The SpecC Methodology

Technical Report ICS-99-56
December 29, 1999

Daniel D. Gajski
Jianwen Zhu
Rainer Doemer
Andreas Gerstlauer
Shuqing Zhao

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{gajski, jzhu, doemer, gerstl, szhao}@ics.uci.edu

The SpecC Methodology

Technical Report ICS-99-56
December 29, 1999

Daniel D. Gajski
Jianwen Zhu
Rainer Doemer
Andreas Gerstlauer
Shuqing Zhao

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{gajski, jzhu, doemer, gerstl, szhao}@ics.uci.edu

Abstract

This report describes the SpecC methodology for system-level embedded system design. The methodology consists of a set of well-defined tasks and design models which allow the easy insertion and reuse of intellectual property. Starting from the abstract executable specification written in SpecC different design alternatives concerning the system architecture (components and communication) can be explored and the specification is gradually refined and mapped to a final HW/SW implementation such that the constraints are satisfied optimally. The final hand-off for manufacturing includes software code compiled for the processors and the RTL descriptions for hardware synthesis.

Contents

List of Figures

The SpecC Methodology

D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

This report describes the SpecC methodology for system-level embedded system design. The methodology consists of a set of well-defined tasks and design models which allow the easy insertion and reuse of intellectual property. Starting from the abstract executable specification written in SpecC different design alternatives concerning the system architecture (components and communication) can be explored and the specification is gradually refined and mapped to a final HW/SW implementation such that the constraints are satisfied optimally. The final hand-off for manufacturing includes software code compiled for the processors and the RTL descriptions for hardware synthesis.

1 Introduction

We define a methodology as a set of models and transformations that can refine an initial, functional system specification into a detailed implementation description ready for manufacturing. The SpecC design methodology we discussed in this report is based on four well-defined models, namely, a specification model, an architecture model, a communication model, and finally, an implementation model.

In the following sections, we will give a detailed description of each model and of the refinement tasks leading from a functional specification model all the way to a cycle-accurate implementation model in SpecC.

2 Overview

The SpecC system-level design methodology starts with the **capture** of the intended functionality in the form of an **executable specification** as shown in Figure 1. This initial **specification model** describes the functionality as well as the performance, power, cost and other constraints of the intended design. It does not make any premature allusions to implementation details. During specification capture the designer may reuse existing code segments, functions or procedures by instantiating them out of an algorithm library.

The synthesis flow of the SpecC methodology consists of two major tasks: architecture exploration and communication synthesis tasks. Through a series of well-defined steps the initial specification is gradually mapped onto a selected target architecture.

Architecture exploration, which refines the specification into an architecture model, includes the design steps of allocation, partitioning of behaviors, channels, and variables, and scheduling. **Allocation** determines the number and types of the system components, such as general-purpose or custom processors, memories, and busses, which will be used to implement the system behavior. Allocation includes the reuse of intellectual property (IP), when IP components are selected from the component library.

Next, **behavior partitioning** distributes the behaviors (or processes) that comprise the system functionality amongst the allocated processing elements, while **variable partitioning** assigns variables to memories, and **channel partitioning** assigns communication channels to busses. **Scheduling** determines the order of execution

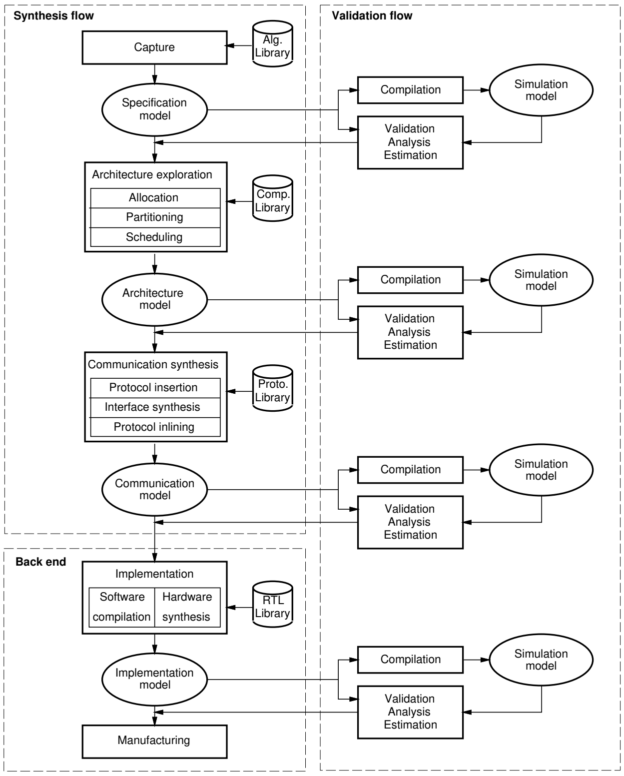


Figure 1: The SpecC methodology.

of the behaviors assigned to either the standard or custom processors after partitioning. In other words, scheduling is used for software and hardware components.

Architecture exploration is an iterative process culminating with an **architecture model** which represents a refinement of the specification model. Estimators evaluate each architecture candidate's satisfaction of the design constraints; until all constraints are satisfied, component and connectivity reallocation is performed and a new architecture with different components, connectivity, partitions, schedules or protocols is generated and evaluated.

Communication synthesis refines the abstract communications between behaviors in the architecture model into an implementation. The task of communication synthesis includes the insertion of communication protocols, synthesis of interfaces and transducers, and inlining of protocols into synthesizable components. In the resulting **communication model**, communication is described in terms of actual wires and timing relationships are described by bus protocols.

The result of the synthesis flow is handed off to the back-end tools, as shown in the lower part of Figure 1. The software part of the hand-off model consists of C code for compilation and the hardware part consists of behavioral C (VHDL) code for high-level synthesis. The back-end tools include compilers and a high-level synthesis tool. The compilers are used to compile the software C code for the chosen processor. The high-level synthesis tool synthesizes the functionality assigned to custom hardware and the functionality of transducers which are necessary for connecting different processors, memories, and IPs.

After software compilation and hardware synthesis, the final **implementation model** is generated, representing a clock-cycle accurate description of the whole system. This description, in turn, then serves as the basis for manufacturing of the system.

In each of the tasks the designer can make design decisions manually by using an interactive graphical user interface, for example, while transformations from one model into another can be accomplished automatically by following the refinement rules or model guidelines which will be described later in this chapter. After each refinement step in the synthesis flow, a corresponding SpecC model of the system is generated, which means that design decisions made in each design task are reflected in the generated models. Thus, in the validation flow that is orthogonal to the synthesis flow in the SpecC methodology, one can perform simulation, analysis and estimation of the SpecC models generated after each task.

After each design step, the design model is statically analyzed to estimate certain quality metrics such as performance, cost, and power consumption. Analysis and estimation results are reported to the user and back-annotated into the model for simulation and further synthesis.

The design can be statically analyzed or simulated after each step for validation of design correctness in terms of functionality, performance, and other constraints. A simulation model is compiled after each step which can be run on the host computer to validate correctness for simulation. For example, at the specification stage, the simulation model is used to verify the functional correctness of the intended design. After architecture exploration, the simulation model will verify the performance of behaviors on different processing elements (PEs). After communication synthesis, the bus-functional model is used to verify the communication and synchronization between processing elements.

At any stage of the refinement process, a standard software debugger can be used to locate and fix the errors if verification fails. Such debuggers enable one to set break points anywhere in the source code and to perform detailed state inspection at any time.

3 Specification

The synthesis flow begins with the capture of a specification of the system being designed. The specification is captured either textually by use of a standard text editor or visually by use of a graphical design entry tool which allows to capture the behavioral and structural hierarchy via a graphical user interface.

An **executable specification** in a formal description language describes the functionality of the system along with performance, cost and other constraints, but without premature allusions to implementation details. The specification should be as close to the computational model of the system as possible.

The source code can be executed with the help of a simulator and a set of test vectors, while errors can be detected with debugger tools. This step verifies the algorithms and the functionality of the system. Obviously, it is easier and more efficient to verify the correctness of the algorithms at a higher abstraction level than at a lower level which includes the implementation details as well.

In our system, we use the SpecC language, described in detail in [ZDG97], to capture the high-level specification of the system under design. SpecC is a superset of C [X3Sec90] and provides special language constructs for modeling concurrency, state transitions, structural and behavioral hierarchy, exception handling, timing, communication and synchronization. This is very different from popular hardware description languages, like VHDL [IEEE98] and Verilog [TM91], which do not include explicit constructs for state transitions, communication, and standard programming languages like C/C++ [Str97] and Java [AG96], which cannot directly model timing, concurrency, structural hierarchy, and state transitions. Thus, SpecC is appropriate for specifying the SFSMD or PSM computational models defined in [GVNG94].

In addition, SpecC is synthesizable and aids the designer in developing "good" designs by providing the features listed above as language constructs, rather than just supporting them in some contrived way. Another important feature of SpecC is its emphasis on **separation** of communication and computation at higher levels of abstraction. This dichotomy is essential for supporting **plug-and-play** of IPs. SpecC achieves this by using abstract function calls in the port interfaces of behaviors. The function calls are themselves implemented by **communication channels**. An executable specification in SpecC includes only the computation portion and uses a model similar to **remote procedure calls** (RPC) for communication. The actual communication methods are resolved and inlined during the refinement process for the final implementation model.

3.1 Specification Model

In the SpecC methodology, the specification model is the model with the highest level of abstraction. It is an accurate model of the intended system in terms of pure functionality but does not reflect its structure or timing. Typically, the specification model executes in zero simulation time. Neither the computation nor any communication is modeled with timing.

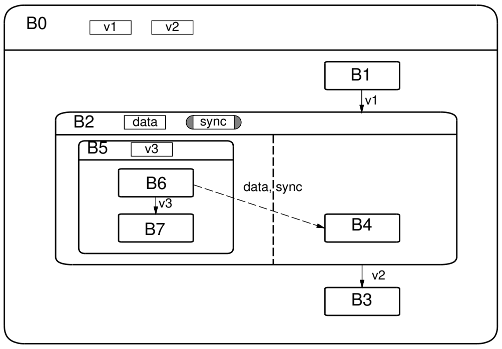


Figure 2: Specification model of design example.

```
behavior B5(in int v1, out int data,
            ISyncOut sync, out int v2) {
  int v3;
  B6 b6(v1, data, sync, v3);
  B7 b7(v3, v2);

  void main(void) {
    b6.main();
    b7.main();
  }
};

behavior B2(in int v1, out int v2) {
  int data;
  CSync sync;
  B4 b4(v1, data, sync);
  B5 b5(v1, data, sync, v2);

  void main(void) {
    par {
      b4.main();
      b5.main();
    }
  }
};

behavior B0() {
  int v1, v2;
  B1 b1(v1);
  B2 b2(v1, v2);
  B3 b3(v2);

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

Figure 3: SpecC code for specification model of design example.

We illustrate our methodology with a simple example design. The specification model of the example is shown in Figure 2. The corresponding SpecC code is shown in Figure 3. The top level behavior *B0* consists of three sequential behaviors: *B1*, *B2* and *B3*. The system starts execution with behavior *B1*. When *B1* completes, the system transitions to *B2*. Finally, the system transitions to *B3* upon behavioral completion of *B2*. Behavior *B2* again is a compound behavior, composed of two concurrent behaviors: *B4* and *B5*. Behavior *B4* is a leaf behavior like *B1* and *B3*. On the other hand, *B5* is hierarchical and consists of two sequential behaviors: *B6* and *B7*.

```
behavior B4(in int v1, out int data,
            ISyncIn sync) {
  void main(void) {
    ...
    sync.recv(); // wait for synchronization
    ...
    x = data;    // use "data"
    ...
  }
};

behavior B6(in int v1, in int data,
            ISyncOut sync, out int v3) {
  void main(void) {
    ...
    data = f(y); // assign "data"
    ...
    sync.send(); // send synchronization
    ...
  }
};
```

Figure 4: Synchronization of shared variable accesses in the specification model.

The leaf behaviors *B1* and *B3* communicate with behavior *B2* via variables *v1* and *v2*, respectively. Inside *B2*, the concurrent behaviors *B4* and *B6* communicate through the shared variable *data*. *B6* synchronizes its execution and hence the shared variable access with *B4* over the synchronization channel *sync*, as shown in Figure 4 and symbolized by the dashed arrow in Figure 2. The channel with simple synchronization semantics, shown in Figure 5, is instantiated out of the SpecC communication library. Finally, as shown in the code (Figure 3), data is communicated from behavior *B6* to behavior *B7* through the variable *v3*, and *B7* in turn produces the output value of *v2*. Note that although all variables are of plain integer type in the example code, in general data communicated can be of arbitrarily complex type.

In general, communication can be modeled in two ways, either as shared variables or by use of channels from the SpecC communication library. Communication channels which are useful for a specification model are those with basic synchronization, such as one-way or two-way handshaking, and channels for data communication such as blocking and non-blocking FIFOs. Communication in the specification over shared global variables or via channels implies nothing about the way it will be implemented later. For the implementation, the communication scheme could be transformed into a message passing or a shared memory mechanism.

The specification model can be composed using any of the constructs supported through the SpecC language. The initial specification should model the system at a very abstract level without prematurely introducing unnecessary implementation details. When developing the initial specification, a designer has to follow certain modeling guidelines in order to achieve optimal results. Basically, the specification should capture the required system functionality in a natural way and in a clear and concise manner, expressing essential features of the specification explicitly.

Table 1 gives the guidelines for developing the initial system specification. The language provides all the necessary support to efficiently describe the desired system features following these guidelines. Hence, each of the modeling concepts like parallelism or hierarchy is reflected in the SpecC description in an explicit and clear way.

```
interface ISyncIn {
  void recv();
};
interface ISyncOut {
  void send();
};

channel CSync()
  implements ISyncIn, ISyncOut
{
  bool  valid = false;
  event e;

  void send() {
    valid = true;
    notify (e);
  }
  void recv() {
    if (!valid) wait(e);
    valid = false;
  }
};
```

Figure 5: SpecC code for the synchronization channel.

Table 1: Specification model guidelines.

Separate communication and computation

Algorithmic functionality has to be detached from communication functionality. In addition, inputs and outputs of a computation have to be explicitly specified to show data dependencies.

Expose parallelism

Allow independent behaviors to run concurrently instead of artificially serializing behaviors in expectancy of a serial implementation. In essence, all parallelism should be made available to the exploration tools in order to increase room for optimizations.

Use hierarchy to group related functionality

Introduce one hierarchical level for each functional group and eliminate localized effects at higher levels. For example, local communication and local data dependencies are grouped and hidden by the hierarchical structure.

Choose proper granularity

The size of leaf behaviors has to be chosen such that optimization possibilities and design complexity are balanced when searching the design space. Basically, the leaf behaviors, which build the smallest indivisible units for exploration, should reflect the division into basic algorithmic blocks.

Identify system states

Use state transitions to explicitly model the steps of the computation in terms of basic algorithms or abstracted, hierarchical blocks.

3.2 Architecture exploration

The first major refinement step in the synthesis flow is the task of architecture exploration which includes allocation, partitioning and scheduling.

Allocation is usually done manually by the designer and basically involves selection of components from a library. In general, three types of components have to be selected from the component library: processing elements, called PEs (where a PE can be a standard processor or custom hardware), memories, and busses. Of course, the component library can include IP components and parts which have already been designed and which can be reused.

The set of selected and interconnected components is called the system target architecture. The task of **partitioning**, then, is to map the system specification onto this architecture. In particular, behaviors are mapped to PEs, variables are mapped to memories, and channels are mapped to busses. **Scheduling** of the behaviors mapped to each PE is then used to serialize execution. In the SpecC methodology, the resulting **architecture model**, like the initial specification, is modeled in SpecC.

Note that in general, exploration is an iterative process. The different tasks can be executed repeatedly and in each iteration the task can be done generally in any order or even simultaneously.

In order to perform architecture exploration, it is crucial to obtain accurate information about the design in a short amount of time. Therefore, the task of **estimation** is central to the whole exploration process. Estimation tools determine design metrics such as performance (execution time) and memory requirements (code and data size) for each part of the specification with respect to the allocated components. Estimation is performed as a combination of static analysis of the specification and dynamic profiling of the design description during functional simulation. Obviously estimation has to support software, hardware and bus components.

The estimation results are back-annotated into the corresponding behaviors and channels of the architecture model. There, they are used during simulation and synthesis in order to obtain feedback on whether the design constraints are met and to drive the decision making process during exploration in order to optimize the design.

3.2.1 Allocation

The task of architecture allocation is the selection of the type and number of components from a given library of system components, such as processors, memories, and busses. Allocation also determines the interconnection among the selected components. All of this has to be done in such a way that the functionality of the system can be implemented, all design constraints are satisfied, and the objective cost function is minimized.

During architecture allocation in the SpecC methodology, three types of components are selected from the component library. First, processing elements (PEs), including standard processors and custom processors, are needed as active elements performing the systems functions. Second, memories are needed to store the processing data. Finally, busses are allocated for the communication among the PEs and memories. Note that for each component type either a synthesizable, custom component or a predesigned IP can be selected. For the system busses, the designer selects the appropriate communication protocol from a library of bus/protocol schemes. In addition, the designer has the option of including custom protocols or customizing available protocols to suit the current application.

The network of selected components is called the **target architecture** of the system. The architecture consists of a set of system ports, a set of system busses, a set of system components, and a connectivity matrix which determines the interconnections among the ports, busses, and components.

3.2.2 Behavior partitioning

Behaviors are partitioned among the allocated processing elements; this decides which behavior is going to be executed on which PE. For example, behaviors to be implemented in software are separated from behaviors to be implemented in hardware. Based on a partitioning decision the design model is refined to reflect the selected partition.

For our design example we assume that two processing elements, *PE0* and *PE1* (a standard processor and synthesizable custom hardware), have been allocated. The specification model from Figure 2 after partitioning is shown in Figure 6. Here, the behaviors *B0*, *B2*, *B3*, *B5*, *B6* and *B7* are assigned to *PE0* (executing in software),

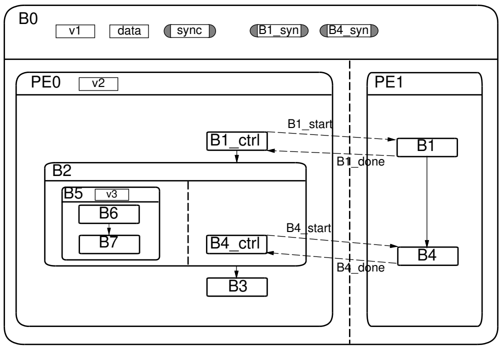
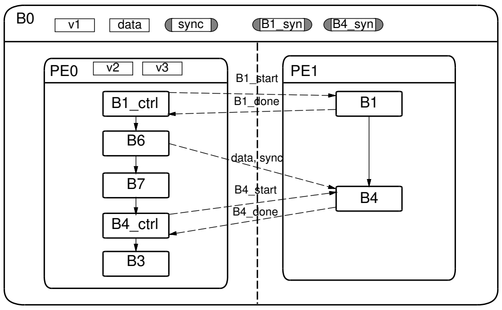
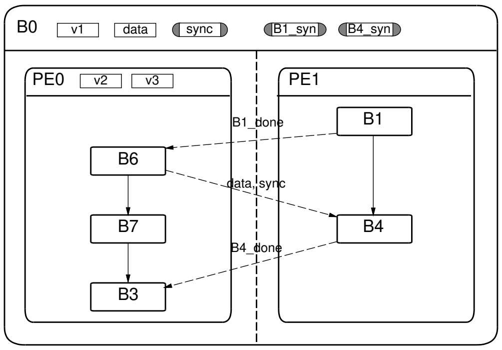


Figure 6: Intermediate model after behavior partitioning.



(a) non-optimized



(b) optimized

Figure 7: Intermediate model after scheduling.

and the behaviors *B1* and *B4* are assigned to *PE1* (implemented in hardware). In order to maintain the execution semantics of the specification, two additional behaviors, *B1_ctrl* and *B4_ctrl*, are inserted which synchronize the execution with *B1* and *B4*, respectively. The variables *v1*, *data* and the channel *sync* are used for communication between behaviors in different components and hence became global system variables. Also, for the additional synchronization of behaviors between components, two global synchronization channels *B1_syn* and *B4_syn* are added which handle the respective *start* and *done* signaling, as shown in Figure 6.

In summary, the steps involved in creating the refined design model after behavior partitioning are shown in Table 2.

3.2.3 Scheduling

The assignment of possibly concurrent behaviors to the inherently sequential PEs requires scheduling. The task of **scheduling** determines the order of execution for the behaviors that execute on a processor. The scheduler ensures that the schedule does not violate any dependencies imposed by the specification and tries to optimize objectives specified by the designer. After a schedule is determined, the design model is refined so that it reflects the sequentialization of the behaviors assigned to the same PE.

In general, scheduling can be either time-constrained or resource-constrained. For **time-constrained scheduling**, the designer supplies a set of timing constraints. Each timing constraint specifies the minimum and maximum time between two behaviors. The scheduler therefore has to compute a schedule in which no behavior violates any of the timing constraints, and which can minimize the number of resources used. On the other hand, for **resource-constrained scheduling**, the designer specifies constraints on the available resources. The scheduler

Table 2: Refinement rules for behavior partitioning.

Introduce additional level of hierarchy
At the top-level of the behavior hierarchy, insert behaviors which represent the components of the system architecture.

Bind behaviors to components
Annotate the component behaviors with the name of the component type out of the component library. Since the inserted behaviors simply group behaviors for each PE, this establishes the correlation of PE behaviors with allocated components in the system architecture.

Group behaviors
Group the behaviors of the specification under the component behavior to which they have been mapped, preserving the structural and behavioral hierarchy of the specification in the parts mapped to each component.

Estimate behavior metrics
Annotate the behaviors with the estimated values of chosen metrics for the components executing the behaviors. For example, in leaf behaviors appropriate `wait()` statements are added to establish correct execution times during simulation.

Add synchronization
For original behavior transitions that now cross component boundaries, introduce additional control behaviors in each component and corresponding global synchronization channels in order to preserve execution semantics.

Move communication
Move variables and channels in the original specification that are used for communication between behaviors mapped to different components to the top level and declare them as global system variables/channels. Add corresponding ports and connections in the structural hierarchy from the top down to the behaviors accessing the variables and channels.

then creates a schedule while optimizing execution time, such that all the subtasks are completed in the shortest time possible, given the restrictions on the resource usage.

Scheduling may be done statically or dynamically. In **static scheduling**, each behavior is executed according to a fixed schedule. The scheduler computes the best schedule at design time and the schedule does not change at run time. On the other hand, in **dynamic scheduling**, the execution sequence of the subtasks is determined at run-time. An **embedded real-time operating system** (RTOS) maintains a pool of behaviors ready to be executed. A behavior becomes ready for execution when all its predecessor behaviors have been completed and all inputs are available. With a **non-preemptive** scheduler, a behavior is selected from the ready list as soon as the current behavior finishes, whereas for a scheduler with **preemption**, a running behavior may be interrupted in its computation when another behavior with higher priority becomes ready to execute.

After a schedule is created, the scheduler moves the leaf behaviors into the scheduled order and also adds necessary synchronization signals and constructs to the behaviors. This refined model then reflects the tasks performed for behavior partitioning including scheduling. Since, in the SpecC system, all design models are captured with the same language, the **scheduled model** is also specified in SpecC.

We illustrate the scheduling process with the intermediate model after behavior partitioning, as shown before in Figure 6. Figure 7 shows how scheduling is performed for this example. As shown in Figure 7(a), the behavioral hierarchy inside *PE0* is flattened and its leaf behaviors are sequentialized. For *PE1*, the behavior changes from (potentially) concurrent to sequential execution.

Due to scheduling, some explicit synchronization can become redundant. Figure 7(b) shows the optimized version of the example. Here, the behaviors *B1_ctrl* and *B4_ctrl*, which were introduced in the partitioning stage, are removed, together with any obsolete synchronization signals.

The rules for creating the refined, scheduled model are summarized in Table 3.

Table 3: Refinement rules for scheduling.

Serialize behavior hierarchy
Inside the PE behaviors, remove all concurrent (parallel, pipe) behaviors and transform the behavior hierarchy according to the selected schedule into a purely sequential execution. Possibly flatten parts of the hierarchy or move behaviors across the hierarchy.

Optimize control
Optimize the scheduled hierarchy by removing unnecessary control and synchronization behaviors and channels.

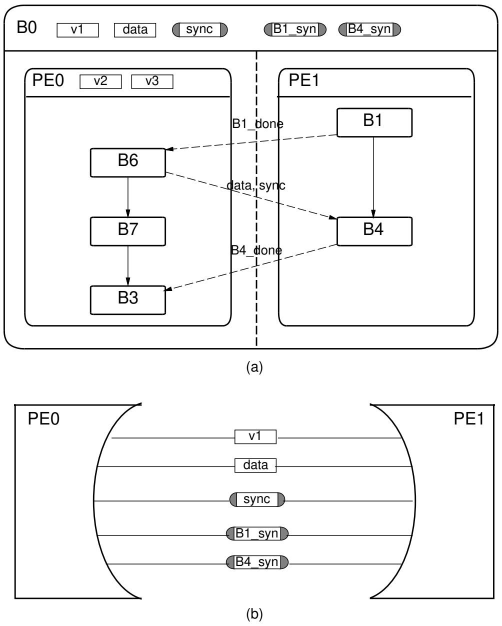


Figure 8: Model after behavior partitioning.

3.2.4 Variable partitioning

Up to this point, communication between the allocated PEs is performed via global, shared variables and channels. Figure 8(b) shows the design example at this point from a structural view which emphasizes communication structure. This representation helps to explain the insertion of communication channels and memory behaviors which is described in this and the next section.

Variables in the system specification need to be assigned to memories. This especially applies to the global, shared variables used for communication between components. These variables have to be mapped either to local memory in the components or to a dedicated shared memory component. In addition, due to memory limitations inside the components, even component-local variables might have to be moved to a shared memory component that has been allocated in the target architecture. This partitioning step of mapping variables to memories is called **variable partitioning**.

Variable partitioning essentially decides whether a variable used for communication is stored in one of the memories allocated outside the PEs, in one of the local memories of the PEs, or whether a local copy of the variable is kept in each accessing PE.

Figure 9 shows our example for a case where the global, shared communication variables *v1* and *data* are mapped to a dedicated, shared memory component *M0*. An additional behavior *M0* representing the memory component is inserted into the design model. Accesses to the variables *v1* and *data* inside the two PEs are replaced with accesses to the shared memory over communication channels *Cv1* and *Cdata*. Variable accesses in leaf behaviors are replaced with `read()` and `write()` calls to the corresponding variable channel. For exam-

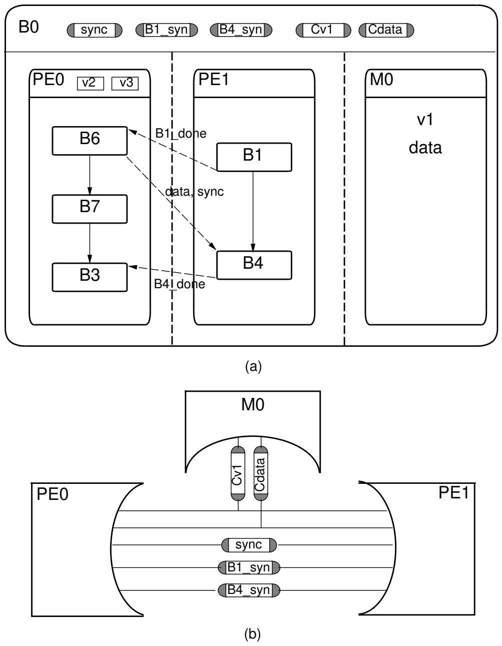


Figure 9: Example model after variable partitioning to a dedicated memory.

ple, statements like `x = data` or `data = f(y)` are now transformed into statements like `x = Cdata.read()` or `Cdata.write(f(y))` instead. The variable channels encapsulate the communication with the memory behavior which is listening on the other end to answer and handle those access requests.

Figure 10 shows the case where the global variables *v1* and *data* are mapped to local memories with a local copy in each PE. Hence, behaviors in the PEs can access their local copies normally. However, in order to preserve semantics updated variable values have to be exchanged at synchronization points among PEs. Variable values are communicated over two global message-passing channels, *Cv1* and *Cdata*. At synchronization barriers, code is added that transfers new values from producers to consumers using the message-passing primitives of these two channels.

In both cases, global channels are not touched during the variable partitioning process. In the refined model after variable partitioning that has been obtained by following rules in Table 4, communication between PEs is handled exclusively over channels and no global variables exist any longer.

3.2.5 Channel Partitioning

In order to refine the abstract communication between components, the global channels need to be mapped onto the allocated busses in the target architecture. This is achieved by grouping and encapsulating global channels in the additional channels representing the system busses.

In other words, abstract channels are partitioned into groups which are assigned to allocated bus channels. Later, during communication synthesis, these virtual bus channels will be refined into time-accurate models according to the selected bus protocols. However, at this point, the virtual bus channels are simply annotated with the type and name of the bus protocol in the IP library.

Figure 11 shows how channel partitioning is performed in our example. Due to the simplicity of this example, channel partitioning here is easy. Since we have to connect only two PEs, we allocate one system bus between the two PEs, represented by the channel *Bus0*. All the channels are assigned to this bus, as shown in Figure 11(a), and all channel accesses in the leaf behaviors are replaced with accesses to this bus channel.

The general refinement rules for channel partitioning, explained in Table 5, are similar to the rules for behavior partitioning. Instead of introducing structure to the computational part of the system, channel partitioning introduces structure to the communication part.

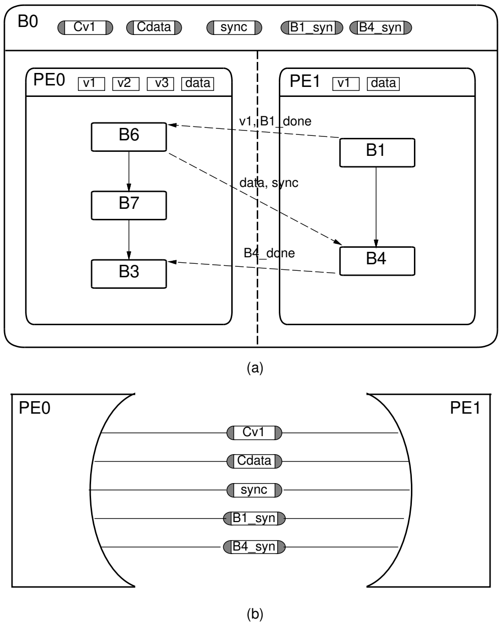


Figure 10: Example model after variable partitioning to local memories.

Table 4: Refinement rules for variable partitioning.

Move variables into components

According to the selected partition, move variables into the (memory or processing) components to which they have been assigned.

Add communication channels

For each variable add a global communication channel. In case a variable has been mapped to a dedicated server component, add a channel for communication with the memory server and connect all components accessing the variable to that channel. In case a local copy of a variable is kept in each component, add a message-passing channel for exchange of updated values.

Update variable accesses

For PEs with no local copy of a variable, replace all variable accesses with `read()` and `write()` calls to the corresponding channel. Otherwise, replace with accesses to local copy and add code at each synchronization point (`wait`) to send or receive updated variable values over the corresponding message-passing channel, in case the local copy was modified before or will be needed after synchronization, respectively. In both cases, update ports and connectivity accordingly.

Optimize communication and synchronization

Optimize variable communication by merging it with any existing communication and/or synchronization. For optimization, remove communication channels and corresponding code if variable is accessed only inside the assigned component.

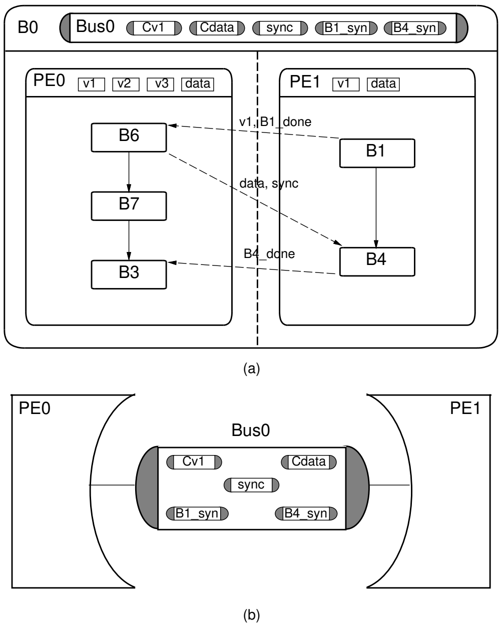


Figure 11: Model of design example after channel partitioning.

Table 5: Refinement rules for channel partitioning.

Add level of hierarchy
Introduce an additional level of hierarchy. At the top-level of the channel hierarchy, insert channels which represent the busses in the system architecture.

Bind channels to busses
Annotate the busses with the name of the bus type and bus protocol out of the IP library. Since the inserted busses simply group communication handled over each bus, virtual bus channels are thereby correlated with allocated busses and their protocols in the system architecture.

Group communication
Group the global communication channels which have been assigned to the same bus under the top-level channel representing the corresponding bus.

Estimate channel metrics
Annotate the channels in each bus with performance metrics estimated from the bus protocol assigned to the channels. For example, appropriate `wait()` statements are added in the communication primitives to establish execution times during simulation.

Update channel accesses
Replace channel accesses in the components with accesses to the corresponding bus interface. Update the ports and connectivity of the structural hierarchy accordingly.

3.2.6 Architecture Model

The task of architecture exploration is completed after allocation, partitioning and scheduling. In the SpecC methodology, the designer determines the sequence of allocation and partitioning tasks, which usually contains several iterations. The designer repeats these steps based on his experience and on the quality metrics obtained with the estimation tools. This designer-driven design space exploration is easily possible in the SpecC environment, because all parts of the system and all models are captured in the same language. This is unlike other environments where, for example, translating C code to VHDL and vice versa must be performed and verified. This design space exploration contributes to the attainment of a "good" system architecture and ultimately to an optimized implementation of the design, with good performance and lower costs.

As a result of architecture exploration, the initial specification model of the design has been refined into the **architecture model**, which accurately reflects the system architecture. Each component and each bus in the real system is represented by a corresponding top-level behavior or top-level channel in the design model.

The architecture model is an abstract model of the system under design, which accurately reflects the functionality and the overall structure of the final implementation. The model is annotated with estimation results for quality assessment during simulation. However, at this point, the model is not yet accurate in terms of cycle-true timing and communication, since execution times and delays have been estimated for all behaviors in the PEs and all channels in the busses. Assuming that the estimated execution times for the leaf behaviors have been inserted into their code in form of `waitfor` statements, the architecture model will reflect these timing delays in the simulation when it is executed.

The architecture model for the design example resulting from the last partitioning step is shown in Figure 11. Figure 12 shows the corresponding SpecC code. As with the code for the original specification model (Figure 3), only the top levels of the hierarchy are listed. An additional level in the behavior and channel hierarchy (component behaviors *PE0* and *PE1*, and bus channel *Bus0*, respectively) is inserted. The code for synchronization of behaviors between components is implemented in separate control behaviors *B1_done*, *B1_ctrl*, *B4_done* and *B4_ctrl*. The control behaviors also contain the code for exchanging updated variable values over message-passing channels (Figure 14) at those synchronization points.

Also, due to the optimizations described in Section 3.2.4, synchronization and message-passing is merged wherever possible. For example, synchronization for *B1* is combined with the message-passing of the updated value of *v1* inside behaviors *B1_ctrl* and *B1_done*. This eliminates the need for the synchronization channel *B1_syn* shown in Figure 11. Similarly, the *sync* channel is eliminated by merging synchronization between behaviors *B6* and *B4* with the communication of the updated *data* value (see Figure 13).

Note that the channels for exchanging updated variable values have standard message-passing semantics (Figure 14). Therefore, they are generally available as part of the SpecC communication library.

The main characteristic features of the SpecC architecture model are summarized in Table 6.

3.3 Communication Synthesis

The purpose of **communication synthesis** is to resolve the abstract communication behavior in the virtual busses of the architecture model through a series of refinements which will lead to an implementation of that communication over actual wires, and which will include accurate timing information.

During communication synthesis, the virtual, abstract communication used in the architecture model is replaced with real communication protocols implemented on the system busses. In other words, the virtual busses in the architecture model are replaced with actual busses and their protocols which are selected during architecture allocation. An application layer that implements the required abstract communication semantics over the native protocols is generated and inserted in the design model on top of the native bus protocols. For incompatible bus protocols, transducers are inserted into the system model to bridge the gap between the different protocols. Finally, the communication protocols are implemented in software and hardware by inlining the communication code into the PEs.

In the SpecC methodology, communication synthesis consists of three tasks, namely, protocol insertion, transducer synthesis, and protocol inlining.

```
behavior PE0(IBus0 bus0) {
  int v1, v2, v3, data;
  B1_ctrl  b1_ctrl(v1, bus0);
  B6       b6(v1, data, v3, bus0);
  B7       b7(v3, v2);
  B4_ctrl  b4_ctrl(bus0);
  B3       b3(v2);

  void main(void) {
    b1_ctrl.main();
    b6.main();
    b7.main();
    b4_ctrl.main();
    b3.main();
  }
};

behavior PE1(IBus0 bus0) {
  int v1, data;
  B1       b1(v1);
  B1_done  b1_done(v1, bus0);
  B4       b4(v1, data, bus0);
  B4_done  b4_done(bus0);

  void main(void) {
    b1.main();
    b1_done.main();
    b4.main();
    b4_done.main();
  }
};

behavior B0() {
  Bus0  bus0;
  PE0   pe0(bus0);
  PE1   pe1(bus0);

  void main(void) {
    par {
      pe0.main();
      pe1.main();
    }
  }
};
```

Figure 12: SpecC code for architecture model of design example.

```
channel Bus0()
  implements IPE0bus0, IPE1bus0
{
  CInt   v1, data;
  CSync  B4_syn;

  void send_v1(int v)       { v1.send(v);       }
  void recv_v1(int *v)      { v1.recv(v);       }
  void send_data(int v)     { data.send(v);     }
  void recv_data(int *v)    { data.recv(v);     }
  void send_b4_syn()        { B4_syn.send();    }
  void recv_b4_syn()        { B4_syn.recv();    }
};

behavior B1_ctrl(int v1, IPE0bus0 bus0) {
  void main(void) { bus0.recv_v1(v1); }
};
behavior B1_done(int v1, IPE1bus0 bus0) {
  void main(void) { bus0.send_v1(v1); }
};

behavior B4_ctrl(IPE0bus0 bus0) {
  void main(void) { bus0.recv_b4_syn(); }
};
behavior B4_done(IPE1bus0 bus0) {
  void main(void) { bus0.send_b4_syn(); }
};
```

Figure 12 (continued): SpecC code for architecture model of design example.

```
behavior B4(in int v1, inout int data,
            IPE1bus0 bus0) {
  void main(void) {
    ...
    // wait for "data"
    bus0.recv_data(&data);
    ...
    x = data;      // use "data"
    ...
  }
};
behavior B6(in int v1, out int data,
            out int v3, IPE0bus0 bus0) {
  void main(void) {
    ...
    data = f(y); // assign "data"
    ...
    bus0.send_data(data); // send "data"
    ...
  }
};
```

Figure 13: Synchronization inside leaf behaviors of the architecture model.

```
interface ISendInt {
  void send(int v);
};
interface IRecvInt {
  void recv(int *v);
};

channel CInt()
  implements ISendInt, IRecvInt
{
  int    buf;
  bool   valid = false;
  event  e;

  void send(int v) {
    buf = v;
    valid = true;
    notify (e);
  }
  void recv(int *v) {
    if (!valid) wait(e);
    *v = buf;
    valid = false;
  }
};
```

Figure 14: SpecC code for message-passing channel.

Table 6: Architecture model guidelines.

Create high-level structure
The top-level behaviors and channels represent the components and busses of the system architecture and their connectivity corresponds to the structure of the architecture. Behaviors grouped under the top-level behaviors specify the functionality (and storage) to be implemented on the corresponding component. Similarly, channels grouped under the top-level channels represents the communication to be implemented over the corresponding system bus.
Sequentialize component functionality
The behavior hierarchy inside the component behaviors is purely sequential, i.e. there are no parallel or pipelined behavior types. Parallelism is available only at the top-level, where all the component behaviors run concurrently.
Specify global communication
The bus channels exclusively contain abstract channels for directed communication of data values, i.e. there are no variables and random-access storage inside the bus channels.
Annotate estimated metrics
Behaviors and channels are annotated with their estimated design metrics for the components and busses to which they are mapped, respectively. For simulation purposes, appropriate delay statements are added to the behaviors and channels.

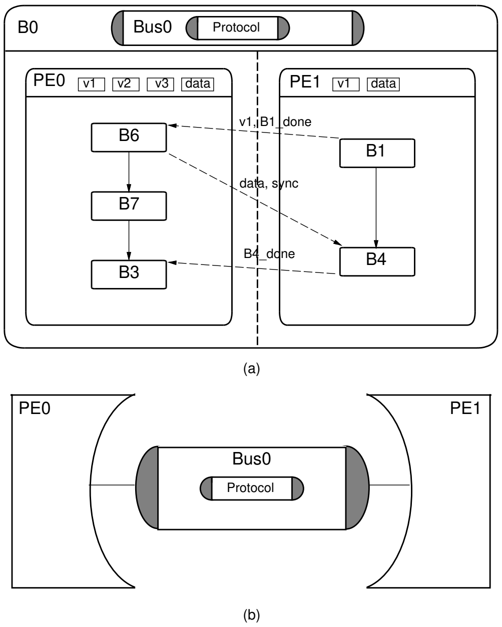


Figure 15: Model of design example after protocol insertion.

3.3.1 Protocol Insertion

The designer selected the appropriate communication protocol for the system busses during the allocation task of architecture exploration. Now, the virtual busses in the architecture model are refined into hierarchical channels which implement the required communication functionality over the actual wires of the bus. Communication is implemented in two layers. The low-level layer is a channel that provides the native communication primitives of the actual bus protocol. On top of that, an application layer implements the abstract communication over the low-level bus protocol, for example by sizing the complex data types used in the application into blocks that can be transported over the bus.

Low-level bus protocol specifications are taken out of a protocol library and are written in the SpecC language in terms of channel primitives that supply common interface function calls to facilitate reuse. For example, a given VME bus description will supply *send()* and *receive()* as would the PCI specification. In this way, we can easily interchange protocols (as channels) and perform some simulation to obtain performance estimates. Later, the remote procedure calls (RPCs) to the channels will be replaced by local I/O instructions for software, or by additional behavior to be synthesized for hardware entities.

This process can be either manual or automatic. The cost of manual refinement is still lower than that of the traditional way, since, on account of the abstraction provided by the channel construct, the user does not have to bother with issues like detailed timing. Automatic refinement will generate code for the application layer which assembles high-level messages from low-level messages, or vice versa.

Figure 15 shows this refinement (summarized in Table 7) for our example. According to the target protocol allocated during architecture exploration, a bus protocol *Protocol*, such as the PCI bus, is inserted in order to carry out the communication between the behaviors. The methods of the virtual bus *Bus0* are refined to use the methods of the bus protocol encapsulated in the channel *Protocol*. Figure 15(b) shows how the channel hierarchy directly reflects the layers of the communication between the PEs.

3.3.2 Transducer Synthesis

The previous section on protocol insertion assumed that all components are synthesizable and therefore that later the communication layers can be inlined into the components, where they would be synthesized together with the other functionality to implement the required protocols at their interfaces.

Table 7: Refinement rules for protocol insertion.

Insert protocol code	
	For each system bus, pull the corresponding protocol channel out of the protocol library.
Generate application layer	
	For each bus, generate the application layer that implements the abstract communication assigned to that bus, using the primitives provided by the corresponding protocol channel.
Replace bus channels	
	Replace the virtual bus channels in the architecture model with the hierarchical combination of application layer and bus protocol channels.

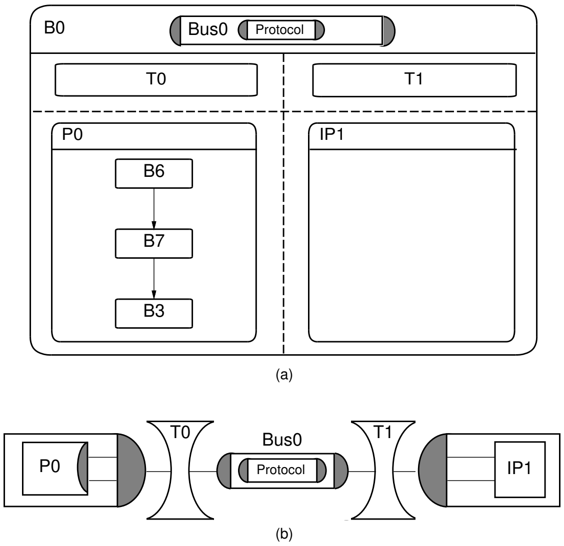


Figure 16: Model with IPs after protocol and transducer insertion.

However, it is possible that the selected bus protocols conflict with the fixed, built-in interfaces of non-synthesizable components. This situation is most likely to occur with hard IP components, processors, or memories, when they are connected to an incompatible system bus.

For such IPs, a model which encapsulates both the proprietary protocols at the IP interface and the (custom or fixed) application layer of the IP communication in a **wrapper** is inserted from the component library during protocol insertion. The corresponding implementation of the communication layers inside the component is part of the IP behavior inside the wrapper.

In the refined design model, introduction of IPs with fixed interfaces necessitates the creation and insertion of a **transducer** which bridges the gap between the IP component and the channels to which the original behavior is connected. Again, it is easy to create such a transducer manually due to the high-level nature of the wrapper and the connected channel. On the IP side, the wrapper provides the layers to implement the communication functionality over the IP protocol which will later become part of the transducer.

It is important to note that the replacement of synthesizable behaviors with IP components is not limited to the communication synthesis stage, but can be executed at any time during architecture exploration and communication synthesis, due to the encapsulation of IP protocols in wrappers. At any time, a behavior can be replaced by a wrapped IP model plus a transducer.

In our example, Figure 16 shows the design model where component *PE0* is replaced with an IP processor that will run the behaviors assigned to it and *PE1* is replaced with a non-synthesizable hard IP component. The fixed IP protocols are encapsulated in wrappers and connected to the bus channel *Bus0* via the inserted transducers *T0* and *T1*. The wrapper processor behavior provides the layers implementing the communication functionality over the processor interface internally to the behaviors running on the processor.

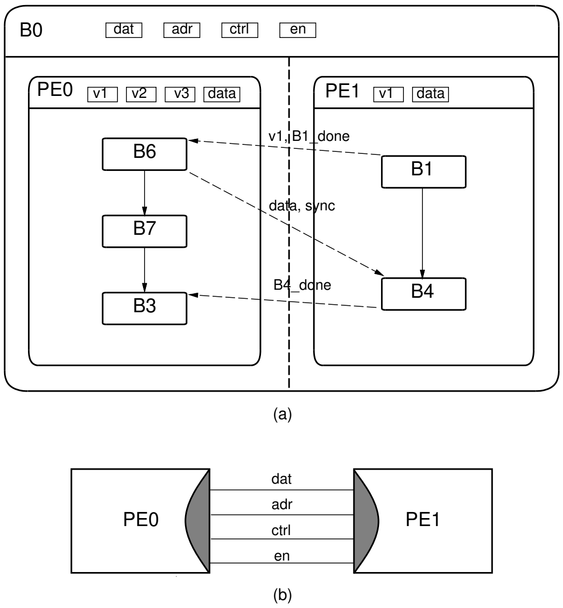


Figure 17: Model after protocol inlining.

As part of protocol insertion described in Section 3.3.1, transducer insertion and synthesis consists of the steps listed in Table 8.

Table 8: Refinement rules for transducer synthesis.

Insert transducers
Insert transducer behaviors between component behaviors and bus channels that have incompatible protocols on their ports.

Encapsulate with wrappers
Replace the component behaviors with their wrapped equivalents that encapsulate the IP interface protocols.

Synthesize transducer code
Create code inside the transducer behaviors which performs a one-to-one mapping of communication primitives on one side to corresponding primitives on the other side.

3.3.3 Protocol inlining

During the final task of protocol inlining, methods located in the channels and wrappers are moved into the connected behaviors, where they will be synthesized together with the rest of the component's functional (computational) behavior. The communication functionality is thereby included in the behaviors. Protocol inlining exposes the variables encapsulated inside the protocol channel, which then represent the wires of the system busses. The port interfaces of the behaviors are therefore composed of bit-level signals, as compared to the abstract function calls before inlining was done.

Figure 17 shows our example with synthesizable behaviors after inlining of the methods of both channels, *Bus0* and *Protocol*, into the behaviors. After this protocol inlining, the protocol channel variables *dat*, *adr*, *ctrl*, and *en* are exposed and serve as interconnection wires between the accordingly created ports of the components.

Finally, Figure 18 shows the model with the IP components after protocol inlining is performed. Here, the methods from all the channels and wrappers are inlined into the transducers which communicate with the IPs via proprietary busses. Again, the bus variables *dat*, *adr*, *ctrl*, and *en* are exposed and serve as interconnection

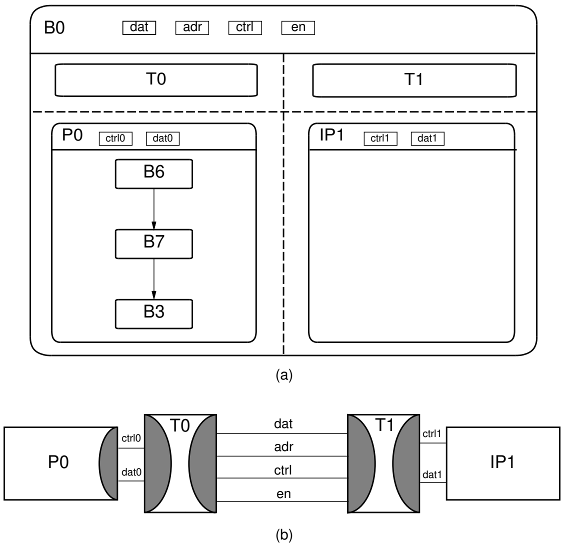


Figure 18: Model with IPs after protocol inlining.

Table 9: Refinement rules for inlining.

Inline communication methods
Move the communication functions provided by the wrappers and channels into the transducer and component behaviors where they are accessed. Variables inside the low-level protocol channels become global, shared variables. Change ports and connectivity from channel interfaces to variable accesses.
Optimize
Cancel away transducers in case the protocols on both sides of a transducer are equivalent after inlining.

wires between the transducers. The processor component contains the corresponding code, which implements the communication layers over the processor interface using the processor's I/O instructions, and which will later become part of the embedded operating system.

In summary, the task of protocol inlining consists of the steps listed in Table 9.

3.3.4 Communication Model

After the communication functions have been inlined into the behaviors, the task of communication synthesis is complete. As a result, the architecture model of the design has been refined into the communication model.

The **communication model** is a design model at a medium level of abstraction. As the architecture model, it is an accurate representation of the design in terms of functionality and overall structure. In addition, the communication model features bit-exact, time accurate communication.

More specifically, the communication model is a bus functional model. The transactions on the system busses are represented bit-exactly with accurate timing in great detail. On the other hand, the components in the system are still represented at a high abstraction level, allowing for fast simulation. However, the execution times of the components are not exact, but are only estimated values.

The communication model of the design example after protocol inlining is shown in Figure 17. Figure 19 lists the corresponding SpecC code. In the example the component interfaces match with the bus protocol (e.g. components *PE1* is a synthesizable custom hardware processor and the bus protocol is equivalent to the protocol of the processor *PE0*). Therefore, no transducers are needed.

The inlined application layer and bus protocols are modeled as explicit **bus drivers** inside the components. The drivers *driver0* and *driver1* translate the functionality of former channel *Bus0* in the architecture model into protocol transactions. Therefore, all the behaviors inside the component remain unchanged.

```
behavior PE0(inout bit[15:0] dat, out bit[15:0] adr,
             out   bit[1:0]  ctrl, out event en) {
  int v1, v2, v3, data;
  CDriver0 driver0(dat, adr, ctrl, en);
  B1_ctrl  b1_ctrl(v1, driver0);
  B6       b6(v1, data, v3, driver0);
  B7       b7(v3, v2);
  B4_ctrl  b4_ctrl(driver0);
  B3       b3(v2);

  void main(void) {
    b1_ctrl.main();
    b6.main();
    b7.main();
    b4_ctrl.main();
    b3.main();
  }
};

behavior PE1(inout bit[15:0] dat, in bit[15:0] adr,
             in    bit[1:0]  ctrl, in event en) {
  int v1, data;
  CDriver1 driver1(dat, adr, ctrl, en);
  B1       b1(v1);
  B1_done  b1_done(v1, driver1);
  B4       b4(v1, data, driver1);
  B4_done  b4_done(driver1);

  void main(void) {
    b1.main();
    b1_done.main();
    b4.main();
    b4_done.main();
  }
};

behavior B0() {
  bit[15:0] dat, adr;
  bit[1:0]  ctrl;
  event     en;
  PE0  pe0(dat, adr, ctrl, en);
  PE1  pe1(dat, adr, ctrl, en);

  void main(void) {
    par {
      pe0.main();
      pe1.main();
    }
  }
};
```

Figure 19: SpecC code for communication model of design example.

```
channel CDriver0(inout bit[15:0] dat, out bit[15:0] adr,
                 out   bit[1:0]  ctrl, inout event en)
  implements IPE0bus0
{
  CBusMaster bus(dat, adr, ctrl, en);

  void recv_v1(int *v) {
    bus.IntA();                   // wait for interrupt
    *v = bus.readBus(ADDR_V1);    // bus read cycle
    *v = *v | (bus.readBus(ADDR_V1) << 16);
  }

  void send_data(int v) {
    // bus write cycle
    bus.writeBus(v & 0xFFFF, ADDR_DATA);
    bus.writeBus(v >> 16,    ADDR_DATA);
  }

  void recv_b4_syn() {
    bus.IntB();                   // wait for interrupt
  }
};

channel CDriver1(inout bit[15:0] dat, out bit[15:0] adr,
                 out   bit[1:0]  ctrl, inout event en)
  implements IPE0bus1
{
  CBusSlave bus(dat, adr, ctrl, en);

  void send_v1(int v) {
    io.raiseA();                  // raise interrupt
    // bus write cycle
    bus.putBus(v & 0xFFFF, ADDR_V1);
    bus.putBus(v >> 16,    ADDR_V1);
  }

  void recv_data(int* v) {
    *v = bus.getBus(ADDR_DATA);   // bus read cycle
    *v = *v | (bus.getBus(ADDR_DATA) << 16);
  }

  void send_b4_syn() {
    io.raiseB();                  // raise interrupt
  }
};
```

Figure 19 (continued): Communication model, bus drivers.

In comparison to the architecture model presented in Section 3.2.6, the additional characteristics of the communication model are listed in Table 10.

Table 10: Communication model guidelines.

Implement bus functionality
At the top level, the behaviors which represent the components of the system architecture communicate via a set of shared variables which represent the wires of the system busses in the target architecture.

Annotate bus timing
The communication between components over their interfaces and the bus wires is modeled with accurate timing whereas the (purely sequential) behavior inside the components is at the functional level with annotated estimated delays for simulation.

3.4 Backend

Communication synthesis, as the last step in the synthesis flow, generates the hand-off model for our system. This model is then further refined using traditional backend tools as shown in Figure 1.

It is the task of the backend to create an optimized implementation for each particular component in the design model. More specifically, the custom hardware and transducer components need to be implemented by a behavioral hardware synthesizer and the software components need to be compiled for the particular processor. After inlining, the communication functionality has become part of the software (bus drivers in the operating system) and hardware/transducer (communication FSMDs) components, and thus will be implemented together with the other parts, possibly employing specific optimizations.

As a result of the backend process, the final implementation model of the system is created. The implementation model will then be used for manufacturing of the system. It is a cycle-accurate RTL description of both the computation and communication in the system.

3.4.1 Software Compilation

C code for each of the allocated processors in the target architecture is created from the communication model. Retargetable compilers or special compilers for each of the different processors are then used to compile the C code into instructions for the target processor.

In the implementation model, the processor behaviors are then replaced with a cycle-accurate description of the target processor executing the generated assembly code. For example, an existing **instruction set simulator** (ISS) of the processor can be hooked into the SpecC implementation model, provided that the simulator supports a suitable C programming interface.

3.4.2 Hardware Synthesis

The hardware portion of the communication model consists of synthesizable, behavioral models in SpecC. For the behavior hierarchy in the SpecC description, C (or VHDL) code can be created, which is then synthesized using standard **high-level synthesis** (HLS) tools. Note that the translation of the SpecC model into synthesizable C (or VHDL) is straightforward, since the component models are free of any special SpecC constructs at this point.

High-level synthesis creates a behavioral or structural RTL model of the hardware components in the form of scheduled register-transfer code or a netlist of RTL components, respectively. This structural or behavioral RTL description can then be modeled in SpecC, to be included in the implementation model for final cosimulation, for example.

3.4.3 Implementation Model

As a result of hardware synthesis and software compilation for each component in the communication model, the final implementation model of the design is generated.

The **implementation model** is the model with the lowest level of abstraction in the SpecC methodology. It is an accurate model of the design implementation in terms of functionality, structure, communication and timing. Note that the implementation model reflects both bus-cycle accurate timing for the communication, as well as clock-cycle accurate timing for the computation performed in the system.

The implementation model differs from the previous communication model only within the synthesizable components. A software component is described in terms of an instruction set architecture, while a hardware component is described with a FSMD model or a RTL netlist, a control unit and a data path, or at least a scheduled (but not bound) description of the operations performed in each clock step. In summary, the implementation model is ready for manufacturing.

Table 11 summarizes the main features of the implementation model, in comparison with the communication model (Section 3.3.4).

Table 11: Implementation model guidelines.

Implement system busses
As in the communication model, component behaviors communicate over shared variables representing the wires of the system busses. Communication is modeled with accurate timing.
Implement system components
The component behaviors are replaced with a cycle-accurate model of the component implementation. Hence, computation is modeled with accurate timing, too.

4 Summary

With the background of a specify-explore-refine paradigm, we have presented an IP-centric methodology for the codesign of embedded systems. The SpecC methodology consists of a set of well-defined tasks and design models which allow the easy insertion and reuse of intellectual property.

More specifically, the design methodology starts with an executable specification of the system under design and eventually creates an implementation model ready for manufacturing. The intermediate tasks of allocation, partitioning, scheduling, and communication synthesis are performed by the designer interactively, either manually or with the help of design automation tools. As explained in this chapter, each task is built upon well-defined models and transformations. Therefore, the new, refined models produced at each step will be automatically generated by the corresponding refinement tools. On the other hand, refinement is driven by the decisions made manually by the designer or automatically by a set of algorithms. In all cases, the designer can focus on the critical design decisions while tools automate tedious exploration and refinement tasks.

The SpecC methodology is based on four well-defined design models of different levels of abstraction: the specification model, the architecture model, the communication model, and the implementation model. With each successive model, more implementation details are introduced. The specification model is purely functional with no timing at all. The architecture model represents the structure of the target architecture, and the functionality in the components and on the busses is annotated with estimated delays. In the communication model, components are still functional (with annotated delays), but the communication over system busses is time accurate. Finally, the implementation model is cycle-accurate both in computation (components) and communication (interfaces and busses).

Please note that because of the modularity of the SpecC model ("plug-and-play"), a design can also be easily represented as a mixture of these models. This is especially useful if parts of a design are further refined as others, or if accuracy is only required for specific portions in the design model. For example, a mixture of communication and implementation models, makes possible a cycle-accurate simulation of certain system components together with bus-functional models for the rest of the system.

References

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[Inc98] IEEE Inc. *IEEE Standard VHDL Language Reference Manual.* New York, 1998.

[Sec90] X3 Secretariat. The C language. In *X3J11/90-013, ISO Standard ISO/IEC 9899*, Washington, DC, USA, 1990. Computer and Business Equipment Manufacturers Association.

[Str97] K. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[TM91] D. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[ZDG97] J. Zhu, R. Dömer, and D. Gajski. Syntax and semantics of the SpecC language. In *Proceedings of the Synthesis and System Integration of Mixed Technologies*, Osaka, Japan, December 1997.