Center for Embedded Computer Systems
University of California, Irvine

System-On-Chip Architecture Modeling Style Guide

Junyu Peng
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-22
July 31, 2004

System-On-Chip Architecture Modeling Style Guide

Junyu Peng
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-22
July 31, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

http://www.cecs.uci.edu

Abstract

Contents

List of Figures

System-On-Chip Architecture Modeling Style Guide

J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski.
Center for Embedded Computer Systems
University of California, Irvine

July 31, 2004

Abstract

1 Introduction

System design in the SoC approach takes an initial specification of the system down to an actual implementation through a series of interactive and automated steps. Starting from a purely functional description of the desired system behavior, an implementation of the design on a heterogoeneous system architecture with multiple processing elements (PEs) connected through system busses is produced at the end of the design flow.

This report describes and defines guidelines and rules for developing SpecC based system models in general and the input to the SoC tools in particular.

1.1 SoC Design Flow

In the SoC design flow (Figure 1), five design models are used to represent the design at different abstraction levels. The design models are executable so that they can be simulated to verify the correctness of the design and obtain design performance metrics at each design step.

The most abstract model is the *specification model* that serves as the input to SoC tools. Specification model is a pure functional model that captures the functionality of the desired design. It should not contain any implementation details.

The *architecture model* is the output of architecture exploration. It exactly reflects the overall computation architecture consisting of processing elements (PEs). The architecture model encapsulates the communication between PEs through abstract message-passing channels.

After network exploration a *network model* is produced to reflect the communication network chosen for the design. It represents the allocation and selection of network stations and the links between them. While the communication is end-to-end between PEs in the architecture model, it is refined into point-to-point in the network model.

Finally, the *communication model* incorporates bus protocols into the model. The communication model can be pin-accurate or transaction-level. The *transaction level model* abstracts away the pin-accurate protocol details.

All the models are captured in SpecC language and have to adhere to the syntax and semantics of the SpecC language. Designers only need to write the specification model for the design and then use the tools to automatically generate the lower level models. SoC tools also support partial specification, which allows designers to start with a specification with incomplete computational functionality. Later designers can modify the computation part of the automatically generated models. However, for the modified models to be valid

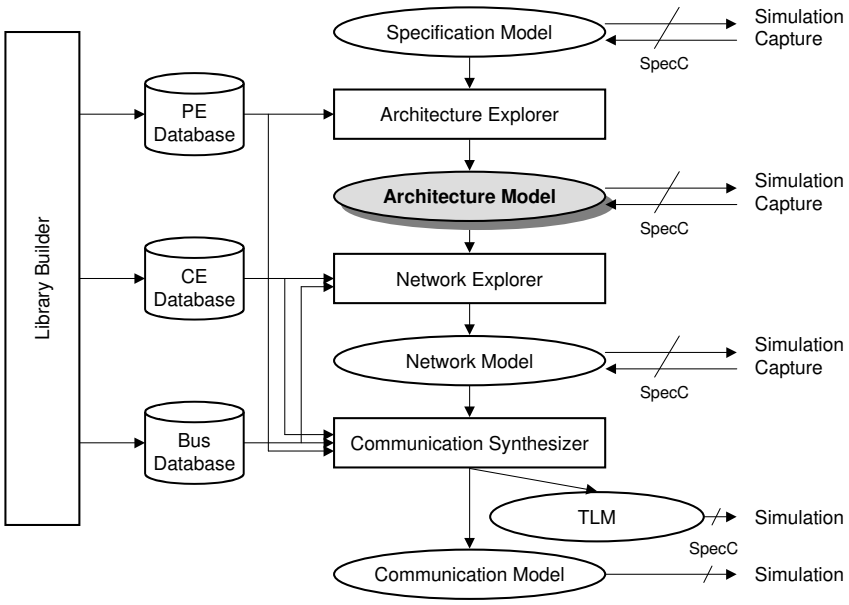


Figure 1: SoC design flow.

for the SoC tools, they must follow certain modeling rules. This report in particular defines the modeling style required for the SoC architecture model, which is highlighted in the figure. [4], [5], [7] and [6] focus on the modeling styles of the other models.

1.2 SpecC Language

The SoC design flow is supported by the SpecC system-level design language ([1]). The SpecC language as an example of a modern system-level design language (SLDL) was developed under support and control of the SpecC Technology Open Consortium (STOC) ([2]) to satisfy all the requirements for an efficient formal description of the models in the SoC design flow. It supports behavioral and structural views and contains features for describing a design at all levels of abstraction.

In the SoC design flow, all five models of the design process, starting with the specification model and down to the implementation model, are described in the SpecC language. One common language removes the need for tedious translation. Furthermore, all the models in SpecC are executable which allows for validation through simulation while reusing one single testbench throughout the whole design flow. In addition, the formal nature of the models enables application of formal methods, e.g. for verification or equivalence checking.

Note that this report is not intended to be a tutorial of SpecC language and we assume that the reader of this report is familiar with the language. This report can be used for two purposes. First, it can help users understand the meaning of the automatically generated architecture model by the architecture explorer. Second, it can help users modify the automatically generated architecture models such that they can be accepted by the network explorer.

The rest of the report is organized as follows. Section 2 presents the overall structure of an architecture model. The major elements of an architecture model are described one by one in detail. Section 3 describes the communication channels allowed in the architecture model. Section 4 describes the modeling of shared

memories in the architecture model. Section 5 describes the guidelines to model processing elements. Finally, Section 6 describes the rules on how to compose the element together to form an valid architecture model.

2 An Overview of Architecture Model

```
import "c_double_handshake";

behavior Stimulus(i_sender input) {                    // Stimuli creator
  void main(void) {
    // while(...) { ... ; input.send(...) ; ... }
  }
};

behavior Monitor(i_receiver output) {                  // Output monitor
  void main(void) {
    // while(...) { ... ; output.receive(...) ; ... }
  }
};

behavior Design(i_receiver input, i_sender output) {   // System design
  // ...

  void main(void) {
    // fsm { ... }
  }
};

behavior Main() {                                      // Top level
  c_double_handshake input, output;

  Stimulus stimulus(input);
  Design   design(input, output);
  Monitor  monitor(output);

  int main(void) {
    par {
      stimulus.main();
      design.main();
      monitor.main();
    }
  }
};
```

Figure 2: Architecture model top-level code.

Figure 2 and Figure 3 show an example template for a valid architecture model. A architecture model has to be an executable SpecC model, i.e. it has to define a `Main` behavior. An architecture model consists of a testbench that surrounds the actual design to be implemented. A testbench consists of stimulating (`Stimulus`) and monitoring (`Monitor`) behaviors that are executing concurrently with the design (`Design`) in the top-most `Main` behavior, and that drive the design under test and check the generated output against known good values.

The actual design to be implemented is modeled by the *design behaviors*, such as `Design` and those composed hierarchically inside `Design` in the figure. Design behaviors form a hierarchy tree by their com-

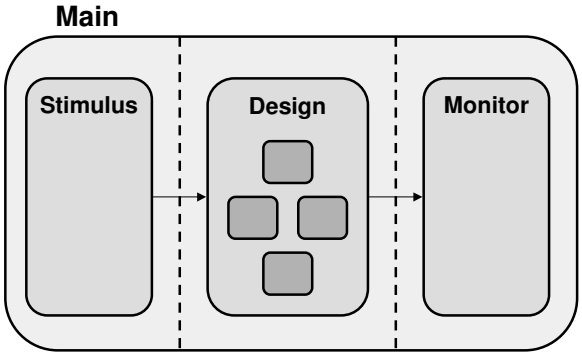


Figure 3: Architecture model top-level structure.

position relations. The root of the tree, for example `Design` in Figure 3, is called the *top-level design behavior*.

Note that the modeling rules and restrictions defined in this manual apply only to the design behaviors since the testbench behaviors will not be considered and touched by tools. Therefor the testbench can be freely described using any valid SpecC code. For example, while the code of the design to be implemented has to be available completely in SpecC source form, the testbench can link against external translation units (libraries) for additional functionality.

In general, it is hard for tools themselves to find out which behaviors are testbench behaviors and which are actual design behaviors. This distinction is to be made by the designers attaching a predefined annotation to the architecture model.

Rule 1 *A architecture model has an annotation `_SER_TOPLEVEL`, which contains the name of the top-level design behavior.*

For the example shown in Figure 3, the annotation would look like the following:

```
note _SER_TOPLEVEL = ''Design'';
```

Once the top-level design behavior is specified, the tools are able to figure out all other design behaviors.

If we zoom inside the top-level design behavior of an architecture model, we can identify a set of finer model elements which are used to capture both the computation architecture and the communication network.

- *PE behaviors* are used to model the processing elements allocated to perform the desired computation;
- *Memory behaviors* are used to model the memories allocated to store data shared by PEs;
- *System channels* are used to model the connection between the processing elements.

Rule 2 *The architecture model has an annotation `_AR_PES` attached to the top-level design behavior, which contains the names, types and other attributes of all PEs and memories that allocated for the design.*

In the following sections, we will define these model elements one by one before we describe how to put them together.

3 System Channels

In an architecture model, SpecC channels are used to abstract inter-PE communication which will be later refined during the communication synthesis. These inter-PE channels are called system channels to distinguish them from the channels that are used exclusively inside a PE.

Rule 3 *A system channel must be one of the following types defined in the SpecC Language Reference Manual:*

(a) `c_double_handshake`*, typed or untyped*

(b) `c_queue`*, typed or untyped*

(c) `c_handshake`

```
interface i_sender {
  void send(const void *d, unsigned long l);
};

interface i_receiver {
  void receive(void *d, unsigned long l);
};

interface i_tranceiver {
  void send(const void *d, unsigned long l);
  void receive(void *d, unsigned long l);
};
```

Figure 4: Interfaces implemented by typed `c_double_handshake` and `c_queue`.

The untyped `c_double_handshake` channel encapsulates un-buffered type-less data transfer. It implements three interfaces: `i_sender`, `i_receiver` and `i_tranceiver` as shown in Figure 4. The untyped `c_queue` channel encapsulates asynchronous, buffered type-less data transfer. It also implements the same interfaces as the untyped `c_double_handshake` channel.

```
interface i_send {
  void send(void);
};

interface i_receive {
  void receive(void);
};
```

Figure 5: Interfaces implemented by `c_handshake`.

The `c_handshake` channel encapsulates one-way handshake synchronization that does not need to carry real data. It implements two interfaces: `i_send` and `i_receive` as shown in Figure 5.

Since data in the application in general is typed, typed double-handshake and queue channels are allowed for system channels. The example code in Figure 6 is used to generate interfaces and double-handshake

```
#include <c_typed_double_handshake.sh>

/* create "float" type interface "i_data_tranceiver" */
DEFINE_I_TYPED_TRANCEIVER(data, float)

/* create "float" type interface "i_data_sender" */
DEFINE_I_TYPED_SENDER(data, float)

/* create "float" type interface "i_data_receiver" */
DEFINE_I_TYPED_RECEIVER(data, float)

/* create "float" type channel "i_data_double_handshake" */
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(data, float)
```

Figure 6: Example of instantiating double-handshake channel.

```
#include <c_typed_queue.sh>

/* create "float" type interface "i_data_tranceiver" */
DEFINE_I_TYPED_TRANCEIVER(data, float)

/* create "float" type interface "i_data_sender" */
DEFINE_I_TYPED_SENDER(data, float)

/* create "float" type interface "i_data_receiver" */
DEFINE_I_TYPED_RECEIVER(data, float)

/* create "float" type channel "i_data_queue" */
DEFINE_C_TYPED_QUEUE(data, float)
```

Figure 7: Example of instantiating queue channel.

channel of a `float` type. The example code in Figure 7 is used to generate interfaces and queue channel of a `float` type.

In an architecture model that is generated automatically by the architecture explorer, the system channels come from two sources. Some of them are specification channels directly coming from the specification model as a result of behavior partitioning. The others are inserted by the architecture explorer to preserve behavior execution order (`c_handshake`), to maintain coherence of distributed data (`c_double_handshake`, `c_queue`) and to handle complex channels such as `c_queue` which are allowed in the specification model.

4 Memories

In the architecture models, memory components are modeled with SpecC behaviors. A memory behavior contains the variables that are stored in it. For other PE behaviors to access the stored variables, the memory behavior implements a *memory interface* which has a set of read and write methods. Since the actual layout of variables inside a memory is not yet determined in the architecture model, the access methods are variable-specific. Basically, for each variable, a pair of read and write methods are implemented by the memory behavior. (Note that for a `struct` type variable, more methods are needed to access its memebers.) Other than implementing the memory interface, the memory behavior does not have other computational functionality.

Rule 4 *A memory behavior does not have any ports, sub-behavior instances or channel instances inside. The* `main()` *method of a memory behavior has an empty body, i.e, does nothing at all.*

Rule 5 *A memory behavior has only one variable* `mem`*, which is a C* `struct` *type that contains all variables stored in the memory.*

The types of variables stored in the memory can be any valid ANSI-C data types except pointer type. Note that SpecC bit-vector type is not allowed.

Rule 6 *A memory behavior implements a memory interface which declares a set of access methods to read-/write individual variables stored in the memory. The signature of an access method observes the following conventions:*

(a) The name of an access method always starts with either "read" or "write". To make the method names unique, a string is mangled from the variable name and appended to "read" or "write". For the members of a `struct` *variable, the hierarchical name is used for mangling.*

(b) If the method is to access an array element, an argument of `int` *type is used to pass the index. Multiple index arguments may be needed to access a nested array.*

(c) A read method has a return type of the variable and a write method always returns `void`*.*

(d) The last argument of a write method is for the written data.

In an architecture model that is produced by the architecture explorer, the memory behavior is completely generated automatically. An example code of memory behavior is shown in Figure 8. This memory contains a scalar variable *v*1 and a `struct` variable *v*2 consisting of an array of `float` variables. In the memory behavior, *v*1 and *v*2 are packed into a `struct`. Four access methods are implemented by the memory behavior to read/write *v*1 and the array elements in *v*2. The signatures of the access methods follow above rules.

```
interface i_mem1 {
  unsigned long int read_v1(void);
  void write_v1(unsigned long int);
  float read_v2_y(int);
  void write_v2_y(int, float);
};

behavior Mem1 implements i_mem1
{
    struct {
      unsigned long int v1;
      struct {
        float y[10];
      } v2;
    } mem;

    void main(void)
    {
    }

    unsigned long int read_v1(void)
    {
      return mem.v1;
    }

    void write_v1(unsigned long int data)
    {
      mem.v1 = data;
    }

    float read_v2_y(int i)
    {
      return mem.v2.y[i];
    }

    void write_v2_y(int i, float data)
    {
      mem.v2.y[i] = data;
    }
};
```

Figure 8: Example code of a memory behavior.

5 Processing Elements

In an architecture model each processing element is represented by a SpecC behavior called *PE behavior*. A PE behavior is a functional description of the computation that is to be executed by the PE. A PE behavior in general is hierarchically composed of smaller behaviors, each of which contains a piece of computation assigned to the PE. The communication inside a PE is modeled using channels and variables. Since later communication synthesis is not going to modify the PE behavior internally, a PE can be freely modeled with any SpecC constructs. For example, any type of channels or behavior compositions can be used as long as the back-end tools are able to handle them.

In the architecture model, variables shared by PEs are moved inside PEs or shared memory components. The PEs must communicate with each other through system channels (see Section 3) and shared memory components (see Section 4) that are connected (or port-mapped) to the PE behavior ports.

Rule 7 *A PE behavior has only interface ports and no variable ports. The interface types allowed are as follows:*

(a) `i_sender` *(typed or un-typed)*

(b) `i_receiver` *(typed or un-typed)*

(c) `i_tranceiver` *(typed or un-typed)*

(d) `i_send`

(e) `i_receive`

(f) memory interface (Section 4)

In the architecture models produced by the architecture explorer, PE behaviors are automatically generated by re-arranging specification behaviors and introducing new behaviors for inter-PE synchronization and data transfer. The channels and variables inside PEs are directly from the specification model.

A design may use both software and hardware PEs. Software PEs are general purpose programmable processors while hardware PEs are application specific hardware units that need to be later synthesized. One common practice for communication between a processor and a hardware device is by means of memory mapped-IO. Basically, the hardware unit uses its internal memory, for example, registers as IO ports that are mapped to the processor's memory space. As such, these registers can be accessed by the processor as if they were memories.

To model the memory-mapped IO, a memory behavior as defined in Section 4 is instantiated inside a hardware PE behavior. The memory behavior implements the same memory interface which is also implemented by the harware PE behavior. Therefore, the memory can be port-mapped to behaviors inside the hardware PE and the hardware PE can be port-mapped to other PE behaviors that need to access the registers in the hardware PE. An example of memory-mapped IO modeling is shown in Figure 9.

Rule 8 *Software PE behaviors do not implement any interfaces and hardware PE behaviors can only implement a memory interface.*

6 Top-Level Design Behavior

In previous sections, we have defined the set of model elements of an architecture model. In this section we present the following rules on connecting the elements together to form an architecture model that is a valid input to the network explorer. An example architecture model is shown in Figure 10.

```
interface i_local_mem {
  short read_x(void);
  void write_x(short);
};

behavior Local_Mem implements i_local_mem
{
  // body omitted for space consideration
};

behavior Computation(i_local_mem m)
{
  void main(void)
  {
    short data;
    data = m.read_x();
    data ++;
    m.write_x(data);
  }
};

behavior HW_with_LM implements i_local_mem
{
  Local_Mem     local_memory;
  Computation   computation(local_memory);

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

  short read_x(void)
  {
    return local_memory.read_x();
  }

  void write(short data)
  {
    local_memory.write_x(data);
  }
};
```

Figure 9: Example code of memory-mapped IO modeling.

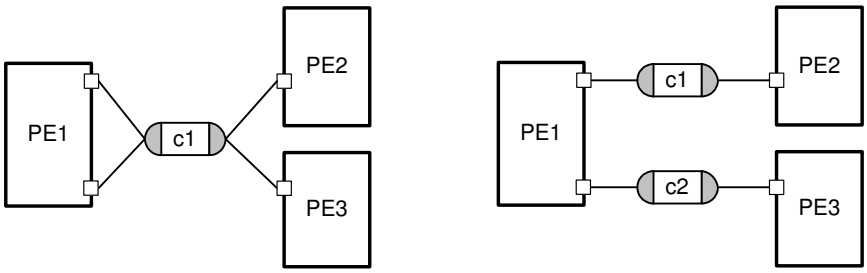
```
behavior Mem1 implements i_mem;
behavior CPU1(i_data_sender, i_mem);
behavior CPU2(i_data_receiver, i_mem);

behavior Design() {
  c_data1_double_handshake c1;
  Samsung M1;
  CPU1 PE1(c1, M1);
  CPU2 PE2(c1, M1);

  void main(void) {
    par {
      M1.main();
      PE1.main();
      PE2.main();
    }
  }
};
```

Figure 10: Example of top-level behavior code.

Rule 9 *The top-level design behavior is a hierarchical behavior that is composed of instances of PE behaviors defined in Section 5 and memory behaviors defined in Section 4. It may also have a set instances of system channels as defined in Section 3. However, it must not have any variables.*

Rule 10 *A top-level design behavior has exactly one method, the* `main()` *method, which contains exactly one statement that is a* `par` *statement.*

According to the definition of a hierarchical behavior, each sub-behavior instance inside the top-level behavior can be called at most once in the `par` statement. The top-level behavior of an architecture model should not have any variables and inter-PE communication must go through the system channels which include `c_double_handshake`, `c_queue` and `c_handshake`.



a) Invalid channel usage b) Valid channel usage

Figure 11: Point-to-point channel connection.

In the architecture model, the system channels are used for pair-wise inter-PE communication. As illustrated in Figure 11, the usage of a system channel on the left side is not allowed because the channel is used by three PEs. The correct usage on the right side introduces additional channel instance to ensure that a channel is used by only two PEs.

Rule 11 *Inside the top-level behavior, a system channel instance can not be connected to more than **two** PE behaviors.*

Furthermore, a system channel is used solely for inter-PE communication and not for intra-PE communication, i.e., a PE sends a data which is received by itself.

Rule 12 *In the architecture model, intra-PE communication over a system channel is not allowed.*

A `c_handshake` channel implements `i_send` and `i_receive` interface which can either send or receive, but not both. Therefore if for each PE only one port is mapped to the same `c_handshake` channel instance, we are sure that the channel instance is only for inter-PE communication.

However, the `c_double_handshake` and `c_queue` channels implement a `i_tranceiver` interface, which allows both sending and receiving from the same PE even if only one port-mapping is permitted for a PE. Therefore, it is users discretion to obey the rule while writing the model. However, SER tools will issue a warning message if either of following situation occurs:

(a) A PE has a `i_tranceiver` port mapped to a `c_double_handshake` or `c_queue` channel instance;

(b) A PE has both `i_sender` port and `i_receiver` port mapped to the same `c_double_handshake` or `c_queue` channel instance.

The architecture models generated by the architecture explorer are guaranteed that no system channels are used for intra-PE communication. If a designer is going to modify the generated models, he must not alter this property. Otherwise, the model would become invalid for later communication synthesis.

Note that the top-level design behavior may have a set of ports to communicate with the testbench behaviors. The top-level design behavior keeps the same set of ports if the architecture model is derived from a specification model. The same interface of the top-level design behavior enables re-use of testbench behaviors for simulation of the architecture model without any modification.

References

[1] R. Dömer, A. Gerstlauer and D. D. Gajski. *SpecC Language Reference Manual, Version 2.0*, SpecC Technology Open Consortium (STOC), Japan, December 2002.

[2] SpecC Technology Open Consortium. http://www.specc.org.

[3] SpecC Compiler V2.2.0, Center for Embedded Computer Systems, University of California, Irvine, June 2004.

[4] A. Gerstlauer, K. Ramineni, R. Dömer, D. Gajski. *System-on-Chip Specification Style Guide*, Technical Report CECS-TR-03-21, Center for Embedded Computer Systems, University of California, Irvine, June 2003.

[5] D. Shin, J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Network Modeling Style Guide*, Technical Report CECS-TR-04-23, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[6] D. Shin, L. Cai, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Transaction-Level Modeling Style Guide*, Technical Report CECS-TR-04-24, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[7] D. Shin, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Communication Modeling Style Guide*, Technical Report CECS-TR-04-25, Center for Embedded Computer Systems, University of California, Irvine, July 2004.