CprE 588
Embedded Computer Systems

Prof. Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University

Lecture #11 – System-Level Design with SystemC

Outline

- System Design Methodology
- Specification Model Generation
- Bus Model Generation
- Implementation Model Generation
- Summary

L. Cai, S. Verma, and D. Gajski, "Comparison of SpecC and SystemC Languages for System Design", *Technical Report CECS-03-11*, Center for Embedded Computer Systems, University of California, Irvine, May 2003.

System Design Methodology

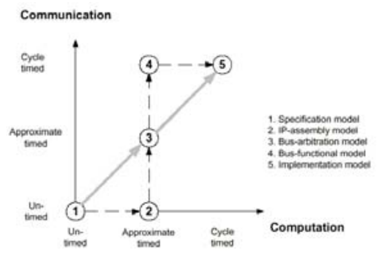


Figure 1. System modeling graph

System Design Methodology (cont.)

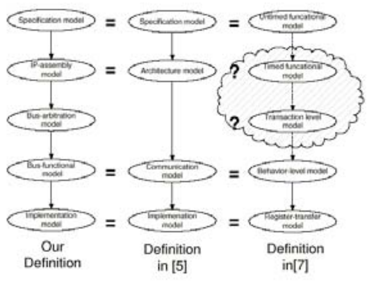


Figure 2. Comparison of defined abstraction models with models in [5] and [7]

System Design Methodology (cont.)

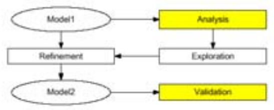


Figure 4. The general flow

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.

Specification Modeling

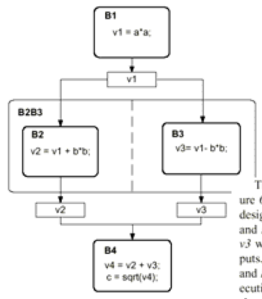


Figure 6. Design example: system behavior

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.

Specification Modeling (cont.)

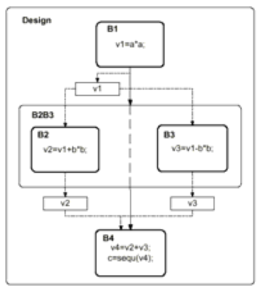


Figure 8. Design example in SpecC: specification model

```
// 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*

Specification Modeling (cont.)

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.

Specification Modeling (cont.)

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] respec-

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.

Specification Modeling (cont.)

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.

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.

SpecC vs. SystemC

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.

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

Specification Modeling

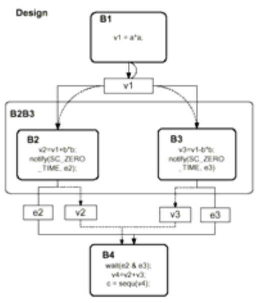


Figure 10. Design example in SystemC: specification model

```
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*

Specification Modeling (cont.)

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.

Architectural Exploration

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.

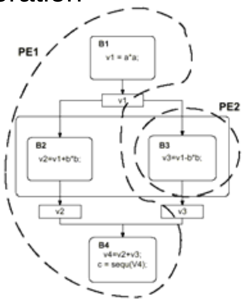


Figure 12. Design example: architecture exploration result

Architectural Exploration (cont.)

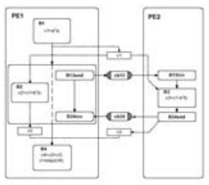


Figure 13. Design example in SpecC: after PE allocation and behavior mapping

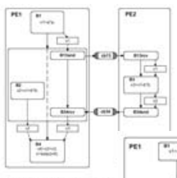


Figure 14. Design example in memory mapping

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.

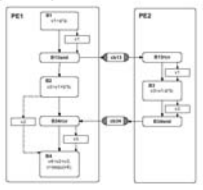


Figure 15. Design example in SpecC: after static scheduling

Architectural Exploration (cont.)

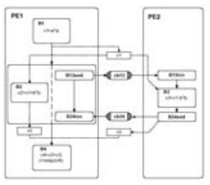


Figure 13. Design example in SpecC: after PE allocation and behavior mapping

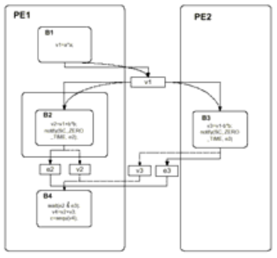


Figure 16. Design example in SystemC: after PE allocation and behavior mapping

Architectural Exploration (cont.)

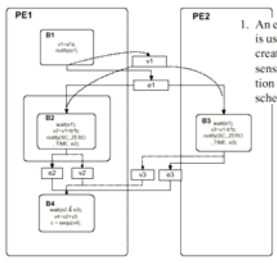


Figure 17. Design example in SystemC: after step 1 of memory mapping

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.

Architectural Exploration (cont.)

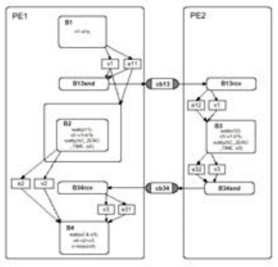
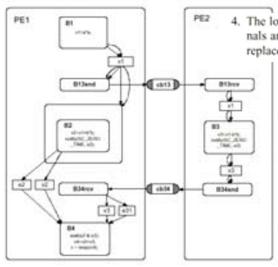


Figure 18. Design example in SystemC: after steps 2 and 3 of memory mapping

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.

Architectural Exploration (cont.)



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.

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

Architectural Exploration (cont.)

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

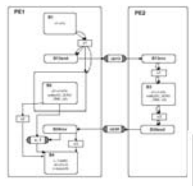


Figure 20. Design example in SystemC: after step 5(a) of memory mapping

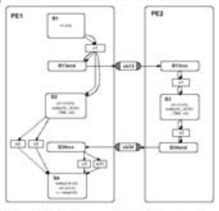


Figure 21. Design example in SystemC: after step 5(b) of memory mapping

Architectural Exploration (cont.)

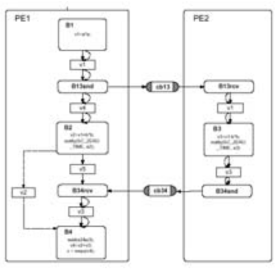


Figure 22. Design example in SystemC: after static scheduling

Architectural Exploration (cont.)

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.

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.

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.

Bus Model Exploration

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.

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.

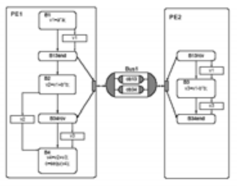


Figure 23. Design example in SpecC: bus-arbitration model

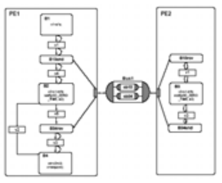


Figure 24. Design example in SystemC: bus-arbitration model.

Bus Model Exploration (cont.)

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.

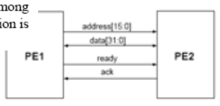


Figure 25. The interfaces of *PE1* and *PE2* when using the double-handshake protocol

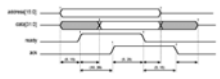


Figure 26. The timing diagram for the double-handshake protocol

Bus Model Exploration (cont.)

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.

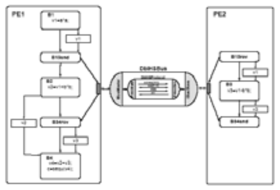


Figure 27. Design example in SpecC: after step 1 of communication refinement.

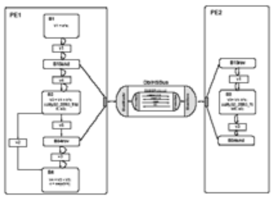


Figure 29. Design example in SystemC: after step 1 of communication refinement.

Bus Model Exploration (cont.)

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.

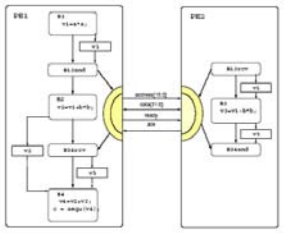


Figure 28. Design example in SpecC: after step 2 of communication refinement.

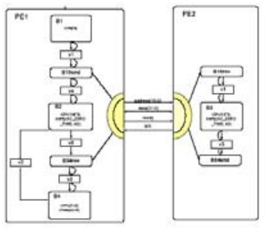


Figure 30. Design example in SystemC: after step 2 of communication refinement.

Implementation Model Generation

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 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.

Summary

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

Summary (cont.)

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