UC Irvine
ICS Technical Reports

Title
SpecC modeling guidelines

Permalink
https://escholarship.org/uc/item/03t1r0k7

Author
Gerstlauer, Andreas

Publication Date
2001-08-05

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

SpecC Modeling Guidelines

Andreas Gerstlauer

Technical Report ICS-00-48
August 5, 2001

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

gerstl@cecs.uci.edu
http://www.cecs.uci.edu/~gerstl

Information and Computer Science
University of California, Irvine

SpecC Modeling Guidelines

Andreas Gerstlauer

Technical Report ICS-00-48
August 5, 2001

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

gerstl@cecs.uci.edu
http://www.cecs.uci.edu/~gerstl

Abstract

Raising the level of abstraction to the system level has been touted as the main solution for closing the productivity gap designers of embedded systems-on-chip (SOCs) are facing increasingly. However, in order to achieve the required productivity gains, a well-defined methodology enabling a synthesis-oriented flow is necessary. The basis for every methodology are clear and unambiguous models at different levels of abstraction.

In this report, we will define the four models that comprise the SpecC system-level design methodology. Using actual code templates, we will show their features and properties in detail. All together, this report provides comprehensive guidelines for modeling a design at each level. In addition to standardizing manually written models, the exact definition of the models builds the basis of all automated tools for exploration, refinement, synthesis or verification.

RECEIVED
APR 15 2002
UCI LIBRARY

Contents

1 Introduction **1**
- 1.1 System Design Flow . . . 2
- 1.2 SpecC Methodology . . . 3
- 1.3 SpecC Language . . . 4

2 Specification Model **4**
- 2.1 Specification Model Example . . . 4
- 2.2 Concurrency . . . 5
- 2.3 Communication . . . 6
- 2.4 Summary . . . 7

3 Architecture Model **9**
- 3.1 Architecture Model Example . . . 9
- 3.2 Storage . . . 11
  - 3.2.1 Local Memory . . . 11
  - 3.2.2 Global Memory . . . 11
- 3.3 Synchronization . . . 13
- 3.4 IP Components . . . 14
- 3.5 Scheduling . . . 16
- 3.6 Time . . . 17
- 3.7 Summary . . . 17

4 Communication Model **18**
- 4.1 Communication Model Example . . . 18
  - 4.1.1 Bus Wires . . . 20
  - 4.1.2 Bus Adapters . . . 20
- 4.2 Protocol Layer . . . 21
- 4.3 Application Layer . . . 23
  - 4.3.1 Synchronization . . . 23
  - 4.3.2 Addressing . . . 25
  - 4.3.3 Data slicing . . . 25
- 4.4 Transducers . . . 25
- 4.5 Arbitration . . . 27
- 4.6 Timing . . . 29
- 4.7 Summary . . . 30

5 Implementation Model **30**
- 5.1 Behavioral RTL . . . 31
  - 5.1.1 Custom Hardware . . . 31
  - 5.1.2 Programmable Processors . . . 33
- 5.2 Structural RTL . . . 34
  - 5.2.1 Clock . . . 35
  - 5.2.2 Controller . . . 35
  - 5.2.3 Datapath . . . 36
  - 5.2.4 Bus Interface . . . 37
- 5.3 Summary . . . 39

6 Summary and Conclusions **39**

References **39**

List of Figures

1 Y-Chart. 1
2 SpecC methodology. 3
3 SpecC models in the Y-Chart. 4
4 Specification model. 5
5 Specification model with explicit dependencies. 6
6 Specification model with message-passing communication. 7
7 Architecture model. 10
8 Shared memory architecture model. 12
9 Architecture model with multiple inter-component behavior transitions. 14
10 Architecture model with IP. 16
11 Communication model. 18
12 PE bus adapters. 21
13 DSP56600 protocol timing diagram. 21
14 Application layer synchronization protocol. 23
15 Communication model with IP. 25
16 Communication model with arbiter. 29
17 Implementation model. 30
18 Custom hardware bus interface FSMD. 32
19 Structural RTL model for custom hardware. 35

List of Listings

1 Specification model. 5
2 Specification model with explicit dependencies. 6
3 Message-passing channel. 7
4 Specification model with message-passing communication. 7
5 Architecture model. 10
6 Shared memory architecture model. 12
7 Global memory component. 12
8 Shared memory accesses in leaf behaviors. 13
9 IP component model. 15
10 Architecture model with IP. 16
11 IP accesses in leaf behavior *B3*. 16
12 Behavior timing. 17
13 Communication model. 19
14 Signal channel for modeling of wires. 20
15 PE bus adapter interface. 21
16 Bus adapter protocol layer. 22
17 Bus adapter application layer. 24
18 Communication model with IP. 26
19 Transducer component model. 27
20 Communication model with arbiter. 28
21 Bus adapter with arbitration. 28
22 Arbiter component model. 29
23 Implementation model. 30
24 Custom hardware behavioral RTL model. 31
25 Custom hardware bus interface FSMD. 32
26 DSP instruction set simulator (ISS) model. 34
27 Structural RTL model for custom hardware. 35
28 Clock generator. 35
29 Custom hardware controller. 36
30 State register. 36
31 Output logic. 36
32 Next state logic. 36
33 Custom hardware datapath. 37
34 Bus interface hardware unit. 38
35 Bus interface controller. 38

SpecC Modeling Guidelines

A. Gerstlauer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

Raising the level of abstraction to the system level has been touted as the main solution for closing the productivity gap designers of embedded systems-on-chip (SOCs) are facing increasingly. However, in order to achieve the required productivity gains, a well-defined methodology enabling a synthesis-oriented flow is necessary. The basis for every methodology are clear and unambiguous models at different levels of abstraction.

In this report, we will define the four models that comprise the SpecC system-level design methodology. Using actual code templates, we will show their features and properties in detail. All together, this report provides comprehensive guidelines for modeling a design at each level. In addition to standardizing manually written models, the exact definition of the models builds the basis of all automated tools for exploration, refinement, synthesis or verification.

1 Introduction

The design of embedded computer systems is the process of implementing a given specification of the desired system on a chip in silicon. Following a formal methodology, defined as a set of models and a set of transformation between the models, the design is gradually refined to lower and lower levels of abstraction.

As depicted by the Y-Chart (Figure 1), four general layers of abstraction are commonly distinguished [1]:

(a) System level

(b) Register-transfer level (RTL)

(c) Gate level

(d) Transistor level

With lower levels, the design process focuses on more and more detailed aspects of the system. At each level, the designer works with a specific set of objects. Objects at higher levels of abstraction are hierarchically composed

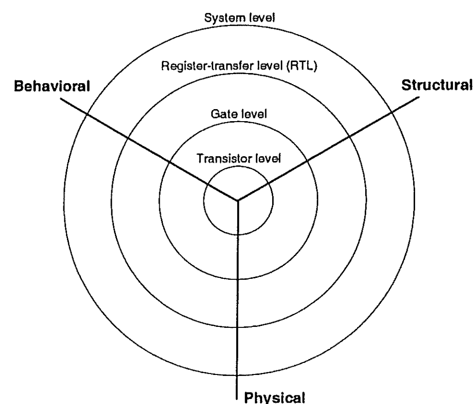


Figure 1: Y-Chart.

of lower-level design objects. For example, at the system level, components of the system architecture are processing elements (PEs) and system busses. At the register-transfer level, in turn, the microarchitecture of PEs is build out of functional units, registers, and so on.

At each layer, the design object at that level can be described or modeled in three different views:

(a) A *behavioral view* describes the functionality of the design in terms of abstract concepts, independent of any implementation details.

Building blocks of a behavioral description are abstract entities that do not represent physical components. Each block describes a piece of functionality that takes inputs, processes them and finishes after producing its output. In a behavioral view, such blocks are then arranged hierarchically to model the control and data dependencies between them.

Parallelism in a behavioral description does not imply true concurrency in hardware. Again, behavioral blocks are abstract representations of algorithms that are free of implementation assumptions.

(b) A *structural view* describes the design as a netlist of lower-level components and their connectivity.

Building blocks of a structural description represent real, physical objects that are connected via wires. As such, each of the blocks is active all the time, constantly processing data. In a structural view, the system is then modeled as a set of non-terminating, concurrent processes representing the way the system is composed out of tangible lower-level components. Dependencies have to be modeled as part of the processes' functionality by inserting synchronization as needed.

Since the processes of a structural description represent real hardware, the parallel composition of the processes reflects the true concurrency available among the set of physical components on the chip or the board.

(c) A *physical view* describes the spatial layout of the lower-level components on the chip. A physical view describes the floorplan of how the components and their interconnect are placed and routed on the chip.

Points in the Y-Chart form specific levels of abstracting a design. In addition to the amount of structure as shown by the layers and views of the Y-Chart, models of the design at certain abstraction levels are defined by the amount of order in the model.

In general, given two events e_1 and e_2, where an event e_i is a tuple (a_i, t_i) of action a_i occurring at time t_i, e_1 and e_2 are ordered iff it can be determined that $t_1 < t_2$ or $t_2 < t_1$. A system is totally ordered if all pairs of events are ordered as is the case with real time on the chip, for example. A system is partially ordered if only subsets of all events are ordered. For example, at higher levels, a relationship between independent parts is not specified. An abstraction level employs a model of time to specify order. Real time is abstracted as discrete logical time. Two unordered events are modeled to occur at the same logical time, leaving the freedom of implementing them in any oder in real time.

1.1 System Design Flow

Design is the process of moving from a behavioral to a structural (and eventually physical) description at a certain level, implementing the desired functionality through an architecture of subcomponents. The subcomponents, in turn, are designed by moving from a behavioral description to a structural (and physical) description of the subcomponent at the next lower level of abstraction. For example, at the system level, the designer will create a system architecture consisting of a set of processing elements (PEs) connected through system busses that implements the desired system functionality. The processing element's functionality, in turn, is implemented by designing a microarchitecture of functional units, registers files, and so on for the PE at the register-transfer level.

A design flow can be bottom-up or top-down. In a bottom-up approach, design moves from the lowest level of abstraction up to the system level by assembling previously designed components such that the desired behavior is achieved at each level. In a top-down approach, design starts with a specification of the system behavior and moves down in the level of abstraction by mapping the desired behavior at each level onto a set of components and specifying the behavior of each component for the next level.

In order to automate the design process with CAD tools, the models and transformations of the design methodology must be formalized. Languages with special support to describe different views of the design at different levels of abstraction in a formal and efficient manner are needed. In addition to the application of formal methods for verification, an executable language allows validation through simulation of the models.

Once the models for the different design views at different abstraction levels are formally defined, CAD tools can automate parts of the design process. Specifically, the formalized process of deriving a structural description from a behavior description of the desired functionality is called *synthesis*. The synthesis processes at the highest levels of abstraction are:

(a) System synthesis
Given a specification of the system behavior, synthesize a system architecture consisting of processing elements and system busses that implements the desired functionality.

(b) High-level/behavioral synthesis
Given a behavioral description of a PE, synthesize a microarchitecture implementation out of RTL components like functional units, register files, and so on.

(c) Logic synthesis
Given a description of the functionality of an RTL component, synthesize a gate netlist that implements the combinatorial/sequential logic for the component.

In this report, we formalize the different models of the SpecC system-level design methodology, representing different views of the design at different levels of abstraction. The SpecC methodology covers system and register-transfer levels of abstraction. Formalizing the models of the methodology forms the basis for developing the corresponding system-level and high-level synthesis tools.

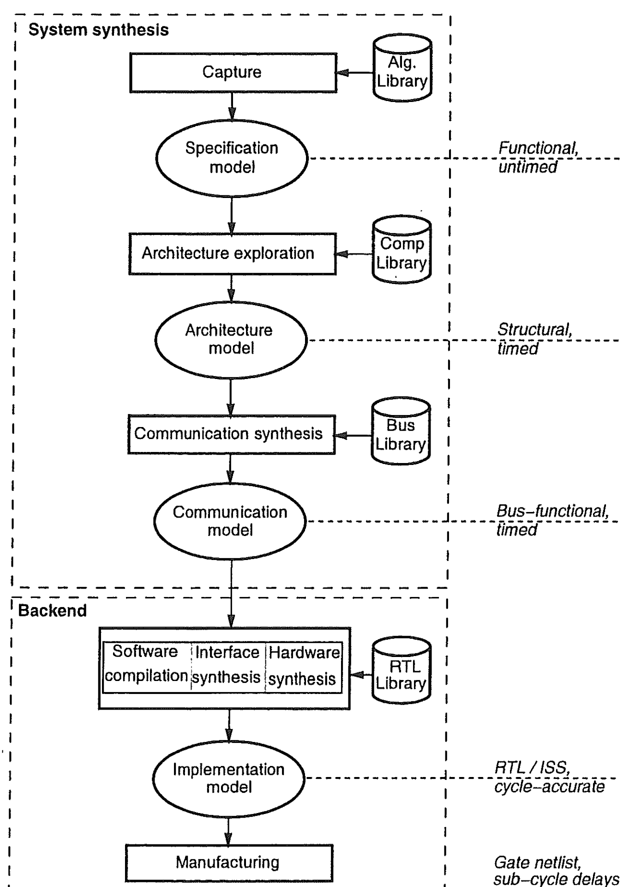


Figure 2: SpecC methodology.

1.2 SpecC Methodology

The SpecC system-level design methodology is shown in Figure 2. The SpecC methodology is a set of four models and three transformation steps that take a system specification down to an RTL implementation [2].

The SpecC design flow consists of two main parts: (a) system synthesis, and (b) a backend for high-level synthesis and compilation. In the SpecC methodology, system synthesis is further subdivided into two orthogonal tasks, architecture exploration and communication synthesis. Architecture exploration implements the computation behavior of the specification on a set of processing elements that form the system architecture. Communication synthesis, on the other hand, implements the communication functionality of the specification over the system busses.

Each system synthesis and backend task refines the model of the design at the current stage of the design process into a new model representing the details of the implementation added during the synthesis step. At the output of each task, the model of the design reflects the implementation decisions made in the previous step. At the same time, each model forms the input to the next task.

The system-level design process starts off with a specification of the desired system behavior. This specification model is written by the user and forms the input to the design process. It is purely functional and free of any implementation details. There is no notion of time yet and only a purely causal ordering of events, i.e. events in the system are limited to synchronization events only which are needed to ensure causality.

In the SpecC methodology, the first task of system synthesis is architecture exploration. Architecture exploration selects a set of processing elements and maps the computation behavior of the specification onto the PEs. Architecture exploration refines the specification model into the intermediate architecture model. The architecture model describes the PE structure of the system architecture and the mapping of computation behaviors onto the PEs, including estimated execution times for the behavior of each PE.

Architecture exploration is followed by communication synthesis to complete the system synthesis process. Communication synthesis selects a set of system busses and protocols, and maps the communication functionality of the specification onto the system busses. Communication synthesis creates the communication model which reflects the bus architecture of the system and the mapping of communication onto the busses.

The communication model is the result of the system synthesis process. It describes the structure of the system architecture consisting of PEs and busses, and the implementation of the system functionality on this architecture. It is timed in both computation and communication, i.e. simulation detail is increased by events for estimated execution and communication delays.

The communication model is a structural view at the system level. At the same time, the specification of the functionality of each PE of the system in the form of a behavioral view at the register-transfer level forms the input to the RTL synthesis of those components in the backend. In a hierarchical fashion, each PE is synthesized separately in the backend and the behavioral view of the PE is replaced with a structural view of its RTL or instruction-set (IS) microarchitecture. The result of this backend process is the implementation model.

The implementation model is a cycle-accurate, structural description of the RTL/IS architecture of the whole system. In a hierarchical fashion, the implementation model describes the system structure and the RTL structure of each PE in the system. Simulation detail is increased down to the clock level, i.e. the timing resolution is in terms of clock events for each local PE clock.

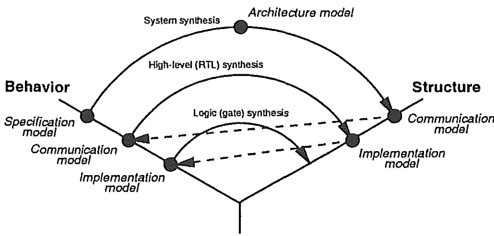


Figure 3: SpecC models in the Y-Chart.

Figure 3 summarizes the four models of the SpecC methodology by their position in the views and abstraction layers of the Y-Chart.

1.3 SpecC Language

The SpecC methodology is supported by the SpecC system-level design language [2]. The SpecC language was developed to satisfy all the requirements for an efficient formal description of the models in the SpecC methodology. It supports behavioral and structural views and contains features for describing a design at all levels of abstraction.

In general, at all levels of abstraction, behavioral and structural views of a SpecC behavior at any point in the code hierarchy are defined as follows:

(a) A behavioral view is modeled as a serial-parallel composition of behaviors. Behaviors terminate after they are finished processing the current input data set and producing corresponding outputs. Behaviors are then arranged hierarchically to explicitly model data and control flow between blocks, describing the desired functionality.

(b) A structural view is defined as a set of non-terminating, communicating, and concurrent behaviors representing the tangible components of the architecture. In SpecC, a structural description is a parallel decomposition of a behavior into subbehaviors that each execute in endless loops and communicate through ports and variables, events, or channels.

Starting with the system behavior description at the top level, behavioral views are replaced with structural views as design progresses down to lower levels.

In the SpecC methodology, all four models of the design process starting with the specification model and down to the implementation model are written and 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, 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.

The purpose of this report is to define the four different models of the SpecC methodology within the framework of the SpecC language and to define how each model is described in SpecC. Based on code templates and examples, we will give guidelines for modeling implementation details available at each level of abstraction and in each design view.

The rest of this report is organized as follows: the report starts with a description of the specification model in Section 2. Section 3 and Section 4 detail the architecture and communication models, respectively. Finally, Section 5 introduces the major aspects of the implementation model. The report then concludes with a summary in Section 6.

2 Specification Model

The specification model is the input of architecture exploration. It is written by the user to specify the desired system functionality. The specification is a behavioral view of the system, i.e. it describes the desired functionality in an abstract manner. The specification model is a purely functional model, free of any implementation details. For example, objects at the specification level are abstract entities that do not correspond to real components.

In general, the specification is hierarchically composed of behaviors. Behaviors are arranged sequentially, concurrently, or in a mix of both, i.e. in a pipelined fashion. Behaviors at the leaves of the hierarchy contain basic algorithms that perform arithmetic and logical operations on data. In addition to temporary data, leaf behaviors will encapsulate any permanent storage required by the algorithm.

The ordering of events in the system is based on causal relationships only and there is no notion of time. The system is partially ordered based on causality as determined by the dependencies between behaviors. Simulation detail is limited to events used for synchronization to ensure causality.

2.1 Specification Model Example

Figure 4 shows an example of a simple yet typical specification model. The corresponding SpecC code is shown in Listing 1. The design is a hierarchical, serial-parallel composition of behaviors. In the example, behavior *B1* is followed by the parallel composition of behaviors *B2* and *B3*. The three leaf behaviors *B1*, *B2*, and *B3* contain algorithms in the form of C code.

```
   // leaf behavior 1
   behavior B1( out type1 v1 )
   {
     void main(void) {
 5      ...
        v1 = ...
     }
   };

10 // leaf behavior 2
   behavior B2( in  type1 v1,
                out type2 v2,
                out event e1 )
   {
15   void main(void) {
        ...
        v2 = f2( v1, ... );
        notify( e1 );
        ...
20   }
   };

   // leaf behavior 3
   behavior B3( in  type1 v1,
25              in  type2 v2,
                in  event e1 )
   {
     void main(void) {
        ...
30      wait( e1 );
        f3( v1, v2, ... );
        ...
     }
   };
35
   // B2 || B3
   behavior B2B3( in type1 v1 )
   {
     type2 v2;
40   event e1;

     B2 b2( v1, v2, e1 );
     B3 b3( v1, v2, e1 );

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

50 // Top-level
   behavior Design()
   {
     type1 v1;

55   B1   b1  ( v1 );
     B2B3 b2b3( v1 );

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

Listing 1: Specification model.

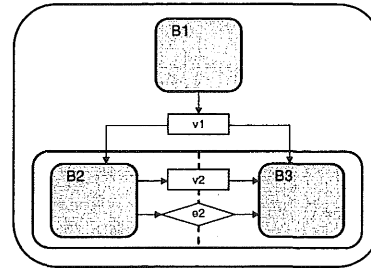


Figure 4: Specification model.

Behaviors communicate through variables attached to their ports. Synchronization of concurrent behaviors is handled through events connected to behavior ports. In this case, behavior *B1* produces the variable *v1* of type *type1* at its output The variable is passed into *B2* and *B3* by connecting *v1* to the corresponding inputs of those two behaviors. The concurrent behaviors *B2* and *B3* communicate through the variable *v2* (of type *type2*) and the event *e1*. Behavior *B2* writes to *v2* and notifies *B3* when the data is ready. Behavior *B3* in turn waits for event notification before reading from variable *v2*.

2.2 Concurrency

In general, concurrent behaviors in the specification model should reflect the available parallelism in the specification. Therefore, they should be as independent as possible. Data or control dependencies between behaviors at the specification level should be explicitly captured through the behavior hierarchy. Instead of concurrent behaviors that communicate or synchronize through variables or events, the behaviors should be split into independent parts that can run in parallel and dependent parts that have to be executed sequentially.

Figure 5 and Listing 2 show the specification model example after splitting the concurrent behaviors *B2* and *B3* to explicitly model the data dependency through the serial-parallel behavior hierarchy. Instead of synchronization via the event *e1*, the dependency on variable *v2* is represented by executing the corresponding parts of the behaviors sequentially.

Note, however, that the modified example introduces an artificial dependency between behaviors *B3_1* and *B2_2*. Depending on the actual implementation, this dependency might result in an unnecessary delay before the execution of behavior *B2_2*. Therefore, the tradeoff between implicit versus explicit parallelism and dependencies will determine whether to cut or combine concurrent threads.

```
    // leaf behavior 2, two parts
    behavior B2_1( in   type1 v1,
                   out  type2 v2 )
    {
 5    void main(void) {
        ...
        v2 = f2( v1, ... );
      }
    };
10  behavior B2_2( in type2 v2 )
    {
      void main(void) {
        ...
      }
15  };

    // leaf behavior 3, two parts
    behavior B3_1( in   type1 v1 )
    {
20    void main(void) {
        ...
      }
    };
    behavior B3_2( in   type1 v1,
25                 in   type2 v2 )
    {
      void main(void) {
        f3( v1, v2, ... );
        ...
30    }
    };

    // B2 || B3, two parts
    behavior B2B3_1( in   type1 v1,
35                   out  type2 v2 )
    {
      B2_1 b2( v1, v2 );
      B3_1 b3( v1 );

40    void main(void) {
        par { b2.main();   b3.main(); }
      }
    };
    behavior B2B3_2( in   type1 v1,
45                   in   type2 v2 )
    {
      B2_2 b2( v2 );
      B3_2 b3( v1, v2 );

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

55  // Top-level
    behavior Design()
    {
      type1 v1;
      type2 v2;
60
      B1     b1   ( v1 );
      B2B3_1 b23_1( v1, v2 );
      B2B3_2 b23_2( v1, v2 );

65    void main(void) {
        b1.main();
        b23_1.main();
        b23_2.main();
      }
70  };
```

Listing 2: Specification model with explicit dependencies.

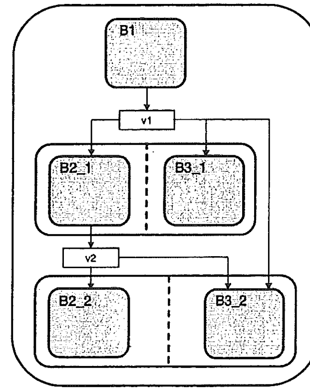


Figure 5: Specification model with explicit dependencies.

2.3 Communication

If the relationship of concurrent behaviors in the specification model extends beyond synchronization through pure events and necessitates some actual form of data communication, the specification needs to clearly separate such communication from the normal computation by encapsulating communication functionality in the form of channels.

In general, behaviors at the specification level communicate via message-passing channels. Behaviors exchange data by sending and receiving messages over communication channels with appropriate semantics. In the case of a sequential composition, message-passing degenerates to simple variables. Data is exchanged by reading and writing from/to the variable. In the case of a parallel composition with simple synchronization only, the synchronization is implemented via a single event. In the general case of data communication between concurrent behaviors, however, a message-passing channel is instantiated.

The specification model instantiates channels out of a SpecC channel library. The library contains channels with abstract communication semantics like buffered and unbuffered message-passing, FIFOs, shared-memory semaphores/mutexes, and so on. By using the predefined channels out of the library, commonly needed communication functionality is available for integration into the specification model.

The simulation model for a channel with blocking message-passing semantics for messages of arbitrary type is shown in Listing 3. Both, the *send()* and *recv()* methods block the sender and receiver until the other end acknowledges receipt or signals readiness to complete the data communication. The double-handshake protocol inside the channel effectively implements the rendevouz-style semantics of blocking message-passing.

Note that the simulation model of the channel does not

```
interface ISend {
  void send( void *data , int size );
};
interface IRecv {
5 void recv( void *data , int size );
};

channel ChMP() implements ISend , IRecv
{
10 void   *buf = 0;           // temporary buffer
   event  eReady , eAck;      // handshake events

   // blocking send
   void send( void *data , int size ) {
15   // copy data to temp. buffer
     buf = malloc( size );
     memcpy( buf, data , size );
     // notify receiver
     notifyone( eReady );
20   // wait for acknowledge
     wait( eAck );
   }
   // blocking receive
   type2 recv( void *data , int size ) {
25   // wait for data
     while ( ! buf )   wait( eReady );
     // read data from temp. buffer
     memcpy( data , buf, size );
     free ( buf );
30   // acknowledge receipt
     buf = 0;
     notify ( eAck );
   }
};
```

Listing 3: Message-passing channel.

imply any specific implementation of the message-passing semantics. The code inside the channel is for simulation of the correct semantics during execution only. It is the task of communication synthesis to refine those abstract channels into an actual implementation of the desired semantics using the available system bus protocols and PE interfaces.

An example of the specification model which uses an abstract message-passing channel for communication between the concurrent behaviors *B2* and *B3* is shown in Figure 6 and Listing 4. The global variable *v2* and event *e1* are replaced with a message-passing channel *C2* that connects the two concurrent behaviors *B2* and *B3* via the channel's sender and receiver interfaces *ISend* and *IRecv*.

Inside the concurrent leaf behaviors *B2* and *B3*, the algorithms operate on local copies of the variable *v2*. Whenever the copies of *v2* need to be updated, they are transfered between the behaviors by calling the *send()* and *recv()* methods of the channel.

2.4 Summary

The purpose of the specification model is to clearly and unambiguously described the system functionality. The specification model is free of any implementation issues. It is

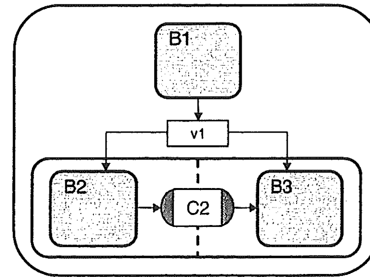


Figure 6: Specification model with message-passing communication.

```
   // leaf behavior 2
   behavior B2( in type1 v1, ISend c2 ) {
     void main(void) {
       type2 v2;
5      ...
       v2 = f2( v1, ... );
       ...
       // send message
       c2.send( &v2, sizeof(v2) );
10     ...
     }
   };

   // leaf behavior 3
15 behavior B3( in type1 v1, IRecv c2 )
   {
     void main(void) {
       type2 v2;
       ...
20     // receive message
       c2.recv( &v2, sizeof(v2) );
       f3( v1, v2, ... );
       ...
     }
25 };

   // B2 || B3
   behavior B2B3( in type1 v1 ) {
     ChMP c2;      // message-passing channel
30
     B2 b2( v1, c2 );
     B3 b3( v1, c2 );

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

   // Top-level
40 behavior Design() {
     type1  v1;

     B1    b1   ( v1 );
     B2B3  b2b3( v1 );
45
     void main(void) {
       b1.main();
       b2b3.main();
     }
50 };
```

Listing 4: Specification model with message-passing communication.

a purely behavioral model specifying the desired functionality of the system. Any hierarchical, serial-parallel composition of behaviors is allowed without implying anything about the structure of the system architecture.

Through the specification model, the user defines the basis for synthesis and exploration. Therefore, the quality of the specification model is critical. Synthesis results can always only be as good as the input description. General guidelines for the specification model are:

Hierarchy At each level of hierarchy, the system should be composed of self-contained blocks with well-defined interfaces enabling easy composition, rearrangement, and reuse. Closely related functionality is grouped through hierarchy. Higher-level behaviors encapsulate tightly coupled groups of subbehaviors such that the ratio of external to internal communication is minimized. On the other hand, the number of subbehaviors per parent should be kept small and manageable. As a guideline, behaviors typically have 2-5 children on average.

At each level, the behavior hierarchy should be clean. Different behavioral concepts shouldn't be mixed in the same level. A behavior is either a hierarchical composition of subbehaviors or a leaf behavior with sequential code. Similarly, a hierarchical behavior is either a sequential, parallel, pipelined or FSM composition of subbehaviors but does not contain arbitrary C code.

Granularity Behaviors at the leaves of the hierarchy define the granularity for exploration. Leaf behaviors contain basic algorithms in the form of C code, reading from their inputs, processing a data set, and producing outputs. An algorithm is a sequence of computational steps that transform the input into the output [3]. Leaf code is split into behaviors along the boundaries defined between reading and writing of data structures. On the other hand, all the code needed to process a complete, consistent data set should be kept together in one leaf behavior.

Also, similar to higher levels of hierarchy, the ratio of communication to computation should be minimized yet the size of the leaf behaviors be kept small and manageable with well-defined, sensible interfaces and possible reuse in mind. As a rule of thumb, what would be a traditional C function will become a leaf behavior with typically half a page to maximally two pages of code.

Communication Computation and communication in the specification model are separated into behaviors and channels, respectively, allowing for a separate implementation of both concepts. Data dependencies should be reflected explicitly in the behavioral hierarchy as transitions between behaviors, either through a sequential composition or conditionally using the `fsm` statement. In this case, channels degenerate to simple variables connecting behaviors, and the need for implicit synchronization through message-passing is eliminated.

All dependencies are explicitly captured through the connectivity between behaviors and no hidden side effects exist. Global variables should be avoided completely. Static variables accessed from a single leaf behavior become member variables of that behavior. Global variables used for communication have to be turned into explicit dependencies in the form of connectivity as behaviors are only allowed to exchange data through their ports.

Encapsulation In general, information should be localized as much as possible. This includes code (functions, methods), storage (variables), and communication (port variables, channels). Each hierarchical unit (behavior) encapsulates and abstracts as many local details as possible, hiding them from the higher levels. Hierarchical behaviors encapsulate dependencies and communication of a group of subbehaviors, providing only an interface to their combined functionality.

At the leaves, behaviors encapsulates all the code and storage needed by the algorithm. As mentioned above, global, static variables become member variables of the leaf behavior. Furthermore, global functions that are called out of leaf behaviors should be avoided. Instead, depending on size and number of callers, consider converting functions into separate leaf behaviors that get instantiated as subbehaviors of the caller, or move global functions into the calling behavior where they become local methods. An exception are small helper functions with a few lines of code that are used ubiquitously and can be considered basic operations (on the same level as additions or multiplications).

Parallelism Any concurrency available between independent behaviors should be exposed through their parallel or pipelined composition. That is, all behaviors that do not have any control or data dependencies (or data dependencies only across iterations) should be arranged to execute in a concurrent fashion. Furthermore, the behavior hierarchy should be constructed in such a way as to maximize the number of independent behaviors and hence the available parallelism.

Dependent behaviors, on the other hand, should generally not be arranged in a concurrent fashion. Instead, their dependencies should be captured explicitly through transitions as explained above and in Section 2.2. An exception are rare (control) dependencies between otherwise highly independent top-level tasks, for example. In those cases, communication and synchronization are modeled using channels between the tasks.

Time The specification model is untimed and all behaviors execute in zero logical time. The only events in the system are events for synchronization in order to specify causality. Synchronization events establish a partial order among concurrent threads of behaviors.

In summary, the specification model hierarchically groups closely related functionality, defines the granularity of the exploration units (behaviors), exposes the available behavior-level parallelism, clearly separates computation from communication, and identifies dependencies through system states, events and transitions.

3 Architecture Model

The architecture model is the intermediate model after architecture exploration. Architecture exploration maps the computational parts of the system specification represented by the SpecC behaviors onto processing elements (PEs) of a system architecture. The architecture model represents this mapping, thus exposing the communication between the components to be implemented by the following communication synthesis task.

The architecture model reflects the PE structure of the synthesized system architecture. Therefore, it represents a structural view of the design at the system level. At the top level of the architecture model, the system is described as a parallel composition of non-terminating, concurrent behaviors representing the PEs of the architecture.

Communication in the architecture model, on the other hand, remains at an abstract message-passing level. Communication between behaviors mapped to different PEs becomes system-global communication. Corresponding message-passing channels are instantiated between PE behaviors at the top level, and behaviors inside the PEs are connected to the channels through the PE's ports.

PEs with fixed, pre-defined external communication semantics are modeled as behaviors that directly provide communication channel functionality at their interfaces. A behavior's channel interface abstracts the PE's internal communication implementation and provides a canonical access for communication with the PE at the message-passing level. Examples are IPs or memories that are not capable of implementing arbitrary communication.

The PE behaviors of the architecture model represent a behavioral view of the PEs. The functionality of each PE is described by grouping the behaviors of the original specification under the PE behaviors according to the selected system partitioning. The original hierarchy is preserved and communication and synchronization behaviors are inserted to preserve the original semantics.

In addition to computation, a PE in general provides system-level storage capabilities. The union of variables inside the behaviors executing in a PE represents the local memory of the PE. A special case of PEs are dedicated system memories which are not capable of executing functionality and only provide variable storage.

All parallelism in the architecture model is captured through the structure of concurrent PEs. Internally, PEs allow a single thread of control only. Behaviorals mapped onto a PE are scheduled to serialize their execution. Static or dynamic scheduling results in a total order among the behaviors inside each PE. Dynamic scheduling emulates parallelism through multitasking yet its time-shared nature allows for only one active behavior at any given time. True parallelism is only available at the PE level with all PEs being constantly active.

The architecture model introduces the notion of time for the computation mapped onto the PEs. Based on estimated execution times on the target PE, behaviors are annotated with timing information. Apart from the total order created by scheduling behaviors inside PEs, execution delays refine the partial order among PEs. Depending on the granularity of the timing information, actions are further ordered in time beyond the pure causality of the specification.

3.1 Architecture Model Example

Figure 7 and Listing 5 show the architecture model for the example design from Section 2.1 (Figure 4 and Listing 1) after mapping the specification onto a system architecture with two components, *PE1* and *PE2*. Behaviors *b1* and *b2* are mapped onto *PE1*, while *b3* is mapped onto *PE2*.

Inside the two PE behaviors, the parts of the original behavior hierarchy that are mapped to the corresponding component are instantiated. In addition, pairs of behaviors, *B13Snd* / *B13Rcv* and *B34Snd* / *B34Rcv* (Listing 5(a)), are inserted into the hierarchy to transfer control and data from *PE1* to *PE2* in oder to preserve the execution semantics of the original specification. The behavior pairs communicate over two system-global message-passing channels, *CB13* and *CB34*, that are inserted between the PEs.

In this example, communication between behaviors mapped to different PEs is transformed into an implemen-

```
// Send data from B1 to B3
behavior B13Snd( in type1 v1, ISend cb13 ) {
  void main(void) { cb13.send(&v1, sizeof(v1)); }
};
5 behavior B13Rcv( out type1 v1, IRecv cb13 ) {
  void main(void) { cb13.recv(&v1, sizeof(v1)); }
};

// Send data from B3 to B4
10 behavior B34Snd( ISend cb34 ) {
  void main(void) { cb34.send( 0, 0 ); }
};
behavior B34Rcv( IRecv cb34 ) {
  void main(void) { cb34.recv( 0, 0 ); }
15 };
```

(a) Communication and synchronization behaviors.

```
// Processing element 1
behavior PE1( ISend cb13, ISend c2, IRecv cb34 )
{
  type1 v1;
5
  B1      b1     ( v1 );
  B13Snd  b13snd( v1, cb13 );
  B2      b2     ( v1, c2 );
  B34Rcv  b34rcv( cb34 );
10
  void main(void) {
    b1.main();       // original behavior B1
    b13snd.main();   // send B1 output to B3
    b2.main();       // original behavior B2
15  b34rcv.main();   // receive B3 output
  }
};

// Processing element 2
20 behavior PE2( IRecv cb13, IRecv c2, IRecv cb34 )
{
  type1 v1;

  B13Rcv b13rcv( cb13, v1 );
25 B3     b3     ( v1, c2 );
  B34Snd b34snd( cb34 );

  void main(void) {
    b13rcv.main();  // receive B3 input from B1
30  b3.main();      // original behavior B3
    b34snd.main();  // send B3 output
  }
};

35 // Top-level
behavior Design() {
  ChMP c2;           // message-passing channels
  ChMP cb13, cb34;

40 PE1 pe1( cb13, c2, cb34 );
  PE2 pe2( cb13, c2, cb34 );

  void main(void) {
    par { pe1.main(); pe2.main(); }
45 }
};
```

(b) Top level hierarchy.

Listing 5: Architecture model.

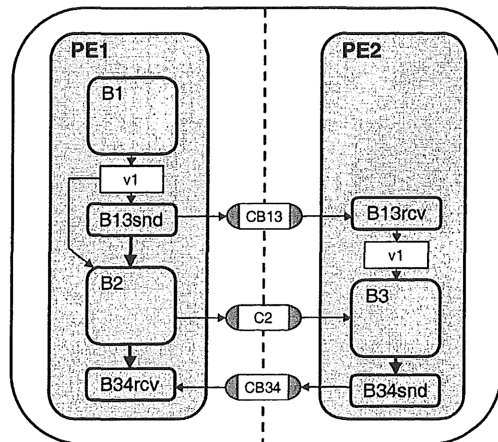


Figure 7: Architecture model.

tation with message-passing between PEs. Local copies of the variable *v1* used for communication between sequential behaviors *B1* and *B3* are created in the local memories of each PE. Inside the PEs, the behaviors operate on the local copies of the variable. In addition, code is inserted to update and synchronize local variable copies over message-passing channels at points where control is transfered between PEs. In the example, the new value of *v1* is communicated through the synchronization and communication behavior pair *B13Snd* / *B13Rcv* and the message-passing channel *CB13* together with transferring control from behavior *B1* on *PE1* to behavior *B3* on *PE2*.

In case of concurrent behaviors mapped to different PEs (e.g. behaviors *B2* and *B3* mapped to *PE1* and *PE2*), communication between the behaviors is transformed into a message-passing implementation as described in Section 2.3 (Figure 6 and Listing 4). The message-passing channel *C2* used for communication between the behaviors becomes a system-global channel connecting *PE1* and *PE2*, and *send()* and *recv()* calls in the behaviors are routed through behavior and PE ports to the global channel.

In the example, behaviors inside the PEs are statically scheduled (see Section 3.5). As shown in Figure 7 and Listing 5, scheduling is done in a straightforward way based on the constraints posed by the behavior dependencies with the goal to exploit the available parallelism. On *PE1*, execution starts with behavior *B1*. After *B1* is finished, behavior *B13Snd* transfers the output of *B1* to *PE2* such that behavior *B3* on *PE2* can then run in parallel with behavior *B2* on *PE1*. Finally, behavior *B2* on *PE1* is followed by behavior *B34Rcv* which waits for the results of *B3* from *PE2*. On component *PE2*, execution starts with behavior *B13Rcv*, waiting for *B1*'s results. Once the data is received from *PE1*, behavior *B3* is started. After *B3* is finished, control is transfered back to *PE1* through behavior *B34Snd*.

3.2 Storage

Member variables of the behaviors in the specification model represent storage that has to be mapped to memories in the implementation of the architecture model. This includes member variables as part of leaf behaviors as well as variables connecting subbehavior ports that are used for communication.

In the implementation, the memory space of the system is formed by the union of the system PE memories. In general, processing elements each have local memories as part of their microarchitecture. If the local memory of a PE can be accessed from other PEs it becomes global system memory. A special case are memory components whose sole purpose is to provide global storage. They are not able to execute any computational behavior and, therefore, do not provide any processing functionality.

Member variables in the specification are mapped to local or global memories in the architecture. Of special interest are variables used for communication between behaviors mapped to different PEs (see Section 3.3). If a member variable connects two subbehaviors mapped to different PEs it becomes a shared variable on the system level. In a message-passing implementation, such variables are mapped to local memories and messages are passed among the components to communicate updated values. In a shared memory implementation, on the other hand, shared variables are mapped to a global memory components which is accessed directly by the PEs.

3.2.1 Local Memory

In the PE behaviors of the architecture model, the union of all its subbehavior's member variables (i.e. of all the behaviors instantiated under the PE behavior in the architecture model hierarchy) represents the amount of local memory occupied in the PE. For example, in the architecture model of Section 3.1 (Figure 7), both *PE1* and *PE2* provide storage for a variable *v1* in their local memories, as specified by the declarations in line 4 and line 22 of Listing 5, respectively.

Unless mapped to global memory (Section 3.2.2), a behavior's member variables will normally be stored in the local memory of the PE the behavior is mapped to. However, member variables that connect subbehaviors mapped to different PEs need to be shared between PEs. In a message-passing implementation, copies of the shared variable are created in the local memories of all PEs accessing the variable. Behaviors inside the PEs then operate on the local copies. In order to implement the shared semantics of the variable and to keep local copies in sync, updated variable values are communicated over message-passing channels between the components at synchronization points, as discussed in detail in Section 3.3.

The example of Section 3.1 implements such a message-passing implementation. As defined by the original specification model (Section 2.1), variable *v1* is shared between behaviors *B1*, *B2*, and *B3*. Since behavior *B3* is mapped to a different PE than behaviors *B1* and *B2*, variable *v1* has to be shared between the PEs of the system architecture. As shown in Figure 7 and Listing 5, local copies of the variable are instantiated in the components *PE1* and *PE2*. Inside the components, the corresponding ports of behaviors *B1*, *B2*, and *B3* are connected to the local copies of *v1*. Finally, the additional communication and synchronization behaviors *B13Snd* and *B13Rcv* (see Listing 5(a)) send and receive updated values of *v1* from *PE1* to *PE2* after behavior *B1* has finished and before behavior *B3* starts to execute.

3.2.2 Global Memory

As discussed in the previous section, the scope of variables stored in the local PE memories usually limits access to behaviors inside the PE. However, if a PE allows other PEs to access variables stored in its local memory, this storage becomes global memory in both scope and lifetime.

Usually, only dedicated shared memory components will support external access of variables stored inside the component. Such memory components provide storage only and can not execute arbitrary functionality, i.e. no behavior can be mapped onto a memory component. On the other hand, it is generally possible for any PE to provide global access to its local memory. In this case, a PE provides global system storage in addition to implementing computation.

In general, any member variable of any behavior running on a certain processing element can be mapped to global memory, for example if the PE's local memory is exhausted. The variable is then removed from the behavior and all accesses to the variable inside the behavior are replaced with global memory accesses.

However, especially the variables used for communication between behaviors mapped to different PEs are candidates for a mapping to global, shared memory. In a message-passing implementation (as described in Section 3.2.1), local copies of such variable have to be created in each connected PE, increasing the total storage cost of the system. In a shared memory implementation, on the other hand, shared variables are mapped to global memory where they can be directly accessed from each PE. Again, accesses to the variable in the leaf behaviors are replaced with accesses to the shared variable in the global memory. Synchronization that is added to preserve the execution semantics of the specification (see Section 3.3) also ensures that global variable accesses are properly ordered according to their sequence in the original specification.

```
// Send data from B1 to B3
behavior B13Snd( ISend cb13 ) {
  void main(void) { cb13.send( 0, 0 ); }
};
5 behavior B13Rcv( IRecv cb13 ) {
  void main(void) { cb13.recv( 0, 0 ); }
};

// Send data from B3 to B4
10 behavior B34Snd( ISend cb34 ) {
  void main(void) { cb34.send( 0, 0 ); }
};
behavior B34Rcv( IRecv cb34 ) {
  void main(void) { cb34.recv( 0, 0 ); }
15 };
```

(a) Synchronization behaviors.

```
// Processing element 1
behavior PE1( IMem m1,
              ISend cb13,
              ISend c2,
5             IRecv cb34 ) {
  B1     b1     ( m1 );
  B13Snd b13snd( cb13 );
  B2     b2     ( m1, c2 );
  B34Rcv b34rcv ( cb34 );
10
  void main(void) {
    b1.main();      // original behavior B1
    b13snd.main();  // B1->B3 transition
    b2.main();      // original behavior B2
15  b34rcv.main();  // wait for B3 to finish
  }
};

// Processing element 2
20 behavior PE2( IMem m1,
              IRecv cb13,
              IRecv c2,
              IRecv cb34 ) {
  B13Rcv b13rcv( cb13 );
25 B3     b3     ( m1, c2 );
  B34Snd b34snd( cb34 );

  void main(void) {
    b13rcv.main();  // wait for B1 to finish
30  b3.main();      // original behavior B3
    b34snd.main();  // send B3 completion
  }
};

35 // Top-level
behavior Design() {
  ChMP c2;            // message-passing channels
  ChMP cb13, cb34;

40 M1 m1();           // Shared memory

  PE1 pe1( m1, cb13, c2, cb34 );
  PE2 pe2( m1, cb13, c2, cb34 );

45 void main(void) {
    par { pe1.main(); m1.main(); pe2.main(); }
  }
};
```

(b) Top level hierarchy.

Listing 6: Shared memory architecture model.

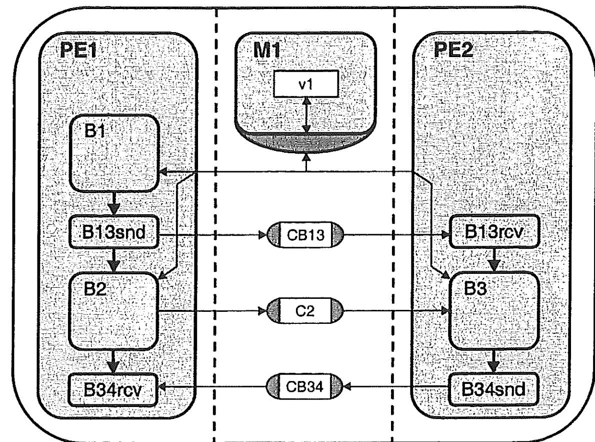


Figure 8: Shared memory architecture model.

```
// Shared memory interface
interface IMem {
  type1 r_v1( void );
  void  w_v1( type1 d );
5 };

// Shared memory component
behavior M1() implements IMem
{
10 type1 v1;

  // Memory read/write interface
  type1 r_v1( void ) { return v1; }
  void  w_v1( type1 d ) { v1 = d; }
15
  void main(void) { /* nothing */ };
};
```

Listing 7: Global memory component.

A shared memory implementation of the architecture model from Section 3.1 is shown in Figure 8 and Listing 6. Instead of a message-passing implementation, the variable *v1*, which is shared between behaviors *B1*, *B2*, and *B3*, is mapped to a dedicated shared memory component *M1*. There are no local copies of *v1* in components *PE1* or *PE2*, and all three behaviors access the variable *v1* inside the global memory *M1* instead.

Like other system components, the dedicated memory component is represented by a behavior which is instantiated at the top level of the architecture model, running in parallel with all other PEs. The code for the global memory component behavior *M1* is shown in Listing 7. Since it is a dedicated memory component that does not execute any computational functionality, the behavior's *main()* method remains empty. In general, if global memory is provided by a processing element, the PE behavior will execute the behaviors mapped to the component in addition to implementing an interface to its memory.

```
// leaf behavior 1
behavior B1( IMem mem )
{
   void main(void) {
5    ...
     mem.w_v1( ... );   // Memory write v1
   }
};

10 // leaf behavior 2
behavior B2( IMem mem, ISend c2 )
{
  void main(void) {
    type2 v2;
15    ...
    v2 = f2( mem.r_v1(), ... );   // read v1
    ...
    c2.send( &v2, sizeof(v2) );
    ...
20 }
};

// leaf behavior 3
behavior B3( IMem mem, IRecv c2 )
25 {
   void main(void) {
     type2 v2;
     ...
     c2.recv( &v2, sizeof(v2) );
30   f3( mem.r_v1(), v2, ... );   // read v1
     ...
   }
};
```

Listing 8: Shared memory accesses in leaf behaviors.

The shared variable *v1* is instantiated as a member variable of the memory behavior (line 10). The memory behavior provides access to the global variables through a channel interface *IMem*. Other PEs can connect to the memory's interface which supplies type-safe methods to read (*r_v1()*) and write (*w_v1()*) shared variables stored inside.

Inside the processing elements, accesses to the shared variable *v1* are replaced with corresponding read or write accesses to the global memory component through behavior ports, PE ports, and the memory interface. Listing 8 shows the updated accesses to variable *v1* in the leaf behaviors. Variable reads are replaced with calls of the memory's *r_v1()* method and variable assignments with calls to the *w_v1()* method.

Since updated values of *v1* are exchanged between PEs via the shared memory, behaviors *B13Snd* and *B13Rcv* (see Listing 6(a)) only perform pure synchronization by exchanging empty messages. No data communication is performed over the message-passing channels. All data transfers are handled through the global memory. On the other hand, the synchronization behaviors ensure that the shared variable *v1* is accessed by *B3* only after *B1* is finished, in consistency with the original specification.

3.3 Synchronization

In the architecture model, synchronization has to be inserted to preserve the execution semantics of the original specification. The behaviors of the specification model are mapped onto a set of concurrent components according to the structural nature of the architecture model. Therefore, synchronization has to ensure that behaviors execute in the proper order according to the transitions in the original specification.

All communication and synchronization between system components in the architecture model is handled via message-passing channels connecting the components. As shown in the architecture model example in Section 3.1 (Figure 7 and Listing 5), for each behavior transition that crosses component boundaries (transitions from *B1* to *B3* and back), a pair of synchronization behaviors (behavior pairs *B13Snd* / *B13Rcv* and *B34Snd* / *B34Rcv*, see Listing 5(a)) that communicate over a message-passing channel (channels *CB13* and *CB34*) is inserted.

By passing messages over the channels, the synchronization behavior pairs ensure that the semantics of the corresponding original behavior transition are preserved among the PEs. In this case, for example, behavior *B13Rcv* blocks execution of *B3* on *PE2* until it receives the message from behavior *B13Snd* that *B1* on *PE1* has finished. Similarly, behavior *B34Snd* on *PE2* notifies *B34Rcv* on *PE1* that *B3* has completed execution.

Along with passing control from one behavior to another, a behavior transition usually represents a transfer of data through the shared variables connecting the ports of the behaviors. If the transition crosses PE boundaries, this data has to be transfered together with passing control. In a shared memory implementation (see Section 3.2.2), data is transfered via a global system memory component and simple synchronization via synchronization behavior pairs and message-passing channels is sufficient for implementation of inter-component transitions.

On the other hand, in a message-passing implementation (see Section 3.2.1), local copies of the shared variables are created inside the components, and local values have to be synchronized across behavior transitions. In this case, communication of data values is combined with control synchronization using the behavior pairs and message-passing channels. For each transition that crosses components, the synchronization message contains all the updated data values shared between the behaviors. Local copies of variables connecting the source behavior's output ports to the target behavior's input ports are transfered in the message for each inter-component behavior transition. The communication and synchronization behavior pairs are responsible for assembling and disassembling messages from/into local variables.

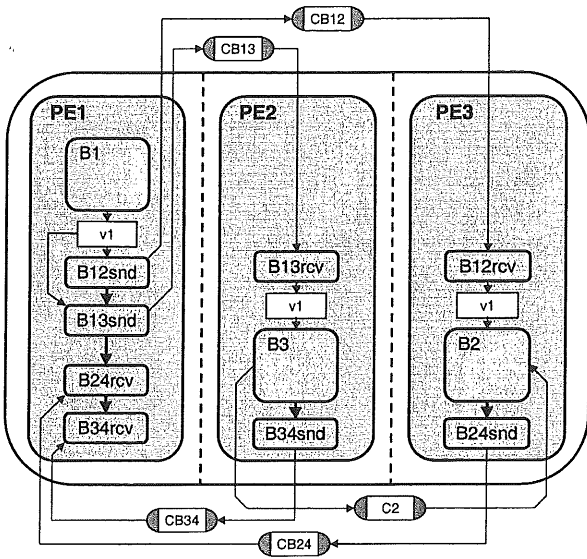


Figure 9: Architecture model with multiple inter-component behavior transitions.

For example, in the architecture model presented earlier in Section 3.1, local copies of the variable *v1* shared between behaviors *B1* and *B3* are synchronized when executing the transition from *B1* on *PE1* to *B2* on *PE2*. As shown in Listing 5(a), the synchronization behaviors *B13Snd* and *B13Rcv* for that transition read the local value of *v1* in *PE1*, pass it in a message over channel *CB13*, and update the local value of *v1* in *PE2*. Together with the synchronization described earlier, this ensures that *B3* doesn't start executing until *B1* has finished and all the output data of *B1* needed by *B3* is available.

In general, there can be multiple inter-component behavior transitions originating from a single behavior on a PE. For each such transition, a message-passing channel and a synchronization behavior pair is inserted. For example, if behavior *B2* had been mapped to a third component *PE3* in our example, an additional channel *CB12* and an additional behavior pair *B12Snd* / *B12Rcv* would have been inserted after *B1* and before *B2* on *PE1* and *PE3*, respectively (Figure 9). Likewise, an additional channel *CB24* and an additional behavior pair *B24Snd* / *B24Rcv* would have been inserted to signal completion of *B2*.

Note that it is part of the implementation issues related to the architecture model to decide in which order the send and receive behaviors are scheduled inside the PEs, e.g. whether *B13Snd* will execute before or after *B12Snd* on *PE1* (for more discussion of scheduling issues see Section 3.5).

Finally, after scheduling has determined the order of synchronization behaviors, an optional code optimization step can be applied to merge consecutive synchronization behaviors inside the same component into a single synchronization behavior that successively sends and receives the necessary messages. For example, in the model from Figure 9, the behaviors *B13Snd* and *B12Snd* could be merged into a single behavior *B1Snd*. Alternatively, behaviors *B12Snd*, *B13Snd*, *B24Rcv*, and *B34Rcv* on *PE1* could be merged into one large synchronization behavior (e.g. *B14Sync*).

3.4 IP Components

Intellectual property (IP) processing elements are characterized by the fact that their computational functionality (behavior), their communication functionality (interface), or both are predefined and fixed. In general, IP supplier and IP intergrator are different entities, either in-house or among a global IP trading marketplace. The IP supplier provides models of the IP component which are integrated into the architecture and following models for validation and synthesis. An IP creator can choose to supply different models of an IP varying in their amount of detail at different levels, trading off accuracy and simulation speed, for example.

In the architecture model, a behavioral model of the IP is required. At this level, the IP model describes the IP functionality annotated with performance and other quality metrics similar to other PE models (see also Section 3.6 about timing annotation). However, in their interfaces to other PEs, IPs, by definition, are not capable of implementing arbitrary inter-PE communication, and it is not possible to simply connect any message-passing channels to the IP in the architecture model, for example.

Therefore, IP models directly provide a channel interface at the message-passing level. An IP's channel interface describes the communication with the rest of the system supported by the IP. It abstracts the underlying IP behavior and IP interface to the external world. Furthermore, IP channel interfaces at this high level are canonical for all IPs of the same class, enabling plug-and-play of IPs without modifications to the rest of the system. For example, different DCT IPs from different suppliers can be easily exchanged since they all provide the same channel interface.

Listing 9 shows an example of an IP model. For this example, we assume that the functionality of *B2* is available in the form of this IP. As part of exploration, we then have the option to implement B2 using the IP instead of mapping it onto a general-purpose PE, depending on quality metrics like performance, cost, and power.

The purely behavioral model in Listing 9(a) is at the highest level of abstraction for integration into the architecture model and as such the minimal requirement an IP

```
// IP interface
interface IIP
{
  // Start IP, send parameters
5 void   start( type1  v1 );
  // Get value of v2 from IP
  type2  v2( void );
  // Wait for IP to finish , get result
  void   done( void );
10 };

// IP model
behavior IP() implements IIP;
//    Declaration only
15 //    Implementation is external

// Annotations
note IP.WMOPS = 13476;  // Quality metrics
```

(a) Behavioral IP.

```
// Bus-functional IP model
behavior IPBF( inout bit[63:0] data,
               in    event      start,
               out   bit[2]     ready,
5              out   event      done );
//    Declaration only
//    Implementation is external

// Behavioral IP model
10 behavior IP() implements IIP
{
  // IP bus
  bit[63:0] dat;
  event     st, dn;
15 bit[2]    rdy;

  // Bus-functional IP instance
  IPBF ip( dat, st, rdy, dn );

20 // Implementation of IP communication
  void start( type1 v1 ) {
    // Put params on bus, notify IP
    dat = v1;
    notify( st );
25 }

  type2 v2( void ) {
    // wait for data, read from bus
    while( ! rdy[1] ) wait( dn );
30   return dat;
  }

  void done( void ) {
    // wait until IP becomes ready again
35   while( ! rdy[0] ) wait( dn );
  }

  // Run internal bus-functional model
  void main(void) { ip.main(); }
40 };
```

(b) Bus-functional IP with wrapper.

Listing 9: IP component model.

supplier must provide. The interface *IIP* defines the possible communication with the IP. Corresponding to its *B2* functionality (compare to Listing 4), the interface provides three message-passing methods: *start()* sends the parameter *v1* to the IP and starts execution of one iteration; *v2()* receives the value of *v2* from the IP during its execution; finally, *done()* waits for the message from the IP that it has finished. Again, note that this is the general interface for all possible IP components that provide functionality equivalent to *B2*.

The actual IP model *IP* then implements the *IIP* interface, modeling the IP functionality (and performance) in response to incoming data and generating outgoing messages from/to other PEs. Usually, the IP supplier will want to protect the details of the IP implementation. Therefore, only the declaration of *IP* is provided. The actual code is supplied in the form of a precompiled library that will get linked into the architecture model for simulation. In addition, the IP is annotated with various information about quality metrics, verification properties, and so on. Note that the IP supplier always has the option to provide full source code. Especially at the behavioral level, source code can serve as additional documentation about IP functionality without disclosing any implementation details.

In Listing 9(b), a slightly more detailed IP model is shown. In this case, the IP model includes a bus-functional IP description *IPBF* that will be needed for the communication model later anyway. As will be explained in Section 4.4, the bus-functional IP model describes communication with the IP as events on the actual IP bus in a timing-accurate manner. In our example, the IP bus includes bidirectional data wires (*data[63:0]*), status lines (*ready[2]* and *done*), and control lines (*start*).

The behavioral IP model *IP* then wraps a channel interface around the bus-functional model. Internally, the behavioral model instantiates *IPBF* and executes its functionality in the *main()* method. The wrapper then implements the message-passing communication of the channel interface by translating them into actions on the IP bus according to the IP protocol. For example, the *v2()* method waits for the corresponding *ready* line to be asserted before reading the value from the *data* bus.

Figure 10 and Listing 10 show the architecture model in which *B2* is mapped onto an instance *IP1* of the *IP* processing element. Instead of *PE3*, the IP component is instantiated in the top level of the design (Listing 10(b)). Compared to the model from Section 3.3 (Figure 9), all communication with *B2* on *PE3* is replaced with direct connections to the IP's channel interface. Inside the communication and synchronization behaviors *B12Snd* and *B24Rcv* (Listing 10(a))), message-passing methods of the IP's channel interface are called for all communication

```
// Send data to IP
behavior B12Snd( in type1 v1, IIP ip1 ) {
  void main(void) { ip1.start( v1 ); }
};

// Receive results from IP
behavior B24Rcv( IIP ip1 ) {
  void main(void) { ip1.done(); }
};
```

(a) Synchronization with IP.

```
// Processing element 1
behavior PE1( ISend cb13, IIP ip1, IRecv cb34 )
{
  type1 v1;

  B1     b1     ( v1 );
  B13Snd b13snd( v1, cb13 );
  B12Snd b12snd( v1, ip1 );
  B24Rcv b24rcv( ip1 );
  B34Rcv b34rcv( cb34 );

  void main(void) {
    b1.main();      // original behavior B1
    b13snd.main();  // B1->B3 transition
    b12snd.main();  // B1->B2 transition
    b24rcv.main();  // wait for B2 to finish
    b34rcv.main();  // wait for B3 to finish
  }
};

// Processing element 2
behavior PE2( IRecv cb13, IIP ip1, IRecv cb34 )
{
  type1 v1;

  B13Rcv b13rcv( cb13, v1 );
  B3     b3     ( v1, ip1 );
  B34Snd b34snd( cb34 );

  void main(void) {
    b13rcv.main();  // wait for B1 to finish
    b3.main();      // original behavior B3
    b34snd.main();  // send B3 completion
  }
};

// Top-level
behavior Design()
{
  ChMP cb13, cb34; // message-passing channels

  IP ip1();        // IP component

  PE1 pe1( cb13, ip1, cb34 );
  PE2 pe2( cb13, ip1, cb34 );

  void main(void) {
    par { pe1.main(); ip1.main(); pe2.main(); }
  }
};
```

(b) Top level hierarchy.

Listing 10: Architecture model with IP.

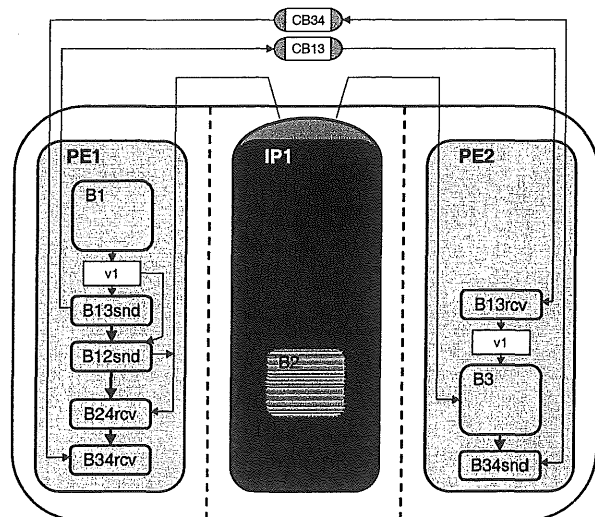


Figure 10: Architecture model with IP.

```
// leaf behavior 3
behavior B3( in type1 v1, IIP ip1 )
{
  void main(void) {
    type2 v2;
    ...
    v2 = ip.v2();   // receive v2 from IP
    f3( v1, v2, ... );
    ...
  }
};
```

Listing 11: IP accesses in leaf behavior *B3*.

with the IP. Similarly, inside leaf behavior *B3*, all channel calls for communication with *B2* are replaced with calls to the IP's corresponding interface methods, as shown in Listing 11.

3.5 Scheduling

By definition, the components of the system architecture are single-threaded in terms of the computation they are executing. According to the inherently sequential nature of components, behaviors mapped onto a PE have to be scheduled in order to serialize their execution. The order of execution of both, the original computation behaviors and the additional communication/synchronization behaviors determines the schedule of computation and communication on each PE.

In the simplest case, static scheduling is performed. The execution order of the behavioral blocks inside the PEs is fixed by introducing artificial dependencies according to the selected schedule. Therefore, the behavior hierarchy inside the components becomes a purely sequential com-

position. Behaviors are executed in the pre-defined order defined by the sequential transitions inside the PEs of the architecture model. For example, as described in Section 3.1, the subbehaviors in each PE of the architecture model from Figure 7 (Listing 5) are executed sequentially in the order determined by the static schedule. Hence, the *PE1* and *PE2* behaviors are a purely sequential composition executing their subbehaviors in the given order.

In a dynamic scheduling approach, on the other hand, the order of execution is determined dynamically during runtime. Behaviors are arranged into potentially concurrent tasks. Inside each task, behaviors are executed sequentially. Tasks can be dynamically forked and joined through `par` statements in the code. A scheduler maintains a pool of task behaviors and dynamically selects a task to execute according to its scheduling algorithm. The scheduler in the architecture model is a behavioral abstraction of the scheduling policy of the underlying operating system.

3.6 Time

After behaviors have been partitioned onto PEs, the concept of time is introduced for the computation represented by the behaviors. Behaviors grouped under a PE are refined to include execution times on the target. As a result, behavior executions among the concurrent PEs are ordered additionally beyond the pure causality established by the inter-PE synchronization.

Behavior execution delays can be based on estimated execution times derived from a model of the target component, for example. Alternatively, execution delays can describe a timing budget allocated for different behaviors. These budgets will later serve as timing constraints for the behavior implementation on the target PEs.

Execution times can be specified on different levels of granularity, ranging from the statement level to the behavior level. Execution delays at the behavior level are used to model average or worst-case execution times of the corresponding behavior. On the other hand, execution times at the basic-block level can accurately model even data-dependent delays. The leaf behaviors are annotated with `waitfor()` statements to model execution time. In addition to providing feedback about logical time during simulation, the annotations serve as constraints for synthesis and verification tools.

Listing 12 shows a code template for a leaf behavior with estimated timing. In this case, execution delays are modeled at the basic block level. At this granularity, data-dependent delays are accurately modeled while keeping the simulation overhead incurred by the `waitfor()` statements at a minimum.

```
   behavior Bx(...)
   {
     void main(void)
     {
5      if ( ... ) {
         ...
         waitfor( T1 );   // execution time 1
       }
       else {
10       ...
         waitfor( T2 );   // execution time 2
       }
       ...
       waitfor( T3 );     // execution time 3
15     Cy.send( ... );
       ...
       waitfor( T4 );     // execution time 4
     }
   };
```

Listing 12: Behavior timing.

3.7 Summary

The architecture model describes the implementation of the computation on the PEs of the system architecture. It is a structural view of the system's PE architecture. It contains behavioral views of the PEs that represent the mapping of computation onto each PE. Communication, on the other hand, remains at an abstract level. The architecture model exposes the communication between PEs which will be implemented in the next step.

In summary, properties of the architecture model are:

(a) At the top level of the behavior hierarchy, the PE structure is modeled as a parallel composition of non-terminating PE behaviors.

(b) PE behaviors communicate via system-global message-passing channels connecting their ports.

(c) PE behaviors with predefined, fixed communication functionality (IPs. memories) directly provide channel interfaces for communication.

(d) Original specification behaviors are grouped under the PE behaviors to specify the functionality to be implemented by each PE.

(e) Member variables of behaviors instantiated inside a PE represent the amount of storage allocated in the local PE memory.

(f) Behaviors inside different PEs communicate by sending and receiving messages over ports and global channels.

(g) True parallelism is limited to the concurrency among PEs. Internally, PEs are single-threaded. Execution of behaviors inside a PE is serialized in time through static or dynamic scheduling.

(h) Computation in the leaf behaviors is annotated with estimated or projected execution times on its target PE.

All in all, the architecture model accurately reflects the implementation of the computational aspects of the system for analysis and validation.

4 Communication Model

The communication model is the final output of the system-level design process after architecture exploration implements computation on the PEs and communication synthesis implements communication over the busses of the system architecture. The communication model represents the mapping of computation and communication onto PEs and busses, respectively.

The communication model is a structural view of the complete system including computation and communication. It shows the PE and bus structure of the final system architecture. The system is described as a netlist of concurrent, non-terminating PEs connected via system bus wires.

Unaltered from the architecture model described in Section 3, the communication model gives a behavioral view of the computation and storage to be implemented by each PE. The functionality of each PE is described by the behaviors grouped under the PE and executing inside. Furthermore, the union of all its behavior's member variables represents the storage allocated inside the PE's local memory.

In contrast, the behavioral view of the communication in the architecture model is replaced with a structural description in the communication model. The abstract channels connecting the PEs in the architecture model are replaced with an implementation of their communication functionality over wires and protocols of system busses connecting the PEs.

Inside the PEs, behavioral models of bus drivers and bus interfaces describe the PE's communication functionality, i.e. the implementation of the message-passing communication over the bus protocols. Those bus adapters specify how the PE implements the semantics of the abstract channels by driving and sampling the wires of the system bus. Behavioral blocks inside the PEs, in turn, connect to the equivalent message-passing channel interfaces provided by the bus adapters.

In general, not all PEs can be programmed or synthesized to implement arbitrary communication functionality. For example, PEs with fixed, pre-defined bus interfaces and protocols like memories or IP components are not capable of connecting to any bus protocol. In those cases, the communication model will include additional transducers that

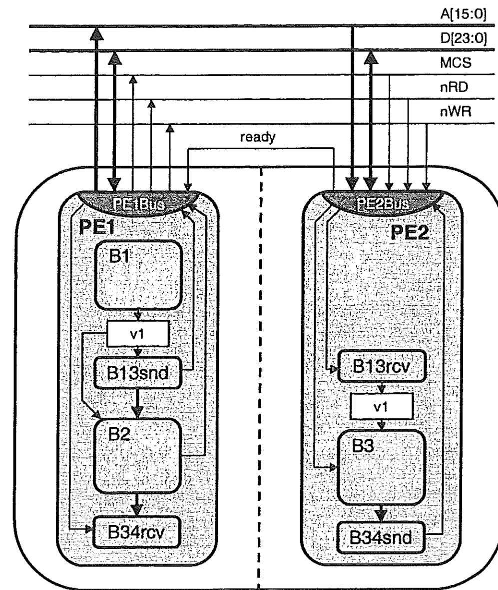


Figure 11: Communication model.

translate between incompatible protocols. Transducers are special PEs that act as bus bridges or bus interfaces, connecting two busses or interfacing a PE to a bus.

As explained in Section 3, the architecture model introduced a total order for the computation inside the PEs. On top of that, the communication model imposes a total order on the communication performed over the busses. Communication on each bus is scheduled. In case of dynamically occurring conflicts between multiple bus masters, arbitration resolves conflicts at run-time in either a distributed fashion or through a centralized arbiter PE.

Finally, the communication model adds timing information for the system communication. Target bus delays are estimated and communication behavior in the drivers and on the busses is annotated with timing information. Increasing timing accuracy to cover both computation and communication further refines the partial order of events in the system architecture. The bus-functional PE models in the communication model accurately describe the behavior and timing of the PEs at their bus interfaces. Therefore, the system model describes their interaction in a timing-accurate manner.

4.1 Communication Model Example

The communication model of the example design from Section 3.1 (Figure 7 and Listing 5) is shown in Figure 11 and Listing 13. As in the architecture model, the system consist of two processing elements, *PE1* and *PE2*. However, instead of abstract channels, the two PEs are connected via a single, shared system bus. During communi-

```
   // Processing element 1
   behavior PE1( out    bit [15:0] A,
                 inout  bit [23:0] D,
                 OSignal MCS,
 5               OSignal nRD,
                 OSignal nWR,
                 ISignal ready )
   {
     // Bus driver
10   PE1Bus bus( A, D, MCS, nRD, nWR, ready );

     type1 v1;

     B1     b1     ( v1 );
15   B13Snd b13snd( v1, bus );
     B2     b2     ( v1, bus );
     B34Rcv b34rcv( bus );

     void main(void) {
20     b1.main();
       b13snd.main();
       b2.main();
       b34rcv.main();
     }
25 };

   // Processing element 2
   behavior PE2( in     bit [15:0] A,
                 inout  bit [23:0] D,
30               ISignal MCS,
                 ISignal nRD,
                 ISignal nWR,
                 OSignal ready )
   {
35   // Bus interface
     PE2Bus bus( A, D, MCS, nRD, nWR, ready );

     type1 v1;

40   B13Rcv b13rcv( bus, v1 );
     B3     b3     ( v1, bus );
     B34Snd b34snd( bus );

     void main(void) {
45     b13rcv.main();
       b3.main();
       b34snd.main();
     }
   };
50
   // Top-level
   behavior Design()
   {
     // System bus wires
55   bit [15:0] A;                          // address
     bit [23:0] D;                          // data
     CSignal MCS, nRD, nWR, ready;          // control

     PE1 pe1( A, D, MCS, nRD, nWR, ready );
60   PE2 pe2( A, D, MCS, nRD, nWR, ready );

     void main(void) {
       par { pe1.main(); pe2.main(); }
     }
65 };
```

(a) Top level hierarchy.

Listing 13: Communication model.

```
   // Send data from B1 to B3 over bus
   behavior B13Snd( in type1 v1, IBus bus ) {
     void main(void) {
       bus.send( CB13, &v1, sizeof(v1) );
 5   }
   };
   behavior B13Rcv( out type1 v1, IBus bus ) {
     void main(void) {
       bus.recv( CB13, &v1, sizeof(v1) );
10   }
   };

   // Send data from B3 to B4 over bus
   behavior B34Snd( IBus bus ) {
15   void main(void) { bus.send( CB34, 0, 0 ); }
   };
   behavior B34Rcv( IBus bus ) {
     void main(void) { bus.recv( CB34, 0, 0 ); }
   };
```

(b) Communication and synchronization behaviors.

```
   // leaf behavior 2
   behavior B2( in type1 v1, IBus bus )
   {
     void main(void) {
 5     type2 v2;
       ...
       v2 = f2( v1, ... );
       ...
       // send message
10     bus.send( C2, &v2, sizeof(v2) );
       ...
     }
   };

15 // leaf behavior 3
   behavior B3( in type1 v1, IBus bus )
   {
     void main(void) {
       type2 v2;
20     ...
       // receive message
       bus.recv( C2, &v2, sizeof(v2) );
       f3( v1, v2, ... );
       ...
25   }
   };
```

(c) Bus communication in leaf behaviors.

Listing 13 (continued): Communication model.

cation synthesis, all message-passing communication between the PEs has been mapped onto that bus.

In this example, it is assumed that *PE1* is a digital signal processor (DSP) from Motorola's DSP56600 family [4] of DSPs. Therefore, the DSP's external bus protocol was chosen as the system bus protocol. The DSP56600 bus consists of an 16-bit wide address bus *A*, a 24-bit wide data bus *D*, and a set of control lines for master chip select (*MCS*) and read/write control (*nRD/nWR*). Details of the protocol and its implementation on the PEs will be explained in Section 4.2.

The DSP56600 bus protocol is a typical master-slave protocol with the DSP (*PE1*) being the master on the bus. In the example, *PE2* is assumed to be a custom hardware component that will be synthesized to implement the protocol as a bus slave listening to requests. On the other hand, *PE2* can signal *PE1* through a *ready* line for synchronization purposes.

4.1.1 Bus Wires

In the communication model, the control wires of system busses are represented by instances of a SpecC channel *CSignal* (shown in Listing 14). The signal channel combines a value and an event into the signal semantics needed for efficient modeling of physical communication. Similar to VHDL signal semantics, an event is generated whenever a value is assigned to the wire. Hence, sampling a wire can be efficiently modeled in the event-driven simulation environment by blocking behaviors/tasks on the wire event.

The signal channel *CSignal* provides two interfaces *ISignal* and *OSignal* for read (*val()* method) or write (*assign()* method) access to the corresponding wire. In addition, the reader interface (*ISignal*) provides a method *waitval()* to efficiently model sampling of the wire until a certain value is reached.

Internally, the signal channel encapsulates the necessary code for simulation of all functionality provided by the wire model. Note that the signal channel code is for simulation purposes only. During synthesis, accesses to the channel's methods will be implemented as corresponding accesses to the real, physical wire.

At the top level of the communication model (Listing 13(a)), signal channels representing the control wires of the system bus are instantiated (line 58). In addition, address and data busses are represented by simple bit vectors of the required width (line 56 and line 57). The PEs then connect to the wires through their ports (lines 2-7 and lines 28-34). Depending on the access direction, PEs connect to the reader and/or writer side of the bit vectors and signal channels.

```
   // Reader interface
   interface ISignal {
     bit[1] val(void);                // get current value
     void    waitval(bit[1] v);       // wait for value
 5 };

   // Writer interface
   interface OSignal {
     void assign(bit[1] v);           // drive signal
10 };

   // Channel implementation
   channel CSignal() implements ISignal, OSignal {
      bit[1] value;
15    event  e;

      void assign(bit[1] v) {
        value = v;
        notify( e );
20    }
      bit[1] val() {
        return value;
      }
      void waitval( bit[1] v ) {
25      while( value != v )
          wait( e );
      }
   };
```

Listing 14: Signal channel for modeling of wires.

4.1.2 Bus Adapters

Inside the PEs of the communication model, bus adapters *PE1Bus* and *PE2Bus* are instantiated (see Listing 13(a), line 10 and line 37, respectively). The bus adapters specify how the communication methods and semantics of the abstract channels from the architecture model are implemented over the bus wires on the corresponding PE.

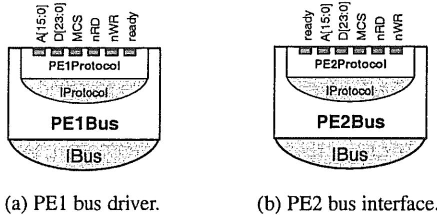
Bus adapters are channels with ports that connect to the bus wires through the PE's ports. At their channel interfaces, on the other side, the bus adapters provide abstract communication methods equivalent to the message-passing methods of the architecture model channels. Instead of the message-passing channels, the behaviors executing inside the PEs then connect to the bus adapter's equivalent interfaces, and the adapters implement the message-passing by driving and sampling the bus wires according to the bus protocol.

As shown in Listing 13(b) and Listing 13(c) for the PE's synchronization and leaf behaviors, respectively, calls to the channel's *send()* and *recv()* methods are replaced with calls to the corresponding methods of the bus adapter interface *IBus* (Listing 15). The bus adapters provide methods for every type of communication handled over that bus, i.e. for sending and receiving messages of arbitrary size in this case. In addition, in order to differentiate between different logical connections mapped onto the same bus, a virtual addressing scheme is introduced at the adapter level. Different transfers over the same adapter are distinguished

```
// Virtual bus addresses
enum { CB13, C2, CB34 } addr;

// Message-passing over bus
5 interface IBus {
    void send( addr a, void *data, int size );
    void recv( addr a, void *data, int size );
};
```

Listing 15: PE bus adapter interface.



(a) PE1 bus driver. (b) PE2 bus interface.

Figure 12: PE bus adapters.

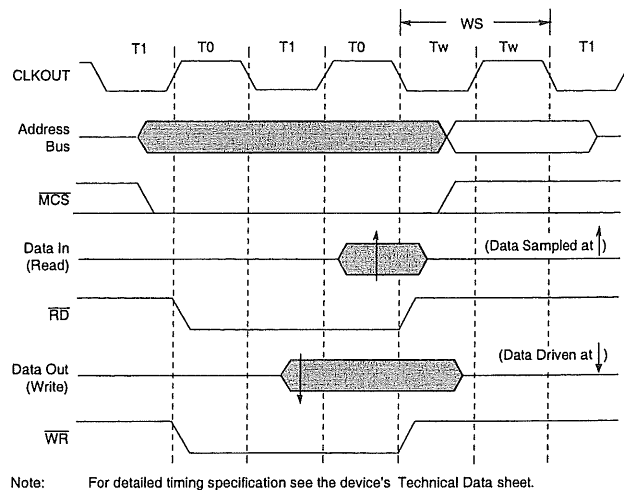


Figure 13: DSP56600 protocol timing diagram.

by their virtual address which, in general, is an identifier for the original connection the transfer belongs to. The bus adapter will then translate virtual addresses into unique, real addresses on the bus.

As depicted in Figure 12, the bus adapter channels are hierarchically composed out of two layers: a high-level application layer and a low-level protocol layer. The protocol layers *PE1Protocol* and *PE2Protocol* perform actual bus transactions by driving and sampling bus wires. At their interfaces to the application layer, they provide methods for all bus primitives supported by the protocol. The application layer, on the other hand, sits on top of the protocol layer and provides the adapter's outer interface to the external world. Using the protocol layer primitives, it performs the necessary synchronization, data slicing, addressing, and arbitration to implement the communication over the bus protocol.

4.2 Protocol Layer

The protocol layer implements the bus protocol for simulation and synthesis. During communication synthesis, a description of the selected bus protocol is taken out of the protocol library in the form of a protocol channel. The protocol channel encapsulates the bus wires and implements the protocol by driving and sampling the wires according to the timing diagram of the protocol. At its interface, the channel abstracts the protocol by providing methods for all primitive transactions like read, write, burst read, burst write, etc. supported by the bus. Protocol channels are then split and moved into the PEs where they become the protocol layer of the PE's bus adapters. In the process, protocol descriptions are adapted to the PE's capabilities (for example by inserting timing estimates), and the application layer is generated on top of the protocol primitives.

The timing diagram for the DSP56600 bus protocol chosen for our example is shown in Figure 13 [4]. A bus transfer starts with the DSP driving the address bus and asserting the *MCS* line. Depending on the direction of the transfer, the DSP then asserts either the *nRD* or *nWR* control line. In case of a bus read, the slave will put the selected data on the data bus where the DSP will read it from before deasserting the *nRD* line again. In case of a bus write, on the other hand, the DSP will drive the data bus and the slave will sample the data when the DSP deasserts the *nWR* line again. Finally, the transfer completes with the DSP releasing the address bus and deasserting the chip select line.

Figure 13 shows the protocol layers of the bus adapters in the *PE1* (DSP) and *PE2* (slave) components for the DSP56600 protocol. The external interface *IProtocol* of the protocol layer (see Listing 16(c)) provides methods for the two simple bus read and write transfers supported by the DSP56600 protocol. The protocol layers *PE1Protocol* and *PE2Protocol* then implement the master and slave side of the *read()* and *write()* primitives by driving and sampling the bus wires according to the sequence of events in the timing diagram. Bus wires are accessed by reading from and writing to corresponding ports of of the protocol channel which, in turn, will connect to the reader and writer interfaces of the bit vectors and signal channels representing the bus wires at the top level.

In the DSP56600 manual [4], the timing diagram from Figure 13 is annotated with additional timing constraints between events on the wires. In the protocol layer code (Listing 16), timing constraints are modeled by enclosing the code sampling and driving the wires in a `do-timing`

```
channel PE1Protocol( out    bit[15:0] A,
                     inout  bit[23:0] D,
                     OSignal MCS,
                     OSignal nRD,
5                    OSignal nWR )
  implements IProtocol
{
  // Bus master read
  bit[23:0] read( bit[15:0] addr )
10 {
    bit[23:0] data;

    do {
      t1 : A = addr;            // assign address
15         waitfor(3);
      t2 : MCS.assign(1);       // assert chip select
           waitfor(12);
      t3 : nRD.assign(0);       // assert read line
           waitfor(5);
20    t4 : data = D;            // sample data bus
           waitfor(18);
      t5 : nRD.assign(1);       // deassert read
           waitfor(7);
      t6 : MCS.assign(0);       // deassert chip select
25  }
    timing {    // constraints
      range( t1; t6; 45; 100 );
      range( t2; t3;  4; );
      range( t3; t5; 33; );
30    range( t3; t4; 30; );
    }

    return data;
  }
35
  // Bus master write
  void write( bit[15:0] addr, bit[23:0] data )
  {
    do {
40    t1 : A = addr;            // assign address lines
           waitfor(5);
      t2 : MCS.assign(1);       // assert chip select
           waitfor(10);
      t3 : nWR.assign(0);       // assert write control
45         waitfor(3);
      t4 : D = data;            // drive data outputs
           waitfor(20);
      t5 : nWR.assign(1);       // deassert write
           waitfor(10);
50    t6 : MCS.assign(0);       // deassert chip select
    }
    timing {    // constraints
      range( t1; t6; 45; 100 );
      range( t2; t3;  8; );
55    range( t3; t5; 20; );
    }
  }
};
```

(a) PE1 bus master protocol.

Listing 16: Bus adapter protocol layer.

```
channel PE2Protocol( in     bit[15:0] A,
                     inout  bit[23:0] D,
                     ISignal MCS,
                     ISignal nRD,
5                    ISignal nWR )
  implements IProtocol
{
  // Bus slave write (answer to bus read)
  void write( bit[15:0] addr, bit[23:0] data )
10 {
    do {
           // wait for chip select
      t1 : MCS.waitval( 1 );
           // address decoding
15    t2 : if (A != addr) goto t1;
           waitfor( 15 );
           // check control line
      t3 : if ( nRD.val() != 0 ) goto t1;
           // drive data bus
20    t4 : D = data;
           // wait for end of cycle
      t5 : MCS.waitval( 0 );
    }
    timing {    // constraints
25    range( t1; t5;   ; 100 );
      range( t2; t3; 4;  50 );
      range( t3; t4;   ;  30 );
    }
  }
30
  // Bus slave read (answer to bus write)
  bit[23:0] read( bit[15:0] addr )
  {
    bit[23:0] data;
35
    do {
           // wait for chip select
      t1 : MCS.waitval( 1 );
           // address decoding
40    t2 : if (A != addr) goto t1;
           waitfor( 20 );
           // check control line
      t3 : if ( nWR.val() != 0 ) goto t1;
           // sample data bus
45    t4 : data = D;
           // wait for end of cycle
      t5 : MCS.waitval( 0 );
    }
    timing {    // constraints
50    range( t1; t5;   ; 100 );
      range( t2; t3; 8;  50 );
      range( t3; t4; 20; );
    }

55  return data;
  }
};
```

(b) PE2 bus slave protocol.

```
// DSP56600 protocol primitives
interface IProtocol {
  bit[23:0] read( bit[15:0] addr );
  void write( bit[15:0] addr, bit[23:0] data );
5 };
```

(c) Protocol layer interface.

Listing 16 (continued): Bus adapter protocol layer.

construct. Constraints are specified as ranges between labels marking events on the wires. For example, there is a minimum delay of 4 time units between asserting the *MCS* and the *nRD* signals in case of a read transfer on the DSP (*PE1*) side (Listing 16(a), line 28). The corresponding slave (*PE2*) write method, therefore, has to have a delay of at least 4 time units between receiving *MCS* and sampling of *nRD* to ensure that the value of *nRD* is correct (Listing 16(b), line 26). For more information about the DSP56600 protocol timing please refer to [4].

In addition to timing constraints, the protocol layer code is annotated with estimated delays by inserting `waitfor()` statements into the code sequence. Those delays are instances of the protocol timing constraints based on an estimation of actual average delays when implementing the protocol on the given PE. Note that the *waitfor()* statements only serve as a feedback about communication timing for simulation, similar to the *waitfor()* statements inserted into the behavior code in the architecture model (see Section 3.6). Interface synthesis as part of the backend process, however, will start from the ranges specified for the timing constraints of the protocol. Based on the constraints and the PE's clock period, a state machine implementing the protocol will be synthesized which will determine the actual, exact protocol delays.

4.3 Application Layer

The application layer wraps around the protocol layer and implements the abstract, high-level communication semantics from the architecture model as a sequence of low-level, primitive bus transactions supported by the protocol. At its interface, the application layer provides message-passing methods equivalent to the architecture model's global channels. Therefore, the behaviors inside the PE can be directly connected to the application layer instead. In order to implement message-passing, the application layer has to perform tasks like synchronization of PEs, arbitration in case of multiple bus masters, addressing of data on the bus, and slicing of abstract data types into bus words. Internally, the application layer instantiates the protocol layer and calls the protocol methods in order to perform the actual bus transfers.

Listing 17 shows the top levels of the two bus adapters, *PE1Bus* and *PE2Bus*, which form the application layers inside the two PEs. The application layers implement the *send()* and *recv()* methods of the *IBus* interface (line 7) introduced earlier (Listing 15). Internally, they each instantiate the corresponding PE's local protocol layer (line 10) described in the previous section (Section 4.2). The protocol layers are connected to the bus wires through corresponding ports of the bus adapters, and they are responsible

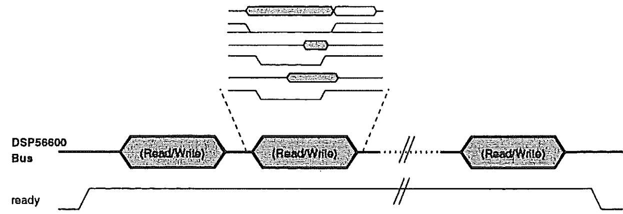


Figure 14: Application layer synchronization protocol.

for actually driving and sampling the bus wires according to the protocol timing diagram. In the code of its methods, the application layer then calls the protocol layer methods to implement the sequence of transactions over the bus.

In the following sections, we will outline each of the tasks performed by the application layer in its implementation of message-passing over the protocol. In this report, we will focus on the modeling aspects for the application layer only. A more detailed description of the communication synthesis process for the application layer can be found in [5].

4.3.1 Synchronization

To implement the blocking semantics of the message-passing communication, the application layer has to perform the proper synchronization of PEs. Depending on the bus, synchronization can be inherent in the protocol. In all other cases, the application layer has to synchronize the communication partners on top of the protocol, possibly over additional wires between the PEs that are driven and sampled by the application layer according to a high-level synchronization protocol.

In our example, rendevouz synchronization is handled through the bus protocol in one direction and through the *ready* signal in the other direction (Figure 14). The slave *PE2* signals its ready status by asserting the *ready* line (line 24 and line 50 in Listing 17(b)). In its calls of the protocol methods, the *PE2Protocol* layer will then listen on the bus for the sequence of transfers as initiated by the DSP, i.e. it will wait repeatedly for the start of each bus transfer. The DSP (*PE1*), on the other hand, first polls the *ready* line (line 26 and line 47 in Listing 17(a)), thereby blocking the DSP until the slave is ready. Once the ready signal is received, the DSP initiates the sequence of transfers through calls to its *PE1Protocol* layer. Through the bus protocol, the DSP will, in turn, wake up the slave which is blocking on the corresponding bus wire events. All together, synchronization in this example is implemented by sending events from the DSP to the slave via the bus protocol whereas events from the slave to the DSP are sent over the *ready* line.

```
   channel PE1Bus( out    bit[15:0] A,
                   inout  bit[23:0] D,
                   OSignal MCS,
                   OSignal nRD,
 5                 OSignal nWR,
                   ISignal ready )
     implements IBus
   {
     // Instantiate protocol layer
10   PE1Protocol protocol( A, D, MCS, nRD, nWR );

     // Send message
     void send( addr a, void *data, int size ) {
       bit[16] Addr;
15     short   *p;

       // Addressing: convert to bus address
       switch( a ) {
         case CB13:
20         Addr = 0x8005;   break;
         case C2:
           Addr = 0x8020;   break;
       }

25     // Synchronization: wait for ready signal
       ready.waitval( 1 );

       // Sliced data transfer
       for ( p = data ; size > 0; size -= 2 ) {
30       // call protocol layer
         protocol.write( Addr, *p++ );
       }
     }

35   // Receive message
     void recv( addr a, void *data, int size ) {
       bit[16] Addr;
       short *p;

40     // Addressing: convert to bus address
       switch( a ) {
         case CB34:
           Addr = 0x800c;   break;
       }
45
       // Synchronization: wait for ready signal
       ready.waitval( 1 );

       // Sliced data transfer
50     for ( p = data; size > 0; size -= 2 ) {
         // call protocol layer
         *p++ = protocol.read( Addr );
       }
     }
55 };
```

(a) PE1 bus driver.

Listing 17: Bus adapter application layer.

```
   channel PE2Bus( in     bit[15:0] A,
                   inout  bit[23:0] D,
                   ISignal MCS,
                   ISignal nRD,
 5                 ISignal nWR,
                   OSignal ready )
     implements IBus
   {
     // Instantiate protocol layer
10   PE2Protocol protocol( A, D, MCS, nRD, nWR );

     // Send message
     void send( addr a, void *data, int size ) {
       bit[16] Addr;
15     short   *p;

       // Addressing: convert to bus address
       switch( a ) {
         case CB34:
20         Addr = 0x800c;   break;
       }

       // Synchronization: assert ready signal
       ready.assign( 1 );
25
       // Sliced data transfer
       for ( p = data; size > 0; size -= 2 ) {
         // call protocol layer
         protocol.write( Addr, *p++ );
30     }

       // Synchronization: deassert ready signal
       ready.assign( 0 );
     }
35
     // Receive message
     void recv( addr a, void *data, int size ) {
       bit[16] Addr;
       short   *p;
40
       // Addressing: convert to bus address
       switch( a ) {
         case CB13:
           Addr = 0x8005;   break;
45       case C2:
           Addr = 0x8020;   break;
       }

       // Synchronization: assert ready signal
50     ready.assign( 1 );

       // sliced data transfer
       for ( p = data; size > 0; size -= 2 ) {
         // call protocol layer
55       *p++ = protocol.read( Addr );
       }

       // Synchronization: deassert ready signal
       ready.assign( 0 );
60   }
   };
```

(b) PE2 bus interface.

Listing 17 (continued): Bus adapter application layer.

4.3.2 Addressing

Virtual addresses on the application side have to be turned into a bus addressing scheme. In general, bus addresses are a combination of source PE, destination PE, and ID of the message to be transfered. Depending on the application, however, the bus addressing scheme can be simplified. For example, if there is a predefined order of messages between two PEs, the message ID can be removed from the address.

If the bus protocol's address bus is wide enough, virtual addresses can be directly converted into bus addresses. Otherwise, address information has to be transfered over the data bus as a header of the message frame, preceding the actual message content. Meta-data in the message header can also contain other information like the size of the message in case of variable-length messages. After synchronization, header data is transfered just like normal data (see the next section, Section 4.3.3) by calling the protocol's bus transaction primitives.

In the case of our example (Listing 17), the virtual, symbolic addresses on the application layer interface are directly converted into 16-bit bus addresses. Although unnecessary in this case since there is a predetermined order of transfers, one address in the range available on the DSP's external bus is assigned to each virtual address *CB13*, *C2*, and *CB34*. Note that since all three messages are uni-directional, each of the virtual addresses needs to be resolved in only one of the two methods on each side.

4.3.3 Data slicing

As part of the application layer, the abstract data types in the messages on the application side have to be sliced into bus words supported by the protocol. In general, slicing is the process of splitting large, complex data structures into a series of bus transfers on the sending side and reassembling the messages from the data received over the bus on the receiving side. Depending on the capabilities of the protocol, data slicing can make use of burst or other block transfer modes, for example.

In addition, slicing has to ensure correct interpretation of the sequence of low-level transfers on both sides in case of different data layout conventions on the PEs. For example, in case of a big-endian PE communicating with a little-endian PE, slicing performs the necessary byte swapping on one of the PEs. In general, different implementations of data serialization on the bus are possible, e.g. based on memory layout, based on a layout imposed by an IP component, or a canonical serialization as part of the bus protocol definition [5].

In the example shown in Listing 17, a simple loop slices the message into 16-bit words that are transfered over the 24-bit data bus. The application layer *send()* and *recv()* methods loop over all the words in the message and transfer the message one word at a time by calling the corresponding *read()* or *write()* methods of the protocol layer.

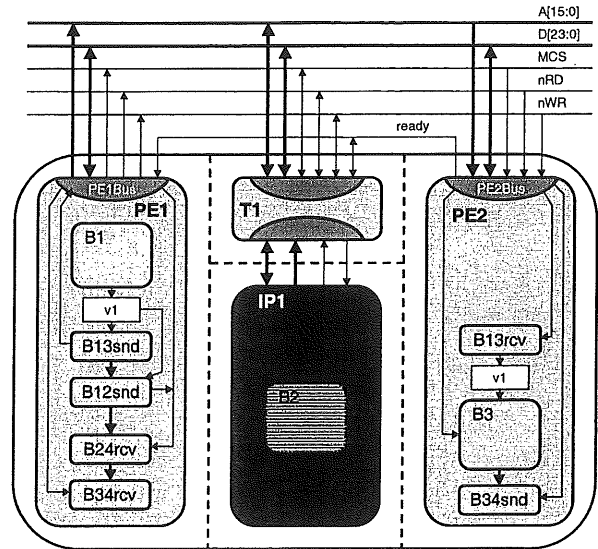


Figure 15: Communication model with IP.

4.4 Transducers

As part of the communication model, additional processing elements that translate between incompatible bus protocols might have to be inserted into the system architecture. Such transducers will act as bridges connecting two busses or as bus interfaces for PEs with fixed, predefined protocols. Especially in the case of IP components, transducers serve as universal glue logic, allowing to interface IP components to arbitrary busses. Their functionality can range from a simple conversion of signal levels up to complete protocol translators that include buffers for transfer rate adaption and decoupling.

In Section 3.4, we introduced an architecture model that included an IP component (Figure 10 and Listing 10). Figure 15 and Listing 18 show the corresponding communication model for the same architecture with communication via a single system bus based on the DSP56600 protocol. The communication model instantiates the bus-functional model *IPBF* of the IP component (line 60, Listing 18(a)) introduced in Section 3.4 (Listing 9(b)). Like the other PE models of the communication model described in the previous sections, the bus-functional IP model describes the behavior of the IP at its bus interface in a timing-accurate manner, i.e. the *IPBF* model generates events in response to incoming stimuli on the wires connected to its ports with correct timing.

Since the IP with its fixed protocol can not be directly

```
   // Processing element 1
   behavior PE1( out   bit[15:0] A, bit[23:0] D,
                 OSignal MCS, OSignal nRD,
                 OSignal nWR, ISignal ready ) {
 5   PE1Bus bus( A, D, MCS, nRD, nWR, ready );

     type1 v1;
     B1      b1     ( v1 );
     B13Snd b13snd( v1, bus );
10   B12Snd b12snd( v1, bus );
     B24Rcv b24rcv( bus );
     B34Rcv b34rcv( bus );

     void main(void) {
15     b1.main();
       b13snd.main();
       b12snd.main();
       b24rcv.main();
       b34rcv.main();
20   }
   };

   // Processing element 2
   behavior PE2( in bit[15:0] A, bit[23:0] D,
25               ISignal MCS, ISignal nRD,
                 ISignal nWR, OSignal ready ) {
     PE2Bus bus( A, D, MCS, nRD, nWR, ready );

     type1 v1;
30   B13Rcv b13rcv( bus, v1 );
     B3      b3     ( v1, bus );
     B34Snd b34snd( bus );

     void main(void) {
35     b13rcv.main();
       b3.main();
       b34snd.main();
     }
   };
40
   // Top-level
   behavior Design()
   {
     bit[15:0] A;                // System bus
45   bit[23:0] D;
     CSignal MCS, nRD, nWR, ready;

     bit[63:0] dat;              // IP bus
     event     st, dn;
50   bit[2]    rdy;

     PE1 pe1( A, D, MCS, nRD, nWR, ready );
     PE2 pe2( A, D, MCS, nRD, nWR, ready );

55   // Transducer instance
     T1 t1( A, D, MCS, MCS, nRD, nRD, nWR, nWR,
            ready, ready, dat, st, rdy, dn );

     // Bus-functional IP instance
60   IPBF ip1( dat, st, rdy, dn );

     void main(void) {
       par {
         pe1.main(); pe2.main();
65       t1.main();  ip1.main();
       }
     }
   };
```

(a) Top level hierarchy.

Listing 18: Communication model with IP.

```
   // Send data to IP
   behavior B12Snd( in type1 v1, IBus bus ) {
     void main(void) {
       bus.send( CB12, &v1, sizeof(v1) ); }
 5 };

   // Receive results from IP
   behavior B24Rcv( IBus bus ) {
     void main(void) {
10     bus.recv( CB24, 0, 0 ); }
   };
```

(b) Synchronization with IP.

Listing 18 (continued): Communication model with IP.

connected to the system bus, a transducer component *T1* is inserted into the communication model. The transducer connects to the IP bus and to the system bus, translating between the two protocols. Like the other PEs, the transducer behavior is instantiated (line 56, Listing 18(a)) and added to the set of concurrent, non-terminating PE behaviors (line 65) at the top level.

The transducer component model is shown in Listing 19. The transducer behavior (Listing 19(b)) connects to the system bus on the one hand and to the IP bus on the other hand through corresponding sets of ports. Since the transducer has to act as both master (for communication with *PE2*) and slave (for communication with the DSP *PE1*) on the system bus, it connects to both reader and writer interfaces of the control lines.

For implementation of the IP protocol, the transducer instantiates an IP bus adapter (line 12). The *T1IP* adapter (shown in Listing 19(a)) copies the channel interface methods of the IP wrapper from the architecture model (Section 3.4, Listing 9). As in the architecture model, the wrapper methods describe the implementation of the IP protocol over the adapter's IP bus ports while providing a set of methods at the message-passing level on the adapter's interface.

Similarly, the transducer contains adapters for master and slave communication over the system bus. In this simple case, the adapters are instances of the *PE1Bus* and *PE2Bus* adapters described in Section 4.1.2 and Section 4.3. Since the transducer's system bus functionality is largely equivalent to the bus communication in *PE1* and *PE2*, we are including copies of their bus adapters in this example. The necessary minor modifications of the adapters to support the additional *CB12* and *CB24* messages of the transducer are, however, not shown here and are left as an exercise to the reader.

In its *main()* method, the transducer then calls the methods provided by the adapters for communication on the IP and on the system bus side. In our example, the order of communication is predetermined and the transducer per-

```
channel T1IP( inout bit[63:0] data,
              out   event     start,
              in    bit[2]    ready,
              in    event     done )
5  implements IIP
{
  void start( type1 v1 ) {
    dat = v1;
    notify( st );
10 }
  type2 v2( void ) {
    while( ! rdy[1] ) wait( dn );
    return dat;
  }
15 void done( void ) {
    while( ! rdy[0] ) wait( dn );
  }
};
```

(a) IP bus adapter.

```
behavior T1( bit[15:0] A, bit[23:0] D, // Bus
             ISignal iMCS, OSignal oMCS,
             ISignal inRD, OSignal onRD,
             ISignal inWR, OSignal onWR,
5            ISignal irdy, OSignal ordy,
             bit[63:0] data,            // IP
             out event start,
             in bit[2] ready,
             in event done )
10 {
  // IP adapter
  T1IP  ip1( data, start, ready, done );

  // Adapters to act as bus master or slave
15 PE1Bus master( A, D, oMCS, onRD, onWR, irdy );
  PE2Bus slave ( A, D, iMCS, inRD, inWR, ordy );

  void main(void) {
    type1 v1;
20  type2 v2;

    // Receive IP parameters from B1 (PE1)
    slave.recv( CB12, &v1, sizeof(v1) );

25  // Start IP execution
    ip1.start( v1 );

    // Receive v2 from IP ...
    v2 = ip1.v2();
30
    // ... and send to B3
    master.send( C2, &v2, sizeof(v2) );

    // Wait for IP to finish
35  ip1.done();

    // Send result back to PE1
    slave.send( CB24, 0, 0 );
  }
40 };
```

(b) Transducer behavior.

Listing 19: Transducer component model.

forms a sequence of data transfers according to this predefined schedule. In the most general case, the transducer will listen on both sides simultaneously in order to handle transfers dynamically as they come in.

In this example, complete messages are received on one side, buffered in the transducer's local memory, and sent out on the other side. In order to reduce latency and memory requirements in the transducer, data transfers could be overlapped, i.e. the transducer could start sending out a words of a message on on side while still receiving remaining parts of the message on the other side. Such transducer optimizations can be part of communication synthesis or the backend process. In the latter case, the implementation model (see Section 5) will include optimized code for the transducer PE. In the former case, the communication model will include a transducer model in which the code of the adapter methods—shown separately here—is inlined into the transducer's *main()* method, flattened, and reordered across method boundaries.

4.5 Arbitration

In case of multiple masters on a system bus, the communication model has to include bus arbitration. If the order of transactions on each bus is statically fixed and predetermined, and if it can therefore be guaranteed that no conflicts will occur (as in the case of the example from Section 4.4), a static arbitration is inherent in the model. Otherwise, PEs have to dynamically resolve bus contention at runtime through an arbitration protocol as part of their protocol or application layers. Arbitration can be distributed or centralized. In a distributed scheme, PEs resolve conflicts among themselves through a distributed arbitration protocol. In a centralized scheme, a central arbiter PE is inserted into the communication model, and the arbiter grants bus requests based on a builtin arbitration algorithm.

For example, if we modify our design such that both *PE1* and *PE2* can act as either master or slave on the system bus, arbitration becomes necessary. Figure 16 and Listing 20 show the modified example including a centralized arbiter component. In this implementation, for each message to be transfered over the bus, the sending PE acts as the bus master while the receiving PE serves as bus slave. Therefore, the protocol and application layers of the bus adapter common to both PEs (shown in Listing 21) contain the *write()* and *send()* methods from the master side (i.e. from *PE1Protocol*, Listing 16(a), and *PE1Bus*, Listing 17(a)) and the *read()* and *recv()* methods from the slave side (*PE2Protocol*, Listing 16(b), and *PE2Bus*, Listing 17(b)). Note that the code for the protocol layer methods is the same as in Section 4.2. However, for simplicity the `do-timing` constraints have been omitted here.

```
// Processing element 1
behavior PE1( bit[15:0] A,   bit[23:0] D,
              ISignal iMCS, OSignal oMCS,
              ISignal inWR, OSignal onWR,
5             ISignal rdy2, OSignal rdy1,
              OSignal req1, ISignal ack1 )
{
  PEBus bus( A, D, iMCS, oMCS, inWR, onWR,
             rdy2, rdy1, req1, ack1 );
10
  type1 v1;
  B1      b1     ( v1 );
  B13Snd  b13snd( v1, bus );
  B2      b2     ( v1, bus );
15 B34Rcv b34rcv( bus );

  void main(void) {
    b1.main();
    b13snd.main();
20  b2.main();
    b34rcv.main();
  }
};

25 // Processing element 2
behavior PE2( bit[15:0] A,   bit[23:0] D,
              ISignal iMCS, OSignal oMCS,
              ISignal inWR, OSignal onWR,
              ISignal rdy1, OSignal rdy2,
30            OSignal req2, ISignal ack2 )
{
  PEBus bus( A, D, iMCS, oMCS, inWR, onWR,
             rdy1, rdy2, req2, ack2 );

35 type1 v1;
  B13Rcv b13rcv( bus, v1 );
  B3     b3     ( v1, bus );
  B34Snd b34snd( bus );

40 void main(void) {
    b13rcv.main();
    b3.main();
    b34snd.main();
  }
45 };

// Top-level
behavior Design()
{
50 bit[15:0] A;
  bit[23:0] D;
  CSignal MCS, nWR, rdy1, rdy2;
  CSignal req1, ack1, req2, ack2;

55 // Arbiter
  Arbiter arbiter1( req1, ack1, req2, ack2 );

  PE1 pe1( A, D, MCS, MCS, nWR, nWR,
           rdy2, rdy1, req1, ack1 );
60 PE2 pe2( A, D, MCS, MCS, nWR, nWR,
           rdy1, rdy2, req2, ack2 );

  void main(void) {
    par {
65    arbiter1.main(); pe1.main(); pe2.main();
    }
  }
};
```

Listing 20: Communication model with arbiter.

```
channel PEProtocol( bit[15:0] A,   bit[23:0] D,
                    ISignal iMCS, OSignal oMCS,
                    ISignal inWR, OSignal onWR )
  implements IProtocol
5 {
  // Bus slave read
  bit[23:0] read( bit[15:0] addr ) {
    bit[23:0] data;
    t1: iMCS.waitval( 1 );
10  t2: if ( A != addr ) goto t1; waitfor( 20 );
    t3: if ( inWR.val() != 0 ) goto t1;
    t4: data = D;
    t5: iMCS.waitval( 0 );
    return data;
15 }
  // Bus master write
  void write( bit[15:0] addr, bit[23:0] data ) {
    t1: A = addr;            waitfor(5);
    t2: oMCS.assign(1);      waitfor(10);
20  t3: onWR.assign(0);      waitfor(3);
    t4: D = data;            waitfor(20);
    t5: onWR.assign(1);      waitfor(10);
    t6: oMCS.assign(0);
  }
25 };
```

(a) Protocol layer.

```
channel PEBus( bit[15:0] A,   bit[23:0] D,
               ISignal iMCS, OSignal oMCS,
               ISignal inWR, OSignal onWR,
               ISignal irdy, OSignal ordy,
5              OSignal req,  ISignal ack )
  implements IBus
{
  PEProtocol p( A, D, iMCS, oMCS, inWR, onWR );

10 // Bus master message send
  void send( addr a, void *data, int size ) {
    short *p;

    irdy.waitval( 1 );    // Synchronization
15
    req.assign( 1 );      // Request bus
    ack.waitval( 1 );     // Wait for acknowledge

    for ( p = data; size > 0; size -= 2 ) {
20    switch( a ) {
        case CB13: p.write(0x8005, *p++); break;
        case C2:   p.write(0x8020, *p++); break;
      }
    }
25
    req.assign( 0 );      // Release bus
  }

  // Bus slave message receive
30 void recv( addr a, void *data, int size ) {
    short *p;

    ordy.assign( 1 );     // Synchronization
    for ( p = data; size > 0; size -= 2 ) {
35    *p++ = p.read( 0x800c );
    }
    ordy.assign( 0 );     // Synchronization
  }
};
```

(b) Application layer.

Listing 21: Bus adapter with arbitration.

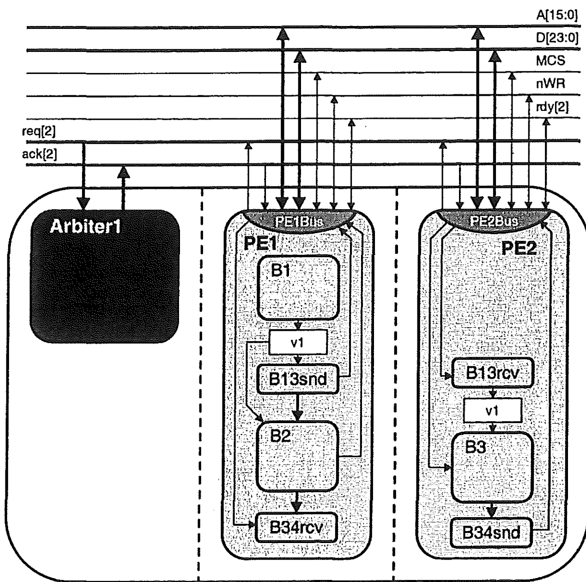


Figure 16: Communication model with arbiter.

The two PEs communicate with the arbiter component *Arbiter1* via two request lines *reqX* and two acknowledge lines *ackX*. As part of its application layer (Listing 21(b)), a PE's first action is to request bus access as a master in the *send()* method by raising its *req* line (line 16). It then waits until it is granted access by the arbiter through the corresponding *ack* line (line 17) before performing the actual data transfers. Finally, the sending PE releases the bus again at the end of the transfer (line 26).

For synchronization, the PEs communicate via two ready lines. Each *rdyX* line signals whether the corresponding PE is ready to receive data. Similar to synchronization in the original communication model example (see Section 4.3.1), the receiving PE drives its outgoing *rdy* line in its application layer *recv()* method while the sending PE blocks on the ready signal coming in from the other, receiving PE.

The communication model instantiates an arbiter component *Arbiter1* and includes it in the set of PEs. The arbiter (Listing 22) receives requests from and grants bus access to the PEs. In an endless loop, the arbiter checks for incoming requests and grants them on a first-come, first-serve basis by sending out *ack* signals. It then waits for the release of the bus as signaled by the PE before continuing to process requests. In case of requests that come in simultaneously, *PE1* has priority over *PE2* in this simple example. Due to the sequential nature of the arbiter, a total order is created among the events on the arbiter's ports, guaranteeing that only one PE is granted access at any given time.

```
   behavior Arbiter( ISignal req1, ISignal req2,
                     OSignal ack1, OSignal ack2 )
   {
     void main(void)
 5   {
       ack1.assign( 0 ); ack2. assign( 0 );

       while( true ) {
         // Priority 1: request from PE1?
10       if( req1.val() ) {
           ack1.assign ( 1 );    // Acknowledge
           req1.waitval( 0 );    // Wait for release
           ack1.assign ( 0 );    // Release bus
         }
15       // Priority 2: request from PE2?
         else if( req2.val() ) {
           ack2.assign ( 1 );    // Acknowledge
           req2.waitval( 0 );    // Wait for release
           ack2.assign ( 0 );    // Release bus
20       }
         // Wait for request
         else {
           wait( req1, req2 );
         }
25     }
     }
   };
```

Listing 22: Arbiter component model.

4.6 Timing

As part of the architecture model, scheduling of behaviors created a total order inside each PE (see Section 3.5). Hence, there is also a total order of events generated at the ports of each PE. In case of a single bus master (i.e. a single driver), this guarantees a total order among the transactions on that bus. In all other cases, arbitration, either statically or dynamically as explained in Section 4.5, will create a total order of bus transactions. Therefore, transactions on each bus in the communication model are totally ordered.

Furthermore, the communication model introduces the concept of time for the communication among the PEs in the system. As shown in Section 4.2, the protocol layers of the bus adapters are annotated with *waitfor()* statements for estimated protocol delays on the target PE. Similarly, the application layer methods can be annotated with timing information based on estimated or budgeted execution delays for the application layer code. As a result, the order of events on the system busses is further refined beyond the order imposed by the sequential PEs (including arbiters) driving the busses.

All together, the communication model provides a timing-accurate description of the interaction between PEs at the system level. From the system's perspective, the bus-functional PE models accurately describe each PE's behavior as seen at its bus interface. Therefore, the communication model allows to validate the order and functionality of the system at the level of PEs communicating via wires.

4.7 Summary

The communication model is the output of the system-level design process and the hand-off to the backend process. It reflects the structure of the system architecture consisting of computation running on PEs and communication over busses. The PEs in the communication model specify the computation and communication behavior to be synthesized into PE microarchitectures in the backend process. The communication model is a timed model in terms of computation and communication. Leaf behaviors and bus adapters are annotated with estimated or projected execution times on the target PE. The backend process will then further refine time into a cycle-accurate model.

In summary, compared to the properties of the architecture model presented in Section 3.7, the properties of the communication model are:

(a) At the top level of the behavior hierarchy, the PE structure is modeled as a parallel composition of non-terminating PE behaviors.

(b) PE behaviors communicate via shared, bit-true variables representing system bus wires.

(c) Bus adapters inside the PEs implement message-passing semantics by driving and sampling the wires of the bus according to the bus protocol.

(d) Behaviors inside different PEs communicate by sending and receiving messages via the PE's bus adapters.

(e) Computation in the leaf behaviors and communication functionality in the bus adapters are annotated with estimated or projected execution times on their target PE.

In terms of behaviors executing inside each of the PEs, the communication model inherits the respective properties (computation functionality, storage, parallelism, scheduling) from the architecture model. Also note that with respect to the properties of the communication model, special PEs like IPs, memories, transducers, or arbiters are no different from the other, general-purpose PEs. As part of the backend process, the implementation of the functionality inside each PE will feed into different flows depending on the type of the PE.

5 Implementation Model

The implementation model is the result of scheduling the functionality mapped onto the PEs (both, computation and communication functionality) into register transfers per clock cycle. Therefore, the implementation model is a cycle-accurate model at the register-transfer level.

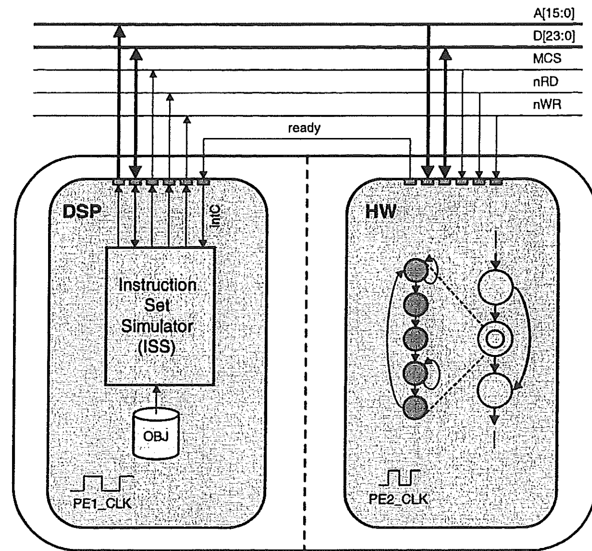


Figure 17: Implementation model.

```
behavior Design()
{
  // System bus wires
  bit[15:0] A;                          // address
5 bit[23:0] D;                          // data
  CSignal MCS, nRD, nWR, ready;         // control

  // PE1 = Processor (DSP)
  DSP pe1( A, D, MCS, nRD, nWR, ready );
10
  // PE2 = Custom HW
  HW  pe2( A, D, MCS, nRD, nWR, ready );

  void main(void) {
15  par {
      pe1.main(); pe2.main();
    }
  }
};
```

Listing 23: Implementation model.

For each PE, the implementation model defines the datapath, the control logic and the clock frequency at which the component runs. In general, the implementation model requires allocation of a datapath, binding of operations, variables, and transfers onto functional units, registers/memories and busses, and the scheduling of register-transfers into clock cycles.

For custom hardware PEs, high-level synthesis creates the implementation model of the hardware PE from the code of the behaviors and adapters inside the PE behavior of the communication model. For programmable processors, the code of the behaviors in the communication model is converted into C code and compiled into assembly code to create the implementation model.

Figure 17 and Listing 23 show the top level of the im-

plementation model for the example design. In this example, *PE1* is implemented as a digital signal processor *DSP* and *PE2* is implemented as a custom hardware PE *HW*. As specified by the communication model, the two components communicate via a bus with 24-bit wide data, 16-bit wide address and four control lines.

The implementation model supports two views of the PEs in the design: a behavioral RTL view and a structural RTL view [6]. In both cases, the steps of allocation, binding and scheduling are required to derive the implementation model. The difference is that the behavioral RTL view does not explicitly represent the datapath architecture and the binding information. However, it corresponds closely to the original C code in the communication model. The structural RTL view, on the other hand, explicitly describes the structure of data path plus control unit. Therefore, structural RTL is closer to the implementation and forms the immediate input to logic synthesis.

5.1 Behavioral RTL

Behavioral RTL specifies the operations performed in each clock cycle without explicitly modeling the units in the PE's datapath. Instead, operations in each cycle are described at the C level. Therefore, behavioral RTL is close to the original, sequential C code. Essentially, behavioral RTL is obtained by scheduling the operations in the C code into clock cycles.

Depending on the type of PE, different styles are needed for the implementation models of the PEs at the behavioral RTL level. For programmable processors, the operations performed in each clock cycle are defined by the assembly code compiled for that PE. On the other hand, for custom hardware PEs the operations in each clock cycle can be explicitly modeled.

5.1.1 Custom Hardware

Listing 24 shows the behavioral RTL code for the custom hardware PE in the implementation model of the example. At the top level, the PE behavior *HW* (Listing 24(b)) remains largely unchanged from the communication model (compare to *PE2* in Listing 13(a)). The *HW* behavior instantiates the bus adapter and the group of subbehaviors mapped onto the custom hardware PE, connects them via variables and interfaces, and executes the subbehaviors in the sequence determined during scheduling.

However, leaf behaviors and bus adapters in the *HW* behavior replaced with refined FSMD models of their state machine implementation. For example, the behavioral RTL code for a leaf behavior *B3* is outlined in Listing 24(a). The code describes the behavior as a finite state machine with datapath (FSMD) model. The FSMD model

```
   behavior FSMD3( in type1 v1, IBus if )
   {
     void main(void) {
       type2 v2;
 5
       // State variable
       enum { S0, S1, S2, ... , Sn } state;
       state = S0;

10     // State machine
       while( state != Sn ) {
         switch ( state )
         {
           ...
15
           case Si:
             v1 += v2;     // datapath func.
             if( v1 )      // next state func.
               state = Si+1;
20           else
               state = Sj;
             break;
           ...
                           // Superstate:
25         case Sj:        // call bus receive FSMD
             bus.recv( C2, &v2, sizeof(v2) );
             state = Sj+1;
             break;
                           // Superstate:
30         case Sj+1: // call f3() FSMD
             f3( v1, v2, ... );
             state = Sj+2;
             break;
           ...
35       }

         // Clock period delay
         waitfor( HW_CLOCK_PERIOD );
       }
40   }
   };
```

(a) FSMD leaf behavior.

```
   behavior HW( in    bit[15:0] A,
                inout bit[31:0] D,
                ISignal MCS,
                ISignal nRD,
 5              ISignal nWR,
                OSignal ready )
   {
     // Bus interface logic FSMD
     HWBus bus( A, D, MCS, nRD, nWR, ready );
10
     type1 v1;

     // FSMD models of leaf behaviors
     FSMD13Rcv b13rcv( bus, v1 );
15   FSMD3     b3    ( v1, bus );
     FSMD34Snd b34snd( bus );

     void main(void) {
       b13rcv.main();
20     b3.main();
       b34snd.main();
     }
   };
```

(b) PE behavior.

Listing 24: Custom hardware behavioral RTL model.

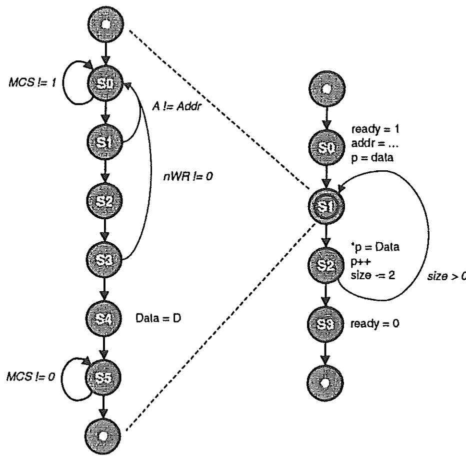


Figure 18: Custom hardware bus interface FSMD.

is the result of scheduling *B3*'s operations into clock cycles and converting the code into states and transitions.

The state machine is modeled by a *state* variable and a `switch-case` statement inside a loop. The state machine starts at state S_0 and runs until the end state S_n is reached. Each case represents a state and specifies the operations and transitions executed in that state. Each state in turn corresponds to one clock cycle. The timing and delay of the PE clock is modeled by inserting a `waitfor()` statement which describes the state delay based on the PE's clock period.

The statements in each state are taken from the original C code of the leaf behavior and represent the datapath operations (register transfers) performed in the corresponding clock cycle. The original control flow in the C code is transformed into state transitions in the FSMD model. In each state, the next state is determined, possibly conditionally as in the case of state S_i, by assigning a new value to the state variable *state*. Variables inside behaviors model the local storage of the component. Depending on the type of storage a variable will be bound to, variable accesses represent reads or writes of the corresponding register file, memory, ROM, etc.

In general, FSMDs can be hierarchical. Superstates are modeled by including function or method calls in a state as shown in states S_j (bus adapter method call) and S_{j+1} (regular function call). While being in a superstate, the FSMD of the callee is executed. Upon entering a hierarchical state, control is transfered to the first state of the sub-FSMD. Control returns to the parent superstate when the end state of the sub-FSMD is reached. For example, the state S_j is a superstate which calls the bus adapter's *recv()* FSMD to transfer a message over the bus.

Similar to the computation in the leaf behaviors, the bus adapter functionality is scheduled into clock cycles and de-

```
channel HWprotocol( in     bit[15:0] A,
                    inout  bit[23:0] D,
                    ISignal MCS,
                    ISignal nRD,
                    ISignal nWR )
  implements IProtocol
{
  // Bus slave read ( answer to bus write )
  bit[23:0] read( bit[15:0] addr )
  {
    bit[23:0] data;

    // State variable
    enum{ S0, S1, S2, S3, S4, S5, S6 } state;
    state = S0;

    // State machine
    while( state != S6 )
    {
      switch( state )
      {
        case S0:    // sample MCS
          if( MCS.val() == 1 ) state = S1;
          break;

        case S1:    // sample address
          state = S2;
          if( A != addr ) state = S0;
          break;

        case S2:    // wait state
          state = S3;
          break;

        case S3:    // sample nWR
          state = S4;
          if( nWR.val() != 0 ) state = S0;
          break;

        case S4:    // sample data
          data = d;
          state = S5;
          break;

        case S5:    // sample MCS
          if( MCS.val() == 0 ) state = S6;
          break;
      }

      // State delay = clock period
      waitfor( HW_CLOCK_PERIOD );
    }

    return data;
  }

  // Bus slave write ( answer to bus read)
  void write( bit[15:0] addr, bit[23:0] data )
  {
    // Omitted ...
    ...
  }
};
```

(a) Protocol layer.

Listing 25: Custom hardware bus interface FSMD.

```
channel HWBus( in    bit[15:0] A,
               inout bit[31:0] D,
               ISignal MCS,
               ISignal nRD,
               ISignal nWR,
               OSignal ready )
  implements IBus
{
  // Protocol layer FSMD
  HWProtocol protocol( A, D, MCS, nRD, nWR );

  // Receive message FSMD
  void recv( addr a, void *data, int size )
  {
    bit[16] Addr;
    short   *p;

    // State variable
    enum { S0, S1, S2, S3, S4 } state;
    state = S0;

    // State machine
    while( state != S4 )
    {
      switch( state )
      {
        // default next state
        state++;

        case S0:
          ready.assign( 1 ); // assert ready
          switch( a ) {      // load addr reg.
            case CB13:
              Addr = 0x8005;  break;
            case C2:
              Addr = 0x8020;  break;
          }
          p = data;          // init loop
          break;

        case S1:    // receive data item
          Data = protocol.recv( Addr );
          break;

        case S2:
          *p++ = Data;         // write into mem.
          if( (size -= 2) > 0 ) state = S1;
          break;               // loop condition

        case S3:
          ready.assign( 0 ); // deassert ready
          break;
      }

      // State delay
      waitfor( HW_CLOCK_PERIOD );
    }
  }

  // Send message FSMD
  void send( addr a, void *data, int size )
  {
    // Omitted
    ...
  }
};
```

(b) Application layer.

Listing 25 (continued): Custom hardware bus interface FSMD.

scribed as an FSMD model. Listing 25 shows the behavior RTL code of the FSMD models for application and protocol layer of the *HW* bus interface. The model for the application layer (Listing 25(b) is similar to the FSMD model of the leaf behaviors model shown above. Each method of the application layer is implemented as a FSMD by scheduling operations into states and transitions. The protocol layer, on the other hand, is a simple FSM driving and sampling the output and input wires of the bus, respectively. The protocol FSM sits directly at the ports of the PEs and implements the bus protocol in terms of the PE's internal clock.

5.1.2 Programmable Processors

In contrast to custom hardware, the behavioral RTL model of programmable processors is based on the execution of assembly output generated by compiling the communication model PE behavior code. Therefore, the behavioral RTL model for programmable components implements an instruction set simulation (ISS) of the assembly code.

Assembly code is generated from the communication model by transforming the behavior hierarchy into a corresponding C function call hierarchy and compiling the resulting C program for the target processor. The C program is then linked against a custom or standard operating system kernel which implements dynamic scheduling, synchronization, communication, and so on.

Bus drivers including interrupt handlers, etc. are generated from the application and protocol layers of the bus adapters. In general, a programmable processor can be connected to the system bus through its builtin bus interface or via a set of general-purpose ports. In the former case, the protocol layer is usually implemented in hardware as part of the processor's microarchitecture. In those cases, the instruction-set architecture of the processor will provide special instructions for bus transfers and usually each protocol layer method directly translates into a single assembly instruction. In the latter case, the protocol layer is implemented in assembly code as a sequence of I/O instructions.

In both cases, application layers are translated into assembly routines that call the protocol layer routines. The mapping of bus wires to processor ports will also determine the implementation of synchronization in the application layer. Depending on whether a synchronization input is connected to an interrupt line or a general purpose input port, an interrupt-driven or polling-based scheme is implemented. In the former case, interrupt handlers that communicate with the application layer routines are generated. All together, interrupt handlers and application/protocol layer routines become the bus drivers of the operating system kernel that is linked to the compiled C program in order to get the final executable.

```
// ISS C/C++ interface
#include "iss.h"

// Instruction Set Simulator (ISS) for DSP
5 behavior DSP( out    bit[15:0] A,
                inout  bit[23:0] D,
                OSignal MCS,
                OSignal nRD,
                OSignal nWR,
10              ISignal intC )
  {
    // DSP bus interface model
    PE1Protocol if( A, D, MCS, nRD, nWR );

15  void main(void)
    {
      // initialize ISS, load program
      iss.startup();
      iss.load("a.out");
20
      // run simulation
      for( ; ; )
      {
        // drive ISS input
25      iss.intC = intC.val();

        // run DSP cycle
        iss.exec();

30      // MOVEM instruction?
        if( iss.IR == MOVEM_RD ) {
          // Simulate external bus read cycle
          iss.DR = if.read( iss.AR );
        }
35      else if( iss.IR == MOVEM_WR ) {
          // Simulate external bus write cycle
          if.write( iss.AR, iss.DR );
        }
        else {
40        // Simulate DSP clock period
          waitfor( DSP_CLOCK_PERIOD );
        }
      }
    }
45 };
```

Listing 26: DSP instruction set simulator (ISS) model.

Different levels of instruction set simulation of the executable are possible. In a compiled instruction set simulation, each assembly instruction is translated into a set of C statements that perform updates of a simulated processor state cycle by cycle [7]. This C code is then wrapped into a behavior and plugged into the implementation model as PE behavior for the processor.

On the other hand, for interpreted instruction set simulation, the behavioral RTL model of the programmable PE consists of a program that reads and interprets the instruction stream. Any instruction-set simulator (ISS) that supports a C-based API can be hooked into the SpecC model. As shown in Listing 26, the external ISS is wrapped into a SpecC behavior that calls the ISS routines via the ISS API (line 2). The core of the processor behavior is a loop which simulates one clock cycle per iteration. The *exec()* function fetches and decodes instructions, performs the corresponding operations in each clock cycle, and updates the simulated processor state accordingly.

In both cases of compiled or interpreted simulation, the simulation model of the processor drives and samples the ports of the PE behavior based on the instruction stream executed. For each I/O instruction, the PE ports are updated from the processor state and vice versa. For example, in the model from Listing 26, the simulated *intC* input of the processor is updated in each cycle by sampling the corresponding input port of the PE behavior (line 25).

Any special bus interface hardware of the processor is simulated through corresponding bus adapters. For example, the model in Listing 26 instantiates the *PE1Protocol* bus adapter to simulate the DSP's bus master interface. For every `MOVEM` instruction encountered in the instruction stream, the corresponding method in the bus adapter is called. The bus adapter simulates the driving and sampling of bus wires in the implementation model as specified by the timing diagrams of the processor hardware for that I/O instruction. Note that this is equivalent to the protocol layer in the bus adapters from the communication model (see Section 4.2).

5.2 Structural RTL

A structural RTL view of the PEs in the implementation model accurately reflects the microarchitecture internal to the system PEs. As a result of the high-level synthesis process, structural RTL explicitly models the allocation of RTL components, the scheduling of register transfers into clock cycles, and the binding of operations, variables and assignments to functional units, register/memories and PE busses. The result is an RTL netlist of sequential and combinatorial logic inside each PE. Structural RTL is the input to traditional logic synthesis which in turn will derive a gate-level netlist from the netlist of units inside each PE.

A structural RTL representation is usually used for custom hardware PEs which have to be synthesized further. Since structural RTL represents the hardware microarchitecture of PEs, at this level there is no difference between models for custom hardware or programmable processors. In both cases, structural RTL is a netlist of functional units, busses, memories and registers. However, in case of predesigned components (IPs, programmable off-the-shelf processors, memories) the level of detail for further synthesis of the hardware is not needed. A more abstract behavioral RTL model is sufficient for effective simulation.

Figure 19 and Listing 27 show the structural RTL view of the custom hardware PE in the example design. The system interface of the component remains unchanged from the communication model or the behavioral RTL view.

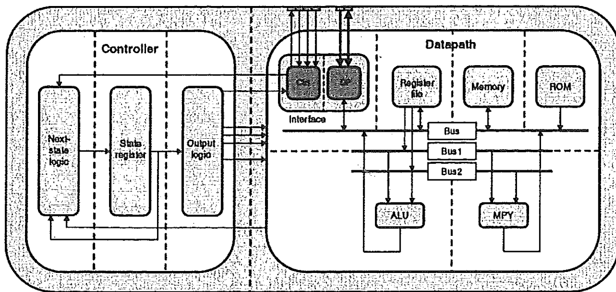


Figure 19: Structural RTL model for custom hardware.

```
behavior HW( in      bit[15:0] A,
             inout   bit[23:0] D,
             ISignal MCS,
             ISignal nRD,
             ISignal nWR,
             OSignal ready )
{
  // Clock signal
  event      clk;

  // Status lines
  bit[15:0]  status;
  event      _status;

  // Control lines
  bit[117:0] ctrl;
  event      _ctrl;

  // Clock generator
  ClkGen     cg( clk );

  // Control
  Control    ctrl( clk,
                   status, _status,
                   ctrl,   _ctrl );

  // Datapath
  Datapath dp( clk,
               A, D, MCS, nRD, nWR, ready,
               ctrl,   _ctrl,
               status, _status );

  // Parallel (structural) composition
  void main(void)
  {
    par {
      cg.main();
      ctrl.main();
      dp.main();
    }
  }
};
```

Listing 27: Structural RTL model for custom hardware.

```
behavior ClockGen( out event clk )
{
  void main(void)
  {
    while( 1 )
    {
      waitfor( HW_CLOCK_PERIOD );
      notify( clk );
    }
  }
};
```

Listing 28: Clock generator.

However, the component itself is now implemented as a purely structural netlist of subcomponents. Subcomponents are represented by subbehaviors. All subbehaviors operate in parallel and are connected via busses and/or wires. Each bus or set of wires is associated with an event that signals a change of the values on the wires.

At the top level of the custom hardware, the PE is comprised of a clock generator *ClkGen*, a controller *Control*, and a datapath *Datapath*. Controller and datapath are connected by a set of control and status lines. Both are driven by the PE's clock signal *clk*.

In general, subcomponents themselves can be further decomposed hierarchically. At each level, however, the same purely structural netlist of behaviors running concurrently and being connected through wires is repeated in the structural RTL view. Therefore, if the hierarchy is flattened all the leaf behaviors will operate in parallel and communicate via wires and corresponding events.

Leaf behaviors of the structural RTL hierarchy model registers and combinatorial logic between registers. Leaf behaviors are reactive, i.e. they are continuously reacting to events on their inputs and create events at their outputs in turn. Structural RTL models hardware as a reactive system with a set of non-terminating processes operating concurrently [8].

5.2.1 Clock

Register transfers cycles are controlled by the common clock event. The clock generator shown in Listing 28 generates the clock by issuing clock events according to the PE's local clock frequency. In an endless loop, a clock event is generated every clock period.

5.2.2 Controller

As shown in Listing 29, the main control unit is hierarchically decomposed into state register, next-state logic and output logic. As previously described for the top level of the PE, subcomponents operate concurrently and are connected through wires and corresponding events.

```
behavior Control ( in  event       clk,
                   in  bit[15:0]   status,
                   in  event       _status,
                   out bit[117:0]  ctrl,
5                  out event       _ctrl )
{
  bit[21:0] state, nextstate;
  event     _state;

10 // State register
  StateReg sr( clk, nextstate,
               state, _state );

  // Output logic
15 OutputLogic ol( state, _state,
                  ctrl,  _ctrl );

  // Next state logic
  NextStateLogic nsl( state,  _state,
20                    status, _status,
                      nextstate );

  void main(void) {
    par {
25    sr.main(); ol.main(); nsl.main();
    }
  }
};
```

Listing 29: Custom hardware controller.

The behavior modeling the state register is shown in Listing 30. The state register continuously reacts to clock events. In an endless loop, the state register is updated with the new value at the input whenever a clock event is received. The corresponding new value is assigned to the current state output and an output event signaling a value change is generated.

The output logic combinatorial block that generates the control signals from the current state value is shown in Listing 31. The output logic is a reactive, non-terminating behavior that is sensitive to changes on the current state value, i.e. the state register output. Whenever the state value changes an evaluation cycle of the output logic is triggered, control values are reevaluated and corresponding control update events generated.

Finally, the next state logic of the controller, shown in Listing 32, is organized similar to the output logic. It generates the next state value from the current state register output and the status output of the datapath. Hence, the non-terminating next-state logic is sensitive to changes of either the state value or the status inputs, and an evaluation cycle is triggered whenever a state or status update event is received.

5.2.3 Datapath

The main datapath of the example design is shown in Listing 33. At the top level, the datapath is hierarchically com-

```
behavior StateReg ( in  event      clk,
                    in  bit[21:0]  next,
                    out bit[21:0]  cur,
                    out event      _cur ) {
5  bit[21:0] state;

  void main(void) {
    while( 1 ) {
      wait( clk );
10    state = next;
      cur   = state;
      notify( _cur );
    }
  }
15 };
```

Listing 30: State register.

```
behavior OutputLogic( in  bit[21:0]   state,
                      in  event       _state,
                      out bit[117:0]  ctrl,
                      out event       _ctrl ) {
5 void main(void) {
    while( 1 ) {
      wait( _state );   // sensitivity
      switch( state ) {
        ...
10      case Si:
          ctrl = "000... 10b";
          break;
        ...
        case Sj:   // send recv() start signal
15        ctrl = "100... 00b";
          break;
        ...
      }
      notify( _ctrl );
20  }
  }
};
```

Listing 31: Output logic.

```
behavior NextStateLogic( in  bit[21:0]  state,
                         in  event      _state,
                         in  bit[15:0]  status,
                         in  event      _status,
5                        out bit[21:0]  next ) {
  void main(void) {
    while( 1 ) {
      wait( _state, _status );   // sensitivity
      switch( state ) {
10      ...
        case Si:
          next = Si+1;
          if( ! status[7] ) next = Sj;
          break;
15      ...
        case Sj:   // wait for recv() done
          if( status[15] ) next = Sj+1;
          break;
        ...
20    }
    }
  }
};
```

Listing 32: Next state logic.

posed as a structural netlist of the different datapath components connected through internal busses. The example shown here is a typical datapath with RAM, ROM, register file, functional units, and three busses. The datapath's subcomponents are then in turn modeled following standard structural RTL design guidelines as outlined in the previous sections of this report. In general, sub-components are register/storage units driven by the clock event, combinatorial logic blocks sensitive to input changes or a hierarchically composition thereof.

The datapath contains a bus interface module *IF*. The bus interface module is itself an FSMD (see Section 5.2.4) that implements message-passing communication over the PE bus. It connects to the PE's bus ports and communicates with the main controller through parts of the control and status vectors. In addition, the bus interface FSMD can exchange data with the memory via the data bus. For this purpose, the bus interface can directly control the memory via the *ifctrl* lines connected to the memory's control inputs.

5.2.4 Bus Interface

The bus interface unit implements the protocol and application layers of the bus communication. It drives the bus wires and executes the correct protocol timing to transfer data words over the bus.

Listing 34 shows the top level of the bus interface. The bus interface is a separate FSMD that communicates with the main state machine through a set of control wires and a common internal data bus. Similar to the top level FSMD for the custom hardware PE, the bus interface module is decomposed into a controller and a datapath communicating via control and status lines. Incoming *start* control signals trigger execution of the bus interface state machine and determine what kind of bus transfer to perform (i.e. message send or message receive). Upon finishing the transfer, the bus interface sends a *done* status signal to the main controller. Data items are exchanged between the bus interface and the main datapath through the data bus and a set of *memctrl* lines that allow the bus interface FSMD to act as a DMA controller for the PE's memory.

The bus interface controller is shown in Listing 35. In this example, the state register, output logic and next-state logic are merged into one combined model. In each clock cycle, as dictated by the sensitivity to the clock event, the non-terminating behavior assigns new values to its outputs and updates the internal state value depending on the current state and the inputs.

The bus interface state machine implements the protocols for sending and receiving messages over the bus wires in one single state machine. The cross-product of the send and receive state machines is optimized to minimize the

```
behavior Datapath ( in     event       clk,
                    in     bit[15:0]   A,
                    inout  bit[23:0]   D,
                    ISignal            MCS,
 5                  ISignal            nRD,
                    ISignal            nWR,
                    OSignal            ready,
                    in     bit[117:0]  ctrl,
                    in     event       _ctrl,
10                  out    bit[15:0]   status,
                    out    event       _status )
{
  bit[1:0] ifctrl;
  event    _ifctrl;
15
  bit[31:0] bus,  bus1,  bus2;
  event     _bus, _bus1, _bus2;

  IF   if( clk,
20           A, D, MCS, nRD, nWR, ready,
             ctrl[117:116], _ctrl,
             bus,           _bus,
             ifctrl,        _ifctrl,
             status[15],    _status );
25
  ROM rom( clk, ctrl[115:94],
           bus, _bus );

  Mem mem( clk, ctrl[93:61] @ ifctrl[1:0],
30         bus, _bus );

  RF  rf ( clk, ctrl[60:30],
           bus, _bus,
           bus1, _bus1,
35         bus2, _bus2 );

  ALU alu( ctrl[29:15],  _ctrl,
           bus,          _bus,
           bus1,         _bus1,
40         bus2,         _bus2,
           status[14:8], _status );

  MPY mpy( ctrl[14:0],   _ctrl,
           bus,          _bus,
45         bus1,         _bus1,
           bus2,         _bus2,
           status[7:0],  _status );

  void main(void) {
50  par {
      if.main();
      rom.main();
      mem.main();
      rf.main();
55    alu.main();
      mpy.main();
    }
  }
};
60
```

Listing 33: Custom hardware datapath.

```
behavior IF( in     event      clk,
             in     bit[15:0]  A,
             inout  bit[23:0]  D,
             ISignal           MCS,
5            ISignal           nRD,
             ISignal           nWR,
             OSignal           ready,
             in     bit[1:0]   start,
             in     event      _start,
10           inout  bit[31:0]  bus,
             inout  event      _bus,
             out    bit[1:0]   memctrl,
             out    event      _memctrl,
             out    bit        done,
15           out    event      _done )
{
  // Control lines
  bit[3:0] ctrl;
  event    _ctrl;
20
  // Status lines
  bit      status;
  event    _status;

25 // Controller
  IFCtrl  ctrl( clk,
                MCS, nRD, nWR, ready,
                inp[1:0],  _start,
                status,    _status,
30              ctrl,      _ctrl,
                memctrl,   _memctrl,
                done,      _done );

  // Datapath (addr. & data reg.)
35 IFDP    dp( clk,
              A, D,
              ctrl,    _ctrl,
              bus,     _bus,
              status,  _status );
40
  // Parallel composition
  void main(void)
  {
    par {
45    ctrl.main();
      dp.main();
    }
  }
};
50
```

Listing 34: Bus interface hardware unit.

```
behavior IFCtrl( in event clk,
                 ISignal MCS, ISignal nRD,
                 ISignal nWR, OSignal ready,
                 in bit[1:0]   start,
5                in event      _start,
                 in bit        status,
                 in event      _status,
                 out bit[3:0]  ctrl,
                 out event     _ctrl,
10               out bit[1:0]  memctrl,
                 out event     _memctrl,
                 out bit       done,
                 out event     _done )
{
15 bit[3:0] state = 0;

  void main(void) {
    while( 1 ) {
      wait( clk );    // sensitivity
20
      done = 0;       // defaults
      state++;
      ctrl = "0000b";
      memctrl = "00b";
25
      switch( state ) {
        case 0:                  // wait for start
          if( start[0] ) state = 1;
          break;
30      case 1:
          ready.assign( 1 ); // assert ready
          break;
        case 2:                  // wait for MCS
          if( MCS.val() != 1 ) state = 2;
35        break;
        case 3:
          ctrl[0] = 1;           // sample A
          break;
        case 4:                  // address match?
40        if( ! status[0] ) state = 2;
          if( start[1] )    state = 9;
          break;
        case 5:                  // check nWR
          if( nWR.val() != 0 ) state = 2;
45        break;
        case 6:
          ctrl[1] = 1;           // sample D
          break;
        case 7:                  // wait for MCS
50        if( MCS.val() == 1 ) state = 7;
          break;
        case 8:
          memctrl[0] = 1;      // data -> mem
          ctrl[3:2]  = "11b";  // dec, check count
55        state      = 15;
          if( status[1] ) state = 2;
          break;
        ...
        case 15:
60        ready.assign( 0 ); // deassert ready
          done = 1;          // transfer done
          state = 0;         // back to start
          break;
      }
65    notify( _done, _ctrl, _memctrl );
    }
  }
};
```

Listing 35: Bus interface controller.

state space. The common state machine is triggered by an external `start` signal. After synchronization and address decoding, the transitions branch into the send or receive protocol depending on the corresponding control inputs. Both branches are joined at the end of the bus cycle and an external `done` signal is asserted.

The accompanying datapath (not shown) contains registers that connect to the external address and data busses. Driven by the controller output, the address and data registers are used to drive and sample the external busses. In addition, the data register connects to the PE's internal data bus in order to exchange data with the local memory. Finally, the bus interface datapath includes counters and comparators for loop control and address decoding.

5.3 Summary

At the top level, the implementation model is equivalent to the communication model (see Section 4.7). The system is a set of concurrent, non-terminating PEs communicating via busses and wires. Internally, on the other hand, PEs represented by the PE behaviors, are further refined and turned into a model of the PE's microarchitectures. The minimal requirement for the PEs in the communication model is that they provide a cycle-accurate description of events on their ports through a behavioral microarchitecture model. Alternatively, more detailed PE models can be used in the communication model, e.g completely structural RTL descriptions.

PE behaviors are interchangeable between communication and implementation model. This allows mixed-level simulations in which a cycle-accurate PE behavior is plugged into an otherwise bus-functional simulation of the design and vice versa. Therefore, different parts of the system can be simulation at different levels of detail, allowing to quickly validate isolated PE's, for example.

In summary, the implementation model is a cycle-accurate model of the system implementation of both, the communication between the PEs and the microarchitecture inside the PEs. In contrast to the bus-functional communication model, the computation inside the PEs is refined down to the register-transfer level. As a result of high-level synthesis of custom hardware and compilation of software for programmable components, the implementation model is the basis for further refinement down to the gate level through logic synthesis or instantiation of hard IP cores.

6 Summary and Conclusions

In this report, we presented and defined the four models of system design which are part of a system-level design methodology from specification down to implementation. The models vertically cover different levels of abstraction, gradually increasing the level of implementation detail as the design flow progresses from top to bottom.

The division of the design flow into four models supports rapid design space exploration by focusing on critical decisions at early stages and providing quick feedback. Unnecessary details are abstracted away at higher levels while important aspects are immediately visible. For example, for validation through simulation high-level models achieve fast simulation speeds while still providing feedback about the crucial aspects at each stage of the design process. Furthermore, model refinement requires only minimal modifications, allowing to leave large parts of the design untouched when exploring different implementations or moving between levels.

Having well-defined, formal models at each step of the design process is the basis for automated synthesis and refinement between the models. With the help of tools, lower-level models can be automatically generated from the model at the next higher level of abstraction based on a corresponding set of refinement rules and transformations. In addition, formal verification can be applied to check properties of the models or to verify equivalence of models at different levels. Therefore, the definition of the models enables fast system-level design exploration paired with a synthesis-based design flow.

References

[1] D. D. Gajski, R. Kuhn. "Guest editors introduction - New VLSI tools." *IEEE Computer*, pp. 11-14, 1983.

[2] D. D. Gajski et al. *SpecC: Specification Language and Design Methodology.* Kluwer Academic Publishers, 2000.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1992.

[4] Motorola, Inc., Semiconductor Products Sector, DSP Division. *DSP56600 16-bit Digital Signal Processor Family Manual*, DSP56600FM/AD, 1996.

[5] A. Gerstlauer, D. D. Gajski. *Communication Software Code Generation.* Technical Report ICS-TR-00-46, University of California, Irvine, August 2000.

[6] H. Lehr, D. D. Gajski. *Modeling Custom Hardware in VHDL.* Technical Report ICS-TR-99-29, University of California, Irvine, July 1999.

[7] J. Zhu, D. D. Gajski. "A Retargatable, Ultra-fast Instruction Set Simulator." In *Proceedings Design, Automation and Test in Europe*, 1999.

posed as a structural netlist of the different datapath components connected through internal busses. The example shown here is a typical datapath with RAM, ROM, register file, functional units, and three busses. The datapath's subcomponents are then in turn modeled following standard structural RTL design guidelines as outlined in the previous sections of this report. In general, sub-components are register/storage units driven by the clock event, combinatorial logic blocks sensitive to input changes or a hierarchically composition thereof.

The datapath contains a bus interface module *IF*. The bus interface module is itself an FSMD (see Section 5.2.4) that implements message-passing communication over the PE bus. It connects to the PE's bus ports and communicates with the main controller through parts of the control and status vectors. In addition, the bus interface FSMD can exchange data with the memory via the data bus. For this purpose, the bus interface can directly control the memory via the *ifctrl* lines connected to the memory's control inputs.

```
behavior Datapath ( in      event       clk ,
                    in      bit[15:0]   A,
                    inout   bit[23:0]   D,
                    ISignal             MCS,
                    ISignal             nRD,
                    ISignal             nWR,
                    OSignal             ready ,
                    in      bit[117:0]  ctrl ,
                    in      event       _ctrl ,
                    out     bit[15:0]   status ,
                    out     event       _status )
{
  bit[1:0] ifctrl ;
  event    _ifctrl ;

  bit[31:0] bus ,  bus1 ,  bus2;
  event     _bus , _bus1 , _bus2;

  IF   if ( clk ,
            A, D, MCS, nRD, nWR, ready ,
            ctrl [117:116],  _ctrl ,
            bus ,            _bus ,
            ifctrl ,         _ifctrl ,
            status [15],     _status );

  ROM rom( clk ,  ctrl [115:94],
           bus ,  _bus );

  Mem mem( clk ,  ctrl [93:61] @ ifctrl [1:0],
           bus ,  _bus );

  RF   rf  ( clk ,  ctrl [60:30],
             bus ,  _bus ,
             bus1 , _bus1 ,
             bus2 , _bus2 );

  ALU alu ( ctrl [29:15],   _ctrl ,
            bus ,           _bus ,
            bus1 ,          _bus1 ,
            bus2 ,          _bus2 ,
            status [14:8],  _status );

  MPY mpy( ctrl [14:0],   _ctrl ,
           bus ,          _bus ,
           bus1 ,         _bus1 ,
           bus2 ,         _bus2 ,
           status [7:0],  _status );

  void main( void ) {
    par {
      if . main ();
      rom. main ();
      mem. main ();
      rf . main ();
      alu . main ();
      mpy. main ();
    }
  }
};
```

Listing 33: Custom hardware datapath.

5.2.4 Bus Interface

The bus interface unit implements the protocol and application layers of the bus communication. It drives the bus wires and executes the correct protocol timing to transfer data words over the bus.

Listing 34 shows the top level of the bus interface. The bus interface is a separate FSMD that communicates with the main state machine through a set of control wires and a common internal data bus. Similar to the top level FSMD for the custom hardware PE, the bus interface module is decomposed into a controller and a datapath communicating via control and status lines. Incoming *start* control signals trigger execution of the bus interface state machine and determine what kind of bus transfer to perform (i.e. message send or message receive). Upon finishing the transfer, the bus interface sends a *done* status signal to the main controller. Data items are exchanged between the bus interface and the main datapath through the data bus and a set of *memctrl* lines that allow the bus interface FSMD to act as a DMA controller for the PE's memory.

The bus interface controller is shown in Listing 35. In this example, the state register, output logic and next-state logic are merged into one combined model. In each clock cycle, as dictated by the sensitivity to the clock event, the non-terminating behavior assigns new values to its outputs and updates the internal state value depending on the current state and the inputs.

The bus interface state machine implements the protocols for sending and receiving messages over the bus wires in one single state machine. The cross-product of the send and receive state machines is optimized to minimize the

```
behavior IF( in     event      clk,
             in     bit[15:0]  A,
             inout  bit[23:0]  D,
             ISignal           MCS,
             ISignal           nRD,
             ISignal           nWR,
             OSignal           ready,
             in     bit[1:0]   start,
             in     event      _start,
             inout  bit[31:0]  bus,
             inout  event      _bus,
             out    bit[1:0]   memctrl,
             out    event      _memctrl,
             out    bit        done,
             out    event      _done )
{
  // Control lines
  bit[3:0] ctrl;
  event    _ctrl;

  // Status lines
  bit      status;
  event    _status;

  // Controller
  IFCtrl  ctrl( clk,
                MCS, nRD, nWR, ready,
                inp[1:0],  _start,
                status,    _status,
                ctrl,      _ctrl,
                memctrl,   _memctrl,
                done,      _done );

  // Datapath (addr. & data reg.)
  IFDP    dp( clk,
              A, D,
              ctrl,    _ctrl,
              bus,     _bus,
              status,  _status );

  // Parallel composition
  void main(void)
  {
    par {
      ctrl.main();
      dp.main();
    }
  }
};
```

Listing 34: Bus interface hardware unit.

```
behavior IFCtrl( in event clk,
                 ISignal MCS, ISignal nRD,
                 ISignal nWR, OSignal ready,
                 in bit[1:0]  start,
                 in event     _start,
                 in bit       status,
                 in event     _status,
                 out bit[3:0] ctrl,
                 out event    _ctrl,
                 out bit[1:0] memctrl,
                 out event    _memctrl,
                 out bit      done,
                 out event    _done )
{
  bit[3:0] state = 0;

  void main(void) {
    while( 1 ) {
      wait( clk );        // sensitivity

      done = 0;           // defaults
      state++;
      ctrl = "0000b";
      memctrl = "00b";

      switch( state ) {
        case 0:                         // wait for start
          if( start[0] ) state = 1;
          break;
        case 1:
          ready.assign( 1 );  // assert ready
          break;
        case 2:                         // wait for MCS
          if( MCS.val() != 1 ) state = 2;
          break;
        case 3:
          ctrl[0] = 1;                  // sample A
          break;
        case 4:                         // address match?
          if( ! status[0] ) state = 2;
          if( start[1] )    state = 9;
          break;
        case 5:                         // check nWR
          if( nWR.val() != 0 ) state = 2;
          break;
        case 6:
          ctrl[1] = 1;                  // sample D
          break;
        case 7:                         // wait for MCS
          if( MCS.val() == 1 ) state = 7;
          break;
        case 8:
          memctrl[0] = 1;         // data -> mem
          ctrl[3:2]  = "11b";     // dec, check count
          state      = 15;
          if( status[1] ) state = 2;
          break;
        ...
        case 15:
          ready.assign( 0 );  // deassert ready
          done = 1;           // transfer done
          state = 0;          // back to start
          break;
      }
      notify( _done, _ctrl, _memctrl );
    }
  }
};
```

Listing 35: Bus interface controller.

state space. The common state machine is triggered by an external `start` signal. After synchronization and address decoding, the transitions branch into the send or receive protocol depending on the corresponding control inputs. Both branches are joined at the end of the bus cycle and an external `done` signal is asserted.

The accompanying datapath (not shown) contains registers that connect to the external address and data busses. Driven by the controller output, the address and data registers are used to drive and sample the external busses. In addition, the data register connects to the PE's internal data bus in order to exchange data with the local memory. Finally, the bus interface datapath includes counters and comparators for loop control and address decoding.

5.3 Summary

At the top level, the implementation model is equivalent to the communication model (see Section 4.7). The system is a set of concurrent, non-terminating PEs communicating via busses and wires. Internally, on the other hand, PEs represented by the PE behaviors, are further refined and turned into a model of the PE's microarchitectures. The minimal requirement for the PEs in the communication model is that they provide a cycle-accurate description of events on their ports through a behavioral microarchitecture model. Alternatively, more detailed PE models can be used in the communication model, e.g completely structural RTL descriptions.

PE behaviors are interchangeable between communication and implementation model. This allows mixed-level simulations in which a cycle-accurate PE behavior is plugged into an otherwise bus-functional simulation of the design and vice versa. Therefore, different parts of the system can be simulation at different levels of detail, allowing to quickly validate isolated PE's, for example.

In summary, the implementation model is a cycle-accurate model of the system implementation of both, the communication between the PEs and the microarchitecture inside the PEs. In contrast to the bus-functional communication model, the computation inside the PEs is refined down to the register-transfer level. As a result of high-level synthesis of custom hardware and compilation of software for programmable components, the implementation model is the basis for further refinement down to the gate level through logic synthesis or instantiation of hard IP cores.

6 Summary and Conclusions

In this report, we presented and defined the four models of system design which are part of a system-level design methodology from specification down to implementation. The models vertically cover different levels of abstraction, gradually increasing the level of implementation detail as the design flow progresses from top to bottom.

The division of the design flow into four models supports rapid design space exploration by focusing on critical decisions at early stages and providing quick feedback. Unnecessary details are abstracted away at higher levels while important aspects are immediately visible. For example, for validation through simulation high-level models achieve fast simulation speeds while still providing feedback about the crucial aspects at each stage of the design process. Furthermore, model refinement requires only minimal modifications, allowing to leave large parts of the design untouched when exploring different implementations or moving between levels.

Having well-defined, formal models at each step of the design process is the basis for automated synthesis and refinement between the models. With the help of tools, lower-level models can be automatically generated from the model at the next higher level of abstraction based on a corresponding set of refinement rules and transformations. In addition, formal verification can be applied to check properties of the models or to verify equivalence of models at different levels. Therefore, the definition of the models enables fast system-level design exploration paired with a synthesis-based design flow.

References

[1] D. D. Gajski, R. Kuhn. "Guest editors introduction - New VLSI tools." *IEEE Computer*, pp. 11-14, 1983.

[2] D. D. Gajski et al. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1992.

[4] Motorola, Inc., Semiconductor Products Sector, DSP Division. *DSP56600 16-bit Digital Signal Processor Family Manual*, DSP56600FM/AD, 1996.

[5] A. Gerstlauer, D. D. Gajski. *Communication Software Code Generation*. Technical Report ICS-TR-00-46, University of California, Irvine, August 2000.

[6] H. Lehr, D. D. Gajski. *Modeling Custom Hardware in VHDL*. Technical Report ICS-TR-99-29, University of California, Irvine, July 1999.

[7] J. Zhu, D. D. Gajski. "A Retargatable, Ultra-fast Instruction Set Simulator." In *Proceedings Design, Automation and Test in Europe*, 1999.

[8] G. Berry, G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation." *Science of Computer Programming*, vol. 19, no. 2, 1992.