IP-CENTRIC METHODOLOGY AND DESIGN WITH THE SpecC LANGUAGE

System Level Design of Embedded Systems

DANIEL D. GAJSKI, RAINER DÖMER AND JIANWEN ZHU
Department of Information and Computer Science
University of California, Irvine
Irvine, California, USA

Abstract.
In this paper, we demonstrate the application of the *specify-explore-refine* (SER) paradigm for an IP-centric codesign of embedded systems. We describe the necessary design tasks required to map an abstract executable specification of the system to the architectural implementation model. We also describe the final and intermediate models generated as a result of these design tasks. The executable specification and its refinements should support easy insertion and reuse of IPs.

Although several languages are currently used for system design, none of them completely meets the unique requirements of system modelling with support for IP reuse. This paper discusses the requirements and objectives for system languages and describes a C-based language called SpecC, which precisely covers these requirements in an orthogonal manner.

Finally, we describe the design environment which is based on our codesign methodology.

1. Introduction

New technologies allow designers to generate chips with more than 10 million transistors on a single chip. The main problem at this complexity is designer productivity. Although the chip complexity measured in number of transistors per chip has increased at the rate of 60 percent per year in the past, the productivity measured in number of transistors designed per day by a single designer has increased only at the rate of 20 percent. This growing gap between the complexity and productivity rates may have the catastrophic effect of slowing down semiconductor industry.

One of the main solutions for solving this problem is increasing the level of abstraction in design of complex chips. The abstraction level increase should be reflected in descriptions, components, tools, and design methodology.

First, modelling or describing designs on the gate or RT level is not sufficient. Moving to executable specifications (behavior) and architectural descriptions (structure) is necessary to improve design productivity.

In order to explore different architectural solutions, we must use higher-level components beyond RTL components, such as registers, counters, ALUs, multipliers, etc. These higher-level components, frequently called IPs, are changing the business and design models. In order to use IPs, we need a methodology that will allow easy insertion of IPs in designs. This new methodology must have well-defined models of design representation, so that IP can be easily inserted or replaced when supplies disappear or IPs get discontinued. In order to achieve easy insertion and replacement, the design models must separate computation from communication, in addition to abstracting those two functions. This way, IP can be inserted by changing only the communication interface to the rest of the design.

Finally, the above IP-centric design methodology must be supported by CAD tools that will allow easy capture of executable specification, architecture exploration with IPs, and RTL hand-off to semiconductor fabs.

In this paper, we present such an IP-centric methodology, starting with an executable specification, define the abstract models used for architectural exploration, synthesis, and hand-off, and describe the necessary tools to support this methodology.

We also describe a C based language, called SpecC, for describing all the models in the methodology, and the SpecC Design Environment which supports all the transformations and explorations indicated in the methodology.

2. Related Work

For system-level synthesis, in particular codesign and coverification, academia, as well as industry, has developed a set of promising approaches and methodologies. Several systems already exist that assist designers with the design of embedded systems. However, none of todays systems covers the whole spectrum of codesign tasks. Instead, most systems focus on a subset of these problems.

2.1. UNIVERSITY PROJECTS

Table 1 lists some system-level projects developed by universities. Although all systems try to cover all aspects of system-level design, each of them really focuses on a subset of the tasks. Also, the target architectures addressed by the tools in many cases are quite specific and do not cover the whole design space.

TABLE 1. System-level Design Projects in Academia.

Project	University	Main Focus
Chinook	U Washington	Simulation, Synthesis
Cosmos	TIMA	Simulation, Synthesis
Cosyma	TU Braunschweig	Exploration, Synthesis
CoWare	IMEC	Interface Synthesis
Lycos	TU Denmark	Synthesis
Polis	UC Berkeley	Modelling, Synthesis
Ptolemy	UC Berkeley	Simulation
Scenic	UC Irvine	Simulation
SpecSyn	UC Irvine	Exploration
Weld	UC Berkeley	Framework

For the specification of embedded systems, standard programming languages are being used, as well as special languages developed to support important concepts in codesign directly. For the latter, two early approaches must be mentioned. Statecharts [7, 15] and SpecCharts [29, 10] use an extended finite state machine model in order to support hierarchy, concurrency and other common concepts. Both have a textual and a graphical representation. SpecCharts is the underlying language being used in the SpecSyn system [11], which is targeted at design space exploration and estimation.

In the Scenic environment [23, 14], the design is modeled with the standard programming language C++. Features not present in the language, like for example concurrency, can be specified by use of classes provided with the Scenic libraries. The SpecC system, as introduced in [39, 6] and described later in Section 4.3, goes one step further. The standard language C is extended with special constructs that support concurrency, hierarchy, exceptions, and timing issues, among others. For simulation, the SpecC language is automatically translated into a C++ program, which can be compiled and executed. This approach makes it possible for the SpecC system to focus on codesign modelling and synthesis while providing simulation, whereas Scenic mainly targets only simulation.

Similar to the specification language, the design representation being used internally in a codesign system is important. Usually every system has its own representation. The Polis system [2], targeted at small reactive embedded systems, uses the codesign finite state machine (CFSM) model [4] to represent the designs. Since this model is formally defined, it is also a suitable starting point for formal verification.

Most codesign systems can be classified as either simulation oriented, or synthesis oriented. A typical representative for simulation oriented systems is the Ptolemy frame work [22, 19]. Ptolemy models a design as a hierarchical network of heterogeneous subsystems and supports simultaneous simulation of multiple models of computation, such as for example synchronous data flow (SDF).

On the other hand, several systems are mainly synthesis oriented. In this category, Cosmos [35, 18] targets at the development of multiprocessor architectures using a set of user-guided transformations on the design. For the Cosyma [9, 16, 30] and the Lycos [26] system, the target architecture is an embedded micro architecture consisting of one processor with a coprocessor, e.g. an ASIC.

Interface and communication synthesis are addressed in particular by the Chinook [5] and CoWare [31] systems. Chinook is targeted at the design of control-dominated, reactive systems, whereas CoWare addresses the design of heterogeneous DSP systems.

As a special framework, the Weld project [3] addresses the use of networking in electronic design. It defines a design environment which enables web-based computer aided design (CAD) and supports interoperability via the internet.

2.2. COMMERCIAL SYSTEMS

A growing number of commercial tools are being offered by the EDA companies. However, they tend to either solve a particular problem as a point tool in the codesign process, e.g. cosimulation, or focus on one particular application domain, e.g. telecommunications.

For modeling and analysis at the specification level, Cadence and Synopsys offers tools (SPW and COSSAP, respectively) to support easy entry and simulation of block diagrams, a popular paradigm used in the communication community.

Another category of simulation tools is targeted at verification for design after backend synthesis. A representative is Seamless CVE from Mentor Graphics, which speeds up cosimulation of hardware and software by suppressing the simulation of information unrelevant to the hardware software interaction. Such information may include instruction fetch, memory access, etc. A similar tool is Eaglei from ViewLogic.

A variety of backend tools exists. The most widely used retargetable compiler is the GNU C compiler. However, since it is designed to be a compiler for general purpose processors, upgrading it into an aggressive, optimizing compiler for an embedded processor with possibly a VLIW datapath and multiple memory banks can be a tremendous effort. Although assembly programming prevails in current practice, new tools are expected to emerge as research in this area matures. The Behavioral Compiler from Synopsys, Monet from Mentor Graphics, and XE of Y-Explorations, are examples of high-level synthesis tools starting from a hardware description language. The Protocol Compiler of Synopsys exploits the regular expression paradigm for the specification of communication protocols and synthesizes interface circuitries between hardware modules.

There is a limited number of commercial tools offered for system-level synthesis. Among the few is the CoWare system, which targets at the hardware software interfacing problem. VHDL+ of ICL, also provides an extension of VHDL, which helps to solve the same problem.

There are a rapidly growing number of vendors for reusable components, or IP products for embedded systems. A traditional software component is the embedded operating system, which usually requires a small memory, and sometimes real time constraints have to be respected. Examples are VxWorks from Wind River, Windows CE from Microsoft, JavaOS from Sun Microsystems, to name just a few. The Inferno operating system from Lucent is especially designed for networking applications. The hardware IP vendors offer cores ranging from the gate and functional unit level, for example Synopsys Designware, to block level, for example Viterbi decoders and processors. They are often provided with a simulation model or a synthesizable model in VHDL or Verilog. While integrating these cores into a system-on-a-chip is not as easy as it appears, new methodologies, such as those proposed in the academia, and new standards, such as those prepared in the VSI alliance, are expected to make the plug-and-play capability possible.

3. System Design Methodology

A methodology is a set of models and transformations, possibly implemented by CAD tools, that refines the abstract, functional or behavioral specification into a detailed implementation description ready for manufacturing. The system methodology [12] starts with an *executable specification* as shown in Figure 1. This specification 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. The specification is captured directly in a formal specification language such as SpecC (see Section 4.3), that supports different models in the methodology.

Since designers do not like to learn the syntax and semantics of a new

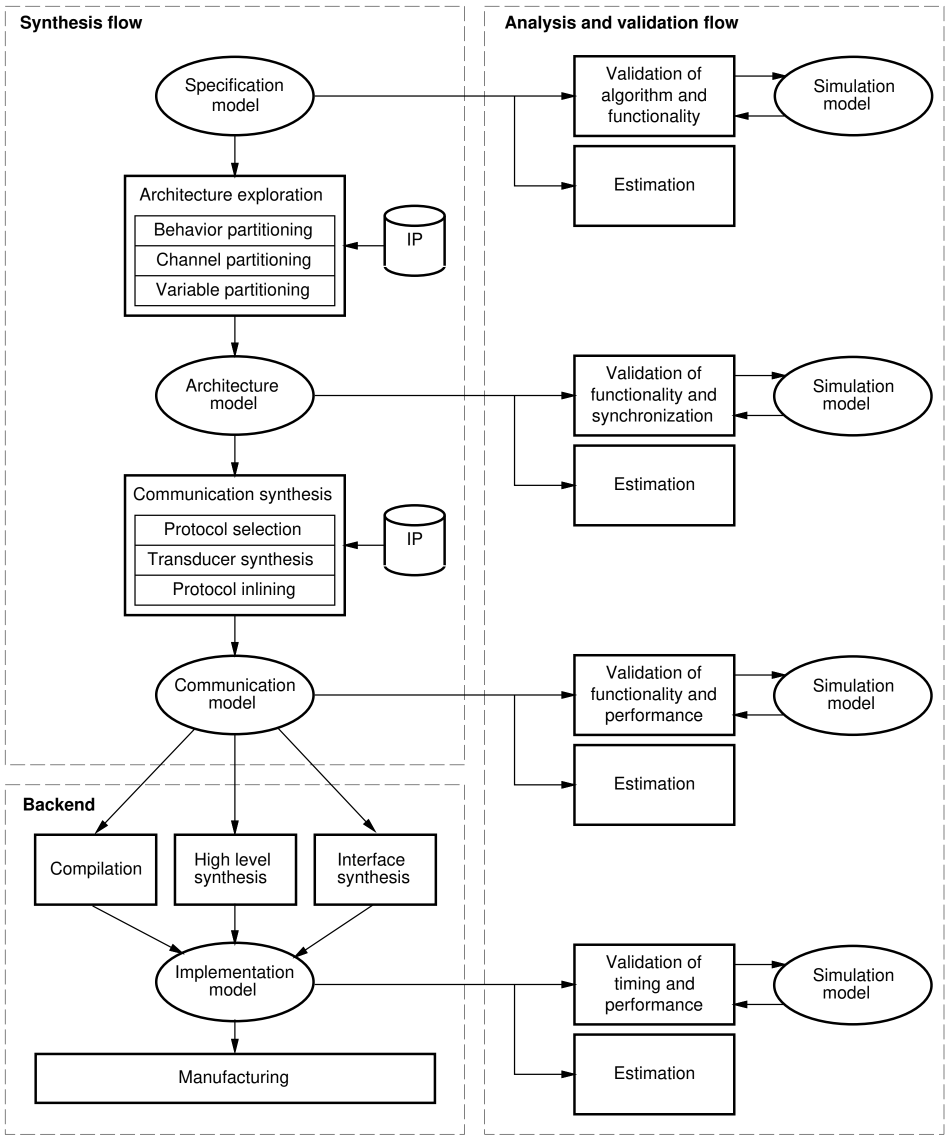


Figure 1. The codesign methodology in the SpecC Design Environment

language, the executable specification can be captured with a graphical editor that generates the specification from well-known graphical forms, such as block diagrams, connectivity tables, communication channels, timing diagrams, bubble charts, hierarchical trees, scheduling charts, and others. Such a graphical editor must also provide support for manual transformations of one model to another in the methodology.

As shown in Figure 1, the synthesis flow of the codesign process consists of a series of well-defined design steps which will eventually map the executable specification to the target architecture. In this methodology, we distinguish two major system level tasks, namely architecture exploration and communication synthesis.

Architecture exploration includes the design steps of allocation and partitioning of behaviors, channels and variables. *Allocation* determines the number and the types of the system components, such as processors, ASICs 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.

Then, *behavior partitioning* distributes the behaviors (or processes) that comprise the system functionality amongst the allocated processing elements, whereas *variable partitioning* assigns variables to memories and *channel partitioning* assigns communication channels to busses. *Scheduling* is used to determine the order of execution of the behaviors assigned to the processors.

Architecture exploration is an iterative process whose final result is the definition of the system architecture. In each iteration, estimators are used to evaluate the satisfaction of the design constraints. As long as any constraints are not met, component and connectivity reallocation is performed and a new architecture with different components, connectivity, partitions, schedules or protocols is evaluated.

After the architecture model is defined, *communication synthesis* is performed in order to obtain a design model with refined communication. The task of communication synthesis includes the selection of communication protocols, synthesis of interfaces and transducers, and inlining of protocols into synthesizable components. Thus, communication synthesis refines the abstract communications between behaviors into an implementation.

It should be noted that the design decisions in each of the tasks can be made manually by the designer, e. g. by using an interactive graphical user interface, as well as by automatic synthesis tools.

The result of the synthesis flow is handed-off to the backend 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 or C code. The backend tools include compilers, a high-level synthesis tool 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 synthesis tool is used to synthesize the functionality mapped to custom hardware. The interface synthesizer is used to implement the functionality of interfaces needed to connect different processors, memories and IPs.

During each design step, the design model is statically analyzed to estimate certain quality metrics such as performance, cost and power consumption. This design model is also used in simulation to verify the correctness of the design at the corresponding step. 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 synchronization between behaviors on different processing elements (PEs). After communication synthesis, the simulation model is used to verify the performance of the system including computation and communication.

At any stage, if the verification fails, a debugger can be used to locate and fix the errors. Usually, standard software debuggers can be used which provide the ability to set break points anywhere in the source code and allow detailed state inspection at any time.

3.1. IP REQUIREMENTS

The use of Intellectual Property introduces additional requirements on the system design methodology. In order to identify the specification segments that can be implemented by an IP, or to replace one IP by another one, the system specification and its refined models must clearly identify the specific IP segment or the IP functionality must be deduced from the description. On the other hand, if the meaning of a model or one of its parts is difficult to discover, it is also difficult to see whether an IP can be used for its implementation.

This situation is well demonstrated in a much broader problem of design methodologies: simulatable vs. synthesizable languages. We know that almost any language (C, C++, Java, VHDL, Verilog, etc.) can be used for writing simulatable models. However, each design can be described in many different ways, all of them producing correct simulation results. Therefore, an IP function can be described in many different ways inside the system specification without being recognized as an IP description. In such a case, IP insertion is not possible. Similarly, replacing one IP with another with slightly different functionality or descriptions is not possible.

For example, a controller, whose computational model is a finite state machine, can be easily described by a `case` statement in which the cases represent the states. Similarly, an array of coefficients can be described with a `case` statement in which the cases represent the coefficient indices. In order to synthesize the description with these two case statements, we have to realize that the first statement should be implemented as a controller and the second as a look-up ROM. If the designer or a synthesis tool cannot distinguish between these two meanings, there is no possibility that a correct implementation can be obtained from that description although it will produce correct simulation results.

Therefore, in order to synthesize a proper architecture, we need a specification or a model that clearly identifies synthesizable functions including IP functions. In order to allow easy insertion and replacement of IPs, a model must also separate computation from communication, because different IPs have different communication protocols and busses connecting IPs may not match either of the IP protocols. The solution is to encapsulate different IPs and busses within virtual components and channels by introducing wrappers to hide detailed protocols and allow virtual objects to communicate via shared variables and complex data structures. In the methodology presented in Figure 1, the executable specification is written using shared variables for communication between behaviors or processes, while models used for architecture exploration use virtual components and channels for easy insertion and replacement of IPs. The final communication model exposes the protocols and uses again shared variables to describe individual wires and busses used in communication. Thus, the architecture exploration is performed on the model that clearly separates computations (behaviors) from communication (channels) and allows a *plug-and-play* approach for IPs.

However, there is a difference between functions defined in a channel and functions in a behavior. While the functions of a behavior specify its own functionality, the functions of a channel specify the functionality of the caller, in other words, when the system is implemented, they will get *inlined* into the connected behaviors or into transducers between the behaviors. When a channel is inlined, the encapsulated variables are exposed serving as communication media, and the functions become part of the caller. This is shown in Figure 2(a) where the channel `C` connecting behaviors `A` and `B` is inlined, assuming that `A` and `B` will be implemented as custom hardware parts. In such custom parts, the computation and communication will be realized by the same datapath and controlled by one controller.

The situation is different when a behavior is not synthesizable, such as in a processor core with a fixed protocol. This can be modelled using a *wrapper* which is a channel encapsulating a fixed behavior while providing higher-level communication functions that deal with the specific protocol of the internal component. For example, a MPEG decoder component with a wrapper can be used by other behaviors simply by calling the decode function provided by the wrapper. Figure 2(b) shows the inlining of the wrapper in component `A` allowing the communication between `A` and `IP` to use the IP protocol. On the other hand, whenever two channels (or wrappers) encapsulating incompatible protocols need to be connected, as shown in Figure 2(c), an interface component or transducer has to be inserted into

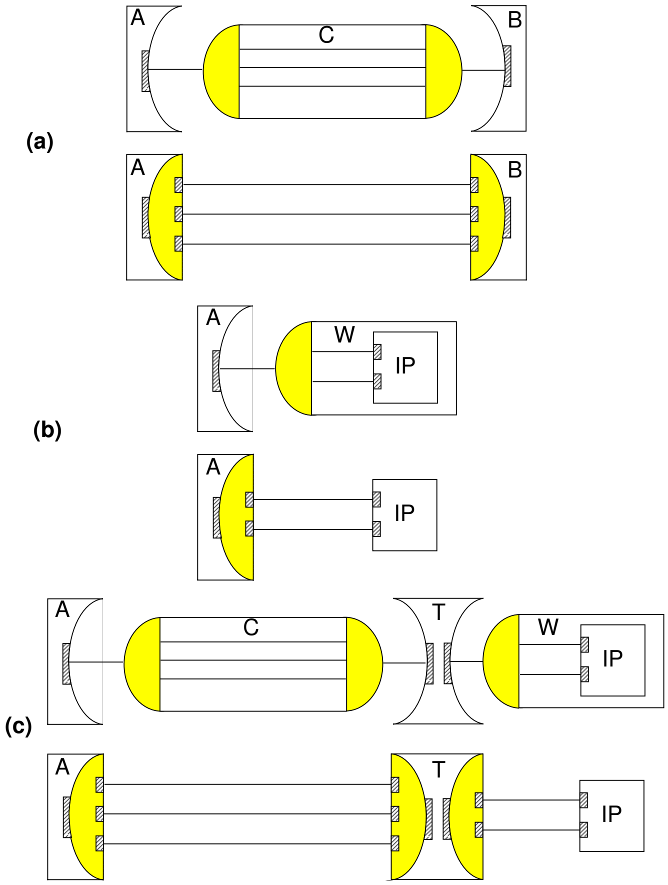


Figure 2. Channel inlining: (a) two synthesizable behaviors connected by a channel, (b) synthesizable behavior connected to an IP, (c) a synthesizable behavior connected to an IP through an incompatible channel.

which the channel functions will be inlined during communication refinement.

Next, we give a detailed description of each refinement task in the synthesis flow of the codesign process.

3.2. SPECIFICATION

The synthesis flow begins with a specification of the system being designed. 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, and errors can be located 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 Section 4.3, to capture the high-level specification of the system under design. SpecC [39] is a superset of C [37] and provides special language constructs for modelling *concurrency*, *state transitions*, *structural and behavioral hierarchy*, *exception handling*, *timing*, *communication* and *synchronization*. This is in contrast to popular hardware description languages, like VHDL [17] and Verilog [34], which do not include explicit constructs for state transitions, communication, etc., and standard programming languages, like C/C++ [33] and Java [1], that cannot directly model timing, concurrency, structural hierarchy, and state transitions. Thus, SpecC is easily used for specifying FSMD or PSM computational models [11].

In addition, SpecC is synthesizable and aids the designer in developing *"good"* designs by providing the above listed features 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 to support *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* [39]. The system behavior includes only the computation portion and uses a model similar to remote procedure calls (RPC) for communication. For implementation, the actual communication methods are resolved and inlined during the refinement process.

In the SpecC Design Environment, the SpecC Editor is used to capture the specification model of the system under design. The editor helps in capturing and visualizing the behavioral and structural hierarchy in the specification. It also supports the specification of the state transition tables, component connectivity and scope of variables and channels with a graphical user interface. Only the behavior of leaf nodes is programmed by use of a standard text editor.

We illustrate our codesign methodology with a simple example. The specification 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 behavior `B1`. When `B1` completes, the system transitions to `B2`. Finally, the system transitions to `B3` on behavioral

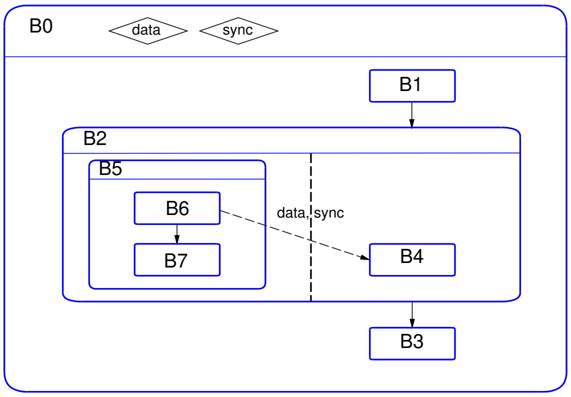


Figure 3. Specification model

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`.

The leaf behaviors `B6` and `B4` communicate using global variables. First, `B6` synchronizes its execution with `B4` by using the `sync` event, as shown with the dashed arrow in Figure 3. Then, data is exchanged via the (possibly complex) variable `data`.

It should be emphasized that in the specification, the communication over shared global variables does not imply anything about the way it will be implemented later. For the implementation, this communication scheme could be transformed into a remote procedure call mechanism, or actually a shared memory model. Also, please note that we use the global variable communication to make the example simple. For a larger system, the designer is free to use, for example, communication via channels (as described in Section 4.3) in the specification model as well.

3.3. 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 means the 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 already designed parts 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. In the SpecC system, the *partitioned model*, like the initial specification, is modeled in SpecC.

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 specification 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 designer (or an automated 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.

In our methodology, architecture exploration is separated in three steps, namely behavior partitioning, channel partitioning and variable partitioning, which can be executed in any order.

3.3.1. *Behavior partitioning*

First, behaviors are partitioned among the allocated processing elements. This decides which behavior is going to be executed on which PE. Thus, it separates behaviors to be implemented in software from behaviors to be implemented in hardware.

For example, given an allocation of two processing elements `PE0` and `PE1` (e.g. a processor and an ASIC), the specification model from Figure 3 can be partitioned as shown in Figure 4. Here, the behaviors `B0`, `B2`, `B3`, `B5`, `B6` and `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 synchronize the execution with `B1` and `B4`, respectively. Also, for this synchronization, four global variables, `B1_start`, `B1_done`, `B4_start` and `B4_done`, are introduced, as shown in Figure 4.

The assignment of behaviors to a sequential PE, for example a processor, requires *scheduling* to be performed. As a preparation step, the approximate execution time for each leaf behavior, which was already obtained from

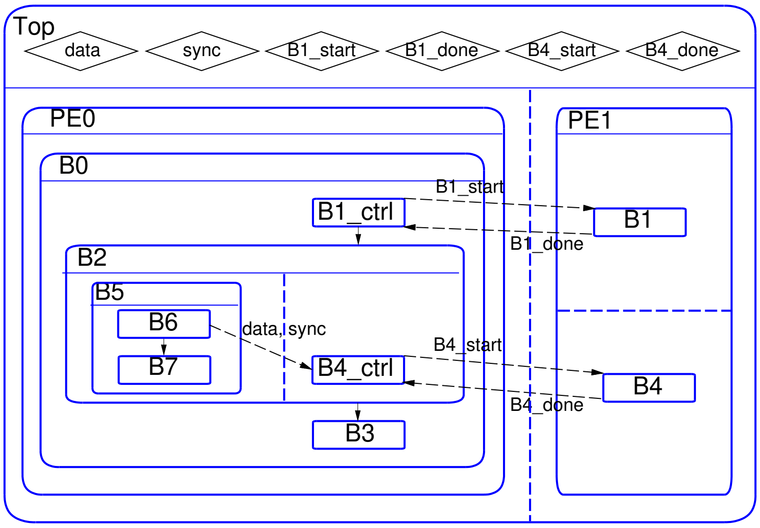


Figure 4. Intermediate model reflecting behavior partitioning

estimators for the partitioning phase, is annotated with the behaviors, so that it can be used during 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 sequential execution of the behaviors.

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 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 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. In this methodology, resource-constraint scheduling is used, since the available resources are already determined during allocation.

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

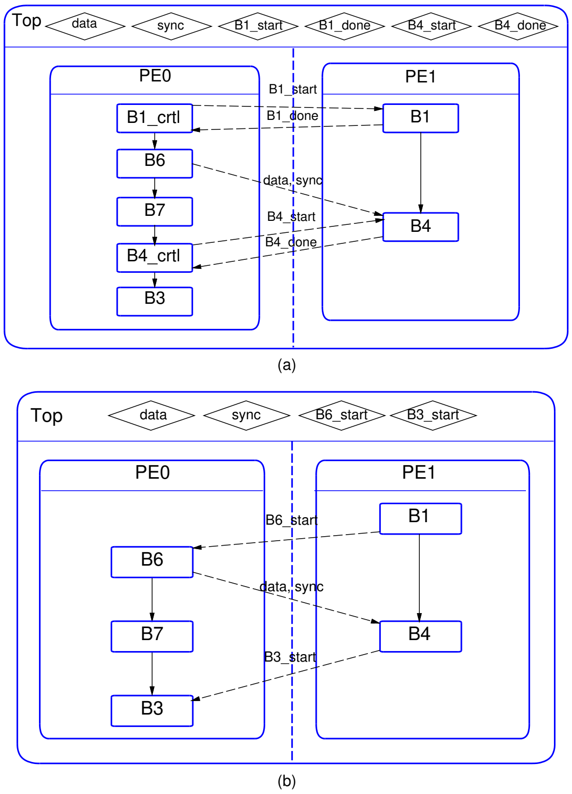


Figure 5. Intermediate model after scheduling: (a) non-optimized, (b) optimized.

run time. On the other hand, in *dynamic scheduling*, the execution sequence of the subtasks is determined at run-time. An *embedded operating system* 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 4. Figure 5 shows how scheduling is performed with the example. As shown in Figure 5(a), the behavioral hierarchy inside `PE1` is flattened and its leaf behaviors are sequentialized. For `PE2`, the behavior changes from (potentially) concurrent to sequential execution.

Due to scheduling, some explicit synchronization can become redundant. Figure 5(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 their synchronization signals.

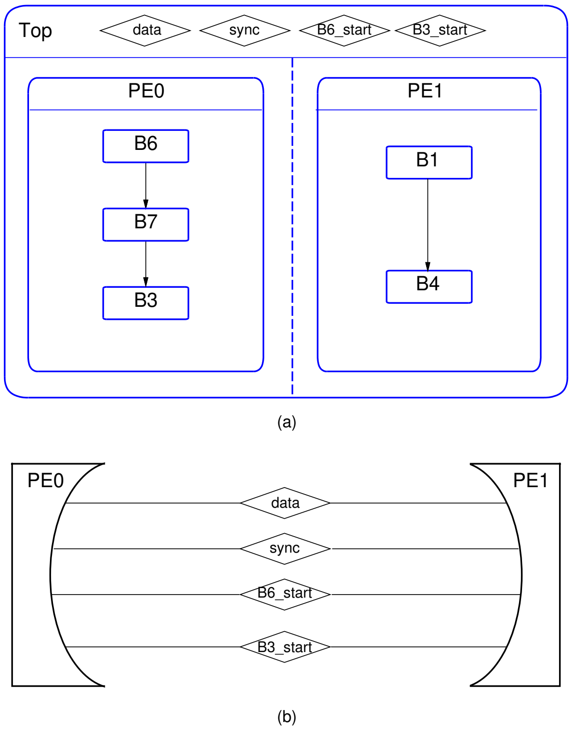


Figure 6. Model after behavior partitioning

After scheduling is done, the task of PE allocation and behavior partitioning is complete. Figure 6 shows the resulting design. In the lower part, it also shows the example from a structural view which emphasizes on the communication structure. This representation helps to explain the insertion of communication channels and memory behaviors which is described next.

3.3.2. *Channel partitioning*

Up to this point, communication between the allocated PEs is still performed via shared variables. In order to refine this abstract communication, these variables are first grouped and encapsulated in virtual channels.

In other words, in order to define the communication structure of the system architecture, channels are allocated into which the variables are partitioned. Later, during communication synthesis, these virtual channels will be refined to system busses.

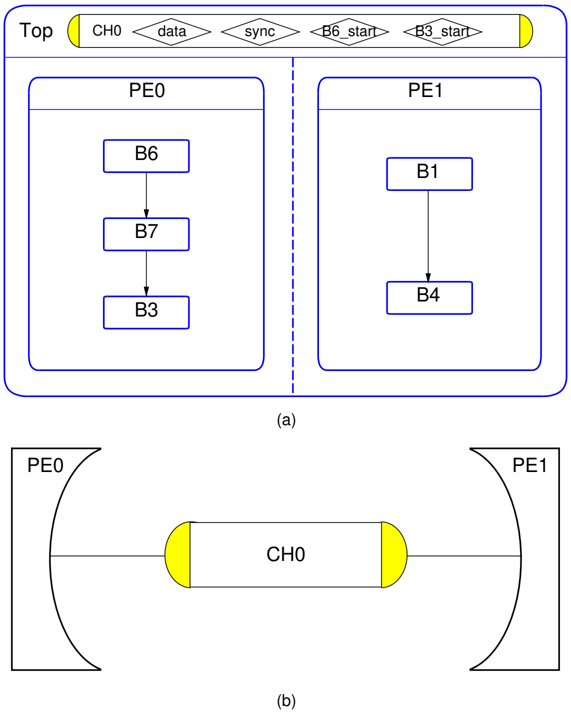


Figure 7. Model after channel partitioning

In our example, channel partitioning is performed as shown in Figure 7. Here, due to the simplicity of the example, channel partitioning is easy. Since we have to connect only two PEs, we allocate one channel `CH0` and group all the variables into this channel, as shown in Figure 7(a).

Please note that in Figure 7, the leaf behaviors of `PE0` and `PE1`, which formerly could access the shared variables directly, are transformed in order to use the protocols supplied by the channel. For example, the behavior `B4`, formerly containing statements like `x = data`, is now transformed into one which uses statements like `x = CH0.read_data()` instead.

3.3.3. *Variable partitioning*

The last partitioning step is the allocation of memory components and the mapping of variables onto these memories. This is called *variable partitioning*.

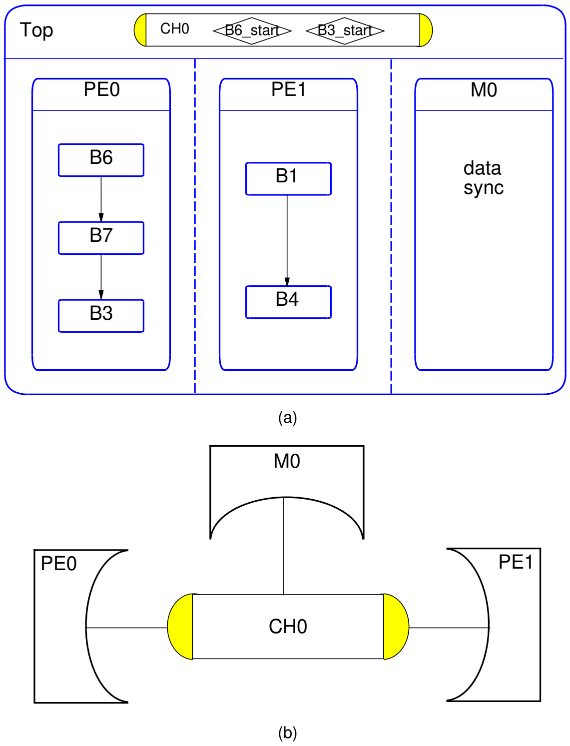


Figure 8. Model after variable partitioning

Variable partitioning essentially decides whether a variable used for communication is stored in a memory outside the PEs or is directly sent by use of message passing. It also assigns variables to be stored in a memory to one of the allocated memory behaviors.

In our example, a single memory behavior `M0` is allocated and inserted in the architecture, as shown in Figure 8. The four variables, that were formerly kept locally in the channel `CH0`, are partitioned into two groups. The possibly complex variable **`data`** and **`sync`** are assigned to the memory `M0`, whereas message passing is used for the synchronization variables **`B6_start`** and **`B3_start`**, as illustrated in Figure 8(a).

Please note that the channel `CH0` is modified only internally in order to accommodate the communication to the inserted memory. Its interfaces to the PEs and the connected PEs themselves are not affected by this refinement step and, thus, need not be modified.

After variable partitioning, the task of architecture exploration is complete. However, it should be emphasized that in the SpecC environment, the sequence of allocation and partitioning tasks is determined by the designer and usually contains several iterations. The designer repeats these steps based on his experience and the performance 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 in contrast to other environments where, for example, translating C code to VHDL and vice versa must be performed and verified. This design space exploration helps to obtain a "good" system architecture and finally an optimized implementation of the design with good performance and less costs.

3.4. COMMUNICATION SYNTHESIS

The purpose of communication synthesis is to resolve the abstract communication behavior in the virtual architecture through a series of refinements that lead to an implementation consisting of processing elements, busses and memories. During this process, new processing elements may be introduced in the form of transducers which serve to bridge the gap between differing protocols.

In our methodology, communication synthesis consists of three tasks, namely protocol selection, interface synthesis and protocol inlining.

3.4.1. *Protocol selection*

The designer selects the appropriate communication medium for mapping the abstract channels from a library of bus/protocol schemes during the task of protocol selection. 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 are written in terms of channel primitives of the SpecC language and 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 additional behavior to be synthesized for hardware entities.

The virtual channels in the design model after architecture exploration can now be refined into hierarchical channels which are implemented in

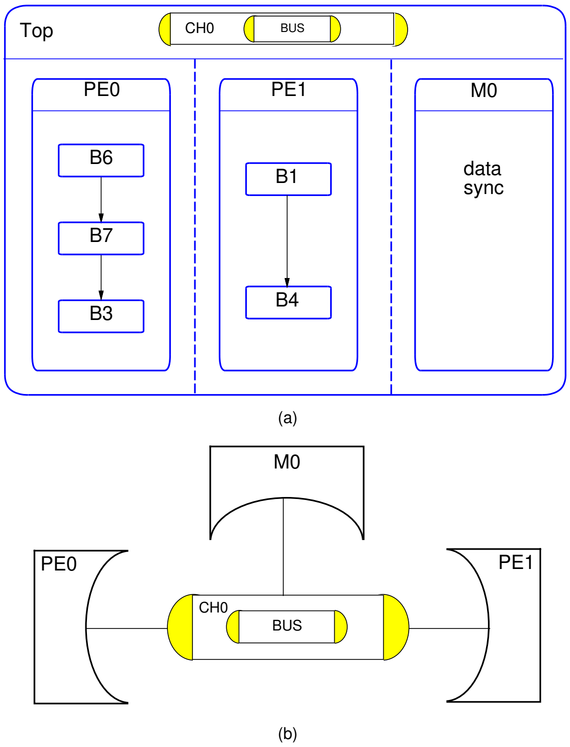


Figure 9. Model after bus allocation

terms of selected lower level channels. This process can be either manual or automatic. The cost of manual refinement is still lower than in the traditional way, since the user does not have to bother about issues such as detailed timing, thanks to the abstraction provided by the channel construct. Automatic refinement will generate code which assembles high level messages from low level messages, or vice versa, that can be delivered by the lower level channel.

In our example, this refinement is shown in Figure 9. A single bus channel `BUS`, e. g. a PCI bus, is selected in order to carry out the communication between the three behaviors. The methods of the virtual channel `CH0` are refined to use the methods of the bus protocol that is encapsulated in the channel `BUS`. It should be noted, that the channel hierarchy, as shown in Figure 9(b), directly reflects the layers of the communication between the PEs.

3.4.2. *Protocol inlining*

During the task of protocol inlining, methods that are located in the channels, are moved into the connected behaviors if these behaviors were assigned to a synthesizable component. Thus, the behavior now includes the communication functionality also. Its port interfaces are composed of bit-level signals as compared to the abstract function calls before inlining was done. The "communication behavior" can then be synthesized/compiled with the rest of the component's functional (computational) behavior.

It should be noted that, since all information necessary is available in the design model, protocol inlining is a fully automatic task that requires no designer interaction.

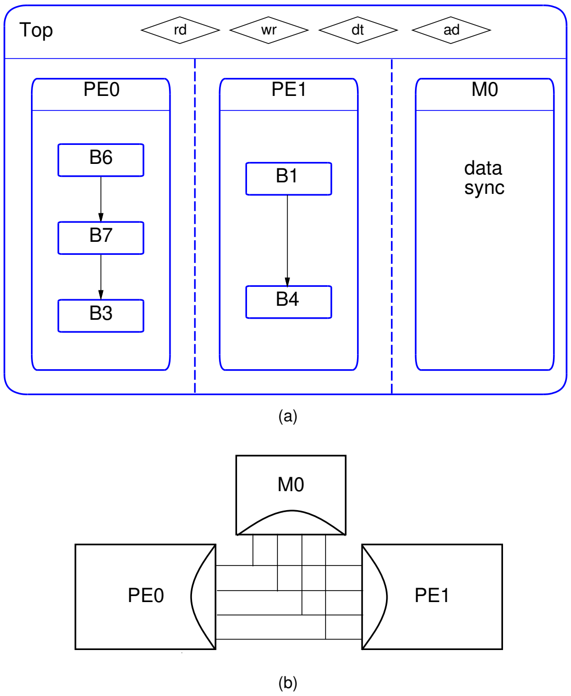


Figure 10. Model after inlining in synthesizable behaviors

In case, in our example, `PE0`, `PE1` and `M0` are all synthesizable behaviors, the methods of both channels `CH0` and `BUS` can be inlined into the behaviors, as shown in Figure 10. After this protocol inlining, the channel variables **`rd`**, **`wr`**, **`dt`** and **`ad`** are exposed and serve as interconnection wires between the accordingly created ports of the components.

3.4.3. *Transducer synthesis*

On the other hand, the designer may decide to use a non-synthesizable IP to implement a behavior in the system architecture. Such an IP can be selected from the component library, which contains both behavior models and *wrappers* which encapsulate the proprietary protocols of communication with the IPs. In the design model, a IP is introduced by creation of a *transducer* which bridges the gap between the IP component and the channels which the original behavior is connected to. Again, such a transducer can be easily created manually thanks to the high level nature of the wrapper and the connected channel.

It should be emphasized that the replacement of synthesizable behaviors with IP components is not limited to the communication synthesis stage. In fact, it is possible at any time during architecture exploration and communication synthesis. The key to this feature is the encapsulation of IP components in wrappers.

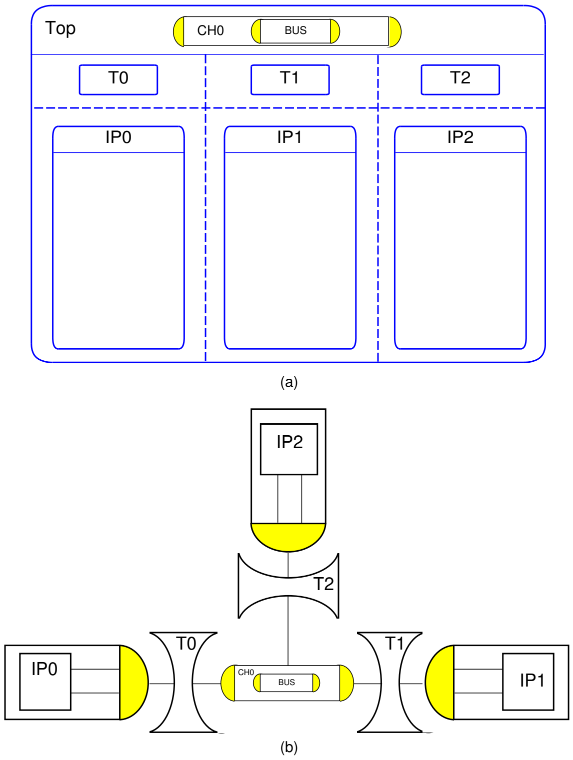


Figure 11. Alternative model with IPs

In our example, Figure 11 shows the design model where the synthesizable behaviors `PE0`, `PE1` and `M0` are all replaced with non-synthesizable IP components encapsulated in wrappers and connected to the channel `CH0` via the inserted transducers `T0`, `T1` and `T2`.

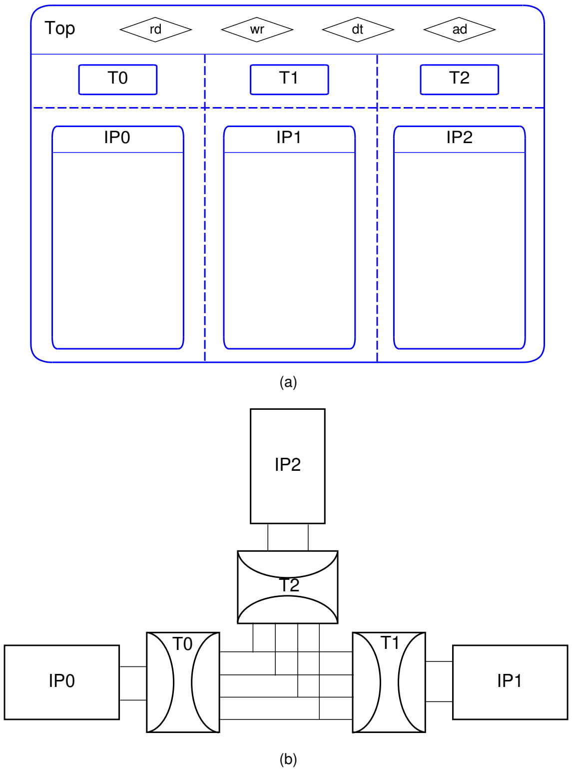


Figure 12. Model with IPs after inlining

Finally, Figure 12 shows this new model with the inserted 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 **`rd`**, **`wr`**, **`dt`** and **`ad`** are exposed and serve as interconnection wires between the transducers.

3.5. HAND-OFF

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 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.

Retargetable compilers or special compilers for each of the different processors can be used to compile the C code. The hardware portion of the model consists of synthesizable, behavioral models in C or VHDL. The behavioral models can be synthesized using standard high-level synthesis (HLS) tools. The interfaces between hardware and software components are also separated in software (device drivers) and hardware parts (transducers). Thus, this is just a special case of hardware and software parts and can be handled in the same way.

Finally, this design process generates the *implementation model* consisting of assembly code executing on the different processors and a register transfer level (RTL) or gate-level netlist of the hardware components. Thus, the implementation model is ready for manufacturing.

4. The Language

With this generic methodology in mind, Section 4.1 discusses the requirements and goals for system description languages and Section 4.2 compares traditional languages with these requirements. Since none of the languages supports all concepts a new modelling language called SpecC is proposed and presented in Section 4.3.

4.1. MODELLING LANGUAGE REQUIREMENTS

For the codesign methodology presented above, it is desirable that *one* language is used for all models at all stages. Such a methodology is called *homogeneous* in contrast to heterogeneous approaches [19, 31], where a system is specified in one language and then is transformed into another, or is represented by a mixture of several languages at the same time.

This homogeneous methodology does not suffer from simulator interfacing problems or cumbersome translations between languages with different semantics. Instead one set of tools can be used for all models and synthesis tasks are merely transformations from one program into a more detailed one using the same language. This is also important for *reuse*, because design models in the library can be used in the system without modification (*"plug-and-play"*) and a new design can be used directly as a library component.

System design places unique requirements on the specification and modelling language being used. In particular the language must be

1. executable,
2. modular and
3. complete.

1. Executability of the language is of crucial importance for simulation. The system specification must be validated to assure that exactly the intended functionality is captured. Simulation is also necessary for the intermediate design models whose functionality must be equivalent to the behavior of the model before the refinement.
2. Modularity is required to clearly separate functionality from communication, which is necessary in a model at a high level of abstraction. It also enables the decomposition of a system into a hierarchical network of components. *Behavioral hierarchy* is used to decompose a system's behavior into sequential or concurrent subbehaviors, whereas *structural hierarchy* decomposes a system into a set of interconnected components.
   Modularity is also required to support design reuse and the incorporation of intellectual property. During refinement, modularity helps to keep changes in the system description local so that other parts of the design are not affected. For example, communication refinement should only replace abstract channels with more detailed ones without modifying the components using these channels. The locality of changes makes refinement tools simpler and the generated results more understandable.
3. Completeness is obviously a requirement. A system language must cover all concepts commonly found in embedded systems. In addition to (a) behavioral and (b) structural hierarchy this includes (c) concurrency, (d) synchronization, (e) exception handling and (f) timing, as discussed in detail in [11]. For explicit modelling of Mealy and Moore type finite state machines, (g) state transitions have to be supported. Furthermore, it is desirable that these concepts are organized orthogonally (independent from each other) so that the language can be minimal. In addition to these requirements, the language should be easy to understand and easy to learn.

4.2. TRADITIONAL LANGUAGES

Most traditional languages lack one or more of the requirements discussed in Section 4.1 and therefore cannot be used for system modelling without problems. Figure 13 lists examples of current languages [34, 17, 15, 29, 37, 1, 39] and shows which requirements they support and which are missing.

Because the traditional languages are not sufficient, a new language must be developed, either from scratch or as an extension of an existing language. The SpecC language [6] represents the latter approach as it is built on top of C.

	Verilog	VHDL	Statecharts	SpecCharts	C	Java	SpecC
Behavioral Hierarchy	○	○	●	●	○	○	●
Structural Hierarchy	●	●	○	○	○	○	●
Concurrency	●	●	●	●	○	◐	●
Synchronization	●	●	●	●	○	●	●
Exception Handling	●	○	◐	●	◐	●	●
Timing	●	●	◐	◐	○	○	●
State Transitions	○	○	●	●	○	○	●

○ not supported ◐ partly supported ● fully supported

Figure 13. Language Comparison

4.3. THE SPECC LANGUAGE

This section introduces the SpecC language and shows how SpecC covers all the requirements discussed before. SpecC is a superset of ANSI-C. C was selected because of its high acceptance in software development and its large library of already existing code.

A SpecC program can be executed after compilation with the SpecC compiler which first generates an intermediate C++ model of the program that is then compiled by a standard compiler for execution on the host machine.

Modularity, providing structural and behavioral hierarchy, and the special constructs making SpecC complete are described next.

4.4. STRUCTURAL HIERARCHY

Semantically, the functionality of a system is captured as a a hierarchical network of behaviors interconnected by hierarchical channels. Syntactically, a SpecC program consists of a set of *behavior*, *channel* and *interface* declarations.

A *behavior* is a class consisting of a set of ports, a set of component instantiations, a set of private variables and functions, and a public `main` function. Through its ports, a behavior can be connected to other behaviors or channels in order to communicate. A behavior is called a composite behavior if it contains instantiations of child behaviors. Otherwise it is

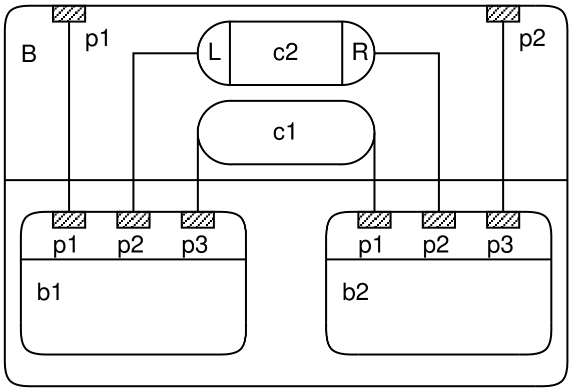


Figure 14. Basic Structure of a SpecC Model

called a leaf behavior. The functionality of a behavior is specified by its functions starting with the `main` function.

A *channel* is a class that encapsulates communication. It consists of a set of variables and functions, called methods, which define a communication protocol. A channel can be hierarchical, for example subchannels can be used to specify lower level communication.

An *interface* represents a flexible link between behaviors and channels. It consists of declarations of communication methods which will be defined in a channel.

For example, the following SpecC description specifies the system shown in Figure 14:

```
interface L { void Write(int x); };
interface R { int  Read (void);  };

channel C implements L, R
{
int Data; bool Valid;

void Write(int x)
   { Data = x; Valid = true; }
int Read(void)
   { while(! Valid) waitfor(10);
     return(Data); }
};

behavior B1(in int p1, L p2, in int p3)
{
void main(void)
   { /* ... */ p2.Write(p1); }
};
```

```
behavior B2(out int p1, R p2, out int p3)
{
void main(void)
   { /* ... */ p3 = p2.Read(); }
};

behavior B(in int p1, out int p2)
{
int c1;
C   c2;
B1  b1(p1, c2, c1);
B2  b2(c1, c2, p2);

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

The example system specifies a behavior `B` consisting of two subbehaviors `b1` and `b2` which execute in parallel and communicate via integer `c1` and channel `c2`. Thus structural hierarchy is specified by the tree of child behavior instantiations and the interconnection of their ports via variables and channels. Behaviors define functionality, and the time of communication, whereas channels define how the communication is performed.

4.5. BEHAVIORAL HIERARCHY

The composition of child behaviors in time is called behavioral hierarchy. Child behaviors can either be executed sequentially or concurrently. Sequential execution can be specified by standard imperative statements or as a finite state machine with explicit state transitions. Concurrent execution is either parallel or pipelined.

For example, we can specify a behavior being the sequential composition of the child behaviors using sequential statements, as shown in Figure 15(a), where `X` finishes when the last behavior `C` finishes. Second, we can use the parallel composition using the `par` construct, as shown in Figure 15(b), where `X` finishes when all its child behaviors `A, B` and `C` are finished. Also, pipelined composition is supported using the `pipe` construct, as shown in Figure 15(c), where `X` starts again when the slowest behavior finishes.

Syntactically, behavioral hierarchy is specified in the `main` function of a composite behavior. For example, with `a`, `b`, and `c` being instantiated child behaviors, the sequence of calls

```
a.main(); b.main(); c.main();
```

simply specifies sequential execution of `a`, `b`, `c`. The `par` and `pipe` statements specify concurrent execution. For example,

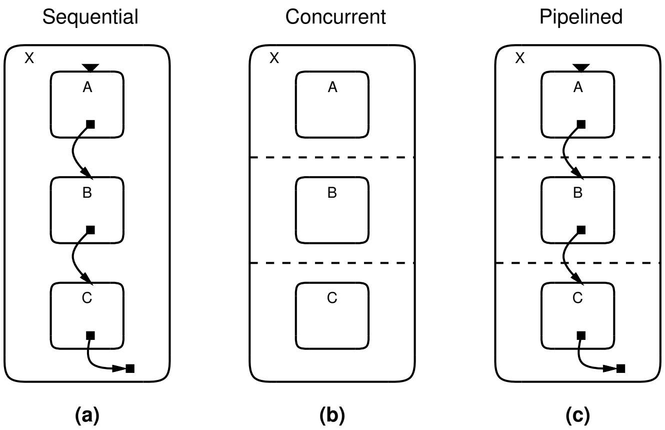


Figure 15. Behavioral Hierarchy

```
par { a.main(); b.main(); c.main(); }
```

executes `a`, `b`, `c` in parallel, whereas

```
pipe { a.main(); b.main(); c.main(); }
```

specifies execution in a pipelined fashion (`a` in the first iteration, `a` and `b` in the second, ...). The `par` statement completes when its last statement finishes, the `pipe` statement implicitly specifies an endless loop.

SpecC also supports explicit specification of state transitions. For example

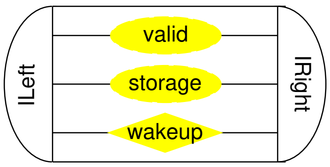
```
fsm { a: { if (x > 0)  break;
   if (x <= 0) goto b; }
      b: { if (y > 0)  goto a;
   if (y == 0) goto b; }
      c: { break; }
     }
```

specifies the state transitions of a finite state machine model with three behaviors `a`, `b`, `c`. Implicitly the first label in the `fsm` statement specifies the initial state (`a`). The FSM exits when a `break` statement is executed.

In summary, behavioral hierarchy is captured by the tree of function calls to the behavior `main` methods.

4.6. SYNCHRONIZATION

Concurrent behaviors usually must be synchronized in order to be cooperative. In SpecC, a built-in type *event* serves as the basic unit of synchronization. Events can only be used as arguments to `wait` and `notify` statements (or with exceptions as explained in Section 4.7). A `wait` statement suspends the current behavior from execution until one of the specified events is notified by another behavior. The `notify` statement triggers all specified events so that all behaviors waiting on one of these events can resume their execution.



(a)

```
interface ILeft {
    void write( int val );
    };
interface IRight {
    int    read( void );
    };
```

```
channel CShared( void )
    implements ILeft, IRight {
    int      storage;
    bool     valid;
    event  wakeup;
    void write( int val ) {
          storage = val;
          valid = true;
          notify( wakeup ); }
    int    read( void ) {
          while ( !valid )
             wait( wakeup );
          valid = false;
          return storage; } };
```

(b)

Figure 16. Example for simple Shared Memory Channel

For example, Figure 16 shows a simple shared memory channel `CShared` that, in addition to a `valid` bit, uses the event `wakeup` to allow only synchronized accesses to its `storage`. With this channel, it is assured that a consumer will always get valid data.

4.7. EXCEPTION HANDLING

SpecC provides support for two types of exceptions, namely *abortion* (or trap) and *interrupt*, as shown in Figure 17.

The `try-trap` construct, illustrated in Figure 17(a), aborts behavior `x` immediately when one of the events `e1`, `e2` occurs. The execution of behavior `x` (and all its child behaviors) is terminated without completing its computation and control is transferred to behavior `y` in case of `e1`, to behavior `z` in case of `e2`. This type of exception usually is used to model the reset of a system.

On the other hand, the `try-interrupt` construct, as shown in Figure 17(b), can be used to model interrupts. Here again, execution of behavior `x` is stopped immediately for events `e1` and `e2`, and behavior `y` or `z`, respectively, is started to service the interrupt. After completion of interrupt handlers `y` and `z` control is transferred back to behavior `x` and execution is resumed right at the point where it was stopped.

For both types of exceptions, in case two or more events happen at the same time, priority is given to the first listed event.

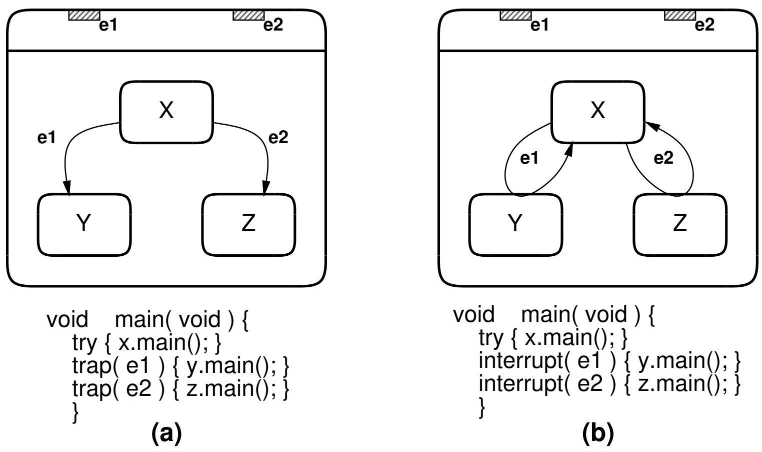


Figure 17. Exception handling: (a) abortion, (b) interrupt.

It should be noted that interrupt and abortion type exceptions can be mixed in SpecC. For example, the following code specifies a behavior `B` with a resetable child behavior `b1` and an interrupt handler `b2`.

```
behavior B (in event IRQ, in event RST)
{
B_sub b1, b2;

void main(void)
   { try { b1.main(); }
     interrupt IRQ { b2.main(); }
     trap      RST { b1.main(); }
    }
};
```

4.8. TIMING

In the design of embedded systems the notion of real time is an important issue. However, in traditional imperative languages such as C, only the ordering among statements is specified, the exact information on when these statements are executed, is irrelevant. While these languages are suitable for specifying functionality, they are insufficient in modeling embedded systems because of the lack of timing information. Hardware description languages such as VHDL overcome this problem by introducing the notion of time: statements are executed at discrete points in time and their execution delay is zero. While VHDL gives an exact definition of timing for each statement, such a treatment often leads to *over-specification*.

One obvious over-specification is the case when VHDL is used to specify functional behavior. The timing of functional behaviors is unknown until they are synthesized. The assumption of zero execution time is too optimistic and there are chances to miss design errors during specification validation. Other cases of over-specification are timing constraints and timing delays, where events have to happen, or, are guaranteed to happen in a *time range*, instead of at a fixed point in time, as restricted by VHDL.

SpecC overcomes this problem by differentiating between two types of timing information, *exact timing* and *timing ranges*. Exact timing is used when the timing is known, for example the execution delay of an already synthesized component. This is specified with a `waitfor` statement which suspends the execution of the current behavior for a specified time. The time is measured in real time units such as nanoseconds. Simulation time is only increased by `waitfor` statements, other statements are always executed in zero time.

Timing ranges are used to specify timing constraints at the specification level. SpecC supports timing information in terms of *timing diagrams* with minimum and maximum time constraints. Timing ranges are specified as 4-tuples $T = \langle l1, l2, min, max \rangle$ with the `range` statement. For example,

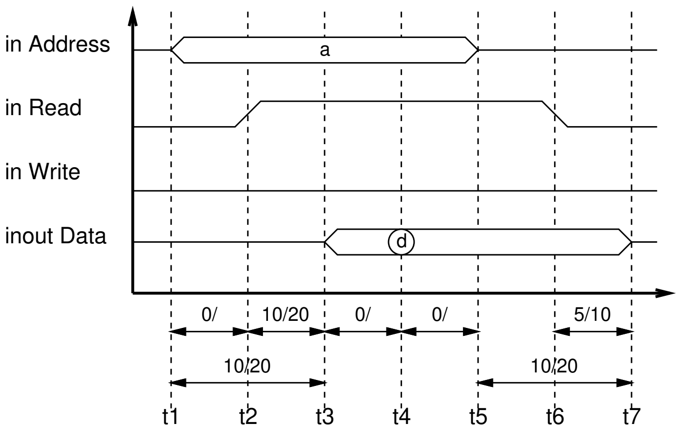
```
range(l1; l2; 10; 20);
```

specifies at least 10 but not more than 20 time units spent between labels `l1` and `l2`.

Consider, for example, the timing diagram of the read protocol for a static RAM, as shown in Figure 18(a). In order to read a word from the SRAM, the address of the data is supplied at the `address` port and the read operation is selected by assigning 1 to the `read` and 0 to the `write` port. The selected word then can be accessed at the `data` port. The diagram in Figure 18(a) explicitly specifies all timing constraints that have to be satisfied during this read access. These constraints are specified as arcs between pairs of events annotated with `x/y`, where `x` specifies the minimum and `y` the maximum time between the value changes of the signals.

Figure 18(b) shows the SpecC source code of a SRAM channel `C_SRAM`, which instantiates the behavior `B_SRAM`, and the signals, which are mapped to the ports of the SRAM. Access to the memory is provided by the `read_word` method, which encapsulates the read protocol explained above (due to space constraints write access is ignored).

Figure 18(c) shows the source code of the `read_word` method at the specification level. The `do-timing` construct used here effectively describes all information contained in the timing diagram. The first part of the construct lists all the events of the diagram, which are specified as a label and its associated piece of code, which describes the changes of signal values.



(a)

```
interface I_SRAM {
   void read_word(bit[15:0] a,
                  bit[15:0] *d);
   };

behavior B_SRAM(
      in bit[15:0] addr,
      inout bit[15:0] data,
      in bool rd,
      in bool wr ) {
   void main( void ) { ... }
   };

channel C_SRAM( void )
    implements I_SRAM {
     bit[15:0]   Address, Data;
     bool        Read, Write;
     B_SRAM sram(
                 Address, Data,
                 Read, Write );

     void  read_word(bit[15:0] a,
                 bit[15:0] *d ) { ... }
   };
```

(b)

```
void  read_word(
       bit[15:0] a, bit[15:0] *d ) {
   do {
      t1 : { Address = a; }
      t2 : { Read = 1; }
      t3 : {  }
      t4 : { *d = Data; }
      t5 : { Address = 0; }
      t6 : { Read = 0; }
      t7 : { break; }
      }
   timing {
      range( t1; t2; 0; );
      range( t1; t3; 10; 20 );
      range( t2; t3; 10; 20 );
      range( t3; t4; 0; );
      range( t4; t5; 0; );
      range( t5; t7; 10; 20 );
      range( t6; t7; 5; 10 );
      }
   }
```

(c)

```
void  read_word(
       bit[15:0] a, bit[15:0] *d ) {
   Address = a;
   Read = 1;
   waitfor(10);
   *d = Data;
   Address = 0;
   Read = 0;
   waitfor(10);
   }
```

(d)

Figure 18. Timing Example: SRAM Read Protocol: (a) timing diagram, (b) SRAM channel, (c) specification level timing, (d) implementation level timing.

The second part is a list of `range` statements, which specify the timing constraints between the events, as explained above.

This style of timing description is used at the specification level. In order to get an executable model of the protocol, *scheduling* has to be performed for each `do-timing` statement. Figure 18(d) shows the implementation of the `read_word` method after an ASAP scheduling is performed. All timing constraints are replaced by delays, which are specified using the `waitfor` construct.

4.9. ADDITIONAL FEATURES

In addition to the concepts explained in the last sections, the SpecC language supports further constructs that are necessary for system-level design. First, SpecC provides explicit support for *Boolean* (`bool`) and *bitvector* (`bit[:]`) types, in addition to all types provided by ANSI-C.

Also, constructs for binary import of pre-compiled SpecC code and support of persistent annotation for objects in the language are provided. Since these constructs are beyond the scope of this paper, please refer to [6] for further details.

In conclusion, the Sections 4.4 to 4.8 show that the SpecC language satisfies the requirements of executability, modularity and completeness, as discussed in Section 4.1.

It has to be emphasized, that the advantage of SpecC lies in its *orthogonal constructs* which implement *orthogonal concepts*. All SpecC constructs are independent of each other, unlike for example signals in VHDL, which are used for synchronization, communication and timing. The SpecC language covers the complete set of system concepts with a minimal set of constructs. Therefore it is easy to learn and easy to understand.

5. Reuse and IP

This section takes a closer look at how well the SpecC language and the SpecC methodology supports the reuse and integration of intellectual property.

Reuse essentially deals with the check-in ("Design for Reuse") and check-out ("Reuse of Designs") of components in the design library. Because all components in the design library are specified using the same SpecC language, reuse becomes easy. Also, the SpecC language encourages the specification of modular components which are decoupled from each other and therefore can be used independently.

In particular, a SpecC design library consists of behaviors, channels, and interfaces. A new design can be developed from scratch and/or composed from existing parts by selecting components from this library. As described earlier, behaviors represent functional units such as hardware components, and channels encapsulate communication such as bus protocols and bus media. Thus computation and communication are clearly separated. Interfaces connect behaviors and channels, as they declare *what* kind of communication is performed. Channels define *how* the communication is performed by implementing the interface.

A behavior's port of type interface can be mapped to any channel that implements that interface. Thus channels delivering the same type of communication can be exchanged without modification of the connected behaviors, for example a PCI bus can be easily replaced with a VME bus ("plug-and-play"). The same applies to behaviors. A behavior can be replaced with another behavior without affecting the channels as long as both implement the same functionality and have compatible ports.

For integration of intellectual property, three IP configurations are possible with SpecC. First, an IP vendor can offer design specifications which still need to be synthesized. This is called Soft-IP and is useful for standard busses and bus protocols for example. In this case the IP consists of a SpecC interface declaration and a channel definition.

On the other hand, Hard-IP integrates already synthesized components such as cores. Here, in addition to the actual core (layout), the IP vendor delivers a SpecC behavior declaration which only specifies the ports of the component, and an object or library file that can be linked to the executable SpecC code for simulation. Note, in this case the IP vendor keeps the implementation of the core secret.

As a third configuration, a combination of Soft-IP and Hard-IP is possible, where the IP consists of a wrapper (a SpecC channel definition with interface) in addition to the Hard-IP parts. This is exactly the situation as described in Figure 2(b), where the wrapper supplies higher-level functions dealing with the communication to the internal component.

6. The System

We have developed the SpecC Environment as shown in Figure 19. The design is specified with the help of the *SpecC Editor* which provides a graphical user interface (GUI). The SpecC Editor is also used for displaying the system models at different design stages and allows the designer to execute transformations on the models in an interactive or automatic manner. Different aspects of the design model are displayed in separate windows. For example, the structural hierarchy of the system under design is displayed in a hierarchy browser, whereas the mapping of ports and variables is shown in a connectivity window. All windows support interactive modification of the design.

In analogy to the methodology described in Section 3, the SpecC synthesis system consists of a set of tools, such as the *Estimator*, the *Allocator* and *Partitioner*, the *Scheduler*, and the *Communication Synthesizer*, which operate on the *SIR*, the SpecC Intermediate Representation. A SIR file can be obtained initially by compiling SpecC source code using the *SpecC Compiler*. It contains the symbol table and the abstract syntax tree of the corresponding SpecC code. It also contains explicit information such as the type of each expression which is implicit in the source.

The SpecC compiler can also automatically generate simulation code

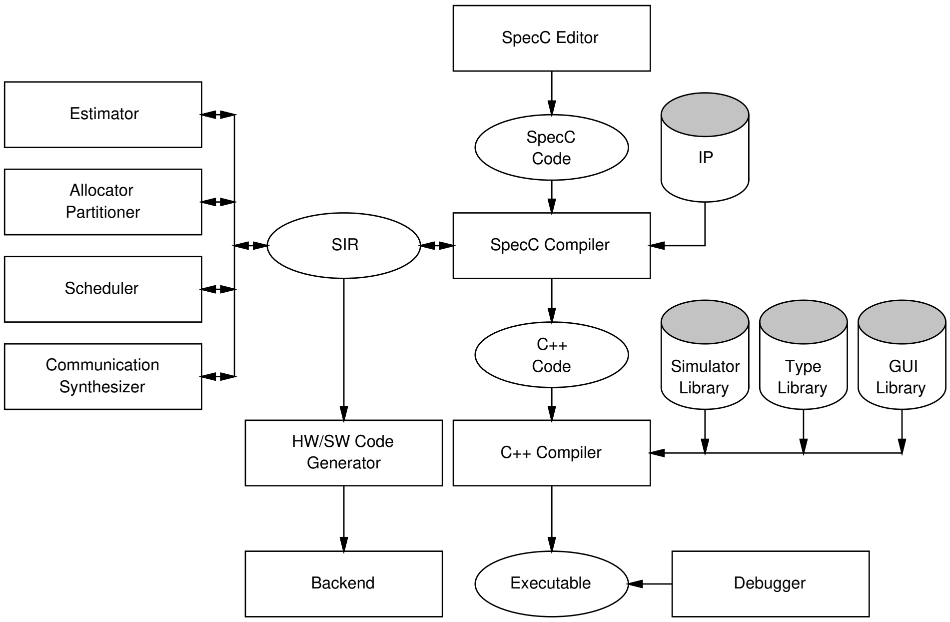


Figure 19. The SpecC Environment (SCE)

in the form of C++, which can then be compiled and linked with a set of predefined libraries in order to generate an executable.

The *Simulation Library* implements a discrete event simulator by maintaining a time wheel which schedules concurrent threads. The *Type Library* provides an implementation of data types such as bitvector and multi-valued logic. The *GUI Library* helps to visualize signal waveforms and supports graphical entry of stimuli.

A standard source-level debugger can be used to debug the executable. The *HW/SW Code Generator* exports the implementation level SIR into C or HDL code.

7. Conclusion

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

In particular, the design methodology starts with an executable specification of the system under design and eventually creates an implementation architecture 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 automatic tools. In other words, for architecture exploration the designer is in the loop.

In order to incorporate IP components and allow "plug-and-play", protocol encapsulation and separation of communication and computation is necessary. A wrapper concept is used to hide details of communication protocols and replace these details with an abstract high-level interface.

For this system design methodology, the language being used is important. Since none of the traditional languages meets all the requirements for system level design, the SpecC language was presented. SpecC precisely satisfies all requirements for codesign languages and explicitly supports structural and behavioral hierarchy, concurrency, state transitions, exception handling, timing and synchronization in an orthogonal way. SpecC encourages reuse and supports integration of IP. Since SpecC is a superset of C, a large library of already existing algorithms can directly be used. SpecC is easy to learn and easy to understand.

Finally, the SpecC Environment was presented. The system is based on the described methodology and the SpecC language.

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.

References

1. K. Arnold, J. Gosling; *The Java Programming Language*; Addison-Wesley, 1996.
2. F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, K. Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS approach*. Kluwer Academic Publishers, April 1997.
3. F. Chan, M. Spiller, R. Newton. "WELD – An Environment for Web-Based Electronic Design". In *Proceedings of the Design Automation Conference*, San Francisco, 1998.
4. M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli. "A Formal Specification Model for Hardware/Software Codesign". In *Proceedings of International Workshop on Hardware-Software Codesign*, Oct. 1993.
5. P. Chou, R. Ortega, G. Borriello. "The Chinook Hardware/Software Co-Synthesis System". In *International Symposium on System Synthesis*, Cannes, France, Sept. 1995.
6. R. Dömer, J. Zhu, D. Gajski. *The SpecC Language Reference Manual*. University of California, Irvine, Technical Report ICS-TR-98-13, March 1998.
7. D. Drusinsky and D. Harel. "Using Statecharts for hardware description and synthesis". In *IEEE Transactions on Computer Aided Design*, 1989.

8. R. Ernst, J. Henkel, T. Benner. "Hardware-software cosynthesis for microcontrollers". In *IEEE Design and Test*, Vol. 12, 1993.
9. R. Ernst, et. al. "The COSYMA Environment for Hardware-Software Cosynthesis of Small Embedded Systems". In *Microprocessors and Microsystems*, Vol. 20, No. 3, May 1996.
10. D. Gajski, F. Vahid, and S. Narayan. "SpecCharts: a VHDL front-end for embedded systems". University of California, Irvine, Technical Report ICS-TR-93-31, 1993.
11. D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, New Jersey, 1994.
12. D. Gajski, J. Zhu, R. Dömer. "Essential Issues in Codesign". In *Hardware/Software Co-Design: Principles and Practice*, edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.
13. R. Gupta, C. Coelho., G. De Micheli. "Synthesis and simulation of digital systems containing interacting hardware and software components". In *Proceedings of the 29th ACM, IEEE Design Automation Conference*, 1992.
14. R. Gupta, S. Liao. "Using a Programming Language for Digital System Design". In *IEEE Design & Test of Computers*, IEEE, 1997.
15. D. Harel; "StateCharts: a Visual Formalism for Complex Systems"; *Science of Programming*, 8, 1987.
16. J. Henkel, R. Ernst. "A Hardware-Software Partitioner Using a Dynamically Determined Granularity". In *Proceedings of the Design Automation Conference*, 1997.
17. IEEE Inc., N.Y. *IEEE Standard VHDL Language Reference Manual*, 1998.
18. T. Ismail, M. Abid, A. Jerraya. "COSMOS: A Codesign Approach for Communicating Systems". In *Proceedings of the International Workshop on Hardware- Software Codesign.* IEEE CS Press, 1994.
19. A. Kalavade, E. Lee. "A Hardware/Software Codesign Methodology for DSP Applications". In *IEEE Design and Test*, Sept. 1993.
20. G. Koch, U. Kebschull, W. Rosenstiel. "A prototyping architecture for hardware/software codesign in the COBRA project". In *Proceedings of the third International Workshop on Hardware/Software Codesign*, IEEE Computer Society Press, 1994.
21. D. Ku, G. De Micheli. "HardwareC – A Language for Hardware Design, Version 2.0". Tech. Rep. CSL-TR-90-419, Stanford University, April 1990.
22. E. Lee and D. Messerschmidt. "Static scheduling of synchronous data flow graphs for digital signal processors". In *IEEE Transactions on Computer-Aided Design*, 1987.
23. S. Liao, S. Tjiang, R. Gupta. "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment". In *Proceedings of the 34th Design Automation Conference*, Anaheim, California, USA, 1997.
24. C. Liem, F. Nacabal, C. Valderrama, P. Paulin, A. Jerraya. "System-on-a-chip cosimulation and compilation". In *IEEE Design & Test of Computers*, 1997.
25. C. Liem, P. Paulin. "Compilation Techniques and Tools for Embedded Processor Architectures". In *Hardware/Software Co-Design: Principles and Practice*, edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.
26. J. Madsen, J. Grode, P. Knudsen. "Hardware/Software Partitioning using the LYCOS System". In *Hardware/Software Co-Design: Principles and Practice*, edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.
27. P. Marwedel, G. Goossens. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.
28. G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw Hill, 1994.
29. S. Narayan, F. Vahid, D. Gajski. "System Specification and Synthesis with the SpecCharts Language". In *Proceedings of the International Conference on Computer Aided Design*, 1991.
30. A. Österling, T. Benner, R. Ernst, D. Herrmann, T. Scholz, W. Ye. "The Cosyma

System". In *Hardware/Software Co-Design: Principles and Practice*, edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.
31. K. Rompaey, D. Verkest, I. Bolsens, H. De Man. "CoWare – A design environment for heterogeneous hardware/software systems". In *Proceedings of the European Design Automation Conference*, 1996.
32. W. Rosenstiel. "Prototyping and Emulation". In *Hardware/Software Co-Design: Principles and Practice*, edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.
33. B. Stroustrup. *The C++ Programming Language*, third edition. Addison-Wesley, 1997.
34. D. Thomas, P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.
35. C. Valderrama, M. Romdhani, J. Daveau, G. Marchioro, A. Changuel, A. Jerraya. "Cosmos: A Transformational Co-design tool for Multiprocessor Architectures". In *Hardware/Software Co-Design: Principles and Practice*, edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.
36. W. Wolf. "Hardware/Software Co-Synthesis Algorithms". In *Hardware/Software Co-Design: Principles and Practice*, edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.
37. X3 Secretariat. *The C Language*. X3J11/90-013, ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association, Washington, DC, USA, 1990.
38. T. Yen, W. Wolf. *Hardware-software Co-synthesis of Distributed Embedded Systems*. Kluwer Academic Publishers, 1997.
39. J. Zhu, R. Dömer, D. Gajski. "Syntax and Semantics of the SpecC Language". In *Proceedings of the Synthesis and System Integration of Mixed Technologies*, Osaka, Japan, Dec. 1997.