IP-Centric Methodology

Daniel D. Gajski, Gaurav Aggarwal, En-Shou Chang
Rainer Dömer, Tadatoshi Ishii, Jon Kleinsmith, Jianwen Zhu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425.
Phone: (714) 824-8059

Abstract

In this report, we describe the *specify-explore-refine* (SER) co-design methodology for design of embedded systems. We describe the necessary design steps in order to map an abstract specification of the system to the final implementation model. We propose a co-design tool based on our co-design methodology. We also present a graphical user interface for the proposed co-design tool.

1 Introduction

The co-design process for embedded systems usually starts from an executable specification as shown in Figure 1. The specification specifies the functionality as well as the performance, power, cost and other constraints of the intended design but no implementation details. The specification can be captured directly in a formal specification language such as SpecC or by the use of interactive graphical entry tool such as SpecEdit. We describe the SpecEdit tool in Section 2. We next give an overview of the refinement transformations in the methodology.

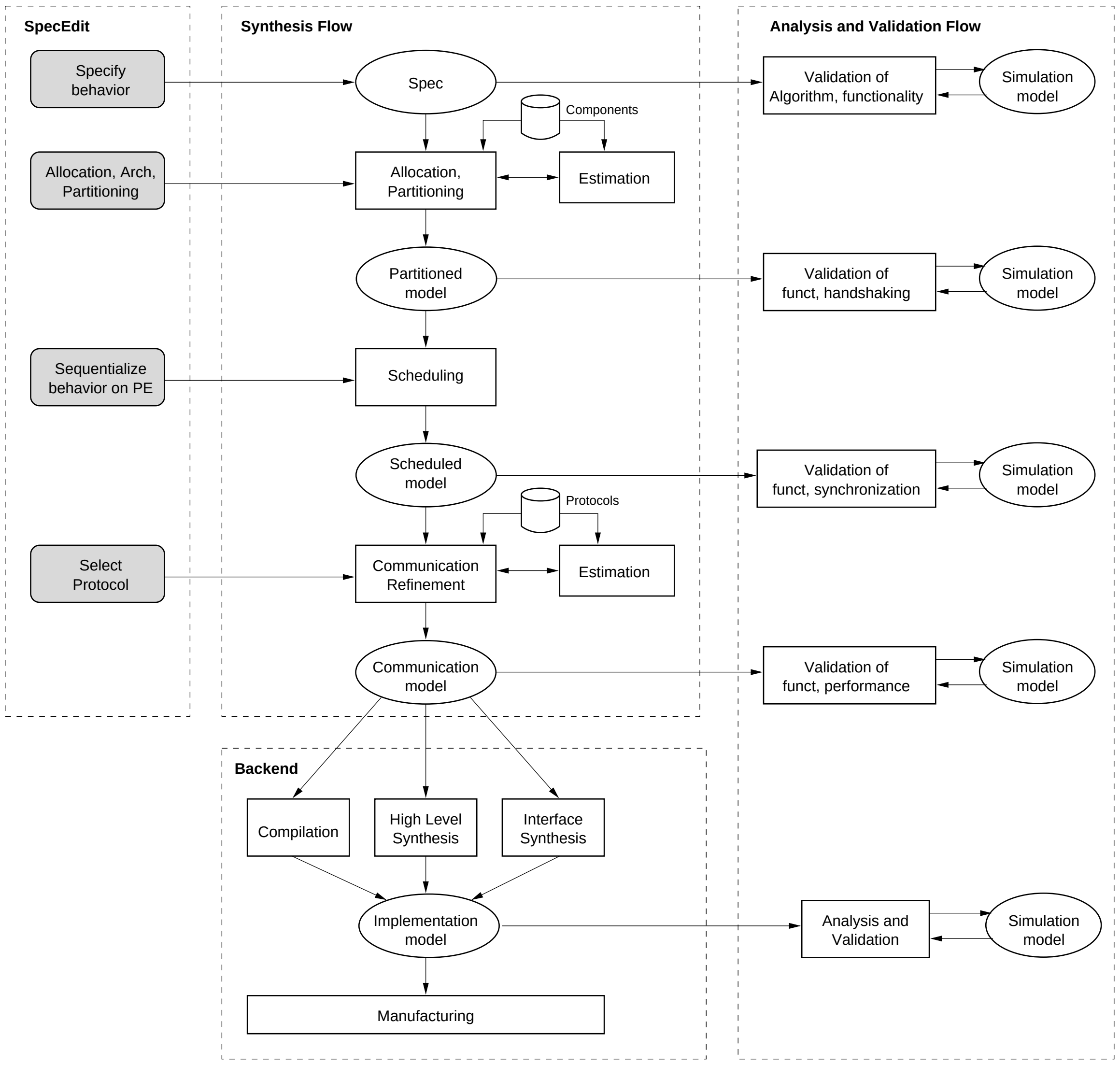


Figure 1: The co-design methodology in the SpecC system

During the co-design process, the designer will go through a series of well-defined design steps which will eventually map the functionality of the specification to the target architecture. These design steps include *allocation, partitioning, scheduling* and *communication refinement*, which form the synthesis flow of the methodology. The task of allocation determines the number and types of the system components, such as processors, ASICs and buses, used to implement the system behavior. The task of partitioning distributes the specification over the system components. The task of scheduling determines the ordering of execution of the sub-behaviors. The task of communication refinement selects the appropriate protocols and resources to implement the abstract communications between the behaviors. The design decision in each of the design task can be made either by the designer using the graphical user interface or by the automatic synthesis tools described in Sections 4, 5 and 6.

The result of the synthesis flow is handed-off to the back-end tools, shown in the lower part of Figure 1. The software part of the hand-off model consists of C code and the hardware part consists of behavioral VHDL code. The back-end tools include compilers, a high level synthesizer and an interface synthesizer. The compilers are used to compile the software C code for the processor on which the code is mapped. The high level synthesizer is used to synthesize the functionality mapped to custom hardware. The interface synthesizer is used to implement the functionality of interfaces.

During each design step, the design model is statically analyzed to estimate certain quality metrics such as performance, cost and power consumption. The same design model will also be used in simulation to verify the correctness of the design at the corresponding step. For example, at the specification step, the simulation model is used to verify the functional correctness of the intended design. After partitioning and scheduling, the simulation model will verify the synchronizations between behaviors in different PEs, which are artifacts generated by these design steps. After communication refinement, the model is used to verify performance of the system including computation and communication. If validation fails, a debugger can be used to locate the errors. Like all standard source level debuggers, the SpecC debugger provides the capability of break points and state inspection during the debugging sessions.

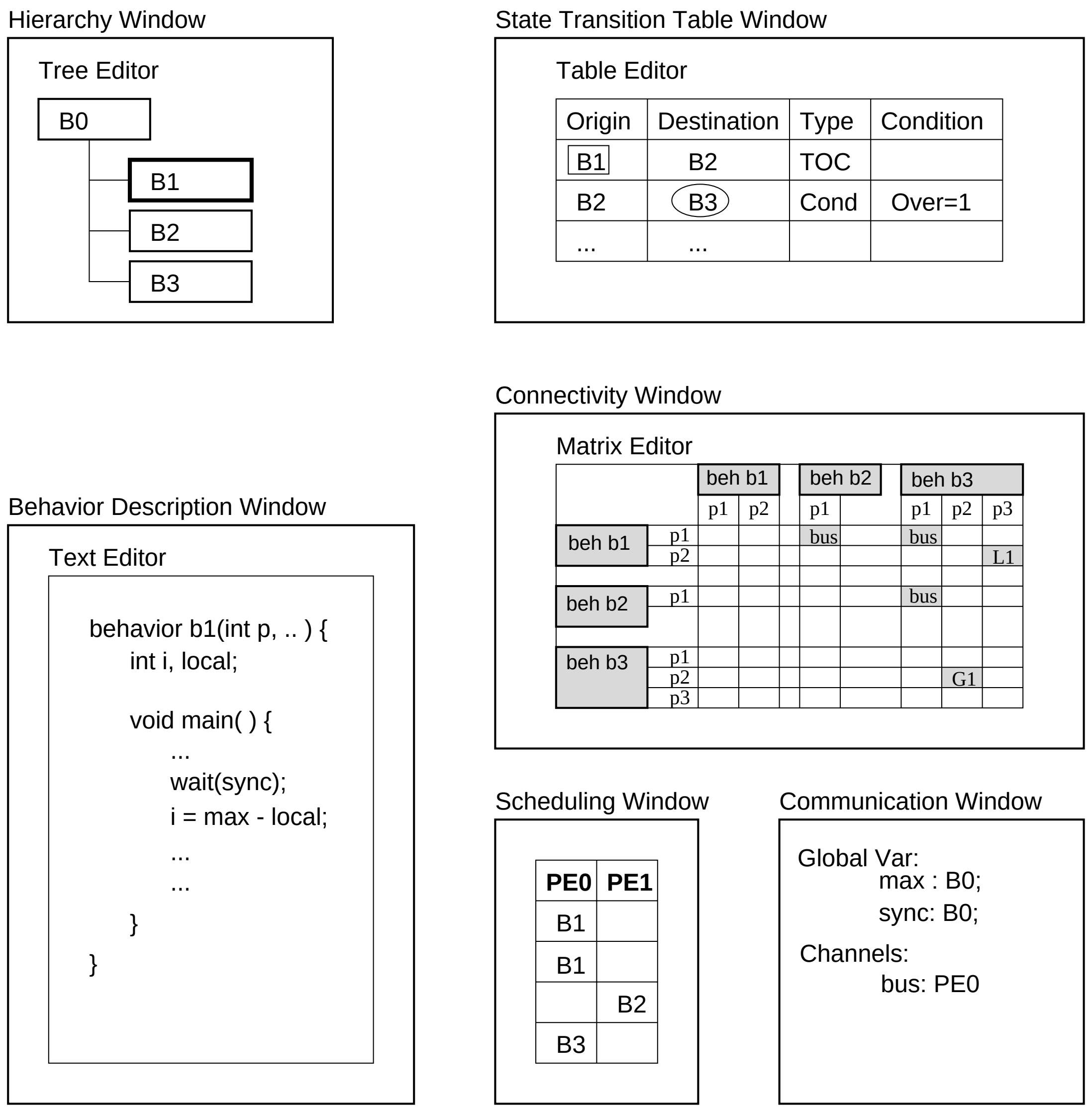
2 SpecEdit

SpecEdit is a graphical user interface (GUI) which is used for specifying the system in the co-design process. It is also used for displaying system models at different levels of abstraction. It also provides support for the transformations that need to be performed at each design step in the methodology. It allows the designer to execute the transformation commands on the models in an interactive manner. These commands can either be simple manual operations or calls to automatic tools that use algorithmic procedures to perform the functions. The interface at a typical processing step in the methodology contains six different windows as shown in Figure 2.

The use of each window is dependent on the functions that need to be performed at any stage. Thus, some windows may be empty or may be used for only displaying information about the model. The functionality of each of the window is as follows.

Hierarchy Window This window is used for displaying and modifying the hierarchy in the specification. It supports creation, deletion and movement of behaviors or a tree of behaviors. It displays the tree view of behavioral hierarchy along with the execution types (sequential, concurrent or pipeline). The Hierarchy Window is located on top-left corner of Figure 2. Each box in the tree represents a behavior and is identified with the name of the behavior. Each box also has an icon associated with it which gives the execution type of the box. In addition, any part of tree can be collapsed to make the display more readable and to fit the display in the current window size. Clicking a behavior box brings up the other windows (described below) for this behavior.

In Figure 2, `B0` is a behavior for node which has children behaviors. `B1` is a leaf behavior which contains an algorithm. `B1` and `B2` will have the same *sequential* icon



Origin	Destination	Type	Condition
B1	B2	TOC	
B2	B3	Cond	Over=1
...	...		

		beh b1		beh b2	beh b3		
		p1	p2	p1	p1	p2	p3
beh b1	p1			bus	bus		
	p2						L1
beh b2	p1				bus		
beh b3	p1						
	p2					G1	
	p3						

```
behavior b1(int p, .. ) {
     int i, local;

     void main( ) {
          ...
          wait(sync);
          i = max - local;
          ...
          ...
     }
}
```

PE0	PE1
B1	
B1	
	B2
B3	

Global Var:
max : B0;
sync: B0;

Channels:
bus: PE0

Figure 2: SpecEdit: authoring tool for SpecC

which implies that these behaviors are executed sequentially.

State Transition Table Window This window is used for specifying the transitions between sequential behaviors in a hierarchical node behavior. This window is located top-right corner of Figure 2. The table in the window consists of four column: *Origin*, *Destination*, *Type* and *Condition*. Each row represents a transition between behaviors. The system transits from a state in *Origin* column to the state in the *Destination* column when a state transition of type "*Type*" occurs. *Type* can be either TOC (transition on completion), Condition (given in the fourth column), Trap or Interrupt. The *Condition* column gives the condition on which the conditional transition occurs. This window is empty if the currently selected behavior is a leaf behavior. The destination field of the table is empty if the behaviors are concurrent.

In Figure 2, when `B1` behaviorally completes (TOC) the system goes into `B2`. If the system is in `B2` and the condition '`Over=1`' occurs, then the system goes into `B3`. The box around `B1` denotes that `B1` is the start state. The oval around `B3` denotes that `B3` is the final state in the state transition diagram.

Behavior Description Window This window is used for programming the behavior of leaf nodes in SpecC. It is shown in the bottom left corner of Figure 2. This window is basically a text editor. It will be disabled for hierarchical behaviors that do not have textual descriptions.

In Figure 2, a skeletal code is shown for the leaf behavior `B1`.

Connectivity Window This window is used for defining connections between sibling behaviors and between a parent and children behaviors. The connectivity window is located in the middle-right of Figure 2. Each row and column has the name and ports of a child behavior. Connection between ports of two behaviors is denoted by the corresponding row and column intersection in the matrix. This connection can either be a global variable name or a channel. The matrix diagonal does not give the connections between the behaviors at this level. Instead, it is used to show the connection between ports of child behaviors and parent behaviors. This window is empty at higher levels of abstraction where there are only global variables.

In Figure 2, port `p1` of `B1` is connected to port `p1` of `B2` and port `p1` of `B3` through a channel `bus`. Since the same name `bus` is used, it implies that the different ports are all connected together through the `bus` channel. Port `p2` of `B1` is connected to port `p3` of `B3` through variable `L1`. The diagonal entry for port `p2` of behavior `B3` is connected to the parent behavior through a global variable `G1`.

Communication Window This window is used for reviewing the global variables, channels and interfaces available for each behavior. These are declared in this behavior and in behaviors in upper hierarchy of this behavior. This window is shown in the bottom right corner of Figure 2.

In Figure 2, the communication window clearly displays the global variables of behavior `B0`.

Scheduling Window This window is used for specifying the schedule of behaviors that execute on each processing element. Each column describes the schedule for a processing element on which more than one concurrent behavior are assigned. The columns list the serialized behaviors for that processing element. The execution duration of the behaviors is illustrated by listing a behavior more than once if it is longer than shortest behavior. Thus, the table uses normalized execution times to display the schedule for each processing element.

In Figure 2, behaviors `B1` and `B3` are assigned to `PE0` while behavior `B2` is assigned to `PE1`. The schedule shows that first `B1` executes for two units of time on `PE0`, then `B2` executes for one unit of time on `PE1` and finally `B3` executes for one unit of time on `PE0`.

We next describe the different refinement transformations in the co-design methodology. These refinement tasks are performed using the SpecEdit interface with the help of automatic tools.

3 Specification

The synthesis flow begins with a **specification** of the system being designed. The specification in a formal description language describes the functionality of the system along with performance constraints but without premature allusions to implementation details. The specification should be as close to the conceptual model of the system as possible. The specification at the behavioral level can be validated for functional correctness. The source code can be debugged with the help of a simulator and a set of test vectors. This step verifies the algorithm and functionality of the specification model. It is easier and more efficient to verify the correctness of the algorithm at a higher level than at a lower level which includes implementation details also.

Typically, a high-level specification will be composed *hierarchically.* Modular decomposition of the system simplifies the development of a conceptual view of the system and facilitates comprehension of the system's functionality. A hardware system will, in general, have either data-driven or control-driven *concurrency.* The specification will then need to describe the concurrency in the system. The specification may model FSMD or PSM. It will then include *state transitions* also. In addition, the specification may include *timing constraints* and describe how *exceptions* have to be handled.

In our system, the graphical user interface, SpecEdit is used to capture the behavioral model of the system language. SpecEdit helps in capturing and visualizing the behavioral and structural hierarchy in the specification. It also helps in specifying the state transition table, connectivity and scope of variables and channels. The behavior of leaf nodes is programmed in the SpecC language using a text editor.

After the system has been specified, a *pre-processing* step encapsulates all global variables in separate channels and adds ports to the behaviors. Thus, after the pre-processing step, every behavior accesses the global variables through ports. This ensures that there are no problems due to scope of variables when behaviors are moved around during the partitioning and scheduling stages.

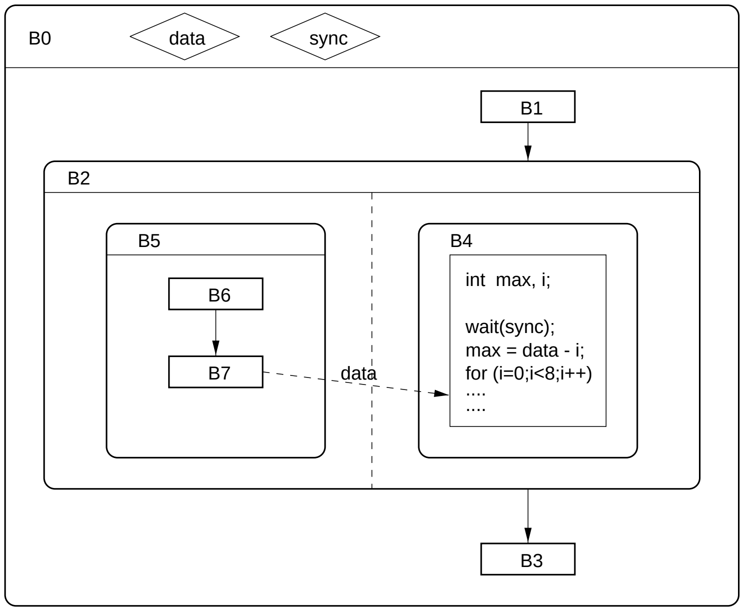
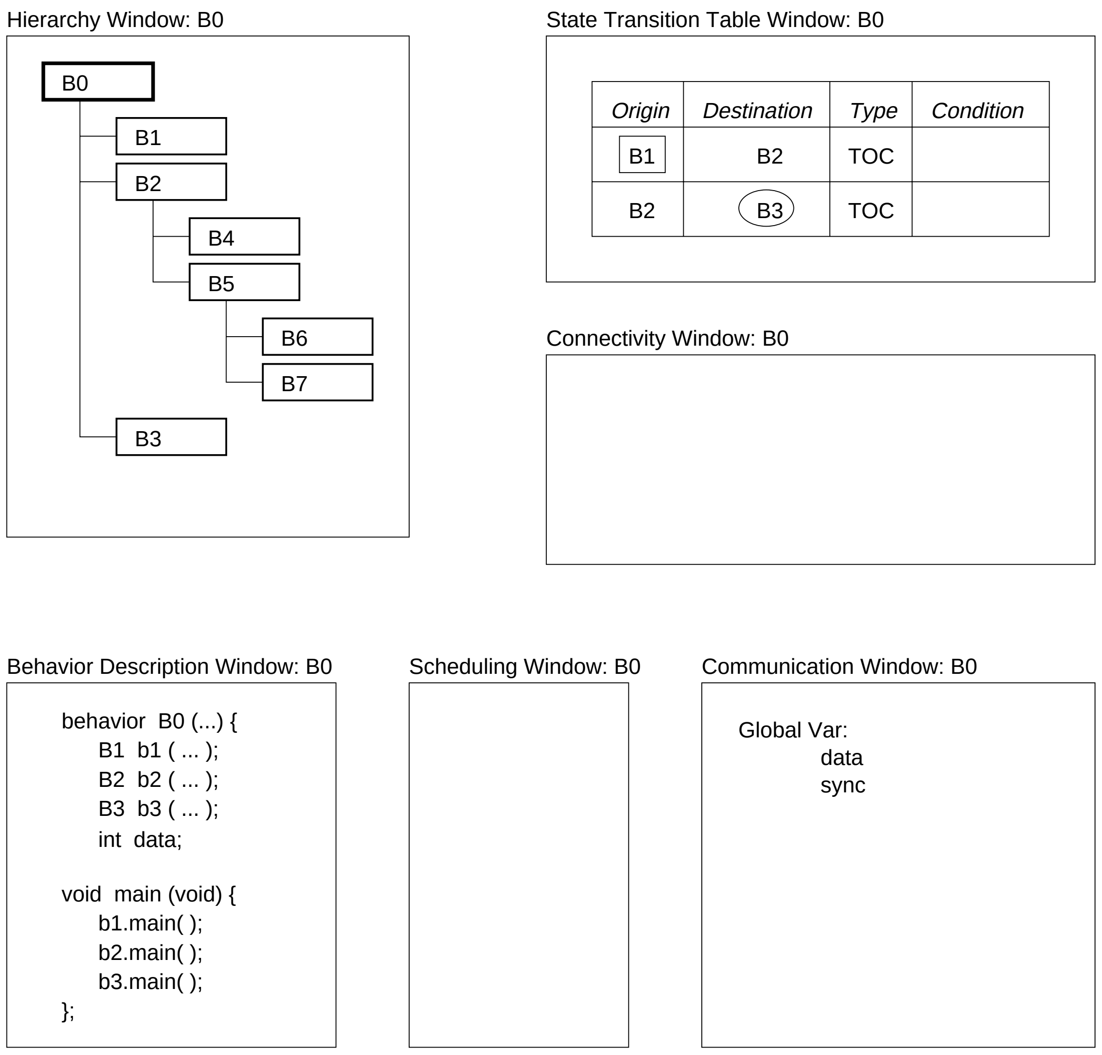


Figure 3: Conceptual specification model

We illustrate our co-design methodology with a simple example. The conceptual model is shown in Figure 3 using the PSM notation. The top level behavior `B0` consists of three sequential behaviors: `B1`, `B2` and `B3`. The system starts execution with behaviour `B1`. When `B1` completes, the system transitions to `B2`. The system transitions to `B3` on behavioral completion of `B2`. Behavior `B2` is composed of two concurrent behaviors: `B4` and `B5`. Behavior `B4` is a leaf behavior like `B1` and `B3`. On the other hand, `B5` is a hierarchical node and consists of two sequential behaviors: `B6` and `B7`.

Behaviors `B7` and `B4` communicate using a global variable: `data`. A global variable `sync` is used to denote if the `data` variable is valid or not. The variable `data` is written into by `B7` and read by `B4`. Initially, the synchronization variable `sync` is 0. `B7` writes into the `data` variable and makes `sync` as 1. `B4` waits while `sync` is 0 since it denotes that the `data` variable has invalid value. `B4` reads the `data` whenever `sync` is set to 1.

The SpecEdit view of the specification model is shown in Figure 4. Each window is for the node selected in the hieararchy window (`B0` in this example). The hieararchy window clearly displays the hierarchy in the system. The state transitition table displays that the behaviors `B1`, `B2` and `B3` are sequential. `B1` is the initial behavior and `B3` is the final. State transitions are of type TOC (transition on completion). The connectivity window is empty because the specification is at a high-level and there is no connectivity in this behavioral model. The scope of global variables is shown in the communication window. Scheduling window is empty since scheduling has not been performed as yet.



Origin	Destination	Type	Condition
B1	B2	TOC	
B2	B3	TOC	

```
behavior  B0 (...) {
    B1  b1 ( ... );
    B2  b2 ( ... );
    B3  b3 ( ... );
    int  data;

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

Global Var:
data
sync

Figure 4: SpecEdit view of the specification

4 Allocation and Partitioning

The first refinement step in the synthesis flow is the task of allocation and partitioning.

Allocation is usually done manually by the designer and basically means the selection of components from a library. In general, three types of components have to be selected from the component library: processing elements (PEs), memories and busses (where the PE can be a standard processor or custom hardware). 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 buses.

In order to perform partitioning accurate information about the design has to be obtained before. This is the task of **Estimation**. Estimation tools determine design metrics such as performance (execution time) and memory requirements (code and data size) for each part of the design with respect to the allocated components. Estimation can be performed either statically by analyzing the specification or dynamically by execution and profiling of the design description. Obviously estimation has to support both software and hardware components. The estimation results usually are stored in a table which lists each obtained design metric for each allocated component.

The table of estimation results can then be used by the partitioner to tradeoff hardware vs. software implementation. It is also used to determine whether each partition meets the design constraints and to optimize the partitions with respect to an objective function.

Usually partitioning is performed sequentially in three steps. The first step is called **behavioral partitioning** and decides which behavior is going to be executed on which PE. Obviously this includes the decision whether a behavior is implemented in software or in hardware.

For example, given an allocation of two processing elements `PE0` and `PE1` (eg. a processor and an ASIC) the specification from Figure 3 can be partitioned as shown in Figure 5. Here

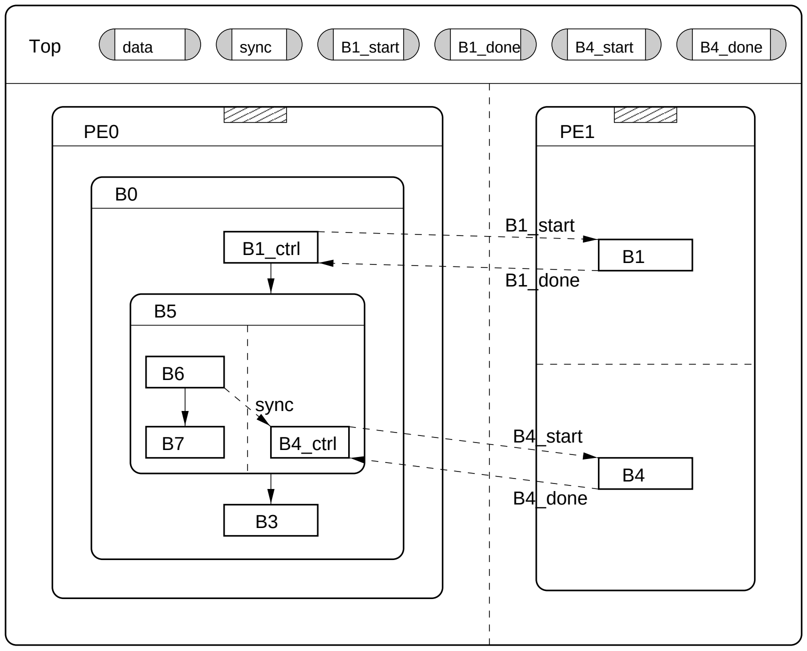


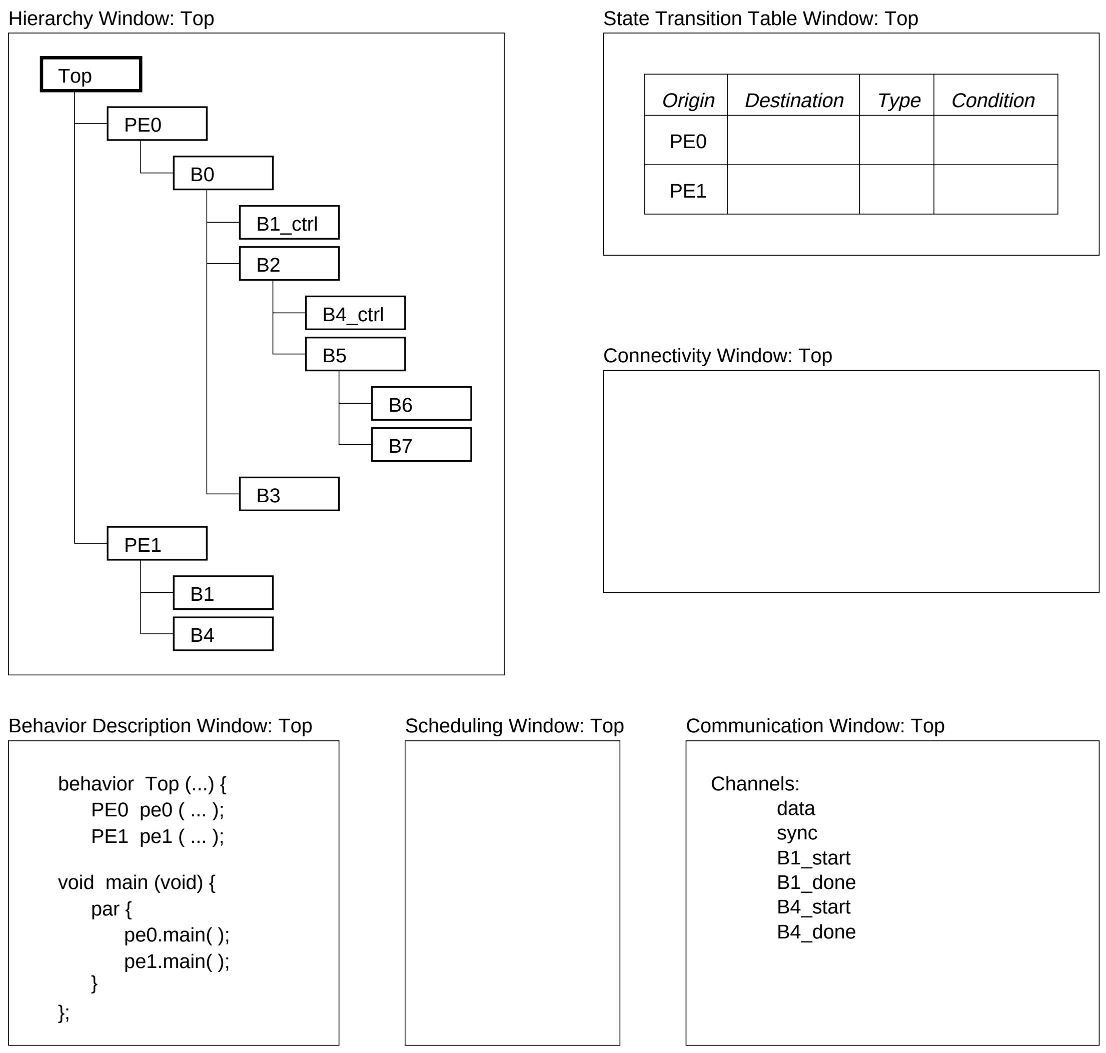
Figure 5: Conceptual model after partitioning

the behaviors `B0, B3, B5, B6, B7` are mapped to `PE0` (executing in software), 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 are synchronized with `B1` and `B4`, respectively.

Figure 6 shows the SpecEdit view of the partitioned model. Here the allocation of `PE0` and `PE1` and the partitioning of the behaviors is displayed explicitly as an additional level of hierarchy in the Hierarchy Window.

The second partitioning step is the mapping of variables onto memories. As explained in Section 3 the SpecC system first transforms all global variables in the specification into separate channels. Therefore the task of **memory partitioning** is to group these variable channels to memory channels which then will be mapped to the allocated memories.

In the third step buses are allocated which connect the PEs and the memories. The task of **bus partitioning** is to map the communication channels onto these buses. Usually this



Origin	*Destination*	*Type*	*Condition*
PE0			
PE1			

```
behavior  Top (...) {
     PE0  pe0 ( ... );
     PE1  pe1 ( ... );

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

```
Channels:
          data
          sync
          B1_start
          B1_done
          B4_start
          B4_done
```

Figure 6: SpecEdit view of partitioned model

is done during communication refinement which is described in more detail in Section 6.

In the SpecC system the sequence of allocation and partitioning tasks is determined by the designer and may contain iterations. Iterating these steps with modified allocation or changed partitioning parameters is called **design exploration**. This exploration of the design space helps to obtain an optimized implementation of the design with good performance and less costs.

5 Scheduling

In the previous stage of allocation and partitioning, each leaf behavior is assigned to a specific type of hardware to be executed. An accurate or approximate execution time for each leaf behavior is then computed. Each leaf behavior with specified hardware type and execution time now is called a subtask. In addition, the execution orders of these leaf behaviors are also passed to the scheduling stage. In the scheduling stage, the SpecC scheduler inputs these subtasks and execution orders, then according to the goal specified by the users, creates a schedule in which these subtasks can be done without violating any execution orders.

Scheduling can be categorized into either time-constrained scheduling or resource-constrained scheduling depending on the goals of scheduling. The designer specifies a set of timing constraints for **time-constrained scheduling**. Each timing constraint specifies the minimum and maximum time between two subtasks. The scheduler needs to figure out a schedule in which no subtask violates any of the timing constraints and tries to minimize the total resources. On the other hand, the designer specifies resource constraints for **resource-constrained scheduling**. The scheduler then creates a schedule such that all the subtasks are completed in shortest time possible given the restrictions on the resource usage.

In addition to the goal of scheduling, there are also two different ways to do scheduling, namely, static scheduling and dynamic scheduling. In **static scheduling**, each subtask is executed following a fixed schedule. The scheduler computes the best schedule before the desired system is synthesized. The schedule will not change at runtime. On the other hand, in **dynamic scheduling**, the execution sequence of the subtasks is determined at runtime. An embedded operation system maintains a pool of ready subtasks. A subtasks becomes ready and is put into the pool once all of its predecessors are complete. As soon as a PE finished a subtask, it picks up another from the ready pool to execute. In this case, the SpecC scheduler focuses on partitioning the subtasks in a way that maximizes PE utilization and minimizes interconnection cost.

Both static scheduling and dynamic scheduling have their advantages. Static scheduling pays no operating system overhead, which usually consumes 10% to 50% system resources. On the other hand, dynamic scheduling can achieve higher PE utilization when execution time of the subtasks change a lot at runtime. However, **Worst Case Execution Time** (WECT) of both scheduling ways are the same. The SpecC scheduler can work in both ways. The scheduler can analyze characteristics of all the subtasks, then uses the best way to do scheduling.

After a schedule is created, the scheduler moves the leaf behaviors into scheduled orders, and adds necessary signals and inserts synchronization instructions into second-level (from leaf) behaviors. The refined specification is then passed to communication synthesis stage.

To make sure the scheduled and the refined specifications are correct, both specifications can be fed into SpecC validation tools and executed in SpecC simulation engine.

Figure 7 shows an example of scheduling performed on the partitioning model from Figure 5. Behaviors `B6`, `B7` and `B3` executing on `PE0` can be serialized in time as `B6`$\rightarrow$`B7`$\rightarrow$`B3`. Behaviors `B1` and `B4` executing on `PE1` can be serialized as `B1`$\rightarrow$`B4`. Note that the scheduling algorithm can do optimizations and remove extraneous behaviors, e.g., `B1_ctrl` and `B4_ctrl`, which were introduced by the partitioning stage. The SpecEdit view of the scheduling model is shown in Figure 8.

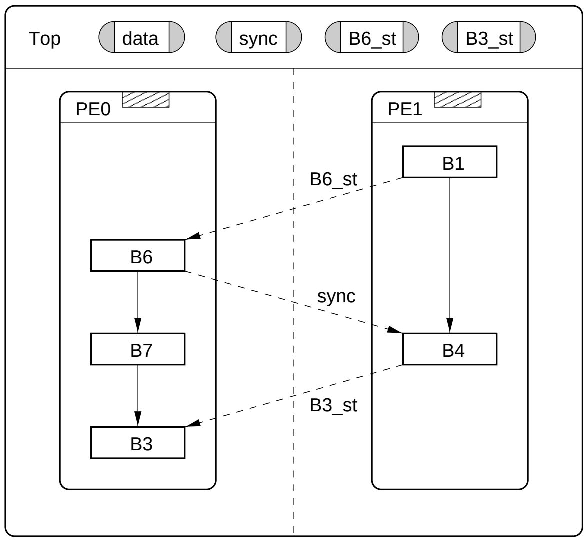


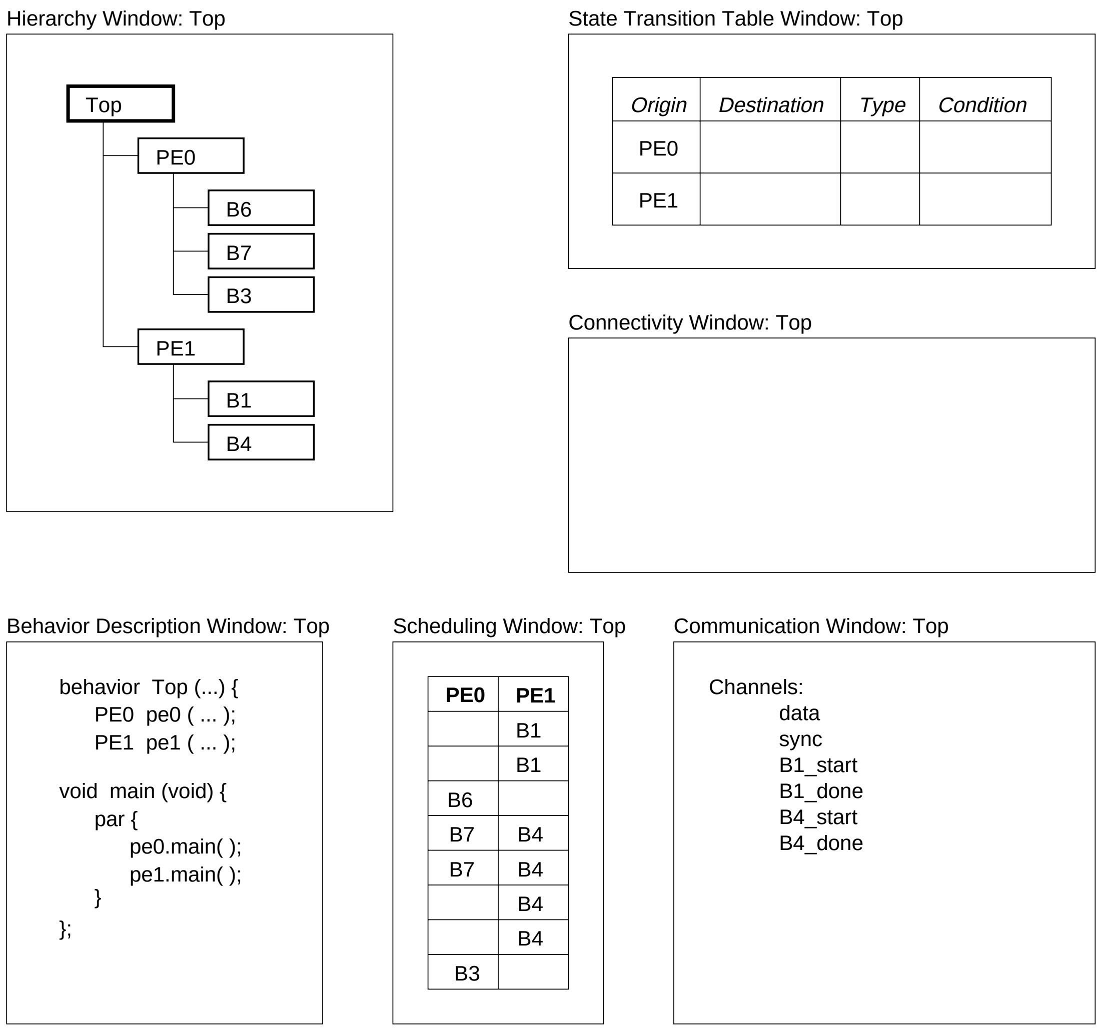
Figure 7: Conceptual model after scheduling

6 Communication Refinement

The purpose of communication refinement is to resolve abstract communication behavior through a series of refinements that lead to a specification consisting of processing elements, buses and memories. During this process, new processing elements may be introduced in the form of interfaces which serve to bridge the gap between differing protocols. It should be noted that communication up to this point is handled through remote procedure calls (**RPC**) supplied in the interface to a given channel.

Communication synthesis consists of three main tasks:

Protocol Selection The designer must select the appropriate communication medium for which to map the abstract channels. A library of generic and or common bus/protocol schemes are supplied. Further, the designer has the option of including custom protocols or customizing available protocols to suit the current application. Protocol specifications contained in the library will be written in terms of channel primitives in the SpecC language and should supply common interface function calls to facilitate


Hierarchy Window: Top
Top
PE0
B6
B7
B3
PE1
B1
B4
State Transition Table Window: Top
Connectivity Window: Top
Behavior Description Window: Top
Scheduling Window: Top
Communication Window: Top

Origin	Destination	Type	Condition
PE0			
PE1			

```
behavior  Top (...) {
    PE0  pe0 ( ... );
    PE1  pe1 ( ... );

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

PE0	PE1
	B1
	B1
B6	
B7	B4
B7	B4
	B4
	B4
B3	

```
Channels:
        data
        sync
        B1_start
        B1_done
        B4_start
        B4_done
```

Figure 8: SpecEdit view of the scheduled model

reuse. For example, a given VME bus description will supply *send()* and *receive()* as would the PCI specification. In this way, we can interchange protocols as channels and perform some simulation to obtain performance estimates. Later, these *RPC*s will be replaced by local I/O instructions for software, or additional behavior to be synthesized for hardware entities respectively.

Interface Generation In the case of previously designed components, those having their own established communication protocols, interfaces to the remaining components in the system must be generated and can take the form of transducers between components. The transducer object may be viewed abstractly as a set of dual, ordered relations between opposing signal groups. That is, a particular "view" of the protocol, either on the sender side or receiver side, is captured and reversed. As such, the transducer generates the signals necessary to satisfy the signaling requirements of either component it is linking. This object can then be realized as a FSM and may be synthesized as another hardware component, merged with another synthesizable component or be translated to software running on an associated processor.

Inlining Behavior that has been partitioned and allocated to an undefined component may have its communication functionality inlined. Namely, the behavior or logic necessary to initiate a communication transaction, formerly located in the channel object is placed inside the partition that utilizes that functionality. This communication behavior can the be handed-off to be synthesized with the rest of the component's functional behavior.

Finally, after protocols have been inlined and/or transducers have been generated, synthesis of the undefined behavioral components including interface components may be completed using current high-level synthesis techniques.

In our example, communication synthesis begins with the scheduled specification as shown in Figure 7. The designer would select one or more appropriate protocols from the design library and allocate instances of this protocol in the form of busses. In our example, a

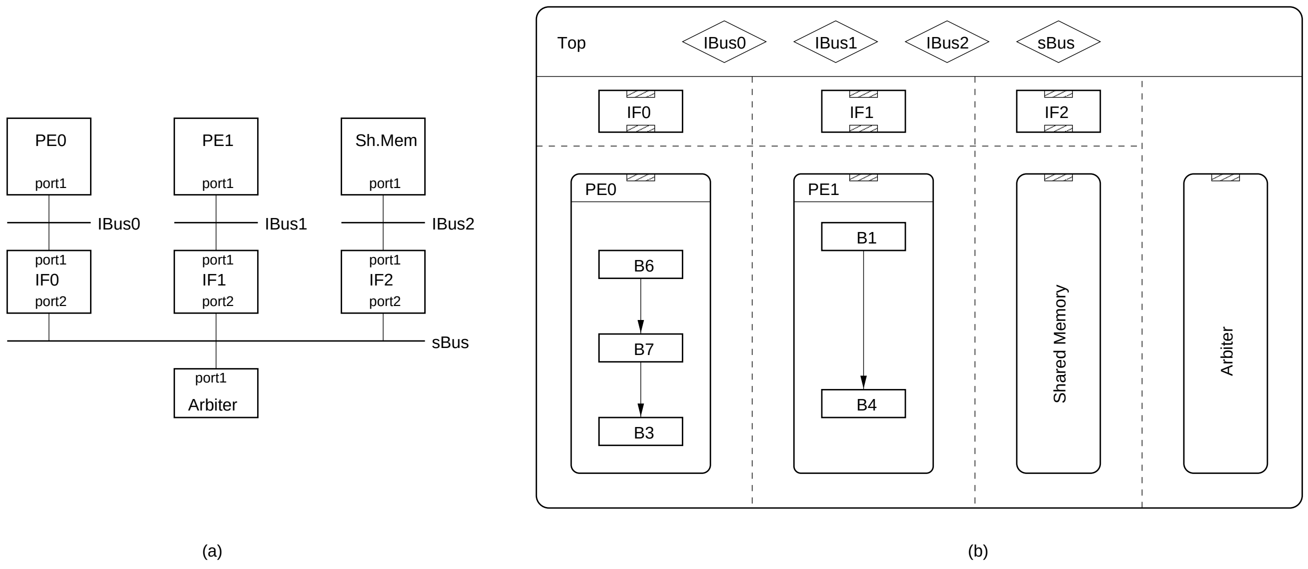
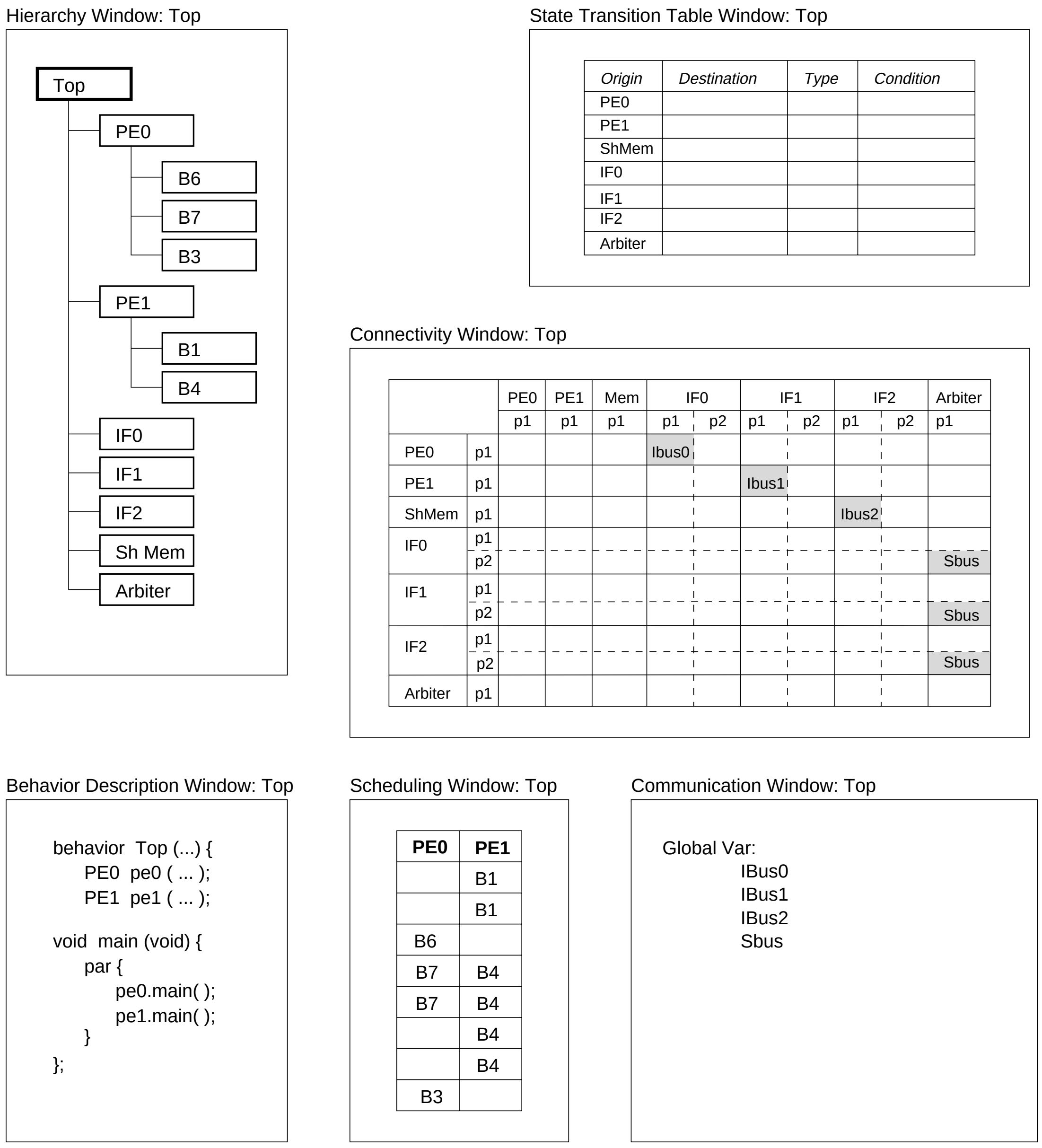


Figure 9: Communication Refinement: (a) Target architecture (b) Conceptual communication model

single protocol was selected and an instance labeled `sBus` was allocated. Channels in the system are then partitioned amongst the busses. Since a single bus was allocated, the partition is straightforward and each channel is mapped to the bus. It should be noted that channels representing global variables may also be mapped to memory elements. In the case of our example, the global variable `data` is mapped to a shared memory, indicated in the Figure 9(a).

Following the allocation and partitioning of bus elements, interfaces must be resolved between processing elements and the bus protocols they are communicating through. Common protocol between a `PE` and a bus will not require further integration. However, if protocols do not match, a transducer must be generated to link the protocols. Figure 9(a) illustrates the connections of the `PEs` to the `sBus` through interface transducers labeled `IF0`, `IF1` and `IF2`. The SpecEdit view of the communication model is shown in Figure 10.

Further refinement may take place. If a given processing element is synthesizable, meaning it is a component yet to be designed, the interface component may be integrated or *inlined*



State Transition Table Window: Top

Origin	*Destination*	*Type*	*Condition*
PE0			
PE1			
ShMem			
IF0			
IF1			
IF2			
Arbiter			

Connectivity Window: Top

		PE0	PE1	Mem	IF0		IF1		IF2		Arbiter
		p1	p1	p1	p1	p2	p1	p2	p1	p2	p1
PE0	p1				Ibus0						
PE1	p1						Ibus1				
ShMem	p1								Ibus2		
IF0	p1										
	p2										Sbus
IF1	p1										
	p2										Sbus
IF2	p1										
	p2										Sbus
Arbiter	p1										

Behavior Description Window: Top

```
behavior  Top (...) {
     PE0  pe0 ( ... );
     PE1  pe1 ( ... );

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

Scheduling Window: Top

PE0	**PE1**
	B1
	B1
B6	
B7	B4
B7	B4
	B4
	B4
B3	

Communication Window: Top

Global Var:
IBus0
IBus1
IBus2
Sbus

Figure 10: SpecEdit view of the communication model

with the processing element's behavior. Additionally, if the processing element represents a general processor and the behavior is slated for software compilation, the interface may be translated to code in terms of I/O instructions of that processor. In such a case, the target architecture shown in Figure 9(a) would reduce to two `PEs` and a shared memory, connected through a single bus `sBus`.

The result of communication synthesis, a component netlist and software code for compilation, can then be handed-off to synthesis tools and target compilers for final synthesis.

7 Hand-off

The last step in the synthesis flow was communication refinement. This step generates the hand-off model for our system. This model is then further refined using traditional back-end tools as shown in Figure 1.

The software portion of the communication (hand-off) model consists of code in C for each of the allocated processors in the target architecture. Compilers for each of the different processors are used to compile the C code. The hardware portion of the model consists of behavioral models of the synthesizable ASICs in VHDL. The behavioral models can be synthesized using high-level synthesis (HLS) tools. The interfaces between hardware and software may also be modeled in behavioral VHDL and synthesized using the same HLS tools. This design process generates the *implementation* model which consists of object code executing on the different processors and a gate-level netlist of the hardware components. The implementation model can then be simulated, verified and manufactured.

8 Conclusion

In this report we have proposed a co-design methodology for the design of embedded systems. We have described the various steps in a methodology that refines an initial specification to a final implementation model. Our methodology uses the SpecC language which provides a minimal and complete set of constructs and features required for the specification of embedded systems. The designer uses SpecC to define the smallest indivisible unit of behavior and the relationship of these units in terms of hierarchy, concurrency, state transitions etc.

We described the synthesis flow of the methodology which consists of well-defined transformations like *allocation, partitioning, scheduling* and *communication refinement.* The result of each transformation is to produce a more refined model. These models are all in the same language (SpecC) and are well defined. These well defined models and transformations provide a good base for formal verification. Additionally, each model described in SpecC is executable, in that it may be compiled, run and profiled, increasing testability. The intermediate models also document the design and make it more manageable and maintainable for future upgrades.

Hardware/software tradeoffs can be easily explored using SpecC since the language and methodology do not distinguish between hardware and software behaviors at the higher levels of abstraction. Further, at these high levels, the use of abstract channels as a communication medium allows easy insertion of IPs. *Plug-and-play* is not possible at the detailed implementation level.

We also described a graphical user interface, *SpecEdit*, for our proposed co-design exploration tools. This interface can be used for specifying the system as well as for performing the refinement transformations. Thus, SpecEdit hides the intricacies of modeling hardware by providing a graphical interface and a C-like environment for specifying the behavior of an embedded system.

This work has opened several avenues of research and uncovered several problems; additional effort will be needed to solve these problems. Implementation of the proposed co-design tool will be the next step toward a viable solution.

Acknowledgements

We would like to acknowledge the support of the various granting agencies who have contributed research funding, without which this work would not have been possible. This work was supported in part by grants from: Hitachi, Grant #-H22003; Toshiba, Grant #-TC-20881; SRC, Grant #-97-DJ-146; and Rockwell, Grant #-RSS-24141.