behavior sensitive to the listed events. Then, when such an event is notified, execution is transferred to one of the exception behaviors. In this case, abortion is specified with the trap keyword, enabling the execution of a1 for e1, and a2 for e2.



```
behavior B1(in event e1, in event e2)
{
 B b, a1, a2;

 void main(void)
 { try { b.main(); }
   trap (e1) { a1.main(); }
   trap (e2) { a2.main(); }
  }
};
```

(a)



```
behavior B2(in event e1, in event e2)
{
 B b, i1, i2;

 void main(void)
 { try { b.main(); }
   interrupt (e1) { i1.main(); }
   interrupt (e2) { i2.main(); }
  }
};
```

(b)

Figure 16: Exception Handling, (a) Abortion, (b) Interrupt.

In contrast to abortion, an interrupt exception will resume the execution of the initial behavior, as shown on the right hand side. In this case, when an event e1 or e2 is notified, an interrupt occurs. Again, the behavior b is stopped immediately in its execution. The appropriate interrupt behavior, such as i1 and i2, is then executed. Once the interrupt behavior finishes, the main behavior b can resume its execution right from the point where it was stopped.

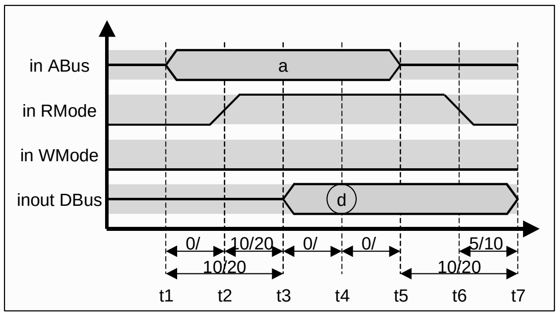
Syntactically, an interrupt is specified with the interrupt keyword, as shown in the code on the right.

4.10 Timing

As stated earlier, the notion of time is an important requirement for system languages. The SpecC language supports two types of timing specification, namely exact timing and timing constraints.

Exact timing, such as delay or execution time, is specified by use of the waitfor statement. The required time delay is given in form of an argument and must be of an integral constant type, evaluatable at compile time. When a waitfor statement is executed, the current behavior is suspended from further execution for the specified simulation time. Any concurrent running behaviors will then be executed until they are suspended as well. Once all active behaviors are suspended, the simulation time will be increased such that the behaviors with the least amount of waiting time can resume their execution.

Note that the simulation time in SpecC is only increased by use of the `waitfor` statement. All other statements execute in zero time.



```
bit[7:0] Read_SRAM(bit[15:0] a)
{
 bit[7:0] d;

 do { t1: {ABus = a;  waitfor( 2);}
      t2: {RMode = 1;
           WMode = 0; waitfor(12);}
      t3: {           waitfor( 5);}
      t4: {d = Dbus;  waitfor( 5);}
      t5: {ABus = 0;  waitfor( 2);}
      t6: {RMode = 0;
           WMode = 0; waitfor(10);}
      t7: { }
     }
  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);
          }
  return(d);
}
```

Figure 17: Timing Diagram for SRAM Read Protocol.

As another form of timing specification, timing constraints are supported in SpecC as well. A set of timing constraint is specified with the `do-timing` construct. The `do` block specifies a set of labeled action statements, whereas the `timing` block contains the actual constraints.

Timing ranges are most useful for the specification of *timing diagrams*, such as the read protocol of a static RAM shown in Figure 17. In order to read data from the SRAM, the address of the requested data is supplied with the address bus `ABus`. Then, the read operation is selected by setting `RMode` to high and `WMode` to low. After the specified time period, the requested value can finally be read from the data bus `DBus`. The timing constraints applying to the protocol are annotated in the diagram.

In SpecC, it is straightforward to capture such a timing diagram, as shown on the right. The actions are specified as labeled assignment statements. The constraints are specified with the `range` construct that specifies the left and right label, and a minimum and maximum time. The minimum and maximum time can be left unspecified, indicating the values negative and positive infinity, respectively.

The execution semantics of a `do-timing` construct are basically the same as for any sequence of compound statements. The labeled statements are simply executed in the specified order.

For an implementation, that actually obeys the specified timing constraints, `waitfor` statements have to be used to specify the time spent for the protocol. One possible implementation is shown in Figure 17 on the right. Here, `waitfor` statements have been inserted in the action block of the timing diagram. For example, there is a delay of two time units between the assignment of the address to the RAM and the assignment of the read/write access mode. Additional 17 time units are spent before the data is read from the data bus, and so on.

The attached timing constraints can be validated during the execution of the protocol by the simulation run-time system. The SpecC simulator internally maintains a list of time stamps when it executes a timing construct. At each action, a time stamp will be created. Then, when the execution of the action block is completed, these time stamps are compared against the specified constraints and any violation is reported to the user.

Note that there are typically many (if not infinite) possible solutions to correctly implement a `do-timing` construct. It is the task of synthesis to determine the solution that best fits other design goals and constraints, such as minimal area or less power consumption.

5 Summary and Conclusion

The SpecC language and the SpecC model clearly separate communication from computation. The model consists of a hierarchical network of behaviors and channels, and supports “plug-and-play” for easy IP reuse.

The SpecC language is built on top of the ANSI-C programming language, the de-facto standard for software development. It is a true superset, such that every C program is also a SpecC program. In addition, the SpecC language has extensions for hardware design. It supports all the concepts that have been identified as requirements for embedded systems design, such as structural and behavioral hierarchy, concurrency, explicit state transitions, communication, synchronization, exception handling, and timing.

The SpecC language is executable and synthesizable. Every construct supported by the language has at least one straightforward implementation in either software or hardware.

SpecC precisely covers the unique requirements for system-level languages. It provides a minimal, orthogonal set of constructs for orthogonal concepts. In other words, SpecC maps embedded system concepts onto independent language constructs in a one to one fashion.

The SpecC language has an impact on the real world. It has gained acceptance in the industry as well as in academia. Moreover, the SpecC Open Technology Consortium (STOC) has been founded to promote and secure the SpecC methodology and language.

The future of SpecC will be determined mainly by two factors, the SpecC consortium and the user base. STOC already has a standardization effort in progress with the goal to create a world-wide standard based on the SpecC language.

However, the real success of SpecC will be determined by its user base. Only with active participation of its users, the SpecC language will have a real impact and can help to solve the huge challenges of SOC design.

References

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

[2] D. Harel. *StateCharts: a visual formalism for complex systems*. Science of Programming, 8, 1987.

[3] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[4] A. Gerstlauer, R. Dömer, J. Peng, D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[5] R. Dömer. *System-level Modeling and Design with the SpecC Language*. Dissertation, University of Dortmund, Germany, 2000.

[6] http://www.cecs.uci.edu/~specc/
[7] http://www.specc.org/