Comparison of SpecC and SystemC Languages for System Design

Lukai Cai, Shireesh Verma and Daniel D. Gajski

Technical Report CECS-03-11
May 15, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{lcai, shireesh, gajski}@cecs.uci.edu

Comparison of SpecC and SystemC Languages for System Design

Lukai Cai, Shireesh Verma and Daniel D. Gajski

Technical Report CECS-03-11
May 15, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{lcai, shireesh, gajski}@cecs.uci.edu

Abstract

In course of system-level design, designers need an efficient system level design language (SLDL), which can serve as the design vehicle. The complexity of the design process at this level is determined upto an extent, by the semantics and syntax definition of the SLDL being used. This report first analyzes the system-level design flow in order to establish the requirements on an SLDL. It then compares SpecC and SystemC, the two popular SLDLs, in terms of the extent they meet these requirements. Finally, it provides the essential modeling guidelines for both the SpecC and SystemC users for the cases where the constructs or the features of the respective languages give rise to an ambiguous design.

Contents

List of Figures

List of Tables

Comparison of SpecC and SystemC Languages for System Design

Lukai Cai, Shireesh Verma and Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

In course of system-level design, designers need an efficient system level design language (SLDL), which can serve as the design vehicle. The complexity of the design process at this level is determined upto an extent, by the semantics and syntax definition of the SLDL being used. This report first analyzes the system-level design flow in order to establish the requirements on an SLDL. It then compares SpecC and SystemC, the two popular SLDLs, in terms of the extent they meet these requirements. Finally, it provides the essential modeling guidelines for both the SpecC and SystemC users for the cases where the constructs or the features of the respective languages give rise to an ambiguous design.

1. Introduction

According to Moore's law, the number of transistors on a chip will keep growing exponentially, propelling the technology towards the System-On-Chip (SoC) era. In order to bridge the gap between growing complexity of chip designs and increased time-to-market pressures, it is unanimous urge of the design community that the design process be shifted to higher levels of abstraction and the reuse of pre-designed, complex system components known as intellectual property (IP) be encouraged.

So far several system level design approaches have been proposed to meet the above criterion, which can be broadly categorized into following three groups:

System-level synthesis This design flow starts from the system behavior, the system architecture is then generated from the behavior, and finally the RTL (register transfer level)/ISS (instruction set simulation) model is generated. The implementation details are added to the design using a step by step process.

Once the components are synthesized they can always be reused. SCE design methodology[5] is based on system level synthesis approach.

Platform-based design Platform-based[8] design approach is a medium path. The difference from the system-level synthesis approach is that, in this case the system behavior is mapped to a predefined system architecture, instead of generating the architecture from the behavior as is the case in system level synthesis approach.

Component-based design It is a bottom-up flow. In this case the existing heterogenous computation/communication components are assembled and wrappers are inserted between them in order to produce the predefined platform. TIMA lab's methodology is based on component-based design[11] approach.

All of above three approaches require a system level design language(SLDL) for modeling the design. In general, a SLDL should have the following two essential attributes:

1. It should support modeling at all the levels of abstraction, from purely functional un-timed model to the cycle-accurate RTL/ISS model.
2. It models should be simulatable, so that functionality and timing constraints of a design can be validated .

Among contemporary system level design languages, SystemC[9][7], SpecC[5][10], and System-verilog[3] are most prominent. SystemC is a C++ class library based language, while SpecC is a super-set extending ANSI-C. Both SystemC and SpecC qualify for the purpose with respect to the above two attributes. System-Verilog is a high-level abstraction extension of Verilog. Unlike SystemC and SpecC, System-Verilog specifically targets RTL-level implementation and verification.

However, satisfying the above two attributes is necessary but not sufficient condition for a SLDL to serve the purpose. The complete set of requirements can only be derived by applying the system level design flow to an example and analyzing the design process. These requirements form the basis for judging the suitability of an SLDL and also for establishing guidelines for the modeling style.

This report has following three goals:

1. Establishing the requirements for a SLDL by analysis of the system-level design flow

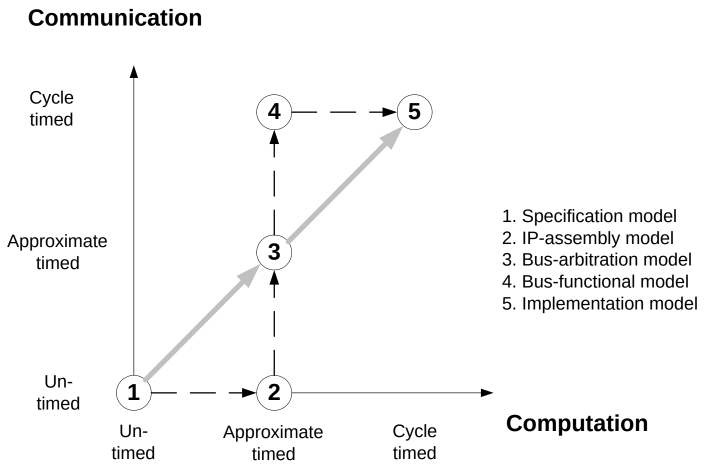


Figure 1. System modeling graph

2. Comparison of SpecC and SystemC in terms of their fulfilment of the requirements derived above

3. Provision of modeling guidelines for both SpecC and SystemC for the cases where the constructs or the features of the respective languages give rise to an ambiguous design

The report is organized as follows. Section 2 presents a general system-level design flow adopted for the purpose of this work and establishes the primary requirements which need to be satisfied by a SLDL. In Sections 3 through 7 we apply consecutive steps (from specification capture to implementation refinement) of the chosen design flow on the design example using both SpecC and SystemC. In Section 8 we perform an overall comparison between SpecC and SystemC on the basis of results obtained in Sections 3 through 7. Finally, we conclude in Section 9.

2. System design methodology

2.1 Abstraction models

The objective of design at the system level is to generate the system implementation from design functionality. In order to reduce the complexity of system design, designers generally define a number of intermediate models. The intermediate models slice the entire design into several small design tasks, each of which has a specific design objective. Since the models can be simulated and estimated, the result of each of these design tasks can be independently validated.

We introduce the system modeling graph shown in Figure 1, which represents the system design domain. X-axis in the graph represents computation and y-axis represents communication. On each axis, three degrees of accuracy are designated: un-timed, approximate timed, and cycle timed. Un-timed computation/communication represents the pure functionality of the design without any implementation details. Approximate-timed computation/communication contains system-level implementation details, such as the selected system architecture, the mapping relations between processes of the system behavior and the processing elements of the system architecture. The execution time for approximate-timed computation/communication is estimated at the system level without cycle accurate RTL/ISS level evaluation. Cycle timed computation/communication contains implementation details at both system level and the RTL/ISS level, such that cycle accurate estimation can be obtained.

Inspired by [5] [7], we define five abstraction models in the system modeling graph. The broken arrows denote a proposed design flow based on the refinement of these models. The five models are:

1. **Specification model.** It represents system behavior/-functionality. It is free of any implementation details. It is an un-timed model in terms of both computation and communication. *Specification model* corresponds to *specification model* in [5] and *un-timed functional model* in [7].

2. **IP-assembly model.** It defines the component structure of the system architecture. The system functionality is partitioned and partitions are assigned to different components. The execution delays of the processes assigned to components are annotated into the model by means of *wait* statements. So, the model is approximate-timed in terms of computation. The communication is modeled at an abstract level and components communicate via message-passing channels.

Hence, this model is un-timed in terms of communication. *IP-assembly model* corresponds to *architecture model* in [5] and is subsumed into *timed functional model* in [7]. [1]

3. **Bus-arbitration model.** In this report, we define bus-arbitration model as the one which models communication in terms of the function calls of channels representing buses. The protocols of communication are selected from the categories, i.e. blocking and non-blocking. Hence, the channels hide unnecessary implementation details including pin-accurate interface and timing-accurate transaction. Each processing element is assigned a bus priority. An arbiter assigns bus grant to processing elements on the basis of their priorities. The time required for each data transaction is inserted into the channel by means of *wait* statements. So, the communication is approximate-timed. The computational behavior is also approximate-timed as is inherited from the previous model. *Bus-arbitration model* is subsumed into *transaction-level model* in [7]. [2]

4. **Bus-functional model.** It defines the bus-functional representation of components. It models the cycle-accurate implementation of bus transactions over the wires and protocols of the system bus. This model has protocols inlined into processing elements. So, it also includes pin-accurate interfaces. This model is cycle-accurate in terms of communication. The computational behavior is same (approximate-timed) as that inherited from the IP-assembly model. *Bus-functional model* corresponds to *communication model* in [5] and *behavior-level model* in [7].

5. **Implementation model.** It defines the components in terms of their register-transfers or instruction-set architecture. The granularity of time and hence of the event order in the system is refined down to individual clock cycles in each component. The communication as well as computational behavior both are cycle-accurate in this model. *Implementation model* corresponds to *implementation model* in [5] and *register transfer model* in [7].

Among above five models, *specification model*, *bus-arbitration model*, and *implementation model* are golden models since they are at pure un-timed, approximate timed, and cycle timed levels respectively. Other two models, namely *IP-assembly model* and *bus-functional model*, are intermediate ones defined to complete the our design flow described in Section 2.2. The refinement flow containing only the golden models is denoted by think gray solid arrows in Figure 1. It fits well into all the design approaches, including system-synthesis, platform-based design, and component-based design approaches.

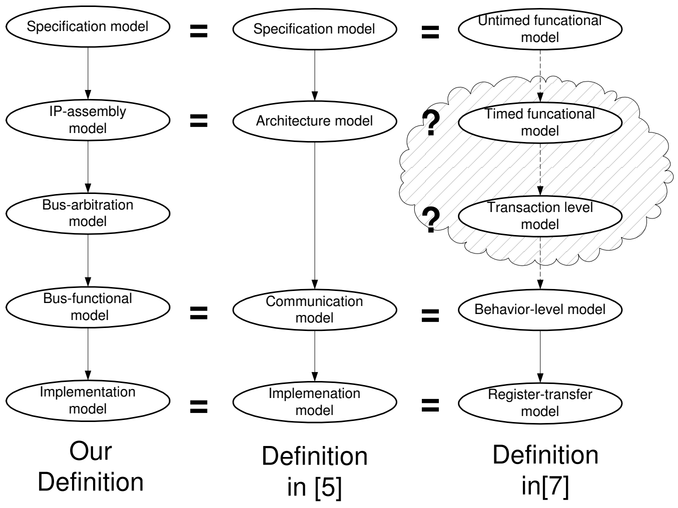


Figure 2. Comparison of defined abstraction models with models in [5] and [7]

The comparison between models defined in this report and models in [5] and [7] is presented in Figure 2. ”=” indicates equivalence of two models. It should be noted that the models defined in this report and those in [5] are taxonomized on the basis of their abstraction levels, while in cases of [7], *timed-functional model* is defined in terms of model’s timing aspect, *transaction level model* is defined on the basis of their communication modeling style, and rest of three models are defined in terms of model’s abstraction levels. Therefore, the *timed-functional model* and *transaction level model* in [7] are not really comparable with other models, which is indicated by cloud and question marks.

2.2 System design flow

Figure 3 demonstrates the system design flow as discussed in this report, which consists of the five abstraction models as discussed earlier.

The left-hand side of Figure 3 shows the refinement engine of the design flow. The right-hand side of Figure 3

[1]In [7], *timed functional model* is defined in terms of model’s timing aspect, rather than its abstraction level. Thus *timed functional model* can be a model representing a pure specification, a component structure of the system architecture, or even a complete system architecture.

[2]In [7], *transaction level model* is defined in terms of model’s communication modeling style, rather than its abstraction level. *IP-assembly model* and *Bus-arbitration model* defined in this report belongs to *transaction level model* in [7].

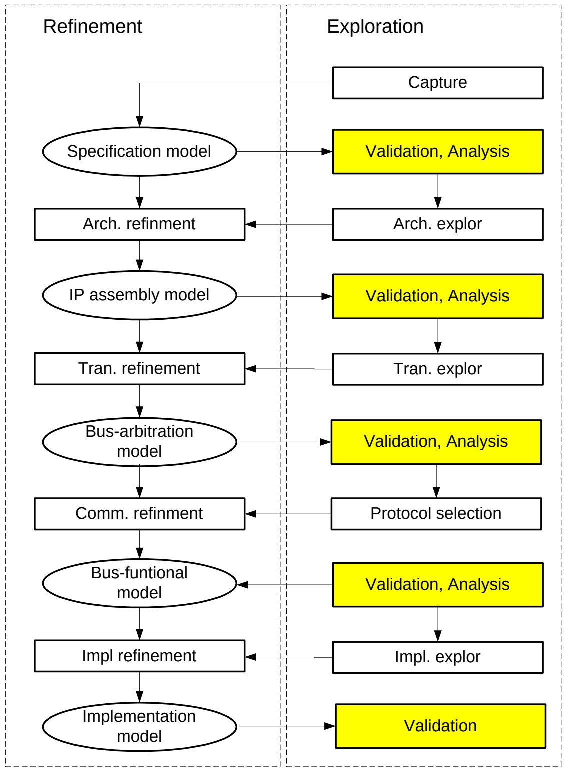


Figure 3. System design flow

shows the four exploration stages of the design flow as follows:

Architecture exploration At this stage, the selection of processing and storage components is performed. The processes of the system behavior are mapped to processing components. The variables of the system behavior are mapped to either local memories of PEs or global memories(storage components).

Transaction exploration This step involves defining the connectivity of processing and storage components via the system bus, i.e. defining the system topology. The channels are mapped onto the system buses (modeled as bus-arbitration model). At this stage the communication protocols are categorized into blocking and non-blocking protocols [3]. In this step, a bus arbitration mechanism is also selected if required.

Protocol selection This step involves explicit selection of bus protocols. The inlining of protocols is performed at this stage and decisions are made as to how different parts of a protocol are distributed among the processing elements.

Implementation exploration At this stage, the allocation, binding and scheduling decisions are made for the behaviors mapped to the custom hardware. The executable C code and RTOS are generated for the behaviors which are mapped to microprocessors. Unlike first three tasks which correspond to those in system level design, this main process of this task is on component design, which refines processes in PEs to the RTL/ISS levels.

[3]if two PEs communicate through an unblocking protocol, then the transmitter starts executing other tasks right after sending the data to the channel, without waiting the receiver to finish receiving data. On the other hand, if two PEs communicate through a blocking protocol, the transmitter cannot start executing other tasks until the receiver receives the data.

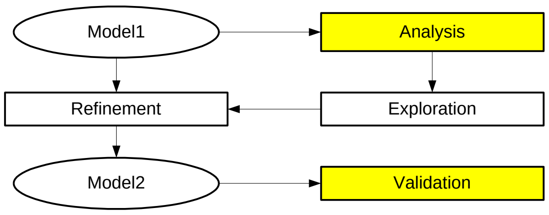


Figure 4. The general flow of synthesis task

The design flow consists of several steps, each of which generates the succeeding abstraction model from the preceding one. As shown in Figure 4, each design step consists of four tasks. In the Figure, *Model 1* is the initial model at start of the step while *Model 2* is the outcome model of the refinement step.

Analysis It involves analysis/estimation of *Model 1* to establish its characteristics.

Exploration This task focuses on design decision which shape the succeeding *Model 2*. It determines the implementation details which need to be added to transform the design to the succeeding *Model 2*, on the basis of the characteristics established in the previous task.

Refinement It focuses on model refinement. At this stage the design decisions made in the previous task are implemented by adding the implementation details to *Model 1*. This produces the succeeding *Model 2*.

Validation Finally, the newly generated *Model 2* is validated.

Although, the key ideas of this design flow are based on system-synthesis approach[5], it also subsumes platform-based and component-based approaches.

If we look at the design flow from point of view of the platform-based design approach, the processing element selection at the architecture exploration stage and interconnection topology generation at the transaction exploration stage are limited in scope.

On the other hand, the component-based design approach such as TIMA lab's[11] starts at the protocol selection stage and finishes at implementation refinement stage of the design flow.

2.3 Requirements on SLDLs

Figure 5 shows the general requirements on SLDLs, which are derived from Figure 4. We evaluate an SLDL in terms of fulfilment of these requirements. They are discussed as follows.

Analyzability In order to establish the characteristics of models, designers should be able to analyze them. So, an SLDL should be conducive to analysis of models at all the levels of abstraction. e.g SpecC features analysis by profiling/estimation.

Explorability The syntax and semantics of a SLDL should allow explicit specification of the characteristics of a model at any level of abstraction. This gives the designers an enhanced latitude in making implementation decisions. e.g. SpecC has *par* and *pipe* constructs for modeling parallelism and pipelined execution respectively.

Refinability The exploration tools should allow specification of design decisions taken, in an explicit format. This allows unambiguous refinement of the model using refinement tools or through manual refinement. Secondly, the modeling styles of the model to be refined and the resulting model after refinement should be consistent.

Validability Models written in SLDL should be able to be validated at all the levels of abstraction. e.g. SpecC and SystemC both allow validation by simulation. Besides validation by simulation, SpecC also allows validation by refinement. Abdi [1] develops theorems and proofs to show that the refinement algorithms for SpecC produce the outcome models which are functionally equivalent to the initial models. Thus, the correctness of the outcome models can be ensured by validating the initial models and refinement algorithms.

In order to study the suitability of SpecC/SystemC for system level design, we model a simple system. We start with capturing the specification model, subsequently implementing the intermediate models following the design flow in Figure 3, and finally develop the implementation model.

At each design step, models are written in both SpecC and SystemC languages. The design guidelines introduced in [6] are followed throughout this process. The two languages are compared in terms of their fulfilment of the requirements on a SLDL as discussed earlier. Finally the modeling guidelines are provided for both SpecC and SystemC users for the cases where the constructs or the features of the respective languages give rise to an ambiguous design.

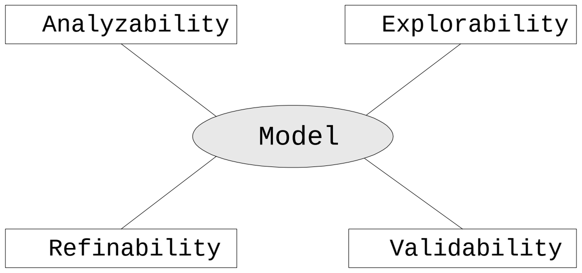


Figure 5. Requirements on system level design languages

Validation is done by simulation in case of both SpecC and SystemC, so they are not compared in term of validability. We compare them in terms of analyzability, explorability and refinability.

3. Specification model generation

In this section, we discuss the method of generating the *specification model* from the design behavior.

3.1 Design Behavior

The behavior of the design example is shown in Figure 6. It has two inputs (*a* and *b*) and an output (*c*). The design consists of five functional blocks: *B1*, *B2*, *B3*, *B4*, and *B2B3*. *B1* computes *v1*. *B2* and *B3* computes *v2* and *v3* with *v1* as input. *B4* computes *c* with *v2* and *v3* as inputs. *B2B3* is a hierarchical block, which encapsulates *B2* and *B3*. The dotted line in *B2B3* represents the parallel execution of *B2* and *B3*. The functionalities of the blocks are shown in Figure 6.

3.2 Specification capture and modeling

The first step of design is modeling *specification model* using SpecC and SystemC. An ideal *specification model* allows smooth refinement at the later design steps and at the same time demands minimum modeling work. The general guidelines for developing the specification model are:

1. The functionality should be modeled hierarchically to allow easy manipulation of complex systems.

2. The granularity of functional blocks should be chosen such that it allows exhaustive exploration. Basically,

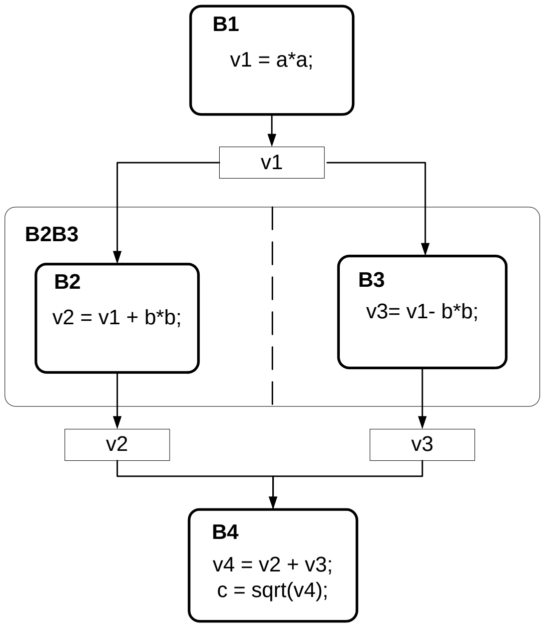


Figure 6. Design example: system behavior

the basic algorithmic blocks should form the smallest indivisible units for exploration.

3. The inherent parallelism of the system should be exposed for the later exploration.

4. The computation and communication each should be modeled such that it does not limit the efficiency of the other.

The modeling features of SpecC and SystemC are analyzed with respect to three aspects:

1. **Computation:** It reflects the capability of modeling functional blocks.

2. **Data transfer:** It reflects the capability of modeling the data exchange among functional blocks.

3. **Execution sequence:** It reflects the capability of modeling the execution sequence among functional blocks.

3.2.1 SpecC

The features of SpecC conducive to developing a specification model, are discussed with respect to the above mentioned three aspects.

Computation SpecC provides two basic computation units:

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

(a) Sequential execution

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

(b) Parallel execution

        void main{
            pipe{
                b1.main();
                b2.main();
            }
        }
(c) Pipelined execution
```

Figure 7. The execution sequence modeled in SpecC

1. **Function**: SpecC *function* follows the same semantics and syntax as the function in C language. A function can be called hierarchically and executes sequentially according to the calling sequence.

2. **Behavior**: SpecC *behavior* is specified by a *behavior* definition. There are two types of behaviors: *leaf behavior* and *composite behavior*. A *leaf behavior* may contain hierarchically called functions but it does not contain any sub-behavior instances. On the other hand, a *composite behavior* consists of sub-behaviors instances. These instances may be executing in parallel, pipeline, or FSM fashion, which are explicitly specified by *par, pipe*, and *fsm* constructs [5] respectively. The sequential execution is the default execution order. e.g. behavior instances *b1* and *b2* are executing, sequentially in Figure 7(a), in parallel in Figure 7 (b), and in pipeline fashion in Figure 7(c).

Data transfer SpecC supports data transfer between behaviors through either *variables* or *channels*. Each behavior has a set of ports which connect to the ports of other behaviors. It also has a set of variables and channels that connect the ports of its sub-behavior instances. Like behaviors, channels also have hierarchical structure.

Execution sequence Functions execute sequentially in SpecC. In case of behaviors, SpecC provides two mechanisms to model execution sequence.

1. **Static scheduling**: In this case, the sequence of execution of behaviors is explicitly specified with *par*, *pipe* and *fsm* constructs[5], the default order of execution being sequential.
2. **Dynamic scheduling**: SpecC uses *event-wait-notify* to schedule behaviors dynamically. SpecC has a data type *event* and the *wait* and *notify* statements which are used for synchronization between behaviors. When the wait statement such as *wait(e)* is executed, the behavior ceases to execute until the waited event *e* of a behavior is notified with *notify e* statement.

Guidelines The guidelines for modeling specification model, using SpecC, with respect to the previously discussed three aspects are:

1. **Computation**: *Leaf behaviors* are used to model smallest indivisible units for ease of exploration, with just *clean*[4]code. Although, the *composite behaviors* can contain a hierarchy of functions, it is advisable to use a hierarchy of behaviors. This allows parallel, pipelined and FSM execution of behavior.
2. **Data transfer**: Data transfer between behaviors is modeled by connecting ports through variables. We do not use channels, as separating computation and communication is not required at this stage.
3. **Execution sequence**: Static scheduling should be preferred for modeling execution sequence between behaviors as far as the design allows. e.g. if two behaviors need to synchronize while they are executing concurrently, then we have to rely on dynamic scheduling.

Example Figure 8 shows the specification model of the design in SpecC. The design contains four leaf behaviors: *B1*, *B2*, *B3* and *B4*. Behaviors communicate through variable *v1*, *v2* and *v3* declared in the behavior *Design*. The thin dotted arrows between behaviors represent the data transfer. *par* construct is used to specify parallel execution between *B2* and *B3*. The execution sequence is indicated by the thick lines/arrows. The SpecC code for behavior *B2B3* and *Design* is shown in Figure 9.

3.2.2 SystemC

The features of SpecC conducive to developing a specification model, are discussed with respect to the previously discussed three aspects.

[4]When we say clean code, we mean it does not contain any sub-behavior calls.

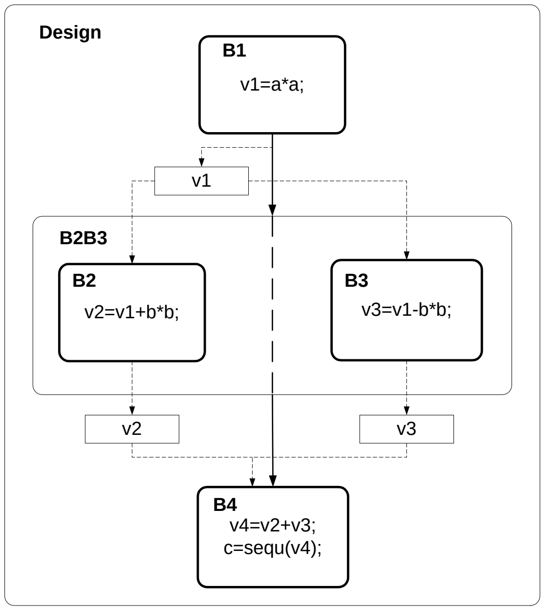


Figure 8. Design example in SpecC: specification model

Computation SystemC provides three basic computation units:

1. **Function**: A *function* is defined in the same way as the that in C language.
2. **Process**: *Processes* are the basic behavioral entities of SystemC. Although, a *process* can contain function calls, it cannot invoke other *processes*. Therefore, hierarchical modeling of processes is not possible.
3. **Module**: *Modules* are structural entities which serve as basic blocks for partitioning a design. Modeling using *modules* reflects structural hierarchy. The modules can be classified into two categories: *leaf module* and *composite module*. A *leaf module* contains processes, which specify the functionality of the module, but it does not contain any module. A *composite module* consists of the instantiation of other modules.

Data transfer Data transfer is modeled by connecting modules' ports through either *signals* or *channels*. The data transfer between modules essentially means data transfer between processes in different modules. The data transfer between processes in a module is performed through either *signals/channels* connected to the modules' port or *signals/variables* declared in the module.

Execution sequence SystemC only supports dynamic scheduling of execution sequence. There are two mecha-

Unit	Modeling hierarchy	Data transfer medium	Execution sequence
Function	yes	variable	sequential
Process	no	signal, channel, variable	static sensitivity dynamic sensitivity(sc_event, sc_signal), channel
Module	yes	signal, channel	static sensitivity dynamic sensitivity(sc_signal), channel

Table 1. Features of computational units of SystemC.

```
// Parallel composition of B2 || B3
behavior B2B3(in  int b, in int v1,
              out int v2, out int v3)
{
  B2 b2(b, v1, v2);
  B3 b3(b, v1, v3);

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

// Top-level behavior, specification model
behavior Design(in  int a, in int b,
                out double c)
{
  int v1;
  int v2;
  int v3;

  B1   b1(a, v1);
  B2B3 b2b3(b, v1, v2, v3);
  B4   b4(v2, v3,c);

  void main(void)
  {
    b1.main();
    b2b3.main();
    b4.main();
  }
};
```

Figure 9. The SpecC code for the specification model of behaviors *B2B3* and *Design*

nisms for dynamic scheduling:

1. **Static sensitivity**: When designers use a static sensitivity mechanism, a list of signals are specified in a "sensitivity list" of a process. If the value of any signal in the sensitivity list of a process changes, the process starts/resumes execution.

2. **Dynamic sensitivity**: The dynamic sensitivity mechanism uses *event-wait-notify* to schedule processes, which is the same as the dynamic scheduling in SpecC. A process can wait and notify a event. If the waited event of a process is notified, the process starts/resumes execution. However, in case of SystemC ports of modules cannot be connected through an event(*sc_event*), therefore, the *event-wait-notify* cannot be used for synchronization between processes in different modules. Designers have to encapsulate events into a channel in order to achieve synchronization between such processes.

Table 1 analyzes the computational units of SystemC for three features: possibility of modeling of hierarchy, means of data transfer and scheduling mechanism for the execution sequence.

Guidelines The guidelines for modeling specification model, using SystemC, with respect to the previously discussed three aspects are:

1. **Computation**: *Leaf modules* should be used to model smallest indivisible units for ease of exploration. Processes are encapsulated inside these *leaf modules* to model their functionality. *Modules* are preferred for modeling the smallest indivisible algorithmic units on account of two reasons. Firstly, *processes* are not capable of modeling structural hierarchy. Secondly, dynamic scheduling of *functions* is not supported.

2. **Data transfer**: Data transfer between *modules* is modeled by connecting their ports through *signals*.

3. **Execution sequence**: The dynamic sensitivity mechanism is recommended for the reasons explained later in Section 4.2.2. However, for cases where a process

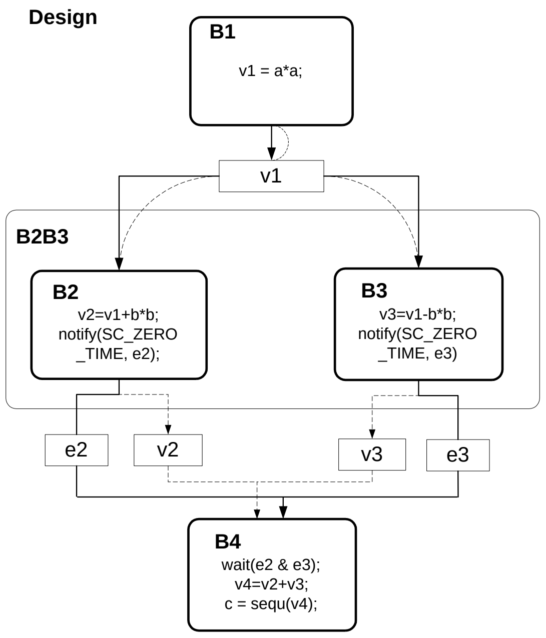


Figure 10. Design example in SystemC: specification model

has to wait for the output of another process, static sensitivity mechanism can be applied.

Example Figure 10 shows the specification model of the design in SystemC. The design contains four leaf modules: *B1, B2, B3* and *B4*. Modules exchange data through signals *v1, v2* and *v3*. The thin dotted arrows between modules represent the data transfer. The thick solid arrows shows the execution sequence. The execution sequence of modules are determined by either static sensitivity mechanism or dynamic sensitivity mechanism. Module *B2* and *B3* are statically sensitive to signal *v1* while module *B4* is dynamically sensitive to events *e2* and *e3* since this situation can not be modeled by static mechanism. *B4* contains statement *wait (e2 & e3)* which directs it to wait for completion of execution of both *B2* and *B3*. Since an event cannot be connected to the port of modules, *e2* and *e3* are declared globally. The SystemC code for module B2B3 and Design is shown in Figure 11.

3.2.3 Comparison

Although SpecC and SystemC share many features, such as dynamic sensitivity mechanism for dynamic scheduling of execution sequence, there are three primary differences which we come across when specification modeling is considered.

```
SC_MODULE(B2B3) {
 sc_in<int> b;
 sc_in<int> v1;
 sc_out<int> v2;
 sc_out<int> v3;

B2 *b2;
B3 *b3;

SC_CTOR(B2B3)
{
    b2 = new B2("b2");
    ... //port binding

    b3 = new B3("b3");
    ... //port binding
}
};

SC_MODULE(Design) {
 sc_in<int> a;
 sc_in<int> b;
 sc_out<double> c;
 sc_signal<int> v1;
 sc_signal<int> v2;
 sc_signal<int> v3;

B1 *b1;
B2B3 *b2b3;
B4 *b4;

SC_CTOR(Design)
{
    b1 = new B1("b1");
    ... //port binding

    b2b3 = new B2B3("b2b3");
    ... //port binding

    b4 = new B4("b4");
    ... //port binding
}
};
```

Figure 11. The SystemC code for the specification model of behaviors *B2B3* **and** *Design*

1. SpecC uses a *behavior*, which is a consolidated representation for both structure and behavior. But in case of SystemC, there is a separation of the basic structural and behavioral entities. The structure is modeled using (*modules*) and behavior is modeled using (*processes*). In summary, SpecC supports behavioral hierarchy which is not available in SystemC.

2. SpecC supports static scheduling while SystemC has to depend only on dynamic scheduling of execution sequence. Therefore, synchronization between concurrently executing processes in SystemC is complex and tedious to model. (e.g. *B4* in Figure 11).

3. SpecC doesn't support static sensitivity mechanism while SystemC does. However, the static sensitivity mechanism in SystemC has disadvantages which will be explained later in Section 4.2.2. These disadvantages can be circumvented by using dynamic sensitivity mechanism, which is same as dynamic scheduling supported by SpecC.

Therefore, we conclude that SpecC is better capable for specification modeling as compared to SystemC.

4. IP-assembly model generation

In this section, we introduce the process of IP-assembly model generation. We first analyze *specification model* and perform *architecture exploration*. We then carry out *architecture refinement* based on the decisions taken after exploration.

4.1 Architecture exploration

Architecture exploration entails selection of processing elements (PEs) and mapping the behaviors on to them. It also involves allocating of local and global memories and mapping the variables of the behaviors on to them. If required, it also determines the RTOS of microprocessor processing elements.

Designers perform architecture exploration on the basis of following two factors.

Complexity of functional blocks The complexity of functional blocks is estimated by profiling the specification model.

Execution sequence The execution sequence of functional blocks is determined by analyzing the specification model.

4.1.1 SpecC

SpecC is a super-set of C langauge. Therefore, SpecC model can be profiled easily. In addition, SpecC has a profiler[4] with a user friendly graphical interface.

The execution of behaviors is primarily scheduled with the use of definitions *par, pipe* and *fsm* constructs and the default execution order is sequential. Therefore, the execution sequence can be directly derived just by reading the specification. e.g. the SpecC code for the top level behavior *Design* and behavior *B2B3* in the taken example are shown in Figure 9. In *B2B3*, *par* construct indicates that behaviors *B2* and *B3* execute in parallel. In *Design*, since none of the *par, pipe*, or *fsm* constructs are used, we know by default the behavior instances *B1*, *B2B3*, and *B4* execute sequentially. Here, behaviors *B2* and *B3* can be mapped to different processing elements in order to exploit their parallelism discovered directly from the specification.

4.1.2 SystemC

SystemC is a C++ library extension. It is difficult to profile the SystemC model accurately because of the C++ class library burden, which obscures the computational needs of the system under consideration from the computational overheads of the SystemC simulator. Therefore, profiling has to be performed with the C specification model.

There are three ways to determine the execution sequence.

1. Specification model can be analyzed manually or automatically by tools. However, this analysis is tedious. e.g. Figure 11 depicts only the structural hierarchy without providing any information about execution sequence. Designers must investigate the processes in each leaf module to establish all the static and dynamic sensitivities. If the specification model is complex and contains many hierarchal levels, analysis is extremely difficult.

2. Languages, such as UML, can be used to record the execution sequence.

3. The execution sequence can be annotated to the SystemC specification.

Guidelines Because SystemC doesn't support a straightforward profiling and doesn't explicitly specify the execution sequence of behavior, we recommend the guidelines for architecture exploration using SystemC as follows:

1. C specification model should be profiled before converting the design into SystemC.

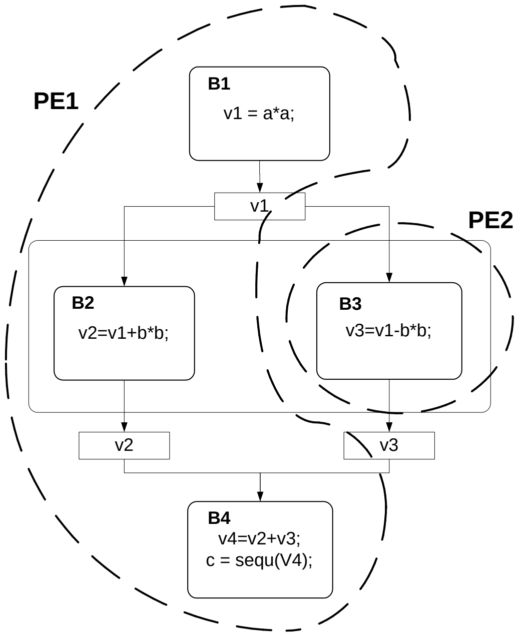


Figure 12. Design example: architecture exploration result

2. The execution sequence should be modeled using languages such as UML or should be annotated to SystemC specification.

4.1.3 Comparison

We conclude that SpecC is better suited for the architecture exploration as compared to SystemC in terms of profiling and determination of execution sequence. A SpecC model can be profiled easily and designers or tools can determine the execution sequence of behaviors by identifying constructs such as *par* and *pipe* etc. This eases the decision making task of architecture exploration. However, SystemC has to rely on C specification model for profiling and on UML/annotations for determining execution sequence.

4.1.4 Example

In the design example, we select two processing elements from the library. We map the parallel behaviors *B2* and *B3* to two different processing elements in order to exploit parallelism. The mapping decision is shown in Figure 12. In our IP-assembly model we do not require global memory. All the variables are mapped to the local memories of the processing elements. We use message passing architecture for communication.

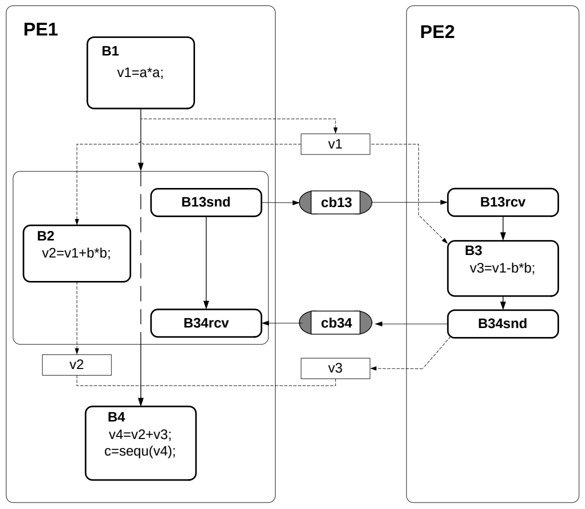


Figure 13. Design example in SpecC: after PE allocation and behavior mapping

4.2 Architecture refinement

Architecture refinement gradually refines the specification model to the IP-assembly model on the basis of decisions taken during architecture exploration. Architecture refinement[6] consists of four steps as follows.

1. **PE allocation.** Processing elements(PEs) and memory components are selected out of the component/IP library and instantiated as part of the system architecture.

2. **Behavior mapping.** Behaviors are mapped onto the processing elements and refined by adding behavior execution delays.

3. **Memory mapping.** Global variables in the specification model are assigned to local memory of a processing element in case of a message-passing implementation or to a dedicated, global memory component in case of a shared-memory architecture.

4. **Scheduling.** Since any processing element(PE) only executes single process at a time, processes inside each PE are sequentialized by either static scheduling or dynamic scheduling.

4.2.1 SpecC

PE allocation and Behavior mapping In our example, at the top level, two behaviors, called *PE1* and *PE2*, are added to represent two processing elements. These two behaviors represent the structural entities. We use the *par* construct to ensure the parallel execution of the two behaviors.

Next, we map behavior *B3* to *PE2* while map rest of the behaviors to *PE1*. The behavioral hierarchy of the specification changes after this mapping. Therefore, the behaviors must be rescheduled . We insert pairs of behaviors communicating via message-passing channels for synchronization between concurrently executing behaviors *PE1* and *PE2*, as shown in Figure 13. The pair *B13snd* and *B13rcv* synchronizes before the starting point of *B3* via channel *cb13*, and pair *B34rcv* and *B34snd* synchronizes after the end point of *B3* via channel *cb34*. *B34rcv* executes after *B13snd*. The behavior containing *B13snd* and *B34rcv* executes in parallel with *B2*.

Memory mapping Memory mapping using SpecC consists of further two steps:

1. A local copy of each global variable is created in each of processing elements where the variable is used.

2. A channel and a pair of behaviors is inserted for each global variable. The channel is needed for data transfers between the local copies of the global variables, created in the processing elements. The pair of behaviors read/write the values of the local copies of variables over the inserted channels.

   If a channel and a pair of behaviors were added to the model for the synchronization during the previous refinement steps, the two channels and two pairs of behaviors are merged at this stage if possible.

In our example, we first create local copies for global variable *v1* and *v3*. Then, we create a channel and a pair of behaviors for *v1*, and a channel and a pair of behaviors for *v3*. Finally, we merge the channel and the pair of behaviors for *v1* with channel *cb13* and behaviors *B13snd* and *B13rcv* inherited from the previous refinement steps. We also merge the channel and the pair of behaviors for *v3* with channel *cb34* and behaviors *B34snd* and *B34rcv*. The final model is shown in Figure 14.

Scheduling Before scheduling, we remove the redundant behavior hierarchy. When using SpecC with the design example, we remove the behavior encapsulating *B13snd*, *B2*, and *B34rcv*.

Static scheduling In general, Static scheduling task consists two steps:

1. Identification of the possible parallelism inside each processing element.

2. Sequentializing the parallel processes.

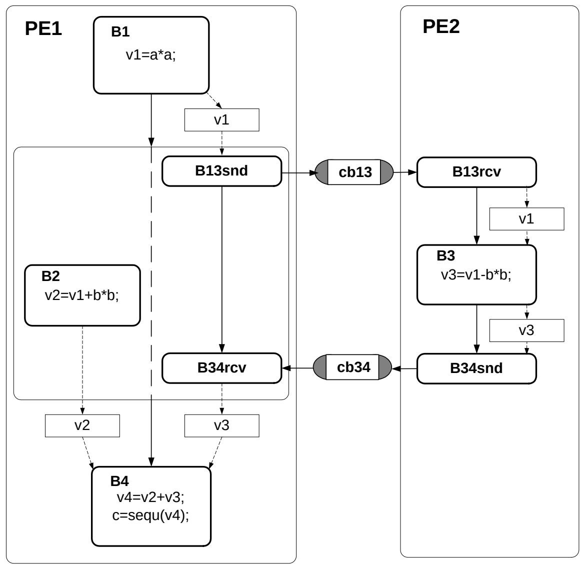


Figure 14. Design example in SpecC: after memory mapping

Since, SpecC supports constructs for explicit specification of the execution sequence, the inherent parallelism is easily identified.

Furthermore, designers can easily serialize the parallel behaviors just by removing *par* construct since default execution sequence is serial. The result of static scheduling of the example is shown in Figure 15.

Dynamic scheduling For the dynamic scheduling of behaviors, a new behavior has to be created in each processing element. The newly created behavior acts as a scheduler. It notifies an event, to the behavior which is ready to execute. All the behaviors start or resume execution when their waited events are notified.

4.2.2 SystemC

PE allocation and Behavior mapping In our example, at the top level, two modules, called *PE1* and *PE2*, are added to represent the two processing elements.

Next, we map module *B3* to *PE2* while rest of the modules are mapped to *PE1*. Since a module in SystemC is a structural entity and the processes are scheduled dynamically , we just move the module *B3* to PE2 without any need for reschedule. The final model is shown in Figure 16.

Memory mapping In the SystemC model, the global variables are represented by global signals. However, the signal is not only used for data transfer, but it is also used for scheduling of execution sequence using the sensitivity list.

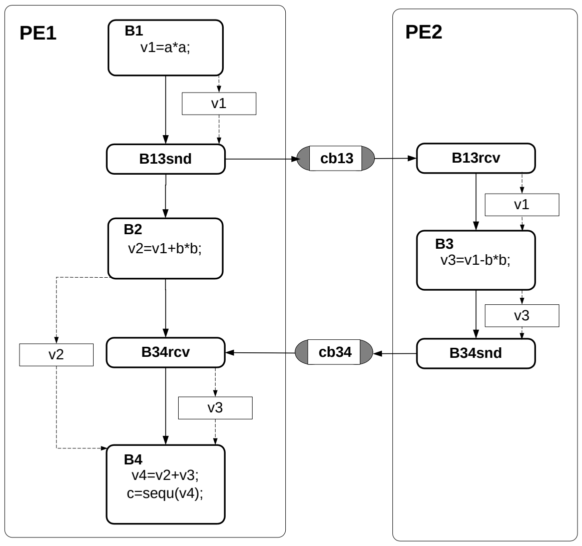
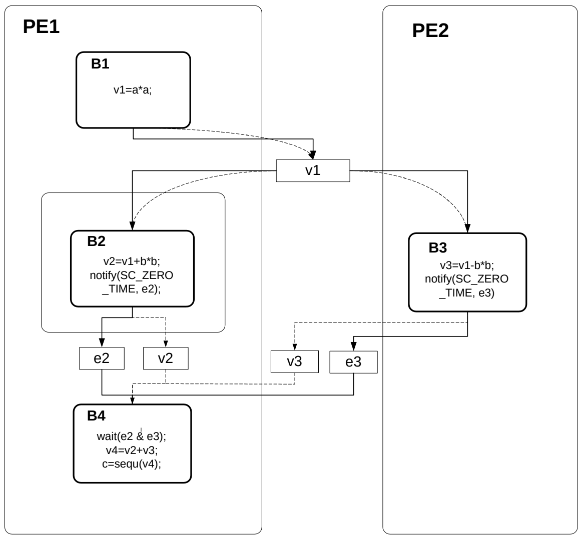
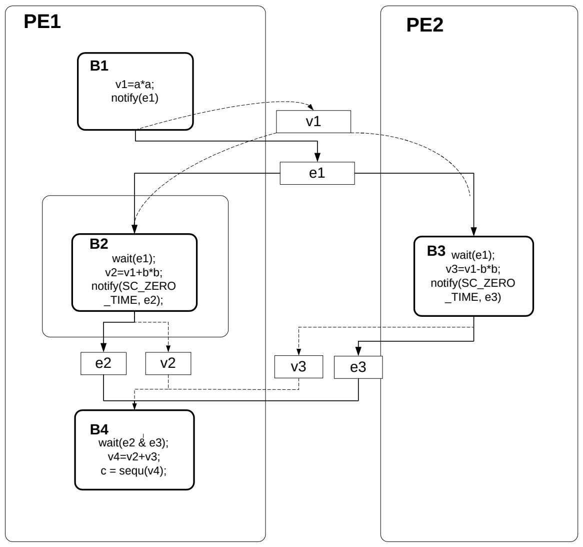


Figure 15. Design example in SpecC: after static scheduling



PE1
PE2
B1 v1=a*a;
v1
B2 v2=v1+b*b; notify(SC_ZERO _TIME, e2);
B3 v3=v1-b*b; notify(SC_ZERO _TIME, e3)
e2
v2
v3
e3
B4 wait(e2 & e3); v4=v2+v3; c=sequ(v4);

Figure 16. Design example in SystemC: after PE allocation and behavior mapping



PE1
PE2
B1 v1=a*a; notify(e1)
v1
e1
B2 wait(e1); v2=v1+b*b; notify(SC_ZERO _TIME, e2);
B3 wait(e1); v3=v1-b*b; notify(SC_ZERO _TIME, e3)
e2
v2
v3
e3
B4 wait(e2 & e3); v4=v2+v3; c = sequ(v4);

Figure 17. Design example in SystemC: after step 1 of memory mapping

Therefore, memory mapping in SystemC not only influences the data transfer, but also interferes with the scheduling of execution sequence.

The architecture refinement step involving memory mapping using SystemC consists of five steps:

1. An event*(sc_event)* is created for each global signal if it is used in the sensitivity list of any process. This newly created event is substituted for the global signal in the sensitivity list. It is used to achieve complete separation of data transfer from scheduling and to perform scheduling using dynamic sensitivity mechanism.

2. A local copy of each global signal and its corresponding created event is maintained in the processing elements where the variable is accessed.

3. A channel and a pair of processes is inserted for each global signal. The channel is needed for data transfers between the corresponding local copies of each global signal, created in the processing elements. The pair of processes read/write the values of the local copies of the signals over the inserted channels. The scheduling after inserting the processes is done using the local copies of the event.

4. The local copies of the corresponding events and signals are merged, if possible. This is done in order to replace the dynamic sensitivity by the static sensitivity.

5. Since, an event in SystemC is not allowed to connect to the port of a module, an event declared in a module

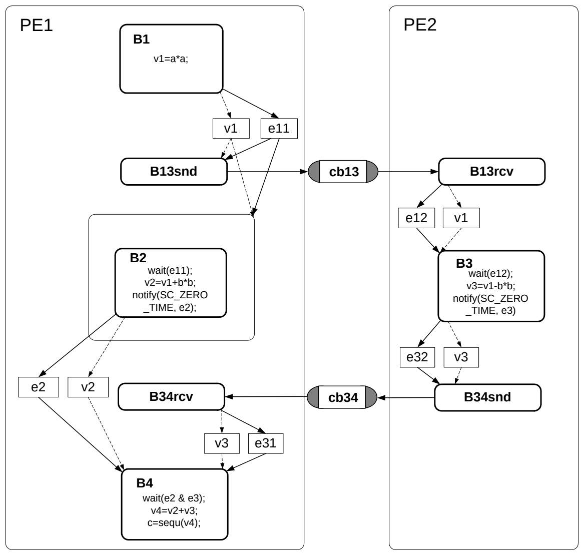


Figure 18. Design example in SystemC: after steps 2 and 3 of memory mapping

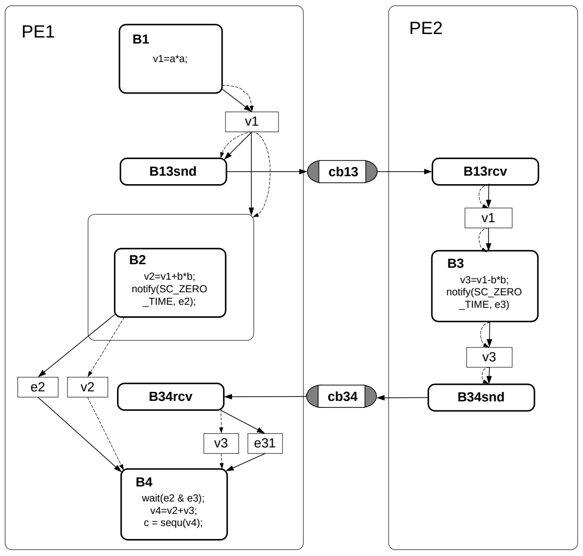


Figure 19. Design example in SystemC: after step 4 of memory mapping

cannot be accessed by its child modules. There are following two possible solutions.

(a) The remaining local events in each processing element are encapsulated by channels.

(b) All the modules in each processing element are removed while keeping the module's processes. Here, an event is used to schedule between processes, rather than between modules.

The model after performing step 1 of memory mapping for the example is shown in Figure 17. In step 1, event *e1* is created for the global signal *v1* to trigger the modules *B2* and *B3* after the execution of *B1* has finished.

The model after performing steps 2 and 3 of the memory mapping for the example is shown in Figure 18. In this model, events *e11* and *e12* are the local copies of event *e1*. Event *e31* and *e32* are the local copies of event *e3*. Global signals *v1* and *v3* also have the local copies in both the processing elements *PE1* and *PE2*. A channel *cb13* and a pair of behaviors *B13snd* and *B13rcv* are inserted for exchange of data between local copies of signal *v1* in both the processing elements. Similarly, a channel *cb34* and a pair of behaviors *B34snd* and *B34rcv* are inserted for exchange of data between local copies of signal *v3* in both the processing elements.

The model after carrying out step 4 of memory mapping for the example is shown in Figure 19. At this step, the variable *v1* and the event *e11* in processing element *PE1*, the variable *v1* and the event *e12* in processing element *PE2*, and the variable *v3* and the event *e32* in processing element *PE2* are merged.

Finally, the model after performing step 5(a) of memory mapping for the design example is shown in Figure 20. Events *e2* and *e31* are encapsulated in the channel *c_1*.

Alternatively, the model after performing step 5(b) of memory mapping for the design example is shown in Figure 21. In this case, all the blocks inside the processing elements *PE1* and *PE2* represent processes instead of modules.

Scheduling When using SystemC with the design example, we have two cases. If we have followed step 5(a) of memory mapping then we remove all the modules in each processing element. Otherwise if we have followed step 5(b) of memory mapping, we already have modules replaced by processes.

Static scheduling As mentioned before, SystemC does not allow explicit specification of the execution sequence. Therefore, identifying the parallelism inside each processing element is difficult. Designers may need UML or SystemC annotation to specify the execution sequence between processes.

On the other hand, in order to serialize the parallel processes in SystemC, designers must add a pair of wait and notify statements, which is tedious if the behavior hierarchy is complex.

In the design example, We apply static scheduling on the model shown in Figure 21. We serialize processes *B13snd*, *B2*, and *B34rcv* such that they execute in the order of *B13snd*, *B2*, and *B34rcv*. A signal *v5* is added to serialize

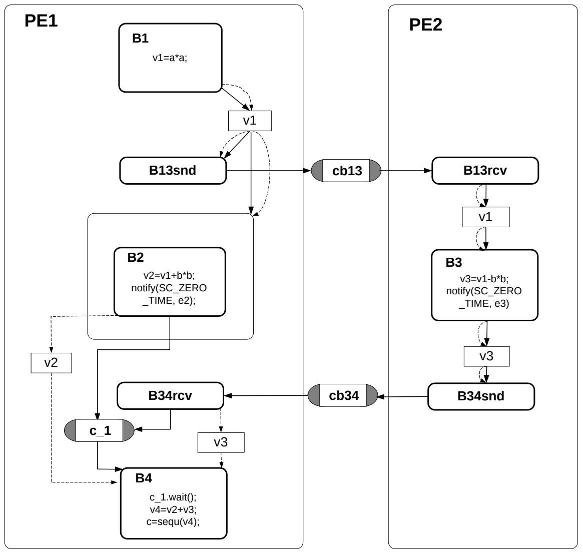


Figure 20. Design example in SystemC: after step 5(a) of memory mapping

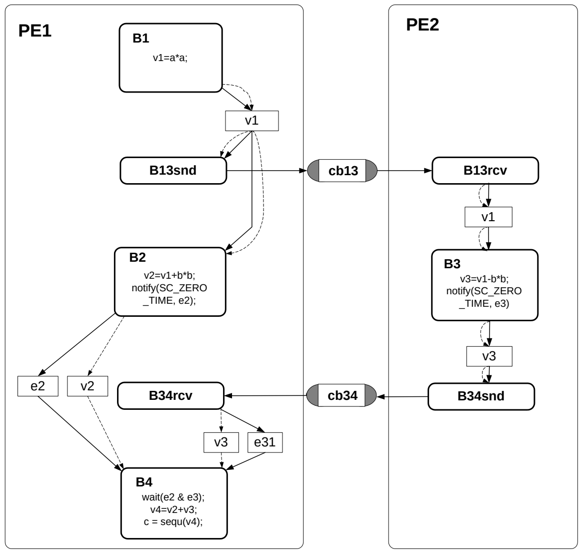


Figure 21. Design example in SystemC: after step 5(b) of memory mapping

processes *B2* and *B34rcv* using the static sensitivity mechanism. Event *e31* is merged with the signal *v3* in order to serialize the processes *B34rcv* and *B4*. Event *e2* is deleted. The model obtained after this architecture refinement step (static scheduling) is shown in Figure 22.

Dynamic scheduling The process of dynamic scheduling of SystemC is same as the scheduling of SpecC.

Guidelines In order to ease the tedious memory mapping task of architecture refinement, designers can follow the guidelines as given below:

1. Avoiding use of static sensitivity in the specification model
2. Following step 5(b) rather than step 5(a) discussed earlier.

4.2.3 Comparison

PE allocation and Behavior mapping The architecture refinement step involving allocation, partitioning and mapping is easier using SystemC compared to that using SpecC. Since the refinement changes the behavioral hierarchy, behaviors in SpecC have to be rescheduled. On the other hand, modules in SystemC can be easily moved across the parent module without the need for reschedule.

Memory mapping In general, the architecture refinement steps involving memory mapping are easier to perform using SpecC compared to that using SystemC. This is can be said on account of the following two reasons:

1. Use of static sensitivity in SystemC leads to interdependence between data transfer and execution sequence scheduling.
2. An event *sc_event* in SystemC cannot be used to connect ports thereby preventing the use of the events of a module by its child modules.

Scheduling In general, the complexities of dynamic scheduling using SystemC and SpecC are similar. But, implementation of static scheduling using SpecC is easier than that using SystemC because SpecC identifies the execution sequence of behavior while SystemC does not.

5. Bus-arbitration model generation

In this section, we describe the process of Bus-arbitration model generation. This section introduces two tasks: *transaction exploration* and *transaction refinement*.

5.1 Transaction exploration

The transaction exploration determines the interconnection of the system components via the system bus or in other words the topology of the system. The channels between the processing elements are mapped onto the system buses.

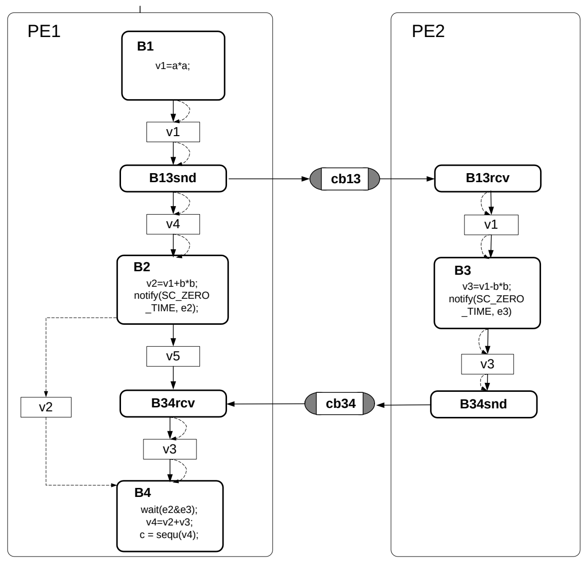


Figure 22. Design example in SystemC: after static scheduling

The bus protocol is chosen from among the two broad categories, blocking and non-blocking. A bus arbitration mechanism is also decided if required. The transaction exploration is determined by the following communication characteristics of the IP-assembly model:

1. Channel topology.
2. Channel traffic.
3. Start and end time of execution of channel.

The first characteristic, channel topology is the primary determinant of the overall system topology. The remaining two characteristics have bearing upon both the bus load and competition, they together determine the bus protocol selection and channel mapping.

5.1.1 SpecC

Designers can easily derive channel topology since SpecC allows explicit specification of the channel topology in the IP-assembly model.

The Designers can obtain channel traffic by profiling using the profiler[4].

However, the start and end time of execution of channel cannot be evaluated from the IP-assembly model because it depends on the outcome of the communication exploration.

5.1.2 SystemC

Designers can easily derive channel topology since SystemC allows explicit specification of the channel topology in the IP-assembly model.

As discussed earlier in Section 4.1, it is tedious to profile SystemC, on account of its C++ library burden.

Again, the start and end time of execution of channel cannot be evaluated from the IP-assembly model because it depends on the outcome of the communication exploration.

5.1.3 Comparison

We conclude that SpecC and SystemC have similar capabilities in terms of support for transaction exploration, except for the determination of channel traffic which is much easily feasible using SpecC on account of its profiling capability.

5.1.4 Example

There are only two processing elements in our design example, hence we select a bus *bus1* to connect *PE1* and *PE2*. Both channels *cb13* and *cb34* are mapped onto *bus1*. We select a blocking protocol for *bus1* with a master-slave arrangement. In this case PE1 is a master and PE2 is slave. We do not need any arbiter here, as there is only one master and one slave.

5.2 Transaction refinement

The decisions taken at the transaction exploration step (or outcome of transaction exploration step) are implemented during the transaction refinement. The transaction refinement consists of two steps:

1. Encapsulation of channels into a hierarchical channel representing the bus.

2. The functionality of the abstract channels representing the buses are implemented using the selected bus protocols i.e. blocking or non-blocking, with the bus modeled at the transaction level.

5.2.1 SpecC and SystemC

The first step of transaction refinement requires the hierarchical channel modeling. Second step requires abstract time modeling i.e. modeling communication delay, in channels. Both SystemC and SpecC support above requirements.

5.2.2 Comparison

We conclude that both the SpecC and SystemC are equally capable for transaction refinement and the refinement process is quite similar for both of them.

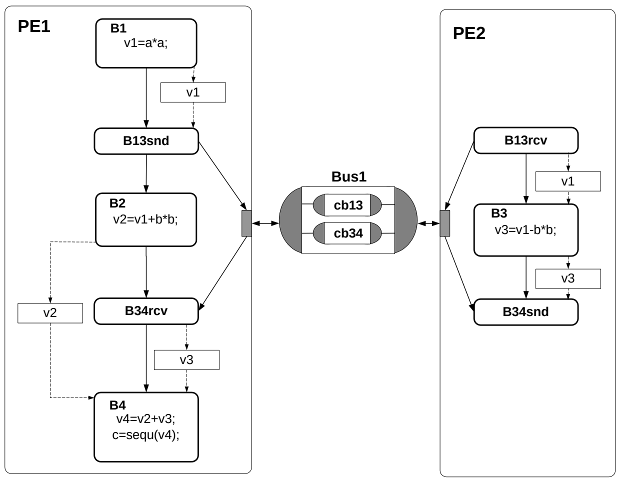


Figure 23. Design example in SpecC: bus-arbitration model

5.2.3 Example

In general, in the bus-arbitration model, bus arbiter is modeled to handle conflicts for the control of the bus. Every processing element is assigned a priority for bus grant. The total communication time for every transaction is annotated by using *wait* statements.

Our design example consists of only two processing elements. *PE1* is the bus master and *PE2* is the slave. Therefore, modeling a bus arbiter and priority assignment to processing elements is not required.

The model of the design example using SpecC after transaction refinement is shown in Figure 23. The difference between the models at steps 1 and 2 of transaction refinement is that the channel in step 2 contains *wait* statements representing the required communication time for each data transaction while step 1 does not. Similarly, the model of the design example using SystemC after transaction refinement is shown in Figure 24.

5.2.4 Guidelines

After the first step of transaction refinement, several channels in the PE-assembly model are encapsulated into a single hierarchical channel. As a result, all the communicating behaviors compete for the access of the newly created unified channel. So, in order to ensure the correctness of the resulting model, the behavior/process involving the channel accesses in each processing element must be serialized. Failure to comply to this may result in an incorrect outcome model which will be illustrated in Section 6.2.

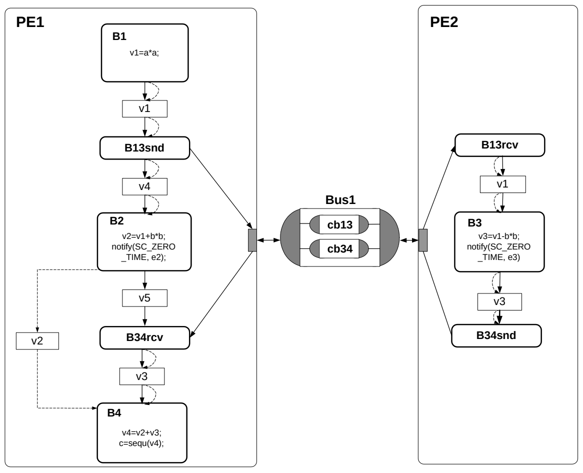


Figure 24. Design example in SystemC: bus-arbitration model.

6. Bus-functional model generation

This section introduces two tasks: *protocol selection* and *communication refinement*.

6.1 Protocol selection

After transaction refinement, the next step is the communication exploration. The communication exploration determines the exact bus protocols for buses from the broad blocking and non-blocking categories. The inlining of protocols is also performed at this stage and decisions are made as how different parts of a protocol are distributed among the processing elements. The communication exploration is determined by the bus protocol selected.

6.1.1 Comparison

SpecC and SystemC have similar capabilities in terms of support for communication exploration.

6.1.2 Example

As discussed earlier in Section 5.1, we already mapped channels *cb13* and *cb34* onto *bus1*. We now explicitly select the double handshake protocol for *bus1*. The double-handshake protocol is a point-to-point protocol in a master-slave arrangement. The master drives the address bus, signals the start of a transfer to the slave via the *ready* line and waits for an acknowledgement from the slave via the *ack* line. The slave, on the other hand, samples the *address* bus upon receiving the *ready* signal and, in case of an address match, acknowledges the transfer by asserting the *ack* line. The data bus can be driven by either the master or the slave

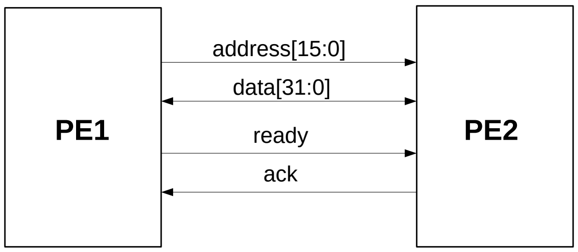


Figure 25. The interfaces of ***PE1*** **and** ***PE2*** **when using the double-handshake protocol**

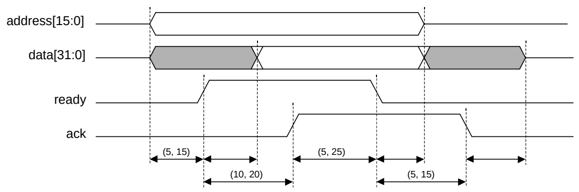


Figure 26. The timing diagram for the double-handshake protocol

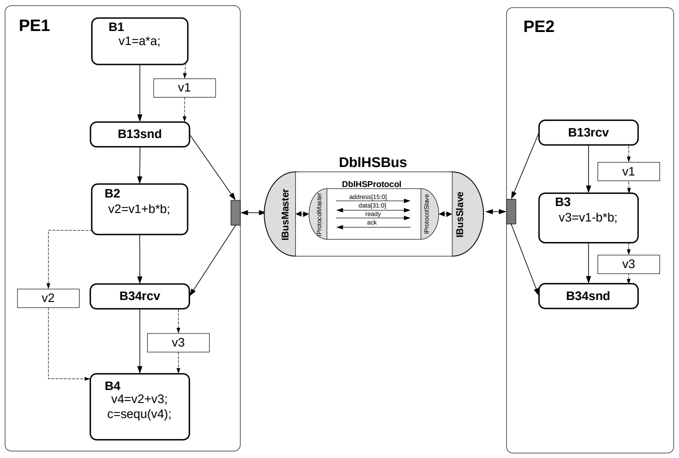


Figure 27. Design example in SpecC: after step 1 of communication refinement.

depending on the direction of the data transfer. The interconnection between *PE1* and *PE2* is shown in Figure 25. The timing diagram of the protocol is shown in Figure 26.

6.2 Communication refinement

The decisions taken at the communication exploration step are implemented during the communication refinement. The communication refinement consists of two steps:

1. The chosen bus protocol is modeled at the bus-functional level.

2. The communication functionality is inlined into the behaviors for implementation on the components. In course of this process, the communication functionality has to be refined and adapted to the component capability.

6.2.1 SpecC and SystemC

The first step requires modeling the channel parameters. The second and the final step requires implementing the functionality of channels in the behaviors/modules. Both SpecC and SystemC support above requirements.

6.2.2 Comparison

We conclude that both the SpecC and SystemC are equally capable for communication refinement and the refinement process is quite similar for both of them.

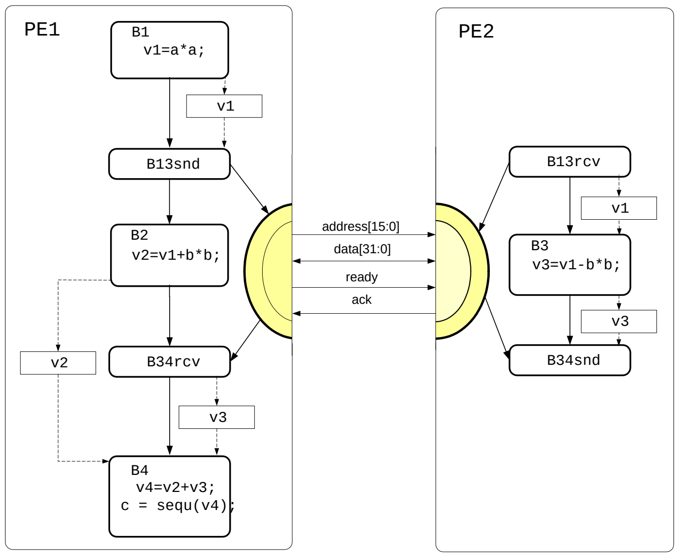


Figure 28. Design example in SpecC: after step 2 of communication refinement.

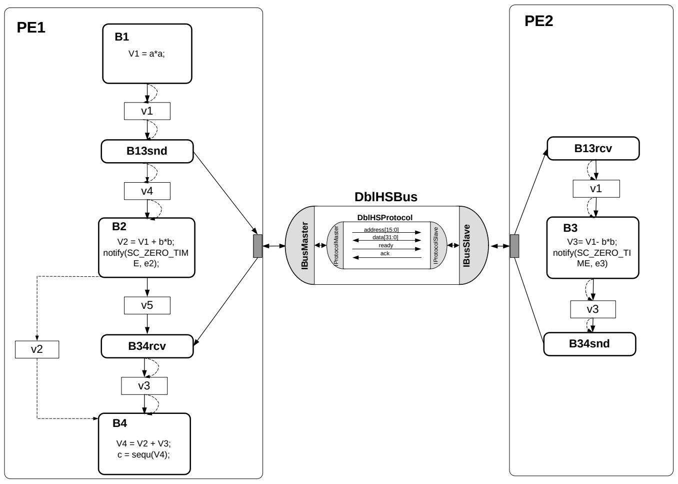


Figure 29. Design example in SystemC: after step 1 of communication refinement.

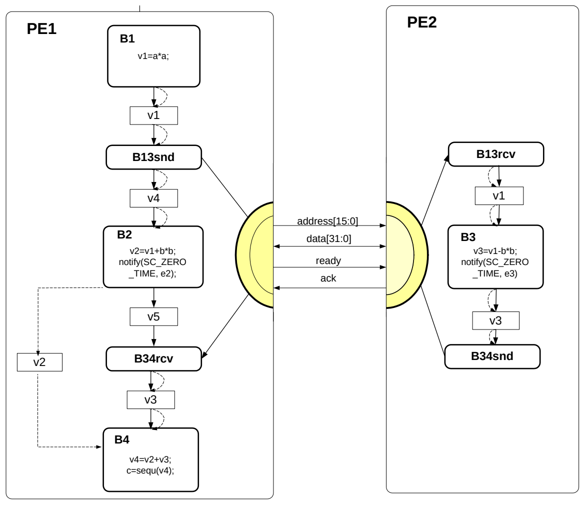


Figure 30. Design example in SystemC: after step 2 of communication refinement.

6.2.3 Example

The models of the design example using SpecC after communication refinement steps 1 and 2 are shown in Figures 27 and 28 respectively. The models using SystemC after refinement steps 1 and 2 are shown in Figures 29 and 30 respectively.

6.2.4 Guidelines

Here, we discuss an example illustrating the need to serialize the channel accesses of each processing element as stated in Section 5.2. e.g., in the design example, if we don't serialize the processes in SystemC during architecture refinement. Module *B13snd* in *PE1* will send signal *ready* to *PE2* then it will wait for the *ack* signal. After the module *B13rcv* in *PE2* asserts the *ack* signal, module *B34rcv* starts execution. Since *B34rcv* observes that *ack* is asserted, it starts receiving data from *PE2*. Obviously, the result obtained is incorrect.

Therefore, during architecture refinement, if dynamic scheduling is used instead of static scheduling, it is recommended to serialize the modules/processes that access the channels.

7. Implementation model generation

This section introduces two tasks: *implementation exploration* and *implementation refinement*.

7.1 Implementation exploration

The last task of design is to refine the bus-functional model to the implementation model. As defined in Section 2 implementation model is specified in terms of the register transfers for the behaviors mapped to custom hardware components and in terms of the instruction set architecture for the behaviors mapped to programmable processors. The implementation exploration achieves decision making for this task.

For the custom hardware components, implementation exploration consists of taking allocation, binding, and scheduling decisions of high-level/behavior synthesis for both the computation and communication/protocol parts. Implementation exploration is not required for programmable processors, because architecture exploration already determines the processor's RTOS and instruction set.

7.1.1 SpecC and SystemC

SpecC and SystemC are both C/C++ based languages. The implementation exploration for custom hardware using both of them is similar to the implementation exploration from C langauge, which has been studied in details in the field of high level synthesis.

7.1.2 Comparison

Therefore, we conclude that the capabilities of SpecC and SystemC are quite comparable with respect to implementation exploration.

7.2 Implementation refinement

The task of implementation refinement is to generate the implementation model, on the basis of the decisions taken during implementation exploration . The following are the steps involved in the implementation refinement.

1. Custom hardware synthesis: The behavior description is synthesized into a netlist of register-transfer level(RTL) components.

2. Software synthesis: The behaviors mapped onto a programmable processor are converted into C code, compiled into the processor's instruction set, and linked against an RTOS if required.

3. Synthesis of bus interfaces and bus drivers: The application and protocol layer [6] functionality is synthesized into a cycle-accurate implementation of the bus protocols on each component. This requires synthesis of bus interface FSMDs on the hardware side and generation of assembly code for the bus drivers on the software side.

Hardware Accellera RTL [2] standard defines five different RTL models, from the most abstract to the least abstract level:

1. Unmapped RTL: The behavior is scheduled to the cycle-accurate model. The Unmapped RTL is equivalent to the programming language code with exception that such code is divided into states, with conditional transition between states added to the code.

2. Storage-mapped RTL: The allocated storage unit such as register, register file, and memory are explicitly specified. The variables of behavior bound to the storage units are replaced by the specified storage units.

3. Function-mapped RTL: The function unit such as adder, shifter are allocated and the computation operations of behavior are replaced by the specified function units.

4. Connection-mapped RTL: The allocated local buses are specified and the variable of behavior bound to the local buses are replaced by the specified local buses.

5. Exposed-control RTL: The model consists of two parts: netlist of datapath components and a controller that assign a constant to each control variable in each state. The value of control variables determines the status/funcationility of each storage, functional or bus component in the datapath.

In this report we do not cover all the five RTL models. We select the unmapped RTL as the final model at the system level. The modeling below unmapped RTL model belongs to the behavioral synthesis problem, which we are not interested into.

The unmapped RTL generation is divided into two steps.

Flattening and merging This step of unmapped RTL generation involves flattening and merging behaviors/processes. This is performed because each custom hardware should contain only one FSM.

Refinement to cycle-accuracy This step involves refinement of the model to a cycle-accurate one. The statements in each cycle in the FSM are determined by implementation exploration/high-level-synthesis.

Software Implementation refinement for software involves conversion of the SpecC/SystemC model into a C model, its compilation into the processor's instruction set and linking against an RTOS if required. Implementation refinement also involves generation of assembly code for the bus drivers.

7.2.1 SpecC

Hardware SpecC supports all the five Accellera RTL models and their step by step refinement. For unmapped RTL generation, the *fsmd* construct in SpecC can be used to model the RTL level FSM, which models behaviors with cycle accuracy. For storage-mapped RTL generation, *buffered variable* construct can be used to replace the behavior variables with modular storage units. For function-mapped RTL generation, functions representing functional units can be used to replace the operations such as ”+”, ”-”. For connection-mapped RTL generation, data type *bit* can be used to replace the behavior variables in order to model the local buses. Finally, for exposed-control RTL generation, SpecC *signal variable* can be used to model the control variables for storage and functional components. Therefore, we conclude that SpecC has complete support for RTL modeling.

Flattening and merging behaviors in SpecC is straightforward. The leaf behaviors are removed and the statements of leaf behaviors are inserted into a hierarchical behavior depending on leaf behaviors' execution sequence.

We use the *fsmd* construct to specify cycle-accurate finite state machine in SpecC.

Software In order to convert a SpecC code to one in C, designers need to remove all SpecC language specific constructs and elements, such as *behavior*, *par*, *notify*, and *wait*. Because of the space constraint, we do not carry out the implementation refinement for software in this report.

```
// design.h
SC_MODULE(PE1) {

  sc_signal<int> v1;
  ...
  SC_CTOR(PE1)
  {
    ...
    SC_THREAD(b1);
    dont_initialize();
    sensitive << a;

    SC_THREAD(b13snd);
    dont_initialize();
    sensitive << v1;
    }
}

// design.cpp
void PE1::b1(){
    v1 = a*a;
};

void PE1::b13snd(){
    bus->write(ADDR_CB13, v1);
};
```

Figure 31. The SystemC code for processes *B1* **and** *B13snd***.**

7.2.2 SystemC

Hardware SystemC provides two ways to model the FSM.

1. Using SC_CTHREAD [7] for implicit modeling of FSM by inserting *wait* statement along with the statements executing every cycle.

2. Using SC_METHOD/SC_THREAD [7] for explicit modeling of FSM by using *switch* statement.

SystemC also provides functions, variables with data type *bit*, and signal *sc_signal*. SystemC uses signal to represent the storage unit in FSMD (SpecC uses *buffered variable*). Hence, we conclude that SpecC and SystemC provide similar support for RTL modeling.

Merging process is specially important for SystemC because it can remove the overhead of implementing static or dynamic sensitivity between processes in the same processing element. The processes should be statically serialized before flattening and merging.

Merging of processes using SystemC should be performed cautiously. This is because SystemC uses signals for data transfer between processes as well as for scheduling, and the value of a SystemC signal is not updated until a delta cycle. Furthermore, SystemC doesn't allow binding of variables to ports of modules, which limits use of variables for data transfer between processes in different modules.

```
// design.h
SC_MODULE(PE1) {

  sc_signal<int> v1;
  ...
  SC_CTOR(PE1)
  {
    ...
    SC_THREAD(b1_b13rcv);
    dont_initialize();
    sensitive << a;

}

// design.cpp
void PE1::b1_b13snd(){
    v1 = a*a;
    bus->write(ADDR_CB13, v1);
};
```

Figure 32. Incorrect SystemC code after merging processes *B1* **and** *B13snd***.**

For example, the SystemC models of process *B1* and process *B13snd* are shown in Figure 31. The incorrect SystemC model after merging *B1* and *B13snd* is shown in Figure 32. In this model, the statements of two processes are just put together. The model is incorrect because *v1* is a SystemC signal, whose value is not updated immediately. Therefore, the value of *v1* which *bus->write(ADDR_CB13, v1)* accesses is the old value, instead of the new value *a*a*. In order to solve this problem, the SystemC signals being used for data transfer between processes should be replaced by the SystemC variables. The correct model after merging *B1* and *B13snd* is shown in Figure 33.

We use SC_THREAD to specify the cycle-accurate finite state machine in SystemC.

Guidelines The SystemC signals being used for data transfer between processes should be replaced by the SystemC variables.

Software Conversion of a SystemC code to one in C involves two steps. First, SystemC code is converted to a C++ code by removing all the SystemC specific constructs and elements, such as *module, port, channel*. Second, C++ code

```
// design.h
SC_MODULE(PE1) {

   int v1;
   ...
   SC_CTOR(PE1)
   {
      ...
      SC_THREAD(b1_b13rcv);
      dont_initialize();
      sensitive << a;

}

// design.cpp
void PE1::b1_b13snd(){
     v1 = a*a;
     bus->write(ADDR_CB13, v1);
};
```

Figure 33. Correct SystemC code after merging process *B1* **and** *B13snd***.**

is then converted to a C code, which is compilable and executable on the microprocessors.

7.2.3 Comparison

Hardware Both the languages have similar capability for modeling cycle-accurate model. However, merging of the processes has quite complex consideration in case of SystemC compared to SpecC where it is fairly easy. Hence, we conclude that SpecC is better capable for implementation refinement compared to SystemC.

Software In both the cases of SpecC and SystemC the language specific constructs and elements need to be removed. Furthermore, since SpecC is C based language and SystemC is C++ based language, an additional step of converting a C++ code to a C code is required for SystemC.

7.2.4 Example

Hardware In this design, we assume both PE1 and PE2 are mapped to the custom hardware. The generated *unmapped RTL* models for SpecC and SystemC are the same, which are shown in Figure 34.

8. Overall Comparison

Table 2 illustrates the differences between system design using SpecC and SystemC in terms of design steps in our

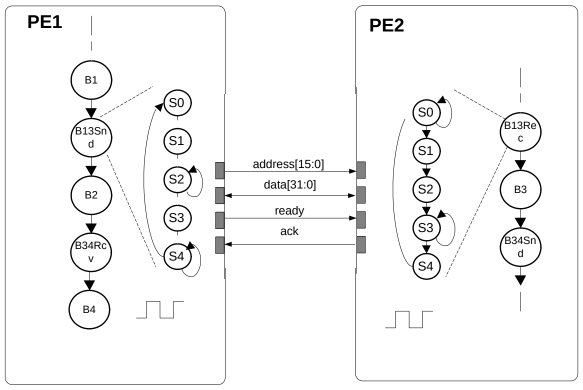


Figure 34. Unmapped RTL model of SystemC and SpecC

design flow. We classify the difficulty of each design task at three levels: easy, medium, and hard, in terms of the used system languages. The bold item indicates that it has the advantage over the normal item in the same row, where each row represents a step. System design using SpecC is easier at six steps, while system design using SystemC is easier at only one step. Therefore, we conclude SpecC is a better suited design language than SystemC considering the whole design flow.

Table 3 shows the overall comparison between SpecC and SystemC in terms of design modeling. Table 3 throws light on the reasons of the differences between SpecC and SystemC, shown in Table 2. Some of the major differences are summarized below:

1. SpecC supports static scheduling using *par, pipe,* and *fsm* constructs, or default sequential execution. Static schedule allows designers to determine the explicitly modeled execution sequence, which is used during architecture exploration. It also eases the static scheduling during architecture refinement. These features are not available in SystemC.

2. SystemC uses *module* as the structural entity and *process* as the behavioral entity. It does not support hierarchical modeling of *process*. Therefore, both *process* and *module* do not fully support behavior entity modeling. On the other hand, SpecC *behavior* supports modeling of behavioral hierarchy.

3. In case of SystemC *variable* and *event* cannot be used to connect the ports of different modules. Therefore, they can only be used either inside the modules or globally. This limits the use of events for scheduling modules and variables for data transfer between modules. On the other hand, SpecC *behavior* supports

Design steps	Sub-steps	SpecC	SystemC
Architecture exploration	Computation profiling	**Easy**	Hard: Tedious C++ library burden
	Executing sequence scheduling	**Easy: Explicit**	Hard: Implicit
Architecture refinement	Allocation and partitioning	Hard: Reschedule required	**Easy**
	Variable mapping	**Easy**	Medium: Data transfer and schedule separation
	Scheduling	**Easy: Explicit**	Hard: Implicit
	Behavior/module flattening	Easy	Easy(removal of modules in PEs)
Transaction exploration	Transaction Profiling	**Easy**	Hard
	Channel topology modeling	Easy	Easy
Transaction refinement	Channel grouping	Easy	Easy
	Transaction protocol insertion	Easy	Easy
Communication exploration	Exact protocol selection	Easy	Easy
	Channel inlining decisions	Easy	Easy
Communication refinement	Bus functional protocol insertion	Easy	Easy
	Channel inlining	Easy	Easy
Implementation exploration		N/R	N/R
Implementation refinement	Process/module merging	**Easy**	Medium: Conversion of signal to variable

Table 2. Overall comparison in terms of exploration and refinement

Abstract models	Model aspect	SystemC	SpecC
Specification model	functional block	module	behavior
	schedule	event, signal	event, definition(par..)
	data transfer	signal	variable
IP-assembly model	structure blocks	module	behavior
	functional blocks	process	behavior
	schedule inside PEs	event, signal	event, definition(par..)
	schedule between PEs	channel	channel
	data transfer inside PEs	signal	variable
	data transfer between PEs	channel	channel
Bus-arbitration model		same as Arch model	same as Arch model
Bus-functional model		same as Arch model	same as Arch model
Implementation model	fsm	switch(SC_THREAD), SC_CTHREAD	fsmd
	function units	function/module	function/behavior
	storage variable	signal	buffered signal
	bus	bit	bit
	control signal	signal	signal

Table 3. Overall comparison in terms of design modeling

scheduling using events and data transfer using variables without any constraint.

4. SystemC uses lower level semantics and syntax to model concepts at higher levels of abstraction. An example is the use of module (which is essentially a structural entity) as behavioral entity in the specification model. Another example is the use of signals for data transfer. Since the value of signal is updated after a delta cycle delay, using signal to model data transferring causes problems such as those described during the process merging step in Section 7.2.

5. SystemC is C++ based language, which is tedious to profile because of C++ library burden. There are no such limitations with SpecC.

6. In case of SystemC, when static sensitivity is used for scheduling, it affects both the data transfer and the execution sequence scheduling. Therefore, designers should only use dynamic sensitivity for scheduling in the specification model. This is not the case with SpecC.

The first four limitations of SystemC can not be circumvented since they follow from the definition of the semantics and syntax of SystemC. However, the last two limitations can be circumvented as discussed there.

9. Conclusion

We first establish the requirements on a SLDL for the system-level design flow. We come up with four essential properties required of a SLDL namely, analyzability, explorability, refinability and validability.

We then compare the capabilities of SpecC and SystemC in terms of the fulfilment of the established requirements on a SLDL. Although SpecC and SystemC share many concepts, SpecC proves better than SystemC in terms of fulfilment of these requirements. This is primarily on account of the clear semantics and syntax definition of SpecC. It should be noted that we choose a general design flow (which befits both SpecC and SystemC) in order to keep our evaluation fair to both.

We also provide design guidelines for SpecC and SystemC users. Although, following the guidelines allows for smooth and efficient system-level design, at the same time, the need of too many guidelines exposes the lack of expressiveness of the language. As we see, there are very few guidelines required for SpecC compared to SystemC where we have numerous guidelines. So we conclude SpecC is superior with respect to this aspect also.

References

[1] S. Abdi and D. Gajski. Formal Verification of Specification Partitioning. Technical Report CECS-TR-03-06, University of California, Irvine, March 2003.

[2] Accellera. RTL Semantics and Methodology, http://www.eda.org/alc-cwg.

[3] Accellera. SystemVerilog 3.0 Accellera's Extensions to Verilog, http://www.accellera.org.

[4] L. Cai and D. Gajski. Introduction of Design-Oriented Profiler of SpecC Language. Technical Report ICS-TR-00-47, University of California, Irvine, June 2001.

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

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

[7] Thorstn Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[8] Keutzer K, Newton AR, Rabaey JM, and Sangiovanni-Vincentelli A. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Dec 2000.

[9] OSCI. http://www.systemc.org.

[10] STOC. http://www.specc.org.

[11] Cesario WO, Lyonnard D, Nicolescu G, Paviot Y, Sungjoo Yoo, Jerraya AA, Gauthier L, and Diaz-Nava M. Multiprocessor SoC platforms: A Component-Based Design Approach. In *IEEE Design and Test of Computers*, Nov-Dec 2002.