System-level Modeling and Design with the SpecC Language

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Universität Dortmund

am Fachbereich Informatik

von

Rainer Dömer

Dortmund

2000

Tag der mündlichen Prüfung:

Dekan/Dekanin:

Gutachter:

System-level Modeling and Design with the SpecC Language

Dissertation

for the degree of

"Doktor der Naturwissenschaften"

submitted to the

Department of Computer Science
at University of Dortmund

by

Rainer Dömer

Dortmund, Germany

2000

For Julia

Abstract

The semiconductor roadmap estimates the design complexity for digital systems to continue to increase according to Moore's law. In the next years, embedded systems with 10ths of millions of transistors on one chip will be standard technology. System-on-Chip (SOC) designs will integrate processor cores, memories and special-purpose custom logic into a complete system fitting on a single die. However, the increased complexity of SOC designs requires more effort, more efficient tools and new methodologies. Increasing the design time is not an option due to market pressures.

System-level design reduces the complexity of the design models by raising the level of abstraction. Starting from an abstract specification model, the system is step-wise refined with the help of computer-aided design (CAD) tools. Using codesign techniques, the system is partitioned into hardware and software parts and finally implemented on a target architecture. Established design methodologies for behavioral synthesis and standard software design are utilized. However, moving to higher abstraction levels is not sufficient.

The key to cope with the complexity involved with SOC designs is the reuse of Intellectual Property (IP). The integration of complex components, which are predesigned and well-tested, drastically reduces the design complexity and, thus, saves design time and allows a shorter time-to-market. Since the idea of IP reuse promises great benefits, it must become an integral part in the system design methodology. Furthermore, the use of IP components must be directly supported by the design models, the tools and the languages being used throughout the design process. For example, it must be easy to insert and replace IP components in the design model ("plug-and-play").

This work addresses the main issues in SOC design, namely the system design methodology, system-level modeling, and the specification language.

First, an IP-centric system design methodology is proposed which is based on the reuse of IP. It allows the reuse and integration of IP components at any level and at any time during the design process. Starting with an abstract executable specification of the system, architecture exploration and communication synthesis

are performed in order to map the design model onto the target architecture. At any stage, the systems functionality and its characteristics can be evaluated and validated.

The model being used in the methodology to represent the system must meet system design requirements. It must be suitable to represent abstract properties at early stages as well as specific details about design decisions later in the design process. In order to support IP, the model must clearly separate communication from computation. In this work, a hierarchical model is described which encapsulates computation and communication in separate entities, namely behaviors and channels. This model naturally supports reuse, integration and protection of IP.

In order to formally describe a design model, a language should be used which directly represents the properties and characteristics of the model. This work presents a newly developed language, called SpecC, which allows to map modeling concepts onto language constructs in a one to one fashion. Unlike other system-level languages, the SpecC language precisely covers the unique requirements for embedded systems design in an orthogonal manner. Built on top of the C language, the de-facto standard for software development, SpecC supports additional concepts needed in hardware design and allows IP-centric modeling. Recently, the SpecC language has been proposed as a standard system-level language for adoption in industry by some of Japan's top-tier electronics and semiconductor companies.

The proposed methodology and the SpecC language have been implemented in the SpecC design environment. In a graphical framework, the SpecC design environment integrates a set of CAD tools which support system-level modeling, design validation, design space exploration, and (semi-) automatic refinement. The framework and all tools rely on a powerful, central design representation, the SpecC Internal Representation (SIR).

Using the SpecC design environment, the IP-centric methodology has been successfully applied to several designs of industrial size, including a GSM vocoder used in mobile telecommunication.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The semiconductor roadmap [SIA97], published by the Semiconductor Industry Association (SIA), estimates the design complexity for digital systems to continue to increase according to Moore's law [Ham99]. Applied to the design of embedded systems, Moore's law estimates the number of transistors on a chip to double every 18 months. The exponential growth of chip capacity is based on the continuing decrease in geometry size and increase in chip density.

In the next years, deep sub-micron design, dealing with process technologies of 0.18μm and below, will allow to integrate 10ths of millions of logic transistors on one chip. This makes it possible to implement complex embedded systems entirely on a single chip. *System-on-Chip* (SOC) designs will integrate system components including processor cores, memories and special-purpose custom logic blocks into a complete system fitting on a single die.

SOC design is desirable especially for multi-media applications and portable devices where embedded systems save space, power and cost. In contrast to traditional ASIC design, which implements one sub-system in application-specific hardware, SOC design consists of the integration and implementation of special-purpose, complex components which are interacting with each other. Typically, a SOC includes one or more microprocessors, several peripheral units, memory blocks, and application-specific logic portions interconnected by on-chip busses.

While the availability of a huge chip capacity enables SOC designs, it, at the same time, significantly raises the complexity of these systems. The increased complexity requires substantially more effort, more efficient tools and new methodologies for building such embedded systems. In fact, the complexity of SOC design is beyond the size that currently established electronic design automation (EDA) tools and methodologies can handle.

The SIA roadmap shows that a *productivity gap* exists between the available chip capacity and the current design capabilities. While the chip capacity grows by 58% per year (according to Moore's law), the support provided by computer-aided design (CAD) tools is estimated to increase by only 21% each year [SIA97]. If this growing gap cannot be overcome, it will result in under-utilization of the available chip capacity and thus unnecessarily increase the cost of embedded systems.

In the past, automated hardware *synthesis* was used to bridge the productivity gap. Logic synthesis and recently behavioral synthesis, also known as high-level synthesis (HLS) [GDW+91], supported designers in order to increase their productivity. Unfortunately, the help of hardware synthesis is not sufficient for SOC design, since embedded systems require more and more software content.

It should be clear that an increase in the design time for embedded systems is not an option in order to solve the productivity problem. The *time-to-market* is critical for the success or failure of a product in the market. Thus, it is necessary to develop and manufacture the next-generation product (and its embedded system) as quickly as possible in order to promote "product-on-demand". Ignoring the market pressures, which require to offer better products with more features for less money in shorter periods of time, is not acceptable.

The threatening under-utilization of available chip capacity due to the productivity gap and the strong market pressures force the electronic industry to search for new design methodologies. More efficient EDA support is required in order to build successful SOC designs. This is the motivation for system-level design which is defined in the following section.

1.1 System-level Design

System-level design (SLD) addresses the problem of the increased complexity of embedded systems by raising the level of abstraction. In contrast to behavioral synthesis, which deals with the implementation of algorithms in application-specific hardware (ASIC design), system-level design focuses on the problem of mapping an abstract specification model of an entire system onto a target architecture (SOC design). As mentioned earlier, a typical target architecture consists of a set of processor cores, memories, peripheral units, and custom hardware blocks. These system components are interconnected by on-chip busses whose implementation is part of system-level design as well.

The cost-effective implementation of complex embedded systems requires a high software (SW) content. Compared to the high cost of developing dedicated hardware (HW), a software implementation is inexpensive. In addition, software can easily be modified if requirements change or new features need to be added. However, a software implementation may not be possible due to performance constraints. It is one task of system-level design to trade-off an inexpensive and flexible software solution versus a high-speed hardware implementation. Therefore, system-level design is also referred to as HW/SW codesign.

Codesign is defined as the design of systems involving both hardware and software. The main task of codesign is the *partitioning* of a single system specification into hardware and software parts. Then, depending on whether a specific component is to be implemented in software or hardware, standard software technologies and established hardware design methods, respectively, are used for the final implementation of the component.

In general, any system consists of parts from different domains. Therefore, system design often is defined as to also include the mechanical domain in addition to the domain of electronics (see for example [CHM+99] and [Sch99]). The inclusion of mechanical aspects extends the coverage of the system model compared to the real system. It also allows trade-offs to be made between mechanical versus electronic implementation of certain features.

On the other hand, these orthogonal domains are quite independent in most cases and thus can be treated separately. This separation significantly simplifies the design tasks as well. Hence, in this work, system design is considered exclusively within the domain of electronics.

Furthermore, some system-level design environments explicitly support the specification and use of analog and mixed signals. While this is useful for sub-systems, for example in the telecommunication area, the majority of embedded systems is specified completely digital. Also, the decision whether a signal is implemented as either analog or digital, can be viewed as an implementation issue that is resolved later in the design flow by back end tools. Within this work, system-level design targets on the design of digital systems [Gaj97], including hardware and software parts.

The system design flow usually starts from a formal, abstract specification of the intended design. After the specification has been validated for functional correctness, it is refined by a sequence of refinement tasks which eventually map the initial specification onto a selected target architecture. Section 1.1.4 discusses in detail the steps in a typical system design process including architecture selection, partitioning, scheduling and communication synthesis.

A very important issue in system-level design is the reuse of predesigned, complex components, often referred to as *Intellectual Property* (IP). In fact, the reuse of IP is the main key to cope with the complexity involved with SOC design. In contrast to redesigning a system completely from scratch, the use and integration of complex components, which are predesigned (possibly by somebody else) and well-tested, drastically reduces the design complexity. Thus, reuse of IP saves a great amount of design and testing time and, hence, allows a shorter time-to-market.

While the idea of IP reuse promises great benefits for system design, there are also problems to be solved. In order to allow easy and seamless integration in a new system, IP components need to be portable to different technologies and must provide standard or flexible interfaces. Good documentation about the IPs functionality, its requirements with respect to the environment, and its performance and other metrics are required as well.

The reuse of IP must become an integral part in the system design methodology. The selection, easy insertion and replacement of IP components ("plug-and-play") in the system must be directly supported by the design models, the tools and the languages being used throughout the design process. These and other issues involved with the reuse of IP are addressed in more detail in Section 1.1.5.

1.1.1 Levels of abstraction

In computer science, a well-known solution for dealing with complexity is to exploit hierarchy and to move to higher levels of abstraction. This effectively reduces the complexity in terms of the number of objects to be handled at one time.

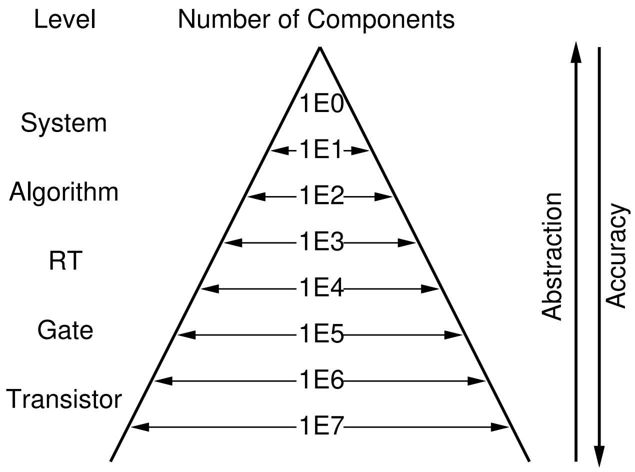

Figure 1.1: Abstraction versus complexity

Figure 1.1 illustrates this for digital systems. An embedded system, which at the lowest level consists of 10ths of millions of transistors, typically reduces to only thousands of components at the register-transfer level (RTL). Furthermore, RTL components are grouped together at the algorithm level. Finally, at the highest, the so-called system level, the one system is composed of only few components which include microprocessors, special-purpose hardware, memories and busses. From

Figure 1.1, it is obvious that a complex embedded system is easier to deal with at the abstract system level than at the detailed gate or transistor level.

The level of abstraction is a trade-off with the level of *accuracy*. A high abstraction level implies low accuracy, and vice versa. The design process of a new system usually starts from a highly abstract specification model and ends with a highly accurate implementation model which reflects the real system with all its details.

The advantage of such a top-down approach is that all necessary design decisions can be made at an abstraction level where all irrelevant details are left out in the model. This allows the design tasks to work with a system model with minimum complexity.

The concepts of abstraction and hierarchy are closely related. In digital systems, *hierarchy* is inherent in the structure of a system. Every system is composed of a set of components, and each component is a (sub-) system that, again, is composed of (sub-) components. In other words, the terms *system* and *component* are recursively defined.

In order to break the recursion in this definition and to clearly identify the system and its components, it is necessary to name the current *abstraction level*. The abstraction level defines the type of the components used and, thus, also determines the system. For example, at the gate level, the components are logic gates and the system is the composition of such gates. One level below, at the transistor level, a single gate can represent an entire system that is composed of a set of transistors.

It should be pointed out that the term *system*, in general, refers to different things in different contexts. For example, a modern aircraft can be viewed as one single system or as a collection of thousands of systems. Within this work, unless stated otherwise, the term system refers to a digital, embedded system which can be implemented by use of application-specific hardware and software running on one or multiple processors.

Please note that this definition of a system is consistent with the term system-on-chip. It is also well-defined with respect to the abstraction level for SOC design, the system level. A precise definition of system-level design will be given in the following section by use of the Y-Chart.

1.1.2 The Y-Chart

The Y-Chart [GK83], shown in Figure 1.2, is a conceptual framework which coordinates abstraction levels in different domains. This can be used to compare and classify different design tools and design methodologies.

The Y-Chart distinguishes three *domains* represented by three axes. A typical design process starts from the behavioral domain which specifies the pure behavior of the system without any implementation details, for example in form of program func-

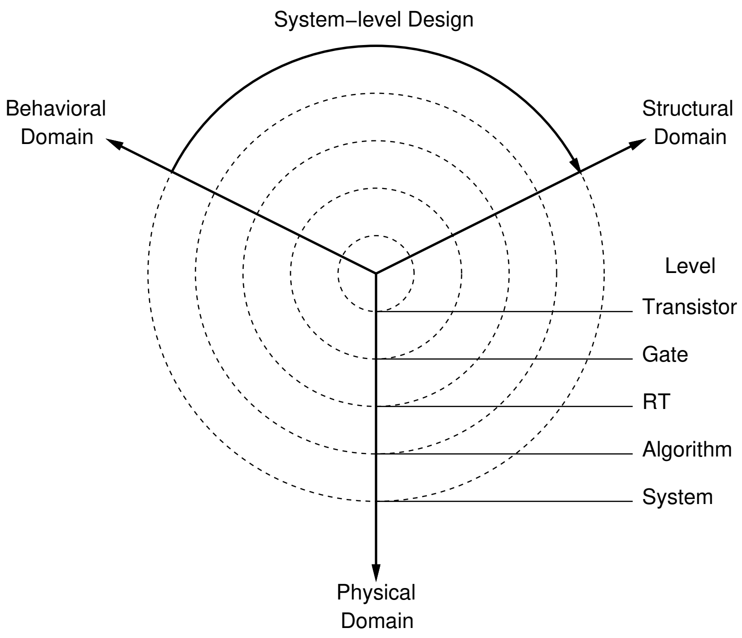

Figure 1.2: System-level design in the Y-Chart

tions or mathematical equations. The design is then mapped onto an architecture in the structural domain. The structural architecture is composed of components, for example logic gates or RT components, depending on the level of abstraction. Finally, an implementation of the design is manufactured in the physical domain.

The level of abstraction, as introduced in Section 1.1.1, is orthogonal to the domains. Starting from the center of the chart, the abstraction level, indicated by the dashed, concentric circles, increases from the transistor level up to the system level.

The Y-Chart allows to illustrate design flows and design tasks as paths on the chart. For example, a complete system design flow starts on the behavioral axis at the system level. After step-wise refinement towards the center of the chart and mapping onto a structural and physical implementation, it finally ends on the physical axis at the transistor level.

On the Y-Chart, *synthesis* is represented by an arc from the behavioral to the structural axis. The definition of system-level design is indicated by the arrow in Figure 1.2. The task of system-level design is to synthesize a structural system architecture from a behavioral system specification.

As another example, high-level synthesis (HLS) is represented by an arc from the behavioral to the structural axis on the RT level.

Furthermore, the tasks of refinement and optimization can be demonstrated on the Y-Chart as well. *Refinement* is represented by an arrow on the behavioral axis from a high to a lower abstraction level. On the other hand, *optimization* can be represented as an arrow at any point in the chart which points back to its starting point. Thus, such optimization is a task that is performed in-place and can occur at any level in any domain.

Recently, the Rugby model [JKH99] was proposed as a new conceptual framework targeted to represent codesign tasks. In contrast to the Y-Chart, the Rugby model explicitly separates software and hardware design. Furthermore, the Rugby model distinguishes five orthogonal dimensions, namely time, data, computation, communication and transformation. As such, the Rugby model is much more complex and not as abstract as the Y-Chart[1].

1.1.3 Models of computation

In order to design an embedded system, a formal model of the system is needed. This section lists the models of computation which are commonly used in system-level design. For an in-depth discussion of these models, please refer to other sources in the literature. Good overviews, including detailed comparisons of the models, can be found in [GVN$^+$94, GZD97c] or [LS96, LSS99], for example.

Models of computation can be classified into language oriented and architecture oriented models. Among the language oriented models, the *control flow graph* (CFG) represents the control flow of a program (for example, `if-then-else` and `loop` statements) in form of a directed graph. A *data flow graph* (DFG) is a (typically acyclic) graph used, for example, to represent expression trees. CFG and DFG can be easily combined into a *control data flow graph* (CDFG), which is a CFG whose nodes contain DFGs. A CDFG is commonly used as an intermediate model for systems specified with imperative programming languages.

Architecture oriented models represent an abstraction of the target architecture for a system. The basis for these models is the *finite state machine* (FSM) model which is a popular model to describe control. A FSM consists of states and transitions between the states. The output of a FSM is either state-based (Moore-type FSM), or input-based[2] (Mealy-type FSM). A FSM model can be easily implemented in hardware as a controller consisting of a state register and a block of combinatorial logic.

The FSM model has several extensions. Combined with the DFG model representing computation, the *finite state machine with datapath* (FSMD) is a typical

[1] The "beauty" of the Y-Chart lies in its simplicity.

[2] The output of a state-based FSM depends solely on the current state, whereas the output of a input-based FSM depends on the current state *and* the current input.

target model for behavioral synthesis. The implementation of a FSMD consists of a controller and a datapath. Very similar to the FSMD model is the *finite state machine with coprocessors* (FSMC) as defined in [JDK+97].

In order to represent complete systems consisting of several concurrent processing elements, more complex models are required. For example, the *codesign finite state machine* (CFSM) model, described in [CGH+93], can be used to represent a set of concurrent executing and communicating FSMs. Alternatively, hierarchy and concurrency can be explicitly added to the FSMD model. This results in the *hierarchical concurrent finite state machine with datapath* (HCFSMD) which allows to have sequential or concurrent sub-states in each state of the FSM.

Finally, programming language constructs can also be added. The *program state machine* (PSM) model, defined in [GVN+94], is a HCFSMD whose leaf states contain program statements. The PSM is a powerful computational model that is used, for example, as the underlying model of the SpecCharts language [GVN93].

Many other models exist with focus on different features. The model of *communicating sequential processes* (CSP), described in [Hoa85], emphasizes communication. The *synchronous data flow* (SDF) model is used in [LM87] to represent data flow intensive applications and digital signal processing. *Petri nets*, first described in [Pet62], are used in several variants and provide a well-defined, formal background for the static analysis of systems.

The model of computation used for embedded systems design should meet certain requirements and objectives. First, it should be *intuitive* to understand so that it is easy to specify the intended system with the model. Second, it must be *executable* in order to allow early system simulation. Furthermore, the model should be *verifiable*, in other words, it should provide support for formal verification. Finally, it must be *synthesizable* so that an implementation of the model can be obtained.

The models listed in this section achieve these goals more or less. It is not possible to decide which model of computation is best suited for the design of embedded systems. For the SpecC system, which is described later in this work, the PSM computational model was chosen. Since the PSM model is close to the target architecture, it simplifies the development of CAD tools. The model also is easy to understand and sufficient powerful for the large complexity of SOC design. The PSM model is directly supported by the SpecC language, the SpecC CAD tools, and the SpecC methodology.

1.1.4 System design process

The system design process starts with a specification of the intended design at a high level of abstraction and ends with an implementation model that accurately describes the implemented system and its components. In order to obtain the implementation from the specification, a set of refinement tasks is applied to the system model. This section defines the necessary tasks in a typical system design process.

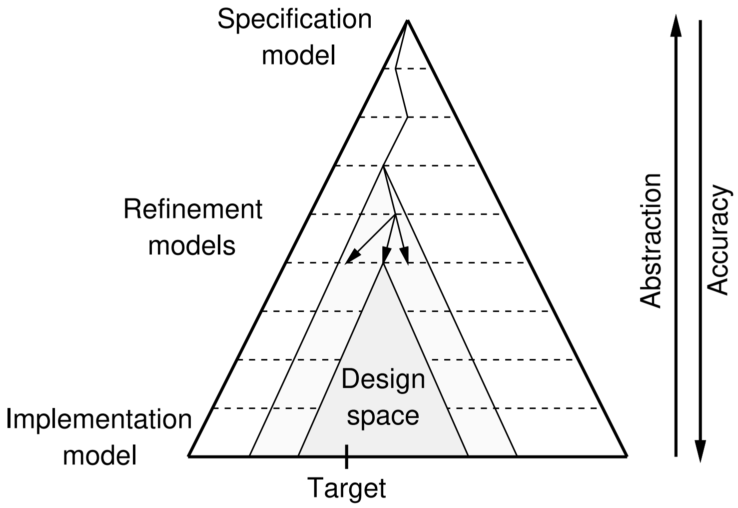

Figure 1.3: Design process using step-wise refinement

Figure 1.3 illustrates a top-down design process using step-wise *refinement*. Starting at the top of the pyramid, the specification model is transformed by a sequence of design tasks into refined models. At each stage, the available *design space*, as indicated by the shaded triangles in Figure 1.3, has to be explored. The goal of this design space exploration is to make a good design decision that will lead to an implementation model close to the target.

Each design decision affects the subsequent one in the way that the available design space shrinks. Obviously, it is important to choose the right model from the set of possible alternatives so that the target stays well inside the design space. Otherwise, if the decision is made in the wrong direction, the implementation will miss the target.

In general, each design task can be performed manually by designers or automatically by CAD tools. Also, both ways can be combined using semi-automatic refinement. Typically, it is up to the designer to make the design decision. Then automated tools are used to actually perform the tedious refinement with the design such that the decision made is reflected in the refined model.

It should be noted that the terms *specification* and *implementation* are relative to a particular design task or abstraction level. The implementation model generated by one task usually serves as the specification model for the next task.

1.1.4.1 Specification

The specification of the intended system is the starting point for the design process. The specification must meet several requirements. First, it should be *complete*. In other words, it should cover the entire design with all its features, its functionality and its requirements. On the other hand, the specification should also be *abstract*. It should not include any premature implementation details.

Furthermore, it is required that the specification is captured unambiguously in a *formal* language so that it can be processed by automated tools. More specifically, the specification must be *executable* so that simulation can be used to validate the functionality of the system from the beginning.

The specification is the first formal and functional description of the system. It serves as an initial model against which all subsequent, refined models will be compared.

1.1.4.2 Validation

In order to ensure the correctness of a system model, it has to be validated. *Validation* can be performed either statically by model analysis or dynamically by simulation.

As mentioned earlier, *simulation* requires the system model to be executable. Simulation validates the functionality of the system model in terms of the outputs generated for given input vectors. At different levels of accuracy, it can also be used to check the correctness of communication, synchronization, and timing.

Simulation usually is performed by a software simulator running on a host work station. However, system simulation in software is typically several orders of magnitude slower than the real system, in particular at low levels of abstraction. Hence, the system can only be validated for a short period of simulation time and a small set of test vectors. If this is not sufficient and more effort and higher cost are acceptable, *rapid prototyping* can be used to increase simulation speed by use of reprogrammable hardware, for example, field programmable gate arrays (FPGA) [Ros97].

It should be emphasized that simulation only validates a system model for the given test vectors and therefore, unless exhaustive simulation is performed, does not cover all possible cases. In contrast to validation, *verification* yields a 100% test coverage. Formal verification is a static analysis technique which can be used to prove certain properties of the system model. Formal verification requires a well-defined, formal model and, because of its complexity, can usually only be applied to very small systems.

In order to evaluate characteristics of a system which are not directly observable from the model, estimation techniques can be used. The task of *estimation* is to quickly determine critical quality metrics of the system such as performance, power consumption, size, cost, and others. Estimation can be performed either statically by analysis of the system model, or dynamically during simulation, for example, by use of profiling.

For estimation, there is a trade-off of accuracy versus time. The emphasis of estimation is on fast, rather than exact, system evaluation. Thus, the use of estimation enables the designer to make a reasonable design decision in short time. This is in contrast to a conservative approach which actually synthesizes all alternatives in order to make an optimal decision, as proposed in [Nie98], for example.

When finally a system has been manufactured, it must be tested for full functionality and no manufacturing defects. The high complexity of SOC designs requires that the chip is prepared for its *testing* already during the design process. Typically, built-in self-test (BIST) and other techniques are used to allow testing of chips with IP cores [ZMD99].

1.1.4.3 Refinement

After the system specification is captured and validated, it is the task of *architecture exploration* to allocate the system architecture, to partition the specification into hardware and software parts, and to map all parts of the design to the components in the architecture. During architecture exploration, estimation is used to determine the quality characteristics of the architecture under consideration. If the metric goals are not satisfied, the system is repartitioned or a different architecture or different components are selected. In the worst case, if no acceptable solution is found, the specification must be changed in terms of goals, constraints, or features.

It is the task of architecture *allocation* to determine the number and types of the processing elements (PE) and the connectivity for the system architecture. The components in the target architecture typically include processors, application-specific hardware, memories, peripheral units and IP cores. These components are interconnected by system or local busses. All components and busses are selected from the component library.

Most parts in a system specification can be implemented in either software or hardware. It is the task of HW/SW *partitioning* to trade off an inexpensive software solution versus a high-speed hardware implementation. Typically, only performance-critical parts of the system are implemented in hardware and all other parts are compiled into software to be executed on the allocated processors.

In general, *scheduling* has to be performed for the software parts of the system, since sequential processors can only execute one thread at a time. Scheduling determines the order of execution for the tasks assigned to a processor. Scheduling can be static or dynamic. A static schedule can be computed at design time if all constraints, including task execution times, delays, and dependencies, are known beforehand and do not change at run-time. Otherwise, dynamic scheduling must be used. In that case, the execution order for all tasks is determined dynamically at run-time, for example, by use of a real-time operating system (RTOS).

At the end of architecture exploration, each object in the specification is mapped to a particular hardware or software component. The quality of this *mapping* depends very much on the granularity of the objects. A coarse grained granularity, which, for example, considers entire processes as smallest, indivisible units, simplifies the refinement tasks since less objects need to be handled, but also limits the implementation options. On the other hand, a fine granularity enables more options allowing a possibly better implementation, but also increases the complexity and, thus, the refinement time.

After architecture exploration, *communication synthesis* must be performed. This includes the selection of communication protocols for the selected busses, hardware interface synthesis, and software driver generation. More specifically, accesses to data, which is assigned to a different PE, must be converted to remote procedure calls (RPC). Then, the RPCs can be implemented by use of the native bus protocol provided by the bus connecting the PEs. For hardware, interfaces need to be synthesized, and for software, device drivers must be generated. In case busses with different protocols need to be connected, protocol transducers must be inserted. In summary, the task of communication synthesis is to refine the abstract communication between the components in the architectural model into an implementation using the actual bus protocols.

The system-level design process is completed with the *back end*. The task of the back end is to make the refined system model available to established design methodologies for behavioral synthesis and standard software design. In order to allow a seamless integration, it is important that the output generated by the back end can be used without modification as input to the subsequent tools.

For the software parts of the system, program code, for example C or assembly code, is generated so that standard compiler, assembler and linker tools can be used for the software implementation. If available, a retargetable compiler can generate code for all the allocated processors. Otherwise, a processor-specific tool set is needed for each type of processor in the system.

For the hardware parts, a synthesizable hardware description is generated, typically in VHDL or Verilog. This description can then be fed into high-level synthesis tools in order to implement the custom hardware.

1.1.4.4 Methodology

In the previous sections, the typical tasks used in the system-level design process have been discussed. It must be emphasized that most of these tasks are interdependent. Moreover, there are cyclic dependencies. For example, the architecture allocation heavily influences the partitioning task, and vice versa. Also, timing constraints are input and output for both scheduling and communication synthesis. Because of these dependencies, there is no sequence of tasks which guarantees an optimum solution.

A heuristic solution to this problem uses an iterative approach. A set of tasks is repeated until an acceptable solution is found. The decision, whether a solution is "good enough" to proceed to the next task, is made by the system designer based on estimation data and his experience.

However, the design tasks must be supported by CAD tools and CAD tools place restrictions on the order they are executed. Thus, the system designer has to follow the guidelines under which the CAD tools were developed. Such a set of guidelines, which refine the abstract specification model into a detailed implementation model ready for manufacturing, is called a *methodology*.

A top-down methodology starts with a specification at the highest level of abstraction and moves down to lower levels while step-wise refining the model. With each step, the design model becomes a more accurate representation of the final implementation.

On the other hand, a bottom-up methodology starts from the lowest level, composing components together. These composed components then can be used in the next step to build even more complex components.

Both methodologies can be combined in order to achieve the best productivity. Usually, the top-down methodology is used until the system is decomposed into components which can be selected from the component library. The component library, on the other hand, is built using the bottom-up strategy.

With this combined approach, only the top-down phase affects the crucial *time-to-market* for the product, because the component library can be built beforehand. Thus, the key to a short design time enabling *"product-on-demand"* is the use of IP components, which are predesigned and can be easily integrated in order to build the product. The system design methodology, which is based on the integration of IP components, is called *IP-centric* [GDZ99a, GDZ99b].

1.1.5 Intellectual Property

As stated earlier, the reuse of IP is a key issue in SOC design. In fact, it is considered a paradigm shift that can be compared to the introduction of high-level synthesis a few years ago. This section elaborates on IP components and the benefits, problems and requirements with IP reuse.

1.1.5.1 IP components

At the system level, predesigned components are frequently called IPs. IP components are independent processing elements, in other words, they have their own flow of control and interact with the other system components via the system busses. Unlike full-custom components, which are synthesized from scratch specifically for the application, IP components are selected from an IP library and are fixed or allow only limited customization.

Typical IP components include memories, processors, and industry standard circuits. Memory IPs, like RAM and ROM blocks, can usually be customized in their size, whereas processor IPs come typically as fixed cores. Processor IPs include embedded micro-controllers, general-purpose, and digital signal processors (DSP). Special-purpose IPs implement industry standards, for example, encoding and decoding algorithms like MPEG, JPEG, etc., or communication devices like PCI or VME bus interfaces.

IP components can be categorized into hard and soft IPs. *Hard IP* components are developed by use of a standard design process and are fully implemented in a specific technology. In particular, for hard IPs, there is a physical representation of the layout, for example, in form of a GDS-II file [KB98]. Since hard IPs are fully implemented, their performance characteristics and other metrics are very accurate and predictable. However, hard IPs are inflexible and limited to a specific target technology.

Soft IP components, in contrast, are very flexible IPs which come typically in form of synthesizable RTL code. Usually, soft IPs can be parameterized or are user-configurable in terms of data size, features, etc. Since soft IPs are synthesizable, they can be implemented in any target technology as well. However, the implementation metrics of soft IPs are not as predictable as for hard IPs, because the final implementation has yet to be synthesized.

IP components can also be classified into internal and external IPs. Since the process of developing the system is decoupled from the development of the IP components, these tasks can be performed independently by separate design teams in possibly different companies. *Internal IPs* are developed inside the same company which builds the system. Typical internal IPs include legacy designs which can be reused from former products that have been proven to be successful.

The use of *external IP* is part of a new business model in the EDA industry. External IP components are developed and provided by IP providers outside the company building the system. While the system house, also called IP integrator, can focus on the problem of the system specification, integration and implementation, IP vendors develop and offer the required IP components. With this approach, the system house benefits from a large library of optimized, well-tested and well-documented components which are available when needed. The IP providers, on the other hand, can take advantage of their expertise in specialized design areas without the need to build and sell complete systems. This business model works well because, in many cases, it is cheaper for the system house to purchase an IP component as to invest time and money to develop it from scratch.

1.1.5.2 IP reuse

The reuse of predesigned components is well-known in the EDA. For example, at the RT level, reuse includes the instantiation of components from the RTL library, such as registers, multipliers, arithmetic-logic units (ALU), etc. Similar to IPs, the components in a RTL library can be internal legacy components or external components supplied by another company.

The advantages of reuse are similar at the RT and the system level. At both levels, reuse of components drastically reduces the time and the cost of the design because the reused components are already designed, optimized, and tested. However, in order to exploit these benefits, several problems have to be overcome.

The main two problems involved with design reuse are component matching and component integration. First, the task of *matching* is to find a corresponding counterpart in the component library for a part of the design specification. A component can only be used in the implementation, if it matches the functionality and meets the constraints in the specification.

Then, the task of component *selection* is to choose one component from the set of matching components which best meets the design goals. Typical design goals are minimal cost or best performance.

Finally, when a suitable component is chosen, it must be integrated with the rest of the design. The task of *integration* is to ensure that the component is properly connected and controlled so that it cooperates with the other system components and works with the right data at the right time.

Component matching and integration are more difficult at the system level than at the RT level because of the higher level of abstraction. At the RT level, the behavioral and structural models of the components are close to the behavioral specification so that mapping and integration are usually straightforward.

For example, the behavioral model of an adder is simply an add operation indicated by a plus sign. The structural model is a component with two bit vector input ports and one bit vector output port. With these models, it is easy to map an addition onto an adder component by feeding the left and right arguments into the input ports and reading the result from the output port[3].

At the system level, however, the tasks of component matching and component integration are not as straightforward because the behavioral and structural models of system components are much more complex.

The functionality of both, the system specification and the IP components, is described by *algorithms* rather than primitive arithmetic operations. Hence, IP matching essentially has to deal with the comparison of algorithms. Whether two algorithms match, however, is undecidable in the general case. Therefore, IP matching requires special handling by the tools[4] or the help of the designer.

The integration of IPs includes similar problems. Instead through plain ports, IP components usually communicate via non-trivial interfaces by use of possibly complex communication protocols. Hence, IP integration typically requires interface synthesis and protocol translation to be performed.

While the matching, selection and integration of IP components are tasks performed by system integrators, IP providers have to deal with the task of IP protection which is discussed in the following section.

1.1.5.3 IP protection

Since the business of IP vendors depends on selling their intellectual property to other companies, IP providers have to protect their IP from being copied, modified, or reverse-engineered. IP protection addresses the security issues for external IPs.

In general, IP components are covered by a copyright and can be further protected by legal contracts and non-disclosure or non-distribution agreements. However, it is usually very difficult to detect and to prove that an IP is used without permission. Therefore, technical measures are taken in addition to legal guarantees.

For hard IPs, protection can be easily achieved by keeping the final implementation with the IP provider. This works well if the IP is provided by the same silicon vendor who also performs the final layout and manufacturing of the system. Instead of the real implementation, the system integrator is supplied with simulation models and estimation data of the IP. With these models, the system can be developed without the need for the real IP. Typically, the deliverables for a hard IP include simulation and timing models at different levels of abstraction, performance, power, and other metrics, a floor plan model, and comprehensive documentation about the functionality and interface specification of the IP [KB98].

[3] Given a properly annotated component library, matching and integration is not significantly more difficult for other RTL components.

[4] For example, the matching of IP components could be indicated by use of a naming convention or some form of annotation recognized by the CAD tools.

For soft IPs, a different approach is necessary. Since the final implementation will be synthesized by the system integrator, the complete, synthesizable model must be made available. In order to still hide the implementation or algorithm details, the IP can be provided in precompiled format without source code. This is basically the same, well-known idea used in the software business to protect proprietary code from being reverse-engineered.

Watermarking can also be used for IP protection. This technique inserts a unique identifier, a so-called watermark, into the component. Such a watermark is typically hidden and difficult to remove. The existence of a watermark ensures that the component can always be identified. Watermarking can be easily applied to hard IPs [KLM+98], but is difficult for soft IPs since it must be ensured that the watermark is not lost during synthesis.

1.1.5.4 IP requirements

This section summarizes the requirements for successful reuse of IP. Different requirements apply to the components, the methodology, the design model, and the tools being used in system level design.

In order to be reusable, IP components must provide support for IP matching, selection and integration. IP matching requires a clearly specified functionality. For IP selection, accurate quality metrics are needed, such as performance, power consumption, size and cost. In order to allow seamless IP integration in a system, IP components must provide standard or flexible interfaces. In other words, the IP interfaces and the communication protocols used must be clearly specified.

Furthermore, IP components need some form of protection and should be highly optimized and well-tested. In order to increase the reusability, IPs should also be customizable to different environments and portable to different technologies. The deliverables for IP components include simulation models at different abstraction levels, quality metrics and comprehensive documentation [KB98, SK+99].

The system design methodology must be IP-centric. In other words, IP reuse must be an integral part of the methodology. The methodology must encourage the reuse of IP by use of guidelines and IP-centric models. Last but not least, the methodology must be supported by suitable tools.

Well-defined, IP-centric models are required for the design and component representation throughout the design process. The design model must allow the easy insertion and replacement of IP components ("plug-and-play") at any time in the design process. This requires that the model clearly separates communication and computation in the design. This ensures that communication and computation portions can be clearly identified and easily replaced with different communication protocols or computation algorithms.

Finally, tools are required to support the user with design maintenance and refinement. System-level tools must recognize and support IP components. While design decisions usually are made by the system designer, CAD tools are needed for all tedious and error-prone tasks during the design process, including specification capture, architecture exploration, communication synthesis, and hand-off to semiconductor manufacturing.

This work addresses the issues of system-level design in general, and, in particular, the problems involved with the reuse and integration of IP components. An IP-centric methodology is presented which is based on well-defined design models and a language that specifically supports the requirements of system-level synthesis.

1.2 Related Work

This section contains a brief overview about related work in system-level design.

While there are efforts, such as the virtual socket interface alliance (VSIA) [BS99], which address general system design issues like the definition of SOC design, system data formats, IP interfaces and modeling guidelines, the majority of interesting projects resemble actual design systems. A subset of such systems for codesign and system-level design is presented in the following section.

Furthermore, Section 1.2.2 lists traditional languages which are commonly used for software, hardware, and system development.

1.2.1 Design systems

For system-level design and codesign, promising approaches and methodologies have been proposed in the academia as well as in the industry. A set of interesting tools and design environments has already been developed.

Table 1.1 lists promising system-level design projects developed by universities. Furthermore, a set of commercial tools and design systems is shown in Table 1.2. It should be noted that many commercial tools have evolved from university projects. For example, CoWare and SystemC[5] originated in academia

Although it is very difficult to classify all these approaches, the main emphasis for each project is noted in Table 1.1 and Table 1.2. Most systems try to cover many aspects of system-level design, but have their strength in the area indicated in the tables. Each of these projects really focuses only on a subset of the tasks. Furthermore, the target architectures addressed by the tools are, in many cases, quite specific and do not cover the whole design space.

---

[5]SystemC originally is Scenic.

Project	University	Main Focus
Chinook	Univ. of Washington	Communication synthesis
Cobra	Univ. of Tübingen	Rapid prototyping
Cool	Univ. of Dortmund	Synthesis
Cosmos	TIMA Laboratory	Synthesis
Cosyma	TU Braunschweig	Synthesis
JavaCAD	Univ. of Bologna	Networked framework
JavaTime	UC Berkeley	Simulation
Lycos	TU Denmark	Synthesis
Polis	UC Berkeley	Formal specification
Ptolemy	UC Berkeley	Simulation
Scenic	UC Irvine	Simulation
SpecSyn	UC Irvine	Exploration
Tosca	Politecnico of Milan	Synthesis
Vulcan	UC Irvine	Synthesis
Weld	UC Berkeley	Networked framework

Table 1.1: System-level design projects in academia

The SpecC design environment described in this work compares well with the set of academia projects listed in Table 1.1. As described later, the SpecC system addresses system specification, simulation, as well as synthesis. However, the main focus of SpecC is design modeling, which is described in detail in Chapter 2.

System-level design and codesign systems can be classified by either homogeneous or heterogeneous specification.

- *Homogeneous specification*: A single language is used for specifying the system including hardware and software parts.
- *Heterogeneous specification*: Different languages are used for specifying the system, for example, VHDL (for hardware) and C (for software).

Examples for both types of systems are given in the next two sections.

1.2.1.1 Homogeneous specification

Chinook: Chinook[6] [COB95] is a codesign tool that addresses in particular interface and communication synthesis. Cosimulation and cosynthesis with timing

[6] Online information about Chinook is available at:
`http://www.cs.washington.edu/research/projects/lis/www/chinook/`

Project	Company	Main Focus
COSSAP	Synopsys, Inc.	Capture
CoWare	CoWare, Inc.	Interface synthesis
Eaglei	Synopsys, Inc.	Simulation
SystemC	Synopsys, Inc.	Simulation
Seamless	Mentor Graphics Corp.	Simulation
SPW	Cadence, Inc.	Capture
XE	Y Explorations, Inc.	Reuse

Table 1.2: System-level design projects in industry

constraints are addressed as well. Chinook is targeted at the design of control-dominated, reactive systems. The system specification is homogeneous since Verilog is used as the only input language.

Tosca: Tosca[7] [BFS95] is a synthesis-oriented system which, just as Chinook, targets at the design of reactive real-time embedded systems. Tosca is an early, pragmatic approach to codesign automation of control-dominated systems. The target architecture consists of a single micro-processor core and several ASICs. Assembly code is generated for execution by the processor and the ASICs are described in VHDL.

Cool: In contrast to the control-dominated systems Chinook and Tosca, Cool [Nie98] is a codesign system for data-flow dominated embedded systems. With Cool, a system is specified in VHDL. The synthesis result consists of assembly code for possibly multiple processors and synthesizable VHDL for possibly multiple ASICs. Cool emphasizes a precise partitioning approach using mixed integer linear programming (MILP) based on exact cost and performance measures.

Vulcan: Vulcan [GM96] is an early, synthesis-oriented system with homogeneous specification. HardwareC is used as description language for both hardware and software. Vulcan starts with a complete hardware solution (everything is implemented in ASICs) and then iteratively moves tasks to a single CPU in order to reduce the costs while obeying the given performance constraints.

[7]Online information about Tosca is available at:
`http://www.cefriel.it/eda/projects/tosca/html/default.htm`

Cosyma: Cosyma [EHB93, HE97, ÖBE+97] is a synthesis-oriented system focusing on hardware/software partitioning. The system is specified in C^x, a variation of the C language. The target architecture consists of one RISC processor with a coprocessor implemented in an ASIC. In contrast to Vulcan, Cosyma starts with an all-software implementation (the complete system is executed on a single CPU) and then moves tasks to the ASIC if the performance constraints are not satisfied.

Lycos: Just as with Cosyma, the target architecture of Lycos [MGK97] is an embedded micro-architecture consisting of one processor with a coprocessor implemented as an ASIC or FPGA. With Lycos, the system is homogeneously specified in either VHDL or the C language. The main emphasis of Lycos is the partitioning task.

Cosmos: Cosmos [VRD+97, IAJ94] targets at the development of multiprocessor architectures using a set of user-guided transformations on the design. In contrast to Cosyma and Lycos, the target architecture consists of possibly multiple processors. In Cosmos, the system is specified in SDL. The generated output consists of VHDL for the hardware, and C for the software parts of the system. It should be noted that the Cosmos system has been extended to support cosimulation with parts in the mechanical domain which are described in Matlab [CHM+99].

SpecSyn: SpecSyn [NVG91, GVN93, GVN+94] is a codesign environment for systems specified in SpecCharts, which is a front end language for VHDL. The main focus of the SpecSyn system is design estimation and design space exploration. The target architecture consists of multiple processors, ASICs and memories, connected via system busses.

Scenic/SystemC: The academic Scenic project [GL97, LTG97, GKL99] recently has been commercialized in form of the SystemC[8] initiative. In Scenic (or SystemC), the design system is described with the software programming language C++. Required modeling features not present in the language, like, for example, concurrency and synchronization, are specified by use of special methods implemented in standard classes provided with the Scenic libraries. Although Scenic targets also at system synthesis, its main focus is simulation. In other words, Scenic is a simulation-oriented system, in contrast to the synthesis-oriented systems listed earlier.

For a more detailed description of Scenic including a comparison with the SpecC system described in this work, please refer to [DG98].

[8] Online information about SystemC is available at: `http://www.systemc.org/`

XE: Although hardware oriented, the explorations environment XE[9] is a commercial tool for system design. Based on a behavioral synthesis system, the strengths of XE are design space exploration and reuse of IP components. In XE, the system is specified with VHDL. The target architecture consists of custom hardware and reused components including processors.

Polis: The Polis[10] system [BGJ+97, CGH+93] is targeted at small reactive embedded systems. Its main focus is a formal approach to codesign enabling formal verification. Polis internally represents a system by use of the codesign finite state machine (CFSM) model. The design specification for Polis is described in Esterel [BG92]. The output consists of a HDL description (e. g. VHDL) for the hardware and C for the software parts.

Cobra: Cobra[11] [KKR94, Ros97] is a prototyping and emulation environment for codesign. VHDL is used as specification and implementation language. In Cobra, the target architecture consists of a set of interconnected field programmable gate arrays (FPGAs).

JavaTime: JavaTime [YMS+99] is a codesign system which focuses on simulation. The standard software programming language Java is used as modeling language. As in Scenic, required modeling features not present directly in the language are specified by use of special methods implemented in a supplied class library. It should be emphasized that, in the JavaTime system, the Java language is used to syntactically describe the system. The standard Java classes, for example the support of internet communication, etc., are not used.

JavaCAD: JavaCAD [DBB99] is another example of a codesign system which uses Java as the specification language. As JavaTime, JavaCAD focuses on simulation. However, JavaCAD also is a networked framework for codesign. In other words, it utilizes the networking capabilities of Java for distributed codesign. In particular, JavaCAD uses networked simulation for protection of IP components, as mentioned in Section 1.1.5.3.

[9] Online information about XE is available at: `http://www.yxi.com/`

[10] Online information about Polis is available at: `http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html`

[11] Online information about Cobra is available at: `http://www.fzi.de/divisions/sim/projects/cobra.html`

1.2.1.2 Heterogeneous specification

Ptolemy: Ptolemy[12] [LM87, KL93] is a typical example of a system design framework with heterogeneous specification. Multiple languages, such as C, VHDL, and Java, can be used for the system specification. Furthermore, heterogeneous models of computation, such as the synchronous data flow (SDF) model, can be mixed and simultaneously simulated in the system. Ptolemy is a typical representative for simulation oriented systems.

CoWare: CoWare[13] [RVB$^+$96, Arn99] is a commercialized codesign environment that, similar to Chinook, addresses interface synthesis for hardware/software communication. CoWare also targets at the simulation and design of heterogeneous DSP systems. Input languages supported include VHDL, Verilog, and C.

SPW: The signal processing work system SPW[14] offered by Cadence is a commercial framework for heterogeneous system specification and cosimulation. SPW is data flow oriented. In other words, SPW addresses in particular DSP and communication systems design. As does CoWare, SPW supports simultaneous simulation with multiple languages, such as VHDL and Verilog for hardware, and C for software.

COSSAP: COSSAP[15] is a block diagram based framework offered by Synopsys. COSSAP is very similar to SPW and targeted at DSP applications as well. A system is specified by use of block diagrams which can be simulated. The output generated consists of synthesizable HDL for the hardware and C code for the software parts of the system.

Seamless: The Seamless[16] co-verification environment (CVE), offered by Mentor Graphics, is another example for hardware/software cosimulation. As CoWare and SPW, Seamless CVE supports VHDL and Verilog for the hardware portions of the system, and C for the software portions.

[12] Online information about Ptolemy is available at: `http://ptolemy.eecs.berkeley.edu/`

[13] Online information about CoWare is available at: `http://www.coware.com/`

[14] Online information about SPW is available at: `http://www.cadence.com/technology/hwsw/cierto_spw.html`

[15] Online information about COSSAP is available at: `http://www.synopsys.com/products/dsp/cossap_ds.html`

[16] Online information about Seamless CVE is available at: `http://www.mentor.com/seamless/products.html`

Eaglei: Eaglei[17], offered by Synopsys, is a cosimulation tool very similar to Seamless. Eaglei focuses on hardware/software co-verification from post-partitioning through a physical prototype. Again, VHDL and Verilog are used for the hardware parts of the system, and C is used for the software.

Weld: The Weld[18] project [CSN98] is a networking framework for heterogeneous systems. It addresses the use of networking in electronic design. The Weld project defines a design environment which enables web-based CAD and supports distributed operation via the Internet.

1.2.2 Languages

As seen with the systems listed in the previous section, a large set of languages is currently being used in embedded systems design. The main reason for this is that the "perfect" language to be used for system-level design has not yet been determined, and it is doubtful if such a language can actually exist. However, this indicates the need for research for a possibly new language targeted specifically at system-level design.

In order to determine how well a specific language is suited for a given purpose, the requirements and goals for the language have to be identified. For example, a typical requirement for languages used in computer science is preciseness. In contrast to languages for human interaction, such as English, German, or Chinese, languages used for automated processing must not allow any misunderstandings. In other words, these languages must be formal and unambiguous.

In addition to these general necessities, many other requirements and goals exist for a system-level design language. In Chapter 4, these requirements will be discussed and identified. Furthermore, a new language called SpecC will be proposed which exactly matches the identified requirements.

In the following sections, some traditional languages used for software design, hardware design, combined software and hardware design (codesign), and system design are briefly reviewed.

1.2.2.1 Software programming languages

Literally hundreds of software programming languages exist today. For real applications, mostly imperative programming languages are used. Among these, some

[17] Online information about Eaglei is available at:
`http://www.synopsys.com/products/hwsw/eagle_ds.html`

[18] Online information about Weld is available at:
`http://www-cad.eecs.berkeley.edu/Respep/Research/weld/index.html`

also have been used for the design of embedded systems. The most important ones are the following three languages.

C: The C programming language [X3/90], originally developed and used with the UNIX operating system, has been officially standardized by the ISO and ANSI. Since then, C has become the de-facto standard for software design.

C++: The C++ programming language [ES90, X3/97, Str97] is an object oriented extension of the C language. It also has been standardized and is being used widely for software development.

Java: Java [AG96] is a recently developed language, whose syntax is very similar to C. Java has gained much of its popularity because it is specifically suited for network applications such as the use of executable code in the world-wide web (WWW).

1.2.2.2 Hardware description languages

Hardware description languages (HDLs) are used for the formal specification and description of hardware. The following is a list of languages commonly being used in industry and academia.

VHDL: VHDL [IEEE87, IEEE93] is a hardware description language standardized by the IEEE. Although VHDL is primarily a simulation language, it is being used widely for synthesis as well[19] [JDK$^+$97].

It should be noted that extended versions of VHDL exist. For example, VHDL+ [ICL97], which is developed by ICL, provides language extensions for interfaces and so-called activities. A comparison of VHDL+ with the SpecC language proposed in this work can be found in [GZG98].

Verilog: Verilog [IEEE96, TM91] is another hardware description language commonly being used for simulation and synthesis[20]. Verilog also has been standardized by the IEEE.

HardwareC: HardwareC [KM90] has been developed specifically as a language for hardware design [Mic94]. Syntactically, HardwareC is similar to the C programming language, but provides additional constructs needed for describing hardware. HardwareC is not as complex and powerful as VHDL or Verilog.

---

[19] For synthesis, only a subset of VHDL can be used since some constructs in VHDL are not synthesizable.

[20] As for VHDL, only a subset of Verilog is synthesizable.

Handel-C: Handel-C [APR+96] is another language used for hardware design which is syntactically similar to C. Semantically, Handel-C is based on the model of communicating sequential processes (CSP). In comparison to the previous hardware description languages, the expressive power of Handel-C is quite limited.

1.2.2.3 Codesign languages

Since codesign consists of the design of systems including both software and hardware, languages combining the features of software programming languages and hardware description languages are preferably being used. Two early approaches should be mentioned.

Statecharts: Statecharts [Har87, DH89] is a state-based specification language for codesign, particularly targeted at the design of reactive systems. Statecharts uses an extended finite state machine model with support of hierarchy, concurrency and other common concepts. Statecharts is based on a visual formalism with a graphical representation and has been extended in several variations.

SpecCharts: SpecCharts [NVG91, GVN93, GVN+94], a combination of Statecharts and VHDL, is based on the program state machine (PSM) model. SpecCharts has a textual and an equivalent graphical representation. It is used in the SpecSyn system for design space exploration and estimation.

1.2.2.4 System-level languages

In addition to the features provided by codesign languages, system-level languages typically include other aspects of a complete system specification as well, for example, constraints in the mechanical domain.

SDL: The specification description language SDL [BHS91, ITU92] is widely used in the field of telecommunication. It is also applied to system design, for example, in the Cosmos system. SDL has been standardized by the ITU.

SLDL: SLDL[21] [Sch99] is a new system-level design language currently being defined in the EDA industry. SLDL focuses on the formal specification of a systems requirements and constraints and allows partial (incomplete) descriptions.

[21] Online information about SLDL is available at: `http://www.inmet.com/SLDL/`

UML: The unified modeling language UML[22] [RJB98] is an industry-standard language for the specification of software systems. UML includes visualization, construction and documentation. The goal of UML is to simplify the process of software design.

1.3 Goals

After the review of a set of promising design systems and important specification and modeling languages in the last two sections, it should be noted that many weaknesses and limitations exist in these approaches. Rather than pointing out specific weaknesses, two major problems should be emphasized.

First, every system presented in Table 1.1 and Table 1.2 only focuses on a subset of the system design tasks and hardly addresses the remaining tasks. In order to cover the whole spectrum of system-level design, it is not even possible to easily combine a set of approaches because of large differences in the methodologies, the models, and the languages being used.

Second, the languages and the design systems are developed separately. Hence, they do not match and modifications and adjustments are necessary. For all design systems listed earlier, the languages used were originally developed for different purposes. Because of this, most systems can only support a subset of the original language, and also are missing features that the language does not support.

In this work, a new approach is taken. Instead of using an existing language, that originally was not designed for system-level design, a new language, called SpecC, is developed that exactly matches the requirements and goals for this task. In addition, a methodology with well-defined design models and explicit support of IP is proposed. The language, the methodology, the models and the implemented design environment are all designed and tuned for the specific requirements and goals of system-level design.

In particular, the following issues need to be solved concurrently and consistently in order to make system-level design successful.

- The system-level language must
  - be executable,
  - be synthesizable,
  - support all hardware-specific concepts, and
  - support all software-specific concepts.

[22]Online information about UML is available at: `http://www.rational.com/uml/index.jtmpl`

- The design models must
  - be well-defined,
  - separate communication and computation,
  - support IP, and
  - support a general (non-restricted) target architecture.
- The design methodology must
  - be well-defined,
  - support highly abstract specification,
  - support validation and verification,
  - support design space exploration,
  - support synthesis, and
  - provide a clear hand-off for the final production.
- The design environment must
  - be a coherent system,
  - contain a complete set of tools, and
  - allow manual and automatic refinement.
- The system design approach must
  - be proven with a set of real-world examples, and
  - gain wide acceptance, in particular in industry.

All these issues will be addressed in the remainder of this work.

1.4 Outline

In order to employ EDA at the system level, the increased level of abstraction and the reuse of IP must be reflected in the system design methodology and, in particular, in the design descriptions, the models and languages, the component library, and the CAD tools. These issues are addressed in the following chapters which present the SpecC system design approach.

The rest of this work is organized as follows:

Chapter 2 introduces the SpecC design model which is based on *behaviors* containing computation and *channels* encapsulating communication. In particular, Chapter 2 describes the models and the guidelines for modeling systems with IP components in the SpecC design environment.

Then, Chapter 3 presents the IP-centric SpecC design methodology. Starting with an abstract, executable specification of the intended system, the SpecC methodology uses step-wise refinement to map the system model onto the target architecture. Using the modeling guidelines defined in Chapter 2, the SpecC methodology is based on four well-defined models representing the design at different stages during the refinement process. The *specification model* is transformed into the *architecture model* by architecture exploration. Then, communication synthesis is applied generating the *communication model*. Finally, the *implementation model* of the system is obtained after software compilation and hardware synthesis.

Chapter 4 discusses the requirements and objectives of system design languages and examines traditional languages regarding their support of the required properties. Since none of these languages satisfies all requirements, a new language, called SpecC, is proposed. The SpecC language is used in the SpecC system to represent the design models throughout the design process. Built on top of C, the SpecC language was developed to directly support all the concepts needed in embedded systems design, including behavioral and structural hierarchy, concurrency, state transitions, timing and exception handling. The SpecC language also features *plug-and-play* support for the reuse of IP.

The implemented SpecC design environment is described in Chapter 5. The SpecC system consists of a set of CAD tools for system validation, analysis, and synthesis, integrated in a graphical user interface (GUI). The main tool in the system is the SpecC compiler which allows the simulation and debugging of SpecC designs.

Chapter 5 also describes the central design representation which all SpecC tools rely on. The so-called SpecC Internal Representation (SIR) offers an application programming interface (API) for the SpecC tool developer, which allows to easily read, write, maintain and transform design models specified with the SpecC language. As such, the SIR provides an abstraction layer above the specific details of the SpecC language and allows the quick development of CAD tools for the SpecC design environment.

Chapter 6 addresses the protection of IP components in the SpecC design environment. Using the SpecC compiler, an IP provider can automatically generate public IP interface descriptions and secret IP simulation libraries for any design model. With this approach, it is ensured that no information about the internal implementation of the IP is revealed and the IP is fully protected against reverse-engineering.

Finally, Chapter 7 summarizes this work and its contributions and concludes with a brief discussion of open issues and future work.

Chapter 2

IP-centric Modeling

As described in the introduction, system-level design starts from an initial design specification which is then transformed, typically by use of several refinement steps, into a final implementation. Throughout this design process, the intended design is represented by a design *model*. A design model is an abstract representation of the real design. The level of abstraction of this model decreases with every refinement step.

The design model itself is typically described by use of a formal language. Many such languages exist already, and one new language, specifically targeted at system-level design, is described in Chapter 4 later in this work. However, it is important to understand that the design model being used in the design process is more important than the design language.

In other words, it must be emphasized that not every description that can be expressed in the language actually represents an usable model for the design process. Rather, the design description must match a well-defined model that can be recognized and processed by the design tools.

More specifically, the use of a well-defined model will also ensure that the design description can be efficiently synthesized. The ability to synthesize a particular design in an efficient manner is more a property of the design model rather than a characteristic of the language.

In order to obtain a well-defined model when specifying a system, modeling guidelines must be followed. Such guidelines will ensure that the described model matches the requirements of the design tools and also fits the design methodology. Modeling guidelines are commonly specified in form of a set of general and also specific rules. For example, please refer to [KB98] or [AG98].

In this and the following chapter, the design models and the modeling guidelines used in the SpecC design environment are presented. This chapter introduces the basic models and their characteristics. Then, Chapter 3 describes the methodology that, based on these models, consists of a set of well-defined transformations performed with these models.

2.1 Computation and Communication

For the design of embedded systems, the key representation for any design is a *block diagram*. Block diagrams consist of a set of blocks and a set of interconnections between the blocks. Block diagrams can also be hierarchical. Thus, each block in a block diagram can itself represent an inner block diagram.

The standard interpretation of block diagrams is that blocks represent components which perform a particular function or computation. These blocks can also interact or communicate with each other through the interconnections in the diagram. It is important to note that there are two types of distinct actions performed by the blocks, namely *computation* and *communication*.

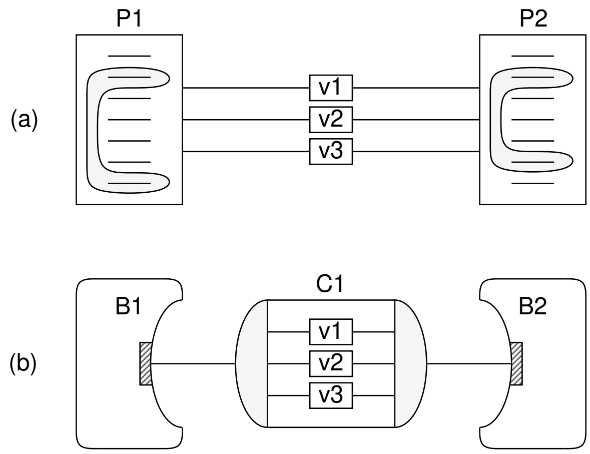

Figure 2.1: Separation of computation and communication

For example, a simple block diagram is shown in Figure 2.1(a). Two blocks, `P1` and `P2`, are communicating via the interconnections `v1`, `v2` and `v3`. These interconnections can represent wires in hardware or shared variables in software. By assigning values to these connections and following a defined protocol, e. g. two-way hand shaking, the blocks can communicate and exchange data.

In this scenario, the blocks `P1` and `P2` contain code for both communication and computation. In Figure 2.1(a), the communication in the code is illustrated as a shaded portion. However, it must be emphasized that there is no way to automatically distinguish the code for communication from the code used for computation.

Because communication and computation are freely intermixed and cannot be identified, it is neither possible to automatically change the communication protocol, nor to switch to a new algorithm to perform the computation.

In order to allow automatic replacement of communication protocols and computation algorithms, the *separation* and *encapsulation* of communication and computation is needed. This is supported in form of *behaviors* and *channels* in the SpecC model, as shown in Figure 2.1(b). Here, the computation is encapsulated in the behaviors `B1` and `B2`, and the communication is contained in the channel `C1`.

More specifically, the channel `C1` encapsulates the communication protocol in form of function definitions such as `read` and `write` or `send` and `receive`. These functions represent the interfaces of the channel. A channel also may contain necessary local functions and the communication media, such as the variables `v1`, `v2` and `v3`. On the other hand, the behaviors only contain computation. In order to communicate, the behaviors call the functions provided by the connected channel.

An important difference between the functions defined in a channel and the functions defined in a behavior is that a behavior is an *active* element, whereas a channel is *passive*. In other words, the functions in a behavior specify the functionality of the behavior itself. On the other hand, the functions in a channel are only executed when they are called from a connected behavior.

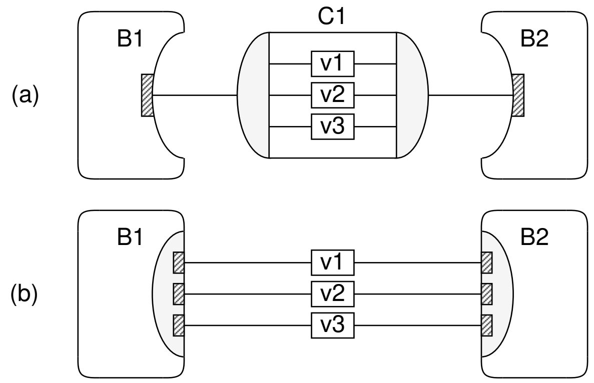

Figure 2.2: Communication inlining: (a) before, (b) after.

This difference is exploited when the model is finally implemented. For the implementation of a channel, its functions are *inlined* into the connected behaviors and the encapsulated communication media are exposed. This is illustrated in Figure 2.2. After the inlining process, the channel `C1` has disappeared. The internal variables `v1`, `v2` and `v3` are exposed and the communication protocol has been integrated into the behaviors `B1` and `B2`. Please note that in this final implementation model communication and computation are no longer separated.

2.2 The SpecC Model

In the SpecC model, behaviors and channels are used to encapsulate communication and computation, respectively. Following the style of standard block diagrams, behaviors and channels can further be composed in form of a structural hierarchy.

2.2.1 Basic structure

The basic structure of a SpecC model is a hierarchical network of behaviors and channels. A simple example is depicted in Figure 2.3.

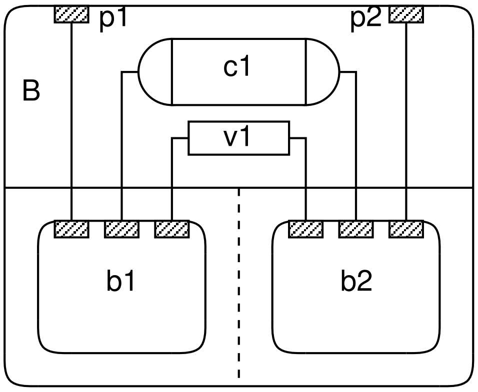

Figure 2.3: Example of a SpecC model

The example shows a behavior `B` which has two ports, `p1` and `p2`, through which it can communicate with its environment. Internally, these ports are connected to two child behaviors, `b1` and `b2`, which execute concurrently. These child behaviors can communicate in two ways. First, both are connected to a shared variable `v1` which, for example, could be written by `b1` and then read by `b2`.

Second, `b1` and `b2` can communicate by use of a communication protocol provided by the channel `c1`. For example, the behavior `b1` could call a function `send` provided by the left interface of channel `c1`. Then, when behavior `b2` calls the `receive` function provided by the right interface, the communication protocol implemented in the channel will ensure that the data is transferred correctly, for example, by use of explicit hand shaking or some specific synchronization mechanism and timing.

Please note that Figure 2.3 only shows one level of the structural hierarchy of the system. The child behaviors `b1` and `b2` could again consist of a network of behaviors and channels. On the other hand, the behavior `B` can be part of a bigger system as well.

2.2.2 Test bench

For any design model, the root of the hierarchy tree typically represents the test bench of the system. Since this is the top level, there are no ports for this behavior. Furthermore, it is a SpecC convention, that this top level behavior is always called `Main`.

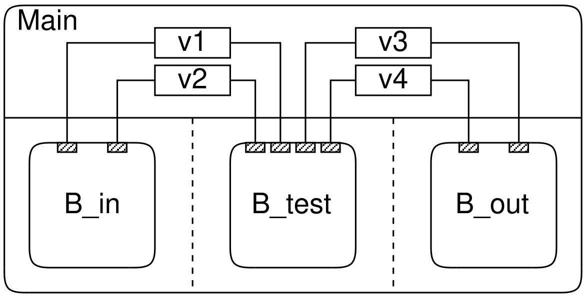

Figure 2.4: Typical test bench model

Figure 2.4 shows a typical example of a test bench model. The actual design model `B_test` is embedded in the test bench `Main` as a child behavior. It is connected to two other child behaviors `B_in` and `B_out`. `B_in` represents a stimuli generator which supplies test vectors to the input ports of the design. The output produced by the design model is observed and verified with the monitor behavior `B_out`.

2.3 Computation Models

In addition to the structural hierarchy described in the previous section, the SpecC model also supports behavioral hierarchy. Behavioral hierarchy is the composition of computation tasks over time. For example, a set of tasks can be executed one at a time or in parallel.

The SpecC behaviors, which encapsulate the computation tasks to be performed by a system, can be classified into eight different models. These behavior models are illustrated in Figure 2.5. Their characteristics are described in the following sections.

2.3.1 Algorithmic program

A SpecC behavior is called a composite behavior if it contains instantiations of child behaviors. Otherwise, it is called a leaf behavior. In Figure 2.5, a leaf behavior is shown in (a). On the other hand, composite behaviors are shown in (b) through (f).

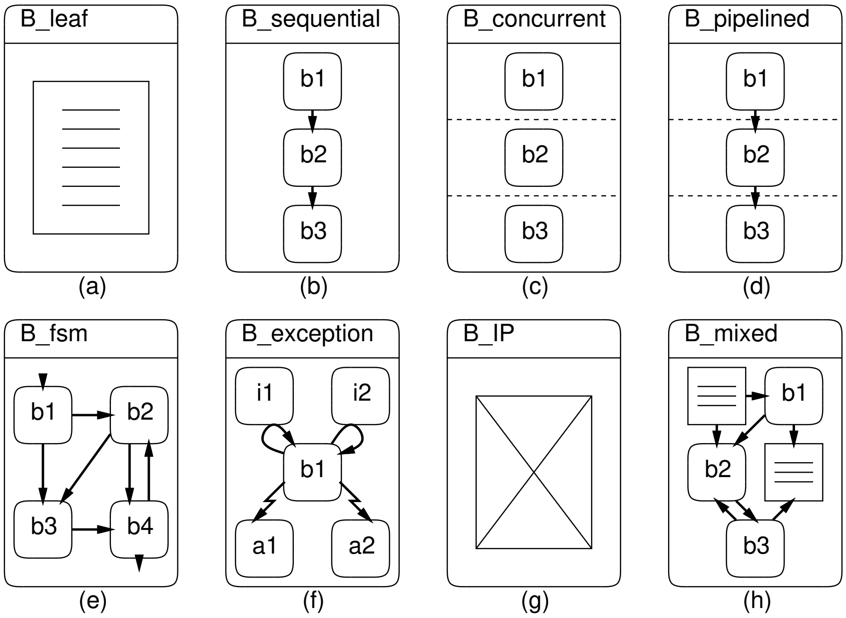

Figure 2.5: Behavior models: (a) leaf behavior, (b) sequential behavior, (c) concurrent behavior, (d) pipelined behavior, (e) FSM behavior, (f) exception behavior, (g) IP behavior, (h) mixed behavior.

The functionality of a leaf behavior is described by an algorithmic program. This program is started when the leaf behavior is activated and the termination of the program also determines the completion of the execution of the behavior.

The program in the leaf behavior can contain any type of programming statements, such as assignments, conditional statements, loop statements and function calls. More specifically, the statements provided by the C programming language can be used in a SpecC leaf behavior. In other words, a leaf behavior is equivalent to a C program.

A very important property of a leaf behavior is that it is *atomic*. In other words, for synthesis and all refinement tasks involved with it, a leaf behavior represents the smallest indivisible unit in the SpecC design model. For example, during the task of partitioning, a leaf behavior will be assigned completely to either hardware or software. It will not be cut into smaller parts.

The atomicity of the leaf behaviors determines the *granularity* of the design model. With a coarse granularity, the design system consists of only few behaviors and most of the functionality of the system is specified inside the leaf behaviors. This simplifies the refinement tasks which are dealing with only a few objects, but, at the same time, it heavily restricts the design space and will typically lead to a sub-optimal solution.

On the other hand, with a fine granularity, the system is specified with many behaviors with only simple functionality. As an extreme example, each arithmetic operation in the design could be specified in a separate leaf behavior. Such a fine granularity implies a large design space, but also requires each refinement task to handle a large set of objects leading to long run-times.

It is the task of the system designer to specify the system with the right granularity. In other words, the system designer has to trade-off a fine grained model with a large design space against a coarse grained model with easy refinement.

2.3.2 Sequential execution

The sequential execution of leaf behaviors can be specified with two types of composite behaviors. First, as shown in Figure 2.5(b), the leaf behaviors `b1`, `b2` and `b3` can be executed in a fixed, unconditional order, one at a time. The execution of the behavior `B_sequential` will start with the execution of `b1` and finally terminate when `b3` has finished its execution.

Second, sequential execution can be specified in a SpecC model in form of a finite state machine (FSM), as shown in Figure 2.5(e). The FSM model allows arbitrary transitions between the child behaviors and, thus, supports conditional execution and loops. The execution of a FSM behavior starts with the indicated initial behavior, such as `b1` in Figure 2.5(e). A FSM behavior terminates when a transition on completion is performed, as shown at `b4`.

2.3.3 Concurrent execution

For the parallel execution of behaviors, again two types of composite behaviors are provided. First, the concurrent execution, as shown in Figure 2.5(c), will execute all child behaviors simultaneously. The execution of `B_concurrent` starts the child behaviors `b1`, `b2` and `b3` at the same time and finishes as soon as all children have completed their execution.

Second, as a special form of concurrency, a pipelined behavior, as shown in Figure 2.5(d), executes its child behaviors in a pipelined fashion. Pipelined execution implies the iterative execution of the children. For Figure 2.5(d), only `b1` will be executed in the first iteration. In the second iteration, `b1` and `b2` will be executed concurrently. In the third and all following iterations, all three children are executed in parallel.

The pipelined behavior also ensures that the data exchanged between the child behaviors is shifted to the next stage each time a new iteration starts. This is described in detail in Section 4.5.2.2.

2.3.4 Exceptions

A special behavior type allows the specification of exceptional execution. As illustrated in Figure 2.5(f), an exception behavior contains one child behavior `b1` for standard execution, and several other child behaviors, such as `i1`, `i2`, `a1` and `a2`, for the handling of exceptions. Two types of exceptions are distinguished, namely *interrupt* and *abortion*.

In case of an interrupt, the behavior `b1` is stopped immediately in its execution and an interrupt behavior, such as `i1` and `i2`, is executed. Once the interrupt behavior finishes, the main behavior `b1` can resume its execution.

In case of abortion, the execution of the behavior `b1` is aborted immediately and will not be resumed. Instead, an abortion behavior, such as `a1` and `a2`, will take over and finish the execution.

The execution of an exception behavior starts with the execution of the main behavior. The execution is terminated when the main behavior completes or an abortion behavior has been executed.

2.3.5 IP model

In order to model IP components, a special IP behavior is supported. The essential property of IP components is that their internals are hidden and cannot be seen from the outside. Therefore, an IP behavior, as shown in Figure 2.5(g), is modeled as a black box whose contents are not accessable. Furthermore, an IP behavior is fixed and cannot be modified during synthesis and refinement.

Because of these restrictions with IP behaviors, special care has to be taken when design models with embedded IPs are transformed. This is described in detail in Section 2.5.

For the sake of completeness, a mixed behavior is shown in Figure 2.5(h). As described later in Chapter 4, the SpecC language allows such behaviors consisting of a mixture of child behaviors and algorithmic code. However, this behavior model is depreciated and should not be used in a well-specified design model[1].

[1]It is possible and also straightforward to automatically convert such mixed behaviors into a set of well-defined behaviors by introducing additional child behaviors and levels of hierarchy. However, currently such a tool has not been implemented yet.

2.4 Communication Models

The communication models mentioned earlier are reviewed in Figure 2.6. There are two models of communication, namely the shared memory model and the channel model.

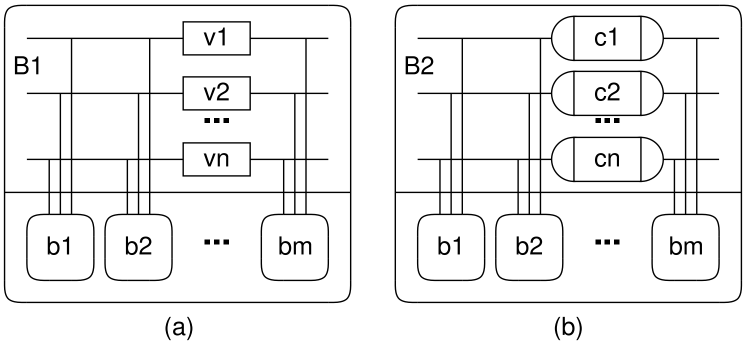

Figure 2.6: Models of communication: (a) shared memory model, (b) channel model.

2.4.1 Shared memory model

The shared memory communication model is realized by use of variables declared in the behavior that encapsulates the communicating child behaviors[2]. As shown in Figure 2.6(a), the variables `v1`, `v2`, ..., `vn` are declared in behavior `B1` and represent communication wires which hold their value over time, acting as a memory. The instantiated child behaviors `b1`, `b2`, ..., `bm` have access to these wires through their ports, so that the stored values can be shared among the connected children.

In the shared memory model, the child behaviors communicate by assigning values to their output ports (send) and observing values at their input ports (receive). While this basic scheme of communication is sufficient for simple cases, communication protocols are typically needed in the more general case, involving synchronization, timing, buffering, error correction, etc. As stated earlier, such communication protocols should be separated from the computation and should be encapsulated in channels, which are described next.

[2]As described in Chapter 4, the SpecC language allows global variables, declared outside of any behavior, to be accessed from the inside of behaviors. Thus, such global variables could also be used for a shared memory communication model. However, this is not recommended since there is no explicit connectivity to these variables. When using local variables in parent behaviors, which can only be accessed through ports, as shown in Figure 2.6, the connectivity is obvious and the model becomes less error prone.

2.4.2 Channel models

In the SpecC model, channels are used to encapsulate communication. Six different channel models are shown in Figure 2.7.

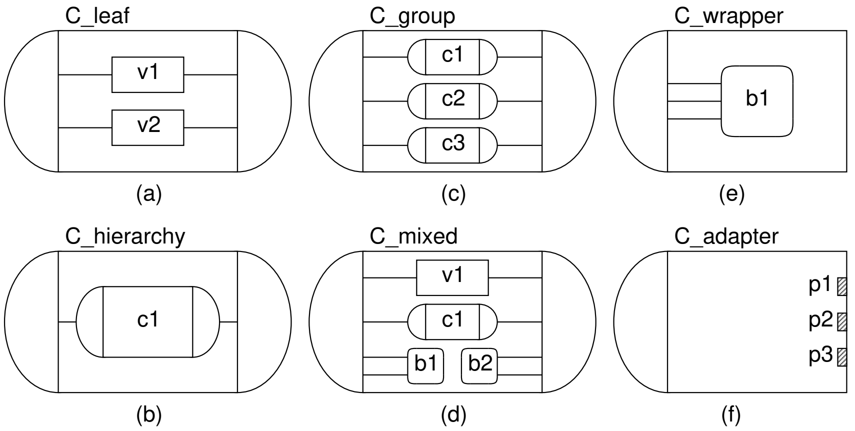

Figure 2.7: Channel models: (a) leaf channel, (b) hierarchical channel, (c) grouping channel, (d) mixed channel, (e) wrapper channel, (f) adapter channel.

A basic or *leaf channel*, as shown in Figure 2.7(a), consists of a set of local variables, such as `v1` and `v2`, and a set of communication functions. The functions of the channel use the local variables to realize the communication. These functions are made available through the interfaces of the channel and can be called by behaviors whose ports are connected to these interfaces.

Similar to behaviors, channels can also be hierarchical, as shown in Figure 2.7(b). A channel is called a *hierarchical channel* if it contains a child channel. A typical example for hierarchy in channels is a communication protocol stack. For example, a channel providing `send` and `receive` functions for large blocks of data might use an internal channel that provides `send_byte` and `receive_byte` functions.

A channel, that instantiates a set of child channels, as shown in Figure 2.7(c), is called a *grouping channel*. This channel model can be used to combine a set of channels into one. For example, a system bus, that is capable of many communication transactions represented by different channels, can be well-modeled as a grouping channel.

Two special channel models, namely wrapper and adapter channels, are used for the communication with fixed behaviors, such as hard IPs, whose ports cannot be modified. A channel is called a *wrapper* if the channel instantiates a behavior, as shown in Figure 2.7(e). Typically, the behavior `b1` represents an IP core with fixed, bit-level ports. In order to raise the abstraction level for the communication, a channel `C_wrapper` is wrapped around the behavior. This channel provides a communication interface which translates high-level operations, such as `send` and `receive`, into the required bit-level transactions. Thus, other components in the system can easily communicate with the IP via common, high-level functions.

An *adapter channel*, as shown in Figure 2.7(f), is very similar to a wrapper channel. However, instead of encapsulating the IP behavior, an adapter channel provides ports to which the behavior can be connected. Thus, an adapter allows to drive low-level wires by use of a high-level, functional interface. Since an adapter can simply be plugged in between incompatible behaviors while leaving both behaviors on the same level in the structural hierarchy, it is preferred, in this work, over the wrapper model.

Similar to the mixed behavior model, the SpecC language described later, also allows mixed channels, as shown in Figure 2.7(d). Although syntactically possible, the mixed channel model is depreciated and should not be used in a well-defined specification.

2.5 Modeling with IP

For a specification model to be IP-centric, it must naturally and explicitly represent the reuse and integration of intellectual property (IP). While IP components must be represented in a way so that they can be easily identified, they must not be used differently than other components. In other words, IP models must not create an exception.

As discussed in the introduction, IP can usually be classified into soft IP and hard IP. Soft IP, which comes in form of synthesizable source code, applies to both, behaviors and channels. For both, the IP models are exactly the same as the non-IP models in the system specification.

On the other hand, hard IP, which represents a fixed core component whose internal structure is hidden from the user, only applies to behavior models. There is no channel model for hard IPs. The reason for this is that channels can only be used in the system specification and during intermediate refinement steps, but need to be inlined for the final implementation. The process of inlining requires knowledge about the internal structure of the channel.

In the following, three models representing IP in a system model are presented, first, the channel model for communication protocol IP, and then, the wrapper and adapter models representing hard IP cores. With all these three models, "plug-and-play" with IPs is possible.

2.5.1 Channel model

A proprietary communication protocol, or a proprietary implementation of a standard protocol, is represented by an IP channel in SpecC. With one exception, such a channel is not different from other channels in the system and therefore can be treated the same way.

The only exception is that an IP channel typically needs to be wrapped by another channel which performs data type conversion. For example, an IP channel might provide native functions to send and receive single bytes and also blocks of 512 bytes of data. However, in order to use this channel in an application that needs to transfer pictures of a certain size, e. g. 1024 by 768 pixels, a data type conversion is required from the picture type into the transferrable block type, and vice versa. This conversion can be easily performed by a channel surrounding the IP channel.

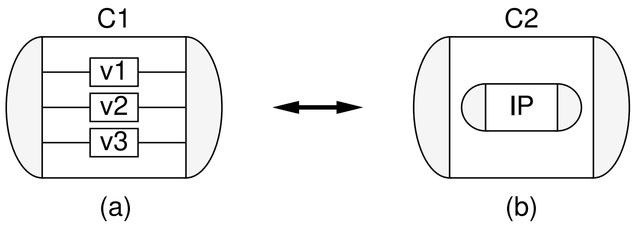

Figure 2.8: IP channel model: (a) virtual channel, (b) IP protocol channel.

Figure 2.8(b) shows this situation. The channel `IP` is encapsulated in channel `C2` which takes care of the necessary conversions. Assuming that in the initial system specification a virtual channel `C1`, shown in Figure 2.8(a), is used to transfer the picture, the channel `C2` can be used as an equivalent replacement at any time. Thus, it is possible to immediately plug in the IP protocol into the system model once the decision for its use has been made ("plug-and-play"). Also, this change is only local and does not affect any other channels or behaviors in the system.

2.5.2 Wrapper model

Similar to the IP channel in the previous section, a hard IP core is wrapped in a channel as well. This IP wrapper model is shown in Figure 2.9(b).

The IP behavior `IP1` contains ports which accurately describe the ports of the real IP core. Typically, these ports are modeled in a bit-exact manner. These behavior ports are mapped to variables in the channel. Communication with the IP is established by use of a set of high-level communication functions provided by the wrapper `W1`. These functions contain the detailed interface protocol to drive the

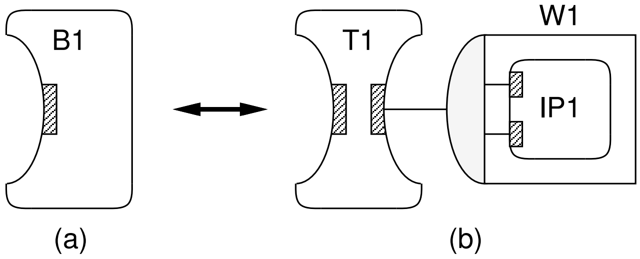

Figure 2.9: IP wrapper model: (a) synthesizable behavior, (b) IP replacement using a wrapper.

variables connected to the IP. Thus, by using the wrapper functions, other behaviors can easily communicate with the IP.

In order to allow "plug-and-play", a *transducer*, such as `T1`, is required in addition to the wrapper `W1`. A transducer is a synthesizable behavior used to connect two channels. Later, in the implementation model, the tranducer will contain two communication protocols, transforming `receive` requests from one protocol into `send` requests of the other, and vice versa. Note that a transducer can be eliminated in an optimization step if the two communication protocols are identical.

The reason for the need of a transducer stems from the fact that two channels cannot be directly connected because they are passive components. In order to connect passive channels, an active behavior is needed in the middle.

In summary, a synthesizable behavior, such as `B1` in Figure 2.9(a), can be replaced by an IP wrapper model, shown in Figure 2.9(b), at any time in the design process without affecting any other objects. The wrapper model consists of a transducer `T1` and the IP behavior `IP1` encapsulated in the wrapper `W1`.

2.5.3 Adapter model

The adapter model for incorporation of IP components is essentially the same as the wrapper model presented in the previous section. However, instead of the wrapper channel, an adapter channel is used to capture the communication functions.

Figure 2.10 shows the equivalence of a synthesizable behavior `B1` and the adapter model which consists of the IP core `IP2`, the adapter `A1` and the transducer `T2`.

2.5.4 Inlining

It has been already mentioned that, in order to obtain a final implementation model, the communication functions from the channels are inlined into the behaviors and the contained variables are exposed, forming the connecting wires. This process

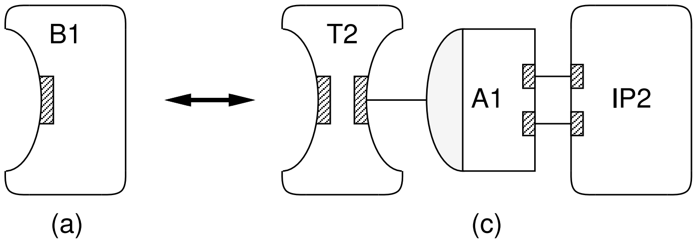

Figure 2.10: IP adapter model: (a) synthesizable behavior, (b) IP replacement using an adapter.

of inlining has been demonstrated in Figure 2.2 for two synthesizable behaviors connected by a standard channel (see page 33).

Although the principles of inlining are the same, the situation is slightly different when IP behaviors, wrappers, adapters, and transducers are part of the system model. Wrappers and adapters need to be inlined since they are essentially channels. IP behaviors are fixed and therefore cannot be modified to incorporate protocols. Transducers, however, can be treated just as standard behaviors.

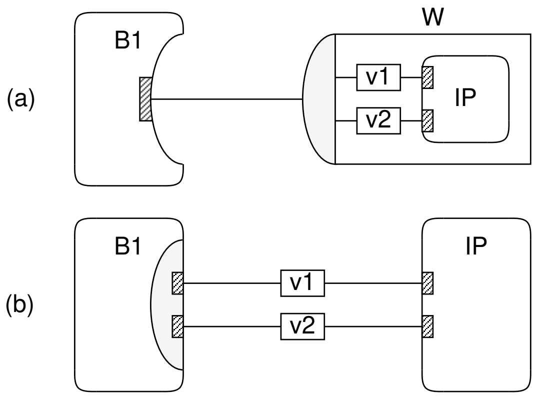

Figure 2.11: Wrapper inlining: (a) before, (b) after.

Three common cases are illustrated in the following. Figure 2.11 shows the process of inlining with a wrapper model. Before the inlining, the wrapper `W` is connected to a synthesizable behavior[3] `B1`. After the wrapper has been inlined, the IP communication protocol has been integrated into the behavior `B1` and the variables `v1` and `v2` are exposed, forming the connecting wires to the IP. Note that the IP behavior has been exposed as well, but was not changed during the process.

[3]Note that the transducer in the wrapper model is nothing else but a synthesizable behavior.

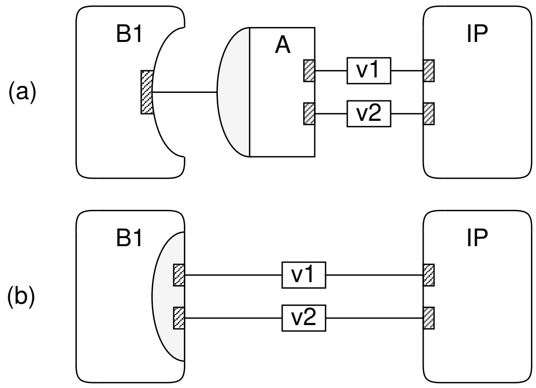

Figure 2.12: Adapter inlining: (a) before, (b) after.

As shown in Figure 2.12, the inlining process is very similar when using an adapter model. After the inlining, the adapter `A` has disappeared. Its communication functions have been incorporated into the behavior `B1`. Please note that the result from this inlining process is exactly the same as the one from the wrapper model, shown in Figure 2.11.

Figure 2.12 also shows that the inlining process for the adapter model does not change anything at all for the behavior `IP` and the wires `v1` and `v2`. This is in contrast to the wrapper model where `IP`, `v1` and `v2` are moved up by one level in the structural hierarchy of the system.

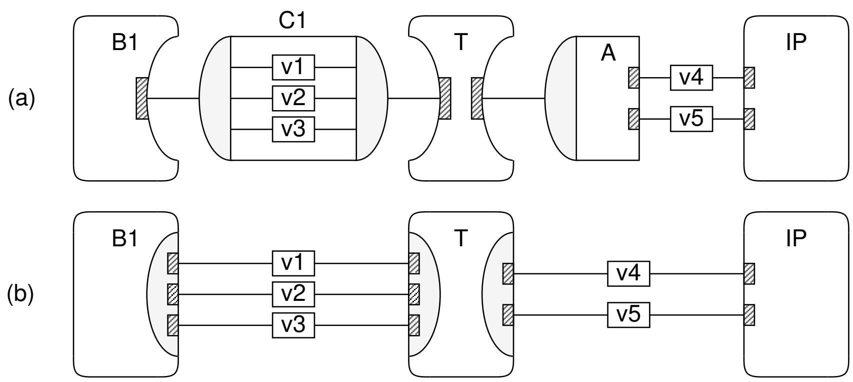

Figure 2.13: Inlining with transducer: (a) before, (b) after.

Finally, Figure 2.13 illustrates the need for transducers. For example, a processor core, represented by the behavior `IP`, needs to be interfaced with the system bus, represented by channel `C1`. Because the communication protocol used by the system bus `C1` is incompatible with the native processor bus, represented by `v4` and `v5`, a transducer `T` is necessary. After the inlining of the channel `C1` and the adapter `A`, the transducer `T` has incorporated both bus protocols and therefore can translate between the system bus and the processor.

Chapter 3

The SpecC Design Methodology

In the previous chapter, the basic SpecC models consisting of behaviors, channels and interfaces, have been introduced. In this chapter, these models are used as building blocks to form and define the models on which the SpecC design methodology is based.

As described in the introduction, a design methodology is a specific design flow that, with the help of CAD tools, transforms an initial, functional specification of the intended design into a detailed, structured implementation. In other words, a methodology consists of a set of model transformations that step-wise refine an abstract specification model of the design into an implementation model ready for manufacturing.

The SpecC design methodology is based on four well-defined models, namely a specification model, an architecture model, a communication model, and finally an implementation model. These models, and the tasks performed with these models, are described in detail in the following sections, starting with an overview.

Please note that the SpecC design methodology presented in this chapter is a refinement of the generic codesign methodology described in [DGZ98, GAC$^+$98, GZD97b, GZD97c]. In contrast, the models and tasks defined in this chapter are of much finer detail and reflect the actual status of the SpecC design environment.

3.1 Overview

An overview of the SpecC design methodology is shown in Figure 3.1 as a directed flow graph. The graph contains two types of nodes, namely *tasks*, indicated as rectangular boxes, and *models*, shown as ellipses. The models represent the input and output of the tasks, as indicated by the arcs in the graph.

The SpecC design methodology consists of a vertical synthesis flow, a horizontal

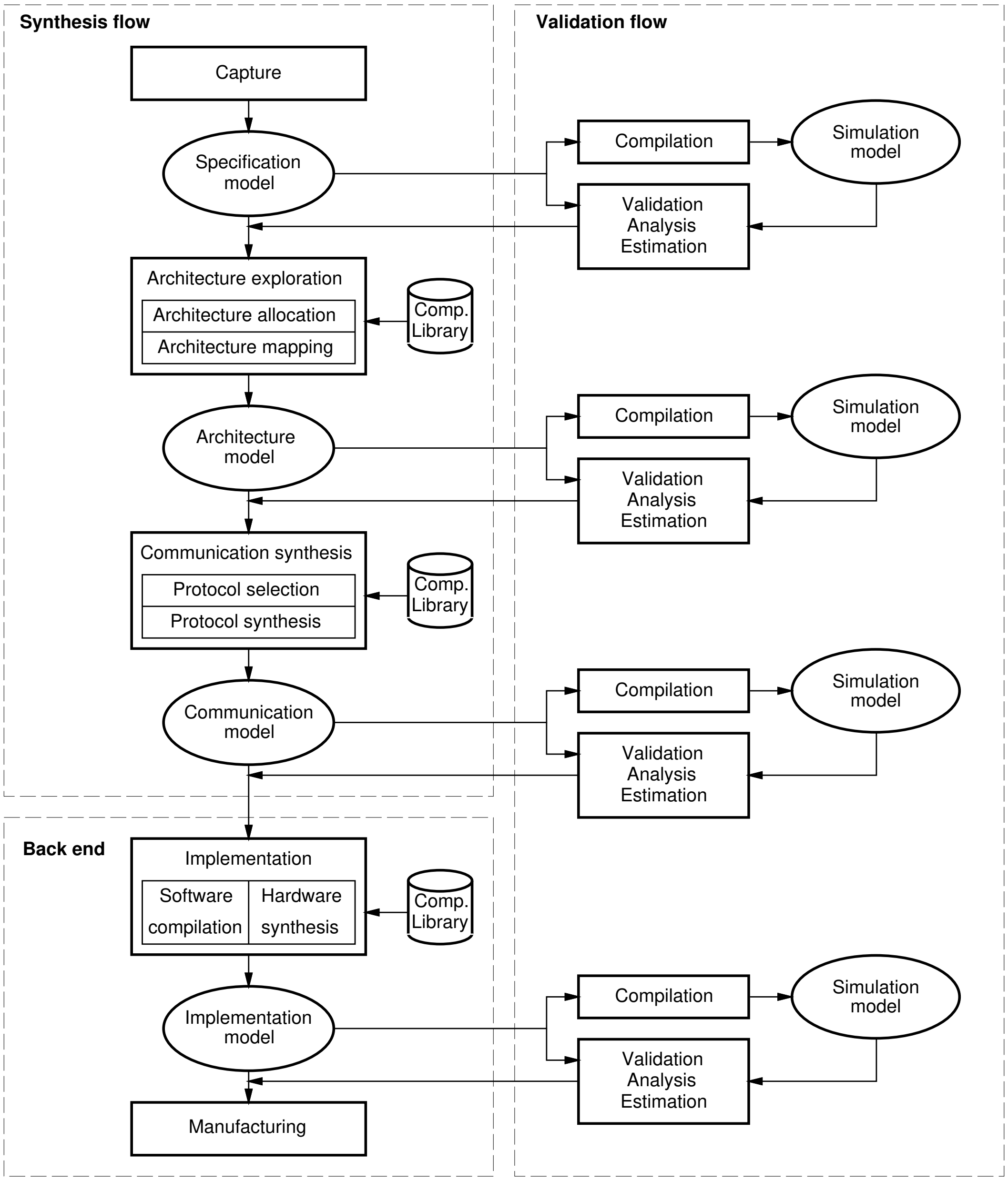

Figure 3.1: System design methodology with the SpecC design environment

validation flow, and a back end, as indicated by the dashed boxes in Figure 3.1.

The *synthesis flow* starts with the *capture* of the intended design, followed by a series of refinement steps. The initial *specification model* of the design is captured by use of a graphical or textual design entry. It consists of an abstract, executable description that includes the functionality and the constraints of the intended design.

The specification model is the input of the first refinement task, called *architecture exploration*. During architecture exploration, the target architecture of the system is determined in two major steps. First, a set of components, such as processors, ASICs, memories and busses, is allocated from the component library, forming the target architecture. Then, the specification model is mapped onto the selected architecture and a refined *architecture model* of the design is generated.

The architecture model is further refined by the task of *communication synthesis*. During communication synthesis, communication protocols are selected, inserted and refined for each bus in the system. Also, interface components will be inserted and realized in the system, if necessary. The result of communication synthesis is output as a *communication model*, which is passed on to the back end.

It is the task of the *back end*, to actually implement each component in the system. For software, binary program code has to be compiled for each processor, using a compiler for the particular instruction set. For hardware, a control unit and a datapath need to be synthesized for each ASIC, by use of behavioral synthesis, for example.

After software compilation and hardware synthesis, an *implementation model* is generated, representing a clock-cycle accurate description of the system. This description, in turn, is used by the final task of *manufacturing*.

Note that the abstraction level of the design model decreases with each refinement step in the synthesis flow. In other words, the design decisions made by each task are reflected in the generated models, making them a more and more accurate description of the final design.

The *validation flow* is organized orthogonally to the synthesis flow. For each of the four design models, *validation*, *analysis* and *estimation* can be performed statically on the model itself. Furthermore, for each design model, a corresponding *simulation model* can be generated by *compilation*, in order to perform dynamic validation. The generated simulation model is a program that can be run on the host computer, simulating the execution of the corresponding model.

The validation flow serves several purposes. First, each design model can be validated for correctness. This includes the correctness of the functionality, as well as the correctness of the performance, the timing, etc., if this is applicable to the model. Second, important characteristics and properties of the model can be obtained, verified, and also be reported to the designer. Furthermore, these results can be fed back into the synthesis flow, supplying data for further design decisions.

Note that the tasks performed in the validation flow are identical for the models at the four different abstraction levels, and therefore can be implemented by the same set of tools.

For the SpecC design methodology, two important features should be emphasized.

First, the SpecC methodology is homogeneous. All design models in the methodology are composed of the basic SpecC models introduced in Chapter 2. Moreover, all these models are represented by use of the same formal language, called SpecC, which will be described in detail in Chapter 4.

This is beneficial in several ways. Not only does this approach avoid cumbersome and error prone translations between languages with different semantics, it also yields a minimal number of design representations which use the same semantics and therefore can be easily compared and verified. Also, this allows for a minimal number of tools which need to be developed and maintained, and these tools can even share the same internal design representation and most data structures. Last, but not least, it makes the use easier for designers, since they only have to learn and deal with one language and one set of models.

Second, is should be pointed out that the design flow in the SpecC methodology only contains small loops, locally within the refinement tasks. This avoids large design iterations which are expensive in terms of both, design time and money.

In the following sections, the SpecC design methodology is described in detail. For each task, the input and output models with their particular characteristics and properties are defined, as well as possible intermediate models. In particular, the four main models are defined, namely the specification model, the architecture model, the communication model, and finally the implementation model.

Please note that, in the following, the *tasks* of the SpecC methodology are specified. The *algorithms* for these tasks, however, are beyond the scope of this work. In other words, it is described *what* the tasks do, not *how* they do it.

3.2 Specification Capture

The synthesis flow of the SpecC methodology begins with the capture of the design specification. The specification is usually captured textually by use of a standard text editor. Alternatively, a graphical design entry tool, such as VisualSpec [AIG99], can be used which allows to enter the specification in form of graphical diagrams and flow charts.

In both cases, the system specification is eventually represented formally by use of the SpecC language. The SpecC language has been specifically developed to represent the design models introduced in Chapter 2 and is described in detail in the next chapter.

The functionality of the intended system is captured in form of an *executable specification*. Thus, the specification model can be easily simulated on a host computer in order to verify that the system and its algorithms work as expected.

Along with the functionality, given design constraints are specified as well. Typical constraints include the required performance, maximal power consumption, maximal manufacturing cost, etc. These constraints are specified in form of annotations to the design description.

It should be emphasized that the specification should be as abstract as possible. Except for the given constraints, it should not include any details which restrict the implementation in any way. This will enable a large design space, leading to a better implementation.

3.2.1 The specification model

In the SpecC methodology, the specification model is the model with the highest level of abstraction. It is an accurate model of the intended system in terms of pure functionality, but does not reflect its structure or its timing.

Typically, the specification model executes in zero simulation time. Neither the computation, nor any communication, is modeled with timing. In other words, there is no `waitfor` statement in the SpecC description of the specification[1].

Communication can be modeled in two ways, either as shared variables, or by use of channels from the SpecC communication library. For a specification model, useful communication channels are channels with basic synchronization, such as one-way or two-way hand shaking, and buffered channels, such as blocking and non-blocking FIFOs. Note that with both types of communication, complex data types may be used for the exchanged data.

The specification model can be freely composed out of any of the basic SpecC models discussed in Chapter 2. A typical specification model is shown in Figure 3.2.

The specification model `Sa` consists of an arbitrary, hierarchical network of behavior and channel models. It includes sequential behaviors (`s1`, `f1`, `f2`), concurrent behaviors (`c1`, `c2`, `p1`), exception behaviors (`e1`), and program code in leaf behaviors (`l1`). Communication is performed via shared variables (`v1`, `v2`, ..., `v13`) or basic channels (`ch1`, `ch2`, ..., `ch5`).

It should be emphasized that all "natural" features, that are inherent in a design, should be specified explicitly in order to obtain a well-written specification model. In particular, any potential concurrency should be expressed by use of concurrent behaviors, since it is difficult to extract such concurrency later, if it is not modeled explicitly.

[1]Please refer to Section 4.10.1 for a description of the SpecC `waitfor` statement.

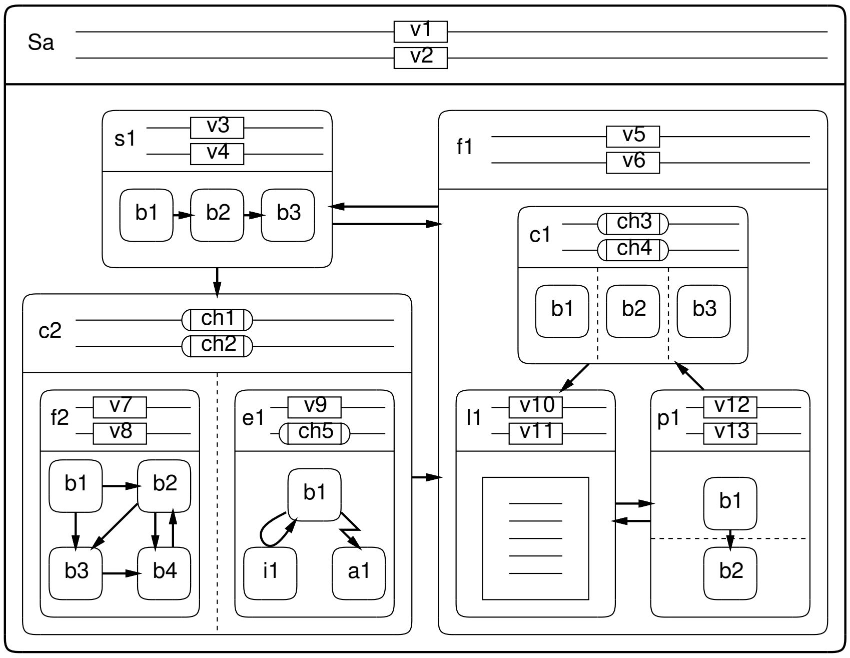

Figure 3.2: Specification model

3.3 Validation and Analysis

As shown earlier in Figure 3.1, validation and analysis are performed in the validation flow for each of the four design models. After the design has been captured, the specification model is validated for functional correctness in order to ensure that the captured model actually behaves as intended and the specified algorithms work correctly. The functionality of each following model is also checked and compared against the initial specification model.

For each refined model, the modified portions and the added features need to be verified as well. More specifically, for the architecture model, the new structural organization and the introduced synchronization between the concurrent components must be validated. For the communication model, the inserted communication protocols must be verified. Finally, the implementation model must be checked whether it actually meets the given design constraints, such as performance, size, etc.

It has been stated in the introduction that, in contrast to general validation, only the formal *verification* of a model guarantees its correctness for all cases. However, the true verification of a medium sized system model is, as of today, still too complex and cannot be performed in reasonable time. Because of this, the SpecC design methodology relies on validation rather than verification. In particular, *simulation* and *estimation* are performed with each design model.

3.3.1 Simulation

In system-level design, simulation is the most common form of design validation. In contrast to static analysis, simulation is dynamic and, thus, requires the design model to be executable.

In the SpecC methodology, simulation is performed in two steps. First, the design model is compiled into a corresponding *simulation model*. More specifically, the SpecC compiler takes the design model, together with a corresponding test bench model, and generates an executable program that is linked with the SpecC simulation library. The simulation library implements the semantics of the simulation. In particular, it maintains an event queue, advances the simulation time, and also takes care of concurrent execution and the synchronization facilities.

Then, the generated simulation model can be run on the host computer, simulating the execution of the corresponding model. Typically, the test bench included in the simulation model will supply the test vectors, automatically check the computed output values, and report any problems to the user.

If any problems occur, a debugger can be used to set break points, interrupt the simulation, and inspect intermediate values, in order to locate and fix the design errors in the model.

It should be noted that there is a trade-off between the time and the accuracy of the simulation. In other words, the length of the simulation time depends on the accuracy of the design model. For example, compared to the specification model, the communication model will need longer time for a simulation, because it performs any communication in a clock-cycle accurate manner. The implementation model will spend even more time for the same simulation, since communication and computation are both cycle accurate.

However, because of the "plug-and-play" capability of the SpecC models, it is easily possible to simulate a model at a mixed level of accuracy, saving simulation time. In particular, only the parts of the system, which need special attention, can be simulated accurately, whereas all other parts can be executed at the pure functional level. For example, in order to observe the detailed behavior of a particular bus transaction, the architecture model can be used where only the particular bus is replaced with the detailed communication model.

3.3.2 Estimation

The task of estimation is to obtain quality metrics from a design model. Although the obtained metrics should be accurate, the main emphasis of estimation is to deliver these values quickly.

In the SpecC methodology, estimated quality metrics are especially needed for the task of architecture exploration. In particular, the trade-off between a software or a hardware solution for each behavior in the design model requires metrics for performance and cost.

More specifically, the execution time and the area of each behavior is estimated for a potential hardware implementation. Also, the execution time, code size and data size will be determined for a potential implementation in software, for each allocated processor. In addition, metrics, such as bit width and throughput, need to be determined for all channel and bus models, since these are needed for the task of communication synthesis.

All these estimation results are annotated in the design model at the particular behaviors and channels. Thus, they are fed back into the synthesis flow so that this data is immediately available when it is needed by the synthesis algorithms.

Estimation is typically performed in form of static analysis of the design model. However, by use of *profiling*, estimation data can also be obtained dynamically during simulation. In the SpecC system, profiling can be used to count the execution frequency of each behavior. Based on these counter values, branching probabilities can be determined, for example, for the conditional transitions in FSM behaviors. These branching probabilities are then used to estimate the average execution time for such behaviors.

3.4 Architecture Exploration

The first major refinement step in the synthesis flow of the SpecC methodology is the task of *architecture exploration*, which includes the traditional design steps of component allocation, hardware/software partitioning and scheduling. More specifically, architecture exploration consists of architecture allocation and architecture mapping, as shown in Figure 3.1 at the beginning of this chapter.

Architecture allocation determines the connectivity and the number and the types of the system components, such as processors, ASICs, memories and busses, which will be used to implement the specified system. Note that this also includes the reuse of intellectual property (IP), when IP components are selected from the component library.

Then, *architecture mapping* is performed for all behaviors, channels and variables in the specification, assigning them to processing elements (PEs), busses and memories, respectively. *Behavior mapping* distributes the behaviors to the allocated PEs. *Variable mapping* assigns variables, which cannot be stored locally in the PEs, to the allocated memories. Finally, *channel mapping* assigns the non-local communication channels to the allocated busses. In addition, *scheduling* is performed to determine the execution order of the behaviors assigned to sequential processors.

Although architecture exploration is described in the following as a set of tasks which are only once and sequentially executed, it is free to be implemented as an iterative process whose final result is the definition of the system architecture. In each iteration, estimation is 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 is evaluated, with different components, connectivity, partitions, or communication.

Such an iterative approach is called *design space exploration*. It will eventually result in a better system architecture and an optimized design implementation with good performance and less cost.

3.4.1 Architecture allocation

Given a library of system components, such as processors, memories and busses, the task of architecture allocation is defined as the selection of the type and number of these components. The interconnection among the selected components must also be determined. Further, the system architecture has to be defined in a way so that the functionality of the system can be implemented, all design constraints are satisfied, and the objective cost function is minimized.

During architecture allocation in the SpecC methodology, three types of components are selected from the component library. First, processing elements (PEs), including standard processors and custom ASICs, are needed as active elements performing the systems functions. Second, memories are needed to store the processing data, and finally, busses are allocated for the communication among the PEs and memories. Note that for each component type, either a synthesizable, custom component can be selected, or a predesigned component, such as an IP.

The network of selected components is called the *target architecture* of the system. In the SpecC methodology, the target architecture is defined by customization of a generic architecture. In other words, parameters are defined for the generic architecture, so that it becomes a specific target architecture for the system.

The generic system architecture is shown in Figure 3.3. The architecture consists of a set of system ports, a set of system busses, a set of system components, and a connectivity matrix which determines the interconnections among the ports, busses, and components.

In order to define a specific target architecture, all parameters have to be fixed.

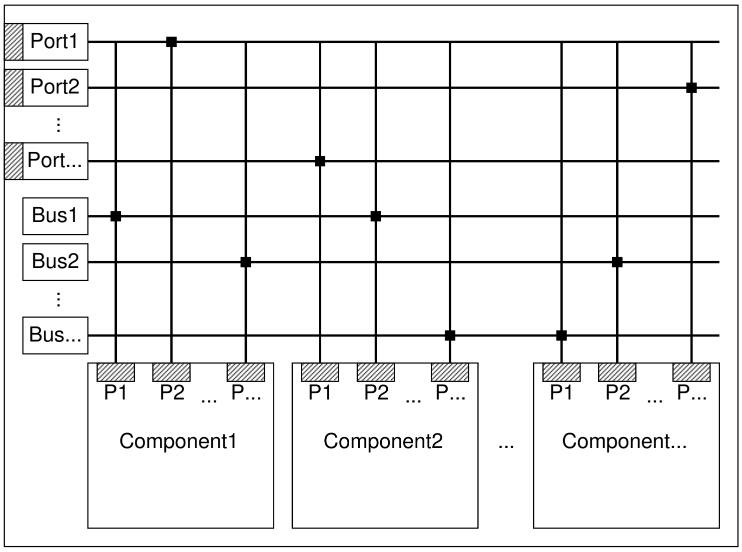

Figure 3.3: Generic system architecture

For each set, the number of elements and the type of each element must be defined. Then, the connectivity matrix is filled, determining whether a connection exists between each component and each bus or port. Note that a connection can only be set, if the connected elements are of compatible type.

Figure 3.4 shows a typical target architecture created as a result of this customization. The architecture consists of two processors, `P1` and `P2`, one ASIC `A1` and four memories, `M1`, `M2`, `M3` and `M4`. Further, an input/output unit `IO1` and three bus interfaces, `I1`, `I2` and `I3`, have been allocated. Note that, because of the selected connectivity, both processors, `P1` and `P2`, and the ASIC `A1`, each have a dedicated local memory, whereas `M3` serves as a global memory for storage of shared data.

3.4.2 Architecture mapping

After the target architecture has been defined, the specification model needs to be mapped onto the architecture. This mapping process is often referred to as *partitioning*[2]. However, because the term partitioning typically is used to describe the assignment of parts from the system model to either hardware or software in general, and not to a particular processing element, the term *mapping* is preferred in this work.

[2]Further, other common terms for the mapping process also include binding, grouping and assignment.

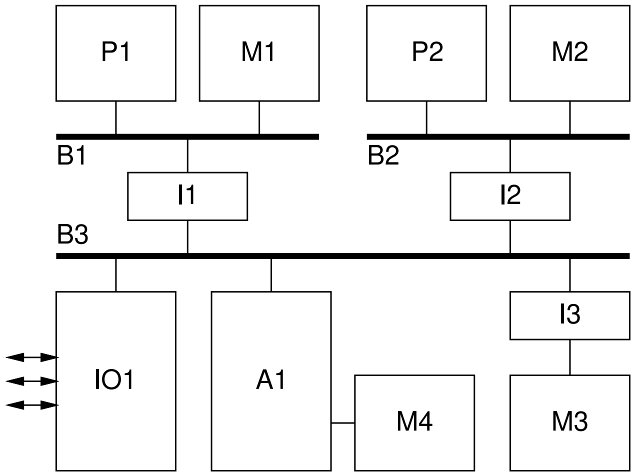

Figure 3.4: Example of a system architecture

Architecture mapping in the SpecC design methodology consists of behavior mapping, variable mapping, and channel mapping. In addition, scheduling is included as well. Note that, technically, these subtasks can be executed in any order, or even simultaneously. For simplicity, however, they are described sequentially in the following sections, starting with behavior mapping.

Please note also, that the creation of the mapping itself is beyond the scope of this chapter. It is assumed that the mapping has been determined by some optimizing algorithm[3]. Rather, it is described how the mapping is applied to the design model in order to reflect the design decision.

3.4.2.1 Behavior mapping

The task of behavior mapping assigns each behavior in the specification model to one of the allocated processing elements and updates the design model according to this decision. Note that behavior mapping includes the core task of codesign, the hardware/software partitioning of the system.

The design model after behavior mapping differs from the specification model in the way that an additional level of hierarchy has been introduced. At the top-level of the structural hierarchy, behaviors are inserted that represent the allocated PEs. In each PE behavior, only the behaviors from the specification model, that have been mapped to the particular PE, are included. Behaviors, which have been assigned to a different PE, are replaced with control behaviors that are used to synchronize the

[3]For information on such algorithms, please refer to [Wol97] or [YW97], for example.

execution of such behaviors.

Note that the inserted PE behaviors simply group the behaviors for each PE together. The correlation of PE behaviors with the allocated components in the library is established as an annotation of the library and component name at the PE behavior.

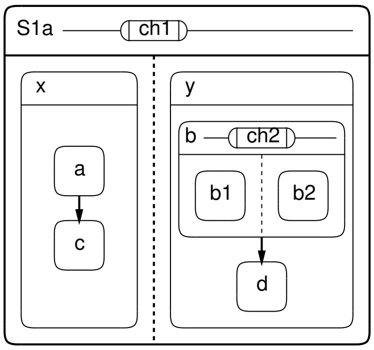

Figure 3.5: Design example `S1` before behavior mapping

The process of behavior mapping is illustrated with the design example `S1` shown in Figure 3.5. The design is specified as two concurrent behaviors `x` and `y`, communicating via channel `ch1`. The behavior `x` consists of two sequential child behaviors `a` and `c`, whereas `y` contains two children `b` and `d`. The behavior `b`, in turn, is composed of the parallel behaviors `b1` and `b2` which can communicate via the channel `ch2`.

For the example, two processing elements, `PE1` and `PE2`, have been allocated. Furthermore, it is assumed that all behaviors are to be executed by `PE1`, except for `c` and `b2` which are assigned to `PE2`.

Given these assumptions, Figure 3.6 shows one possible design generated as a result after behavior mapping. The two allocated processing elements `PE1` and `PE2` have been introduced as top-level, concurrent behaviors reflecting the two components of the selected system architecture.

Since most of the behaviors were assigned to `PE1`, its structural composition is almost the same as the initial design. Only the behaviors `c` and `b2` have been replaced with `c_ctrl` and `b2_ctrl`, respectively. These controller behaviors consist of a start and a wait behavior, e. g. `c_s` and `c_w`, which serve to synchronize `PE1` with `PE2`. `PE1` can be seen as a client which sends a start signal to `PE2` and then waits for the behavior `c` to be completed. The server `PE2` waits in a ready state `c_r` for commands from `PE1`, and sends a done message back in `c_d` once the behavior `c` has been executed.

Note that two new channels, `c_syn` and `b2_syn`, have been introduced for the

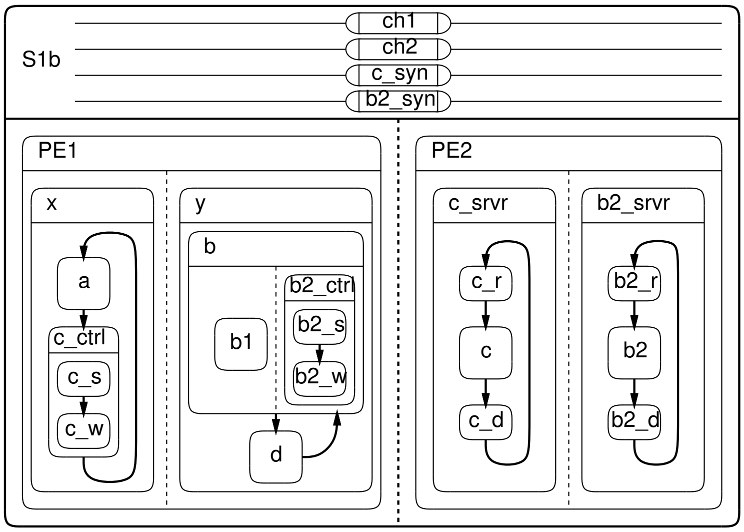

Figure 3.6: Design example `S1` after behavior mapping

synchronization between `PE1` and `PE2` regarding the behaviors `c` and `b2`. Furthermore, the channel `ch2` has been moved up to the top-level of the hierarchy so that the behaviors `b1` and `b2` can still communicate.

Please note also that, after the behavior mapping has been performed, infinite loops have been introduced for the behaviors `x`, `y`, `c_srvr` and `b2_srvr` in Figure 3.6. This reflects the fact that processing elements, such as processors and ASICs, never terminate.

So far in the design process, the behaviors in the design specification have been grouped into the allocated PEs according to the selected mapping. However, the behaviors assigned to sequential executing PEs, such as processors, still need to be serialized. This is the task of scheduling which is described next.

3.4.2.2 Scheduling

The assignment of concurrent behaviors to a sequential PE, for example a processor, requires *scheduling* to be performed. The task of *scheduling* determines the order of the execution for these behaviors. Hereby, the scheduler ensures that the selected order does not violate any dependencies or timing constraints imposed by the specification model, while optimizing the execution time and other objectives specified by the designer.

As mentioned in the introduction, scheduling can be performed either statically or dynamically. With a static scheduler, the schedule is determined beforehand and the behaviors will be executed in a fixed order. On the other hand, a dynamic scheduler, determines the execution order at run-time. Typically, this is implemented by use of a real-time operating system (RTOS). In the SpecC methodology, however, a static scheduler is used [CG99].

After a satisfactory schedule is determined, the design model is refined so that it reflects the sequential execution of the behaviors in the sequential PEs. Note that the design model is only changed inside the scheduled PEs. Everything else is left unchanged.

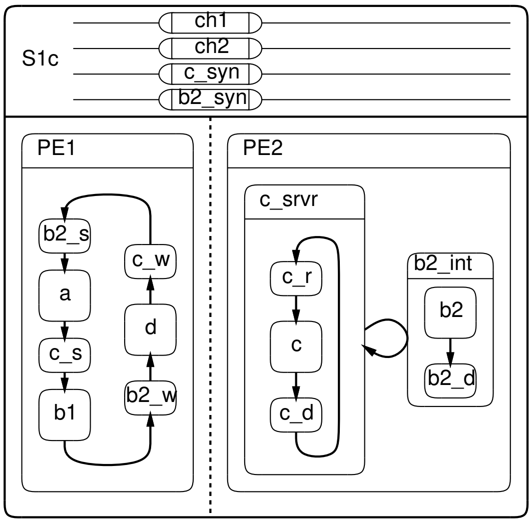

Figure 3.7: Design example `S1` after scheduling

The process of scheduling can be demonstrated continuing the design example `S1` from Figure 3.6, assuming that both, `PE1` and `PE2`, are sequential components. A scheduled model of this example is shown in Figure 3.7. Note that the top-level structure of the design has not changed. Only the internal structures of `PE1` and `PE2` have been modified so that there is no concurrency left.

In `b2_s`, `PE1` first sends a start signal to `PE2` in order to initiate the execution of `b2`, and then executes behavior `a`. After that, `c` is given a start signal in `c_s` and `b1` is executed. Before `PE1` can continue with behavior `d`, it has to wait in `b2_w` for `b2` to finish, because performing `d` in parallel would violate the execution order specified in Figure 3.6. Finally, `PE1` waits for the completion of `c` and then repeats the whole sequence.

In contrast to `PE1`, which executes in a single loop, a solution with use of an interrupt model has been selected for `PE2`. The main execution of `PE2` consists of the behavior `c_srvr` including `c` and its synchronization points `c_r` and `c_d`.

However, whenever PE2 receives a signal to start b2, the interrupt handler b2_int is called which will execute b2 immediately. Once b2 is finished, PE1 is notified in b2_d and the execution of c_srvr can continue. Please note that the behavior b2_r from Figure 3.6 has been replaced by this interrupt model.

It should be emphasized that the schedule found for this example takes advantage of scheduling both PEs simultaneously. In other words, a global scheduling approach for the whole design is used, as opposed to two local schedulers working independently in PE1 and PE2.

3.4.2.3 Variable mapping

Variables used in the system specification need to be assigned to memories. Such memories are either standard memory components allocated in the target architecture, or local memories within the PEs. However, local memory space in PEs is usually quite limited. ASICs can store only a small set of variables in register files, and processor cores typically contain only very small built-in memories.

For variables mapped to memories, communication functions, such as Read and Write, need to be used by the PEs in order to access these variables. The same applies when a PE needs to read or write a variable stored within another PE.

In the SpecC design model, such variable access functions are represented explicitly by so-called *variable channels*, which are introduced and maintained automatically. These variable channels encapsulate the necessary functions which communicate with the memory component that actually contains those variables.

Later in the design flow, the variable channels will be grouped into virtual busses which, in turn, will then be refined into the allocated system busses.

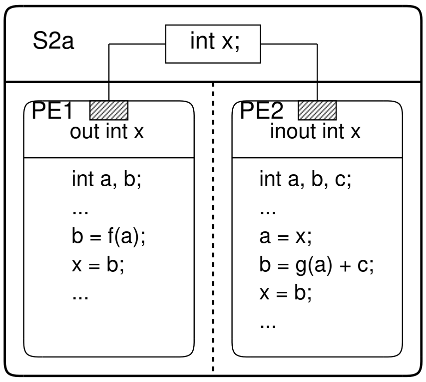

Figure 3.8: Design example S2, initial specification

The refinement step of variable mapping is illustrated with the simple design example `S2` shown in Figure 3.8. The design consists of two behaviors, `PE1` and `PE2`, which initially communicate via a shared integer variable `x`. More specifically, `PE1` writes the result of a function `f` through its output port into the connected variable `x`. For simplicity, the output port is named `x` as well[4]. On the other hand, `PE2` reads the shared variable `x` through its port, computes a function `g` with the value, and writes the result back into `x`. Note that the port of `PE2`, which again is named `x`, is bidirectional, allowing both read and write access.

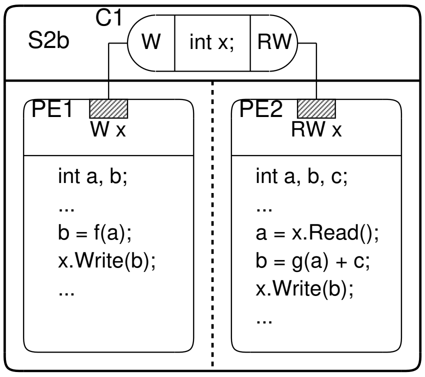

Figure 3.9: Design example `S2` before variable mapping

In a preprocessing step, the shared variable `x` is first encapsulated in a variable channel `C1`, as shown in Figure 3.9. The channel `C1` provides a left interface `W` for write access, and a right interface `RW` for bidirectional access to `x`. These interfaces are used as the new port types for `PE1` and `PE2`. Furthermore, in order to access `x` in the channel `C1`, the direct assignments to the ports are replaced with the function calls `Read` and `Write` provided by `C1`.

As a result, the design has been transformed so that all shared variables are replaced with channels and all communication is performed by explicit `Read` and `Write` function calls. This preprocessing step prepares the next step and is also needed for communication synthesis performed later in the design process.

Under the assumption that the variable `x` has been assigned to a memory `M1`, the design model can be further refined, as shown in Figure 3.10. The memory `M1` is placed into the design as a new behavior in parallel to `PE1` and `PE2`, and the former shared variable `x` is declared as a local variable in `M1`.

The functionality of the memory `M1` can be specified as an infinite loop that serves incoming requests for reading and writing to the storage `x`. In Figure 3.10, the function `Cmd` is used to determine the type of the request. For a read request `R`,

[4]In the SpecC language, the scope of a port name is limited to the behavior body. Thus, there is no naming conflict between the port `x` and the external variable `x`.

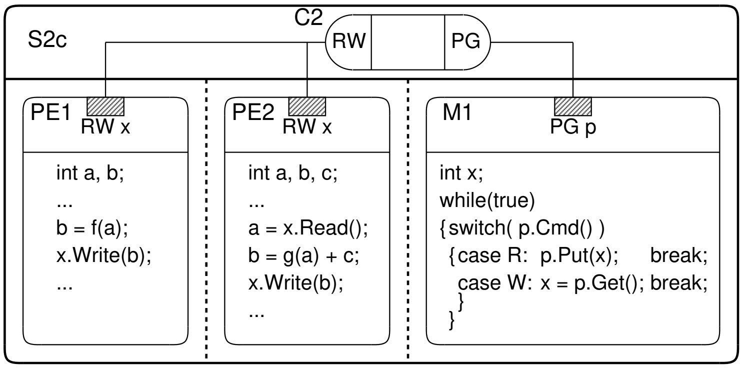

Figure 3.10: Design example `S2` after variable mapping

the value of `x` is output by use of the function `Put`. For a write request `W`, `x` is set to a new value obtained with `Get`.

The functions `Cmd`, `Put` and `Get` are provided by a new channel `C2` which replaces the former channel `C1`. `C2` contains two interfaces. The interface `PG` connects to the memory `M1`, and the interface `RW` connects to the behaviors `PE1` and `PE2`. Note that `PE1` and `PE2` need not to be changed, because the interface `RW` is the same as before[5].

3.4.2.4 Channel mapping

After behavior mapping and variable mapping have been performed, the design model consists of a set of PE and memory behaviors connected by a typically large set of variable channels. In particular, there is one channel for every variable in the design that is transferred between any of the PE and memory components.

In order to obtain the architecture model, the variable channels need to be mapped onto the allocated busses in the target architecture. More specifically, the variable channels are combined by use of grouping channels, as defined in Section 2.4.2. Each grouping channel is called a *virtual bus*, representing a particular bus in the system architecture.

Later, during communication synthesis, these virtual busses will be replaced with cycle-accurate models of the allocated busses. For the architecture model, however, the virtual busses are only annotated with the real bus name.

The refinement step of channel mapping is illustrated with the design example `S3` shown in Figure 3.11. The design consists of two processing elements, `PE1` and `PE2`, and two memories, `M1` and `M2`. For simplicity, only three variables, `a`, `b` and

[5]For space reasons, the interface `W` is ignored and `RW` is connected to both behaviors `PE1` and `PE2`.

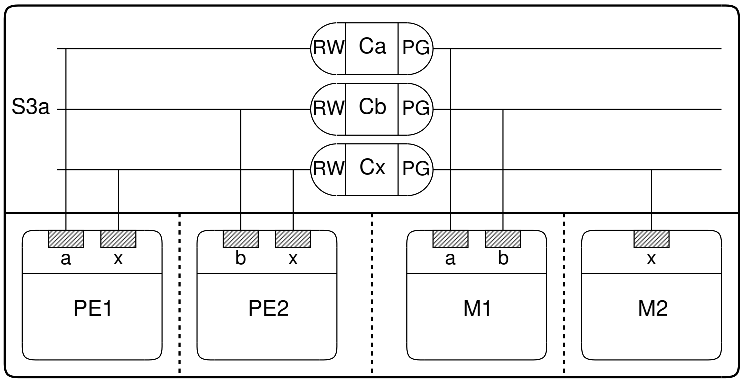

Figure 3.11: Design example `S3` before channel mapping

`x`, are used in the design. Variables `a` and `b` are stored in memory `M1`, whereas `x` is stored in `M2`. For each variable, there exists a corresponding channel that contains the required access functions `Read`, `Write`, `Put` and `Get`, as discussed earlier with Figure 3.10. `PE1` can access the variables `a` and `x` by use of the channels `Ca` and `Cx`, whereas `PE2` has access to `b` and `x` via channels `Cb` and `Cx`, respectively.

Since this is a small design, one single bus is sufficient to connect all four components. In other words, it is assumed that all three channels, `Ca`, `Cb` and `Cx`, are to be mapped onto the same bus.

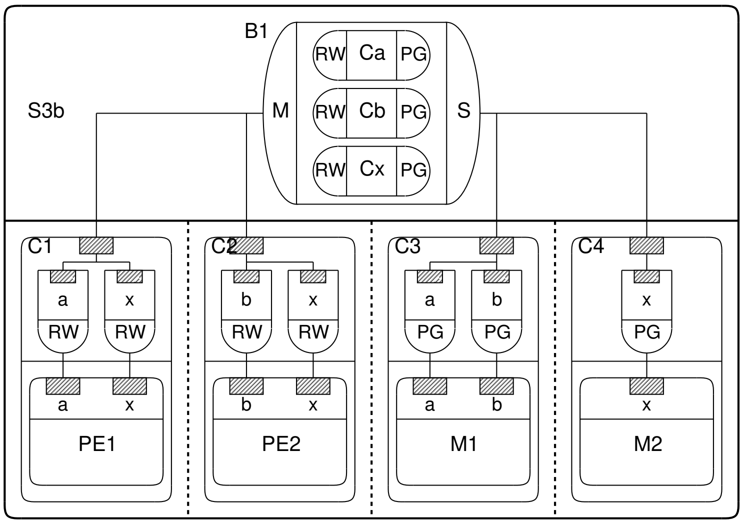

Figure 3.12: Design example `S3` after channel mapping

The result of channel mapping for this example is shown in Figure 3.12. The allocated bus `B1`, represented as a grouping channel, has been inserted into the design, containing the channels `Ca`, `Cb` and `Cx`. The bus `B1` provides two interfaces, a master interface `M` for use by `PE1` and `PE2`, and a slave interface `S` for the memories `M1` and `M2`.

The bus `B1` introduces a new communication layer that references each variable in the design by a unique `ID`. More specifically, the master interface `M` provides `Read` and `Write` functions similar to the `RW` interfaces of the internal channels. However, these functions take a variable `ID` as an additional argument in order to identify which one of the internal channels is to be used. For example, the function call `B1.Read(IDx)` will in turn call `Cx.Read()`, and the call `B1.Write(42,IDa)` will in turn call `Ca.Write(42)`. The same scheme is used for the memory interface `S` with the functions `Put` and `Get`.

The added communication layer is also reflected by the newly introduced component models `C1`, `C2`, `C3` and `C4`, which encapsulate `PE1`, `PE2`, `M1` and `M2`, respectively. Furthermore, in order to compensate the change in the communication protocol, adapter channels have been inserted for each component port. These adapters essentially provide the reverse functionality of the channel `B1`. In other words, the adapters will supply the required `ID` to each function call. For example, an adapter `a` will convert the function call `Read()` into `Read(IDa)` and the call `Write(27)` into `Write(27,IDa)`.

Please note that the level of hierarchy added to the design model due to the bus grouping channels, component behaviors and adapters, does not imply any decrease in performance of the final system. The process of inlining will eliminate the structural overhead.

Please note also that the design model obtained after channel mapping has been performed, accurately reflects the system architecture. Each component and each bus in the real system is represented by a corresponding top-level behavior or top-level channel in the design model.

3.4.3 The architecture model

After behavior, variable, and channel mapping have been performed, the task of architecture exploration is complete. As a result, the initial specification model of the design has been refined into the architecture model.

The architecture model is an abstract model of the system under design, that accurately reflects the functionality and the overall structure of the final implementation. However, the model is not accurate yet in terms of timing and communication.

Communication is performed by use of channels representing virtual busses. As such, communication still uses the original, possibly complex data types and takes zero time.

For the computation parts, execution times have been estimated for all behaviors in the PEs. Assuming that the estimated execution times for the leaf behaviors have been inserted into their code in form of `waitfor`[6] statements, the architecture model will reflect these timing delays in the simulation when it is executed.

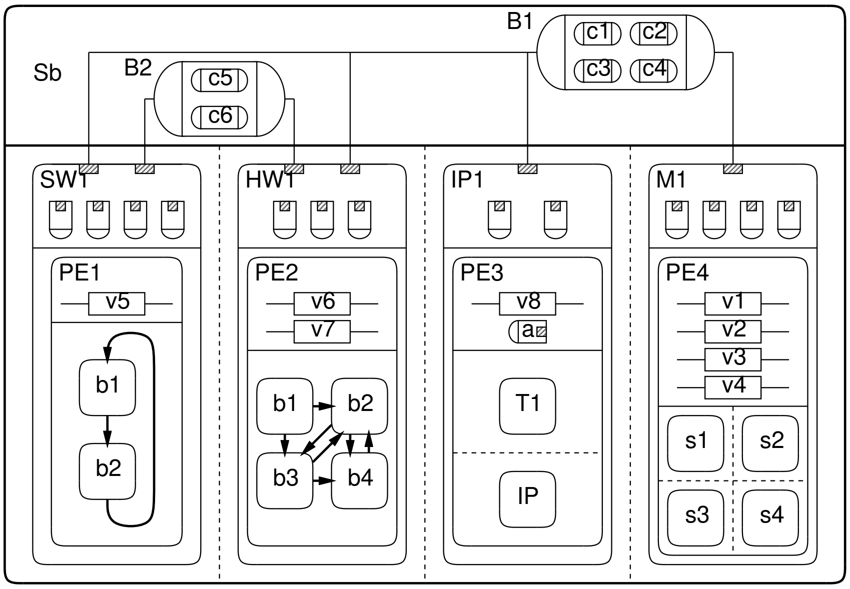

Figure 3.13: Architecture model

A typical architecture model is shown in Figure 3.13. The architecture model `Sb` consists of four components, namely a processor `SW1`, an ASIC `HW1`, an IP core `IP1`, and a memory `M1`. Internally, each of these components consists of a set of adapters for the added communication layer discussed with Figure 3.12, and a PE behavior which, in turn, contains a set of behaviors describing the functionality of the particular component.

Communication is performed via the virtual busses `B1` and `B2`. `B1` serves as a system bus, connecting all four components. On the other hand, `B2` is a local bus between the processor `SW1` and the ASIC `HW1`.

Note that, in contrast to the specification model shown earlier in Figure 3.2, the architecture model clearly reflects the structure of the target architecture.

[6]Please refer to Section 4.10.1 for a description of the SpecC `waitfor` statement.

3.5 Communication Synthesis

In the SpecC methodology, it is the task of communication synthesis[7] to further refine the generated architecture model into the communication model. The communication model will accurately reflect the detailed communication between the components in the design, including cycle accurate timing. Thus, the purpose of communication synthesis is to resolve the abstract communication in the architecture model into an implementation.

During communication synthesis, the virtual communication protocol used in the architecture model is replaced with real communication protocols implemented on the system busses. In other words, the virtual busses in the architecture model are replaced with the actual busses selected during architecture allocation. On top of the native bus protocols, an application layer communication protocol is selected and inserted in the design model. For incompatible bus protocols, transducers are further inserted into the system model which bridge the gap between the protocols by translating the transactions between those busses. Finally, the communication protocols are implemented in the PEs by use of inlining.

Communication synthesis includes the interfacing of hardware and software components. For synthesizable hardware components, the ports of the components can be easily adapted to different busses. This, however, is not true for software components, because processor ports are fixed. In order for software to communicate with connected hardware, processor specific *device drivers* are needed. Since the implementation of device drivers is a special problem in communication synthesis, it is ignored in this section. Two case studies with the SpecC methodology, which involve the communication between a processor and an ASIC, can be found in [GZG+99] and [KG98].

In the SpecC methodology, communication synthesis is separated in three tasks, namely protocol selection, transducer insertion and protocol synthesis. These are described next.

3.5.1 Protocol selection

Communication synthesis deals with communication protocols which, in general, are organized in several *layers*. A communication protocol stack typically starts at the lowest level with the physical layer and extends over several intermediate layers up to the application layer at the highest level.

In the SpecC methodology, two communication layers are distinguished. The low-level layer, called the *bus layer*, is dependent on a particular bus. It contains the native communication functions provided by the bus. The bus layer is stored

[7] In the literature, communication synthesis is sometimes referred to as *interface synthesis*.

in form of a channel in the bus library and is selected for the design as part of the architecture allocation.

On the other hand, the high-level layer, called the *application layer*, is independent from the allocated busses. Rather, it consists of an application specific communication protocol, built on top of the bus layer.

As the first step of communication synthesis, it is the task of protocol selection to select and customize the application layer for the particular design.

The application layer essentially provides two necessary services which enable the PEs in the design to exchange data of any data type, including user-defined records and multi-dimensional arrays.

The first service, called *sizing*, converts the data types used in the application into blocks that can be transported via the busses. For example, assuming that a native bus protocol only supports the transfer of single bytes and small blocks of 256 bytes, an array of 1024 integers could be transferred as a sequence of 16 blocks[8].

The second service, called *addressing*, basically replaces the `ID` mechanism discussed with Figure 3.12. In order to identify particular variables during the communication and in the memories, unique addresses are assigned to each of them. Each variable is then referenced by its address, identifying a particular PE and the location in the PE.

In the design model, the application layer is represented by a hierarchical channel that encapsulates a low-level bus channel.

3.5.2 Transducer insertion

After the communication protocols have been determined for each bus in the design, it is possible that the selected protocols conflict with the built-in protocols of some components. In particular, this situation occurs often times with hard IP components, processors and memories, when these are connected to the system bus.

In case of a protocol mismatch, a transducer needs to be inserted. As discussed in Chapter 2, the transducer then acts as a translator for the two protocols.

Please note that the creation and insertion of a transducer can be easily automated, because of the "plug-and-play" feature of the SpecC model.

The refinement step of transducer insertion is illustrated with the design example `S4` shown in Figure 3.14. The design consists of a behavior `PE1` and a memory `M1`, connected by a virtual bus `VB1`, as discussed earlier with Figure 3.12. For simplicity, the adapters known from Figure 3.12 have been left out in Figure 3.14. Instead, the code shown in the behaviors uses the virtual bus protocol provided by the interfaces

[8]This assumes that the size of integer is 4 bytes.

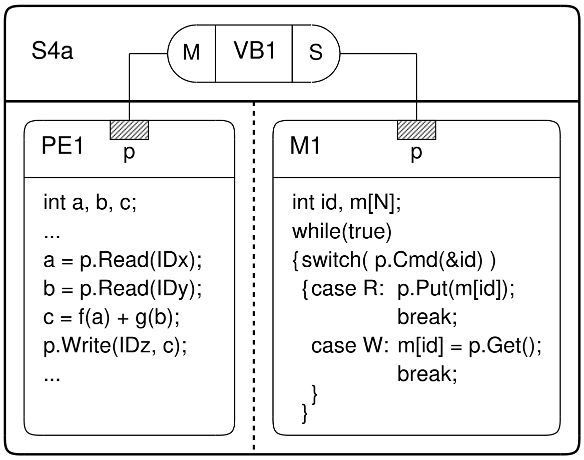

Figure 3.14: Design example `S4` before communication synthesis

`M` and `S` directly[9].

Assuming that architecture exploration has already been performed for the design, `PE1` has been assigned to a synthesizable ASIC and the memory `M1` has been allocated as a standard memory core. Also, the virtual bus `VB1` is to be implemented as a particular system bus. Further, it is assumed that the native bus of the selected memory `M1` differs from the allocated system bus. Thus, a transducer is required to translate the transactions on the system bus into requests on the memory bus, and vice versa.

Figure 3.15 shows the design `S4` after the required transducer has been introduced. The transducer `T1` has been inserted as a new, synthesizable component, running concurrently with `PE1` and `M1`. The virtual bus `VB1` has been reconnected and another virtual bus `VB2` has been inserted, so that any communication between `PE1` and `M1` is performed through the transducer `T1`.

Please note that `PE1` and `M1` have not been modified during the transducer insertion and, up to this point, all three components still communicate via the virtual bus protocol. The real protocol for the selected system bus and the selected memory will be inserted next during the task of protocol synthesis.

3.5.3 Protocol synthesis

After the transducers have been inserted, the virtual communication protocol used so far in the design model can finally be replaced with the actual bus protocols

[9]The code shown in the behaviors can actually be obtained by inlining of the adapters in Figure 3.12.

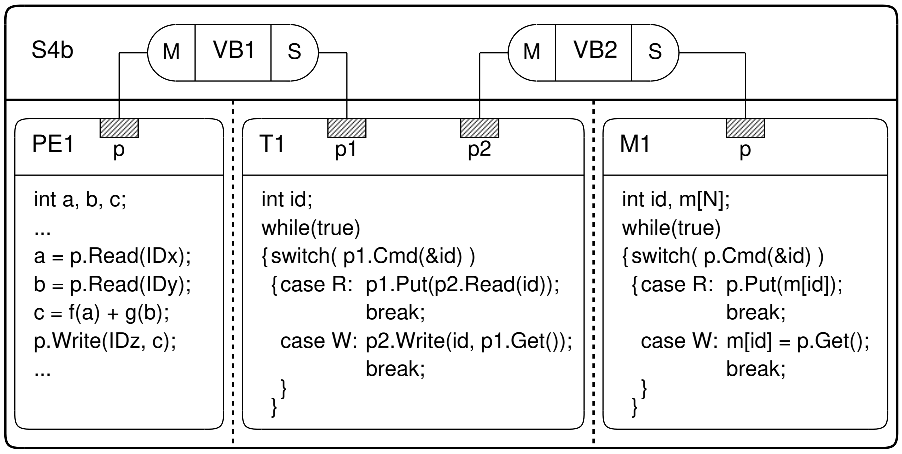

Figure 3.15: Design example `S4` after transducer insertion

embedded in the added application layer.

In the design model, this change is just a matter of "plug-and-play". For each bus, the grouping channel, that represents the virtual bus, is replaced with the hierarchical channel that contains the application layer with the encapsulated real bus channel. At the same time, the adapter channels, that were used to supply the `ID` for each variable, are replaced with new adapters that now provide the addressing for the variables.

Note that the intermediate design model obtained at this point is fully functional and also features bus-cycle accurate communication. However, the application layer communication protocol, in particular the operations necessary for sizing and addressing, are still performed in zero time. In order to obtain accurate execution times for these functions, they need to be inlined into the connected PEs.

Inlining is the last step of communication synthesis. As described in Chapter 2, inlining is performed for each channel in the design. It moves the functions contained in the channel into the connected behaviors and exposes the encapsulated variables which then represent wires.

The process of protocol synthesis can be demonstrated continuing the design example `S4` from Figure 3.15. Note that in Figure 3.15, the components `PE1`, `T1` and `M1` still communicate via the initial protocol provided by the virtual busses `VB1` and `VB2`. More specifically, `PE1` uses the `Read` and `Write` functions of the interface `M`, whereas `M1` calls `Put` and `Get` of the interface `S`. Further, all these functions use an `ID` to identify the particular variable being accessed.

Figure 3.16 shows the example `S4` after the actual bus protocols `SB` and `MB` have been inserted, replacing the virtual busses `VB1` and `VB2`, respectively. Please

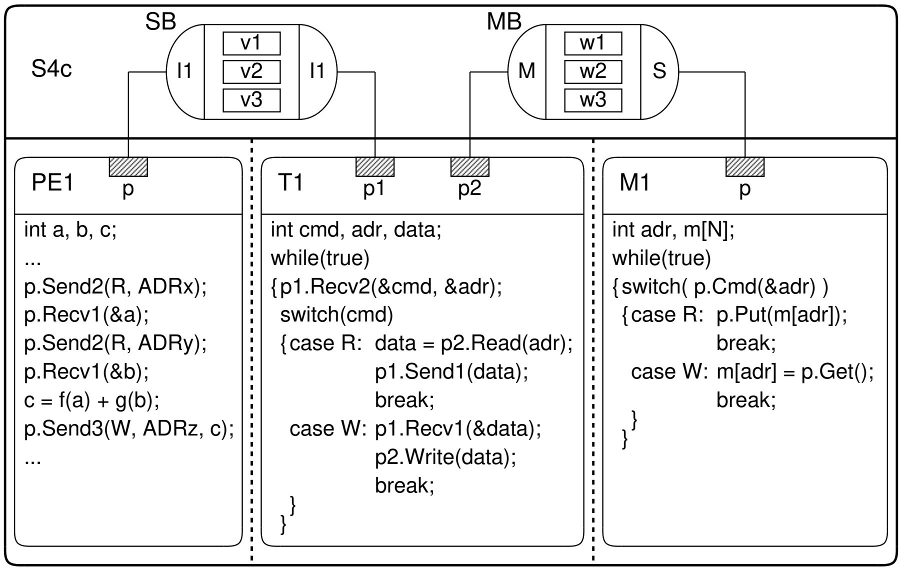

Figure 3.16: Design example `S4` after protocol insertion

note that in Figure 3.16, the application layer has already been inlined into the behaviors in order to demonstrate the change in the communication protocol[10]. Thus, the code shown in the behaviors uses the native communication functions of the selected busses. For this example, the bus `SB` is assumed to provide the functions `Send1`, `Send2`, `Send3`, as well as the equivalent `Recv` functions, whereas the memory bus `MB` provides `Put` and `Get` functions in the same manner as the virtual protocol before.

Note that, while sizing has been ignored, address assignment is shown with the example. Instead of the `IDs` in Figure 3.15, the memory addresses `ADRx`, `ADRy` and `ADRz` are used in Figure 3.16 to identify the variables.

The result of the final inlining process with the example is shown in Figure 3.17. The channels `SB` and `MB` have disappeared. Instead, the former encapsulated variables `v1`, `v2`, `v3`, and `w1`, `w2`, `w3` are used as communication wires. The ports of the components `PE1`, `T1` and `M1` have changed accordingly. Also, the code in the behaviors has been changed so that the low-level bus protocols become visible[11].

[10] Without inlining, the protocol change would have been invisible, since the code in the channel is not shown.

[11] For space reasons, only very small code fragments are shown.

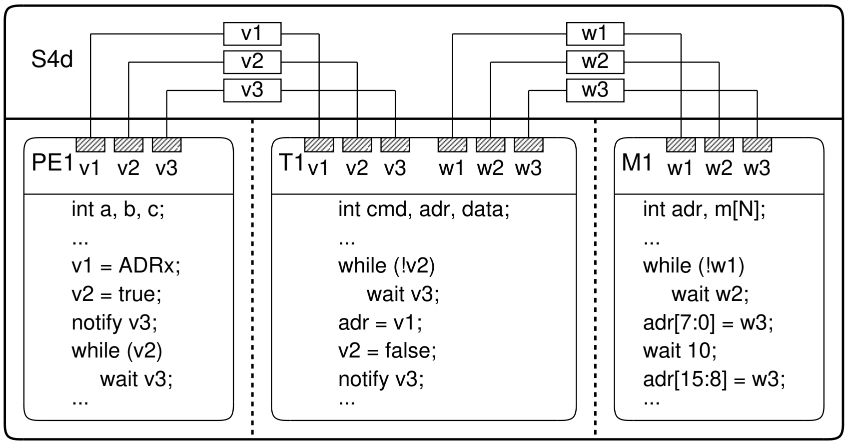

Figure 3.17: Design example `S4` after protocol inlining

3.5.4 The communication model

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

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

More specifically, the communication model is a bus functional model. The transactions on the system busses are represented accurately in great detail, bit by bit and cycle by cycle. On the other hand, the components in the system are still represented at a high abstraction level, allowing fast simulation. However, the execution times of the components are not exact, rather they are only estimated values.

A typical communication model is shown in Figure 3.18. Compared to the architecture model shown earlier in Figure 3.13, the two virtual busses `B1` and `B2` have been implemented, represented by the wire variables `B1a`, `B1b`, `B1c`, and `B2a`, `B2b`, respectively. Further, three transducers have been introduced. `T1` bridges the system bus `B1` to the native bus `B4` of the `IP` component. Similar, the system bus is connected to the processor `SW1` and the memory `M1` by the transducers `T2` and `T3`, respectively. On the other hand, the ASIC `HW1` connects to the system bus `B1` and the processor bus `B2` directly, since the necessary communication protocols have been inlined into the ASIC.

In order to emphasize the inlined communication protocols in Figure 3.18, the

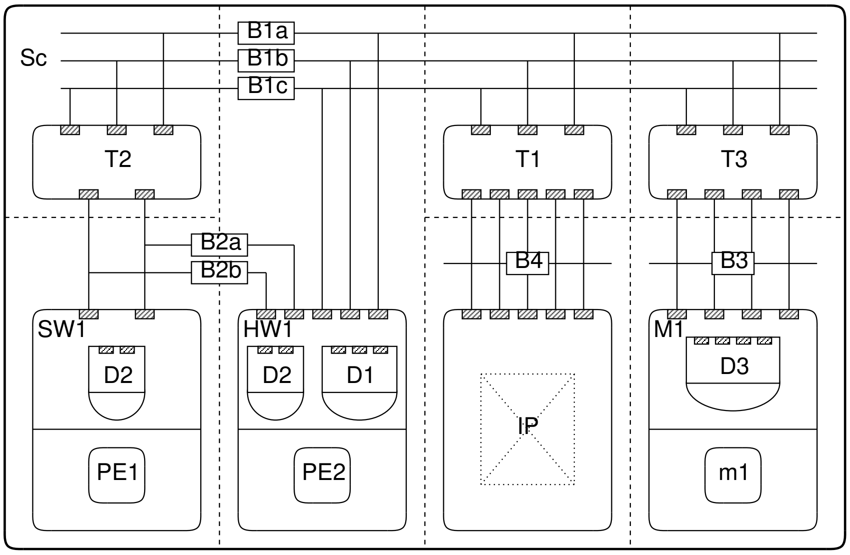

Figure 3.18: Communication model

application layer and the bus protocols are combined and shown as explicit adapters, called bus *drivers*. `D1`, `D2` and `D3` drive the busses `B1`, `B2` and `B3`, respectively. Please note that there is no driver shown for the bus `B4`, since this is the native bus of the `IP` with a built-in protocol.

3.6 Back end

The communication model is also the resulting model of the synthesis flow in the SpecC methodology, as shown in Figure 3.1. It is handed-off to the back end of the design flow.

It is the task of the back end to implement each particular component in the design model by use of standard EDA tools. More specifically, the components assigned to application specific hardware need to be implemented by a hardware synthesizer and the software components need to be compiled for the particular processor.

Please note that there is no need for a special interface synthesis tool any more, since the transducers in the system are standard hardware components which can be synthesized the same way as the allocated ASIC components.

3.6.1 Hardware synthesis

For each component in the design model, that is to be implemented as custom hardware, hardware synthesis has to be performed.

Since currently the SpecC language is not accepted directly by any hardware synthesizer, the SpecC code in the particular behavior needs to be translated into an acceptable language, such as a synthesizable VHDL subset, for example. Note that this translation should be straightforward, since there are no constructs in the component model left, which are not acceptable for hardware synthesis.

After this translation, traditional behavioral or high-level synthesis (HLS) [Mic94, Mar93, LMD94] can be performed, producing a netlist of RTL components as a result. Please note that the generated RTL netlist can be translated back into a SpecC model, since the SpecC language is capable of describing a hardware design model at this level of abstraction as well.

3.6.2 Software compilation

For the processor components in the design model, the according SpecC code is first translated into the standard C++ language, by use of the SpecC compiler. Then, any standard C++ compiler for the particular target processor can be used to produce the final machine code. Alternatively, a retargetable compiler, such as the GNU C/C++ compiler[12], that is capable of compiling C++ code for several target processors, can be used as well [LP97, MG95, Lie97].

In order to create a final SpecC implementation model of the design, the generated machine code can be used with an instruction set simulator of the target processor. Provided, that the instruction set simulator supports a suitable programming interface, for example in C, then this simulator can be easily hooked to the SpecC simulator.

As a result, a cycle-accurate simulation of the instruction set architecture (ISA) of the processor is possible for each software component in the SpecC implementation model.

3.6.3 The implementation model

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

The implementation model is the model with the lowest level of abstraction in the SpecC methodology. It is an accurate model of the design implementation in terms of

[12]Online information about the GNU C/C++ compiler is available at:
`http://www.gnu.org/software/gcc/gcc.html`

functionality, structure, communication and timing. Note that the implementation model reflects both, bus-cycle accurate timing for the communication, as well as clock-cycle accurate timing for the computation performed in the system.

The implementation model differs from the previous communication model only within the synthesizable components. A software component is described in form of an instruction set architecture. On the other hand, a hardware component consists of a network of RTL components, forming a control unit and a data path.

In summary, the implementation model is ready for manufacturing.

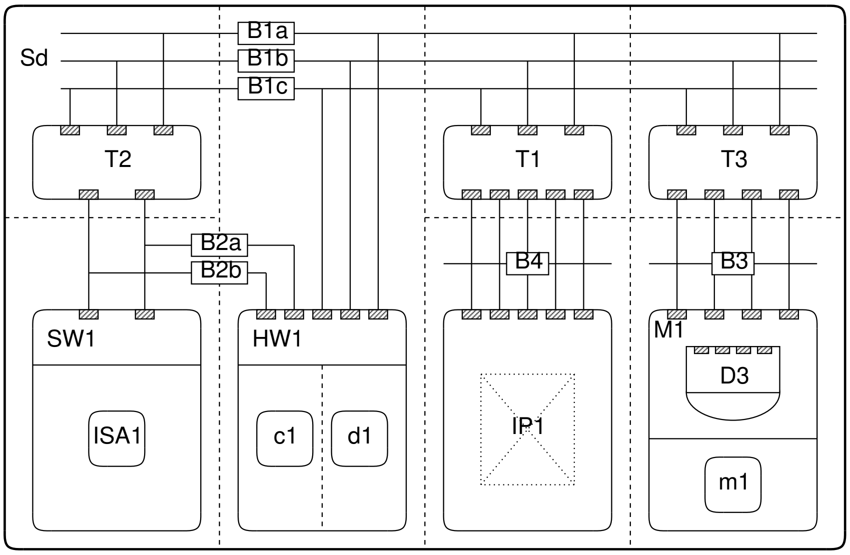

Figure 3.19: Implementation model

A typical implementation model is shown in Figure 3.19. In contrast to the communication model in Figure 3.18, only the processor `SW1` and the ASIC `HW1` have changed. The software component `SW1` is modeled as an instruction set architecture `ISA1`. On the other hand, the hardware component `HW1` consists of a controller behavior `c1` and a data path behavior `d1`.

In this chapter, the SpecC design methodology was described, which is based on four well-defined design models, namely the specification model, the architecture model, the communication model, and the implementation model.

Please note that, because of the modularity of the SpecC model ("plug-and-play"), a design can also be easily represented as a mixture of these models. This is especially useful if parts of a design are further refined as others, or if accuracy is only required for specific portions in the design model.

Chapter 4

The SpecC Language

For the system design methodology presented in Chapter 3, it is desirable that a single language is used for all models at all stages. Such a homogeneous methodology does not suffer from language interfacing problems or cumbersome translations between languages with different semantics. Instead, all models are consistent and one set of tools can be used for all models at all stages. Also, synthesis tasks are merely transformations from one program into a more detailed one specified with the same language.

Using a single language throughout the design process is beneficial for reuse of IP as well. Design models from the component library can be reused in the system without modification (*"plug-and-play"*) and a new design can be inserted immediately as a library component.

As stated already in the introduction, a general requirement for any system language is that it is formal and unambiguous. In order to employ automated refinement and synthesis tools, the design process must start from a formal specification.

These, and other similar requirements are satisfied by many languages, but this does not imply that all these languages are well-suited for the purpose of system-level design. The real quality of a language is determined by its *expressive power*. The expressive power of the language must match the purpose it is used for and must be sufficient to precisely describe the models and concepts needed during the design process. In other words, it is critical that the selected language meets the goals and requirements, but does not include unneeded features.

The goal of this chapter is the identification of a minimal and orthogonal set of properties which are necessary to specify and model embedded systems on different levels of abstraction. Once these properties have been identified and characterized, a language can be chosen or developed which explicitly supports these properties of embedded systems.

In the following section, the unique requirements and objectives for system-level languages are analyzed. Then, some of the traditional languages listed in the introduction are compared to these requirements in Section 4.2.

Since none of these commonly used languages completely meets the identified requirements, a new language called SpecC [GZD97a, ZDG97b, ZDG97a] is proposed. It is also shown that SpecC precisely covers the requirements of system-level design in an orthogonal manner.

4.1 Language Requirements

The major requirements for a language being used for system-level design are easily identified. In particular, such a language must be

- executable,
- synthesizable,
- modular, and
- complete.

In addition, a well-defined language should be

- orthogonal,
- minimal, and
- easy to understand.

4.1.1 Executability

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. Then, simulation is also necessary for the intermediate design models during the synthesis process. Here, the functionality of the refined design can be compared against the behavior of the model before the refinement.

4.1.2 Synthesizability

Synthesizability is a requirement whose importance cannot be ignored. In general, every construct provided by the modeling language should have at least one possible implementation. If this is not the case, a synthesizable subset of the language must be defined and only constructs from this subset can be used. Such a language subset, however, is essentially another language.

In other words, the requirement of synthesizability places a limitation on the descriptive and expressive power of the language being used. For example, many languages, such as VHDL, offer features which are simulatable but not synthesizable.

It should be obvious that, for a codesign language, it is desirable that the provided constructs can be implemented in either hardware or software. This makes it possible to trade-off a hardware implementation against a software implementation, and vice versa. However, it is also acceptable to have only one possible implementation. For example, the implementation of general pointers is only possible in software. On the other hand, parallel execution, in general, can only be implemented in hardware.

Furthermore, it is acceptable if the language contains constructs which need to be refined into a set of lower-level constructs in order to be implementable. Such constructs allow a highly abstract system specification without the loss of synthesizability.

4.1.3 Modularity

Modularity is required to clearly separate functionality from communication. 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 child behaviors, whereas *structural hierarchy* decomposes a system into a set of interconnected components [GZD97c].

Modularity is also required to support design reuse and the incorporation of IP. 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.

4.1.3.1 Behavioral hierarchy

The specification of behavioral hierarchy is defined as the process of decomposing a behavior into distinct child behaviors, which can be either sequential or concurrent.

The *sequential decomposition* of a behavior can be represented as either an algorithmic program or a state machine. On the other hand, the *concurrent decomposition* of behaviors allows child behaviors to run in parallel or in pipelined fashion.

Figure 4.1 shows a behavior `X` consisting of three child behaviors `A`, `B` and `C`. In Figure 4.1(a), the child behaviors are running sequentially, one at a time, in the

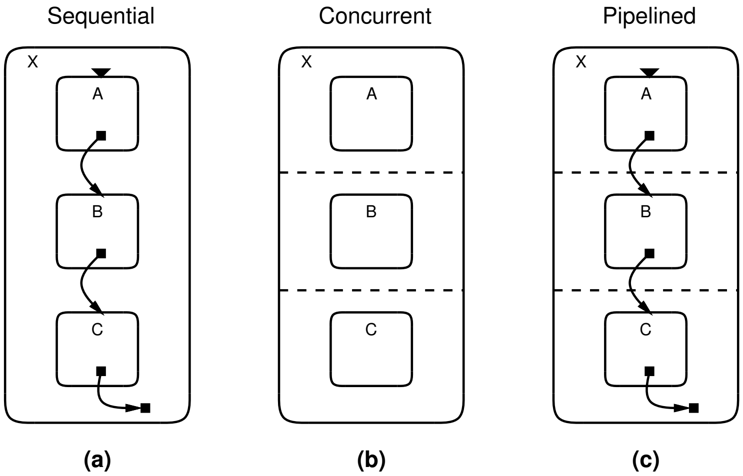

Figure 4.1: Behavioral hierarchy

order indicated by the arrows. In Figure 4.1(b), `A`, `B` and `C` run in parallel. In other words, they all will start when `X` starts, and `X` will finish when all of them have completed. In Figure 4.1(c), `A`, `B` and `C` run in pipelined mode, which means that they represent pipeline stages which concurrently process a stream of data, passing the data through all stages.

4.1.3.2 Structural hierarchy

With structural hierarchy, a system specification is represented as a set of interconnected components. Each of these components, in turn, can have its own internal structure, which is specified with a set of lower-level interconnected components, and so on. Structural hierarchy is typically represented as a set of block diagrams.

4.1.4 Completeness

Completeness is an obvious requirement that needs to be further refined. For a system language, completeness implies that all concepts commonly found in embedded systems design need to be supported.

The concepts needed for modeling embedded systems have been studied for several years. An in-depth discussion and definitions of these concepts can be found, for example, in [GVN+94] and [GZD97c, GZD97b]. In addition to behavioral and structural hierarchy, which have been discussed in the previous section, the important concepts include concurrency, synchronization, exception handling, timing, and explicit state transitions. These are briefly reviewed in the following sections.

4.1.4.1 Concurrency

Concurrency is a necessary feature of any system-level language. Concurrency can be classified into two groups, data-driven or control-driven, depending on how explicitly the concurrency is indicated in the language. Furthermore, a special class of data-driven concurrency, called pipelined concurrency, is of particular importance to signal processing applications. For more details about these concurrency classes, please refer to [GZD97c].

4.1.4.2 Synchronization

Concurrent behaviors usually need to be synchronized in order to be able to communicate or to cooperate. For example, one behavior may generate data that needs to be received by another behavior, or several behaviors have to execute some task simultaneously. In such cases, these behaviors need to be synchronized in such a way that one is suspended until the other reaches a certain point in its execution.

Common synchronization methods can be classified into two schemes, namely control-dependent and data-dependent synchronization. One example of control-dependent synchronization is the use of `fork` and `join` constructs for processes or threads. An example of data-dependent synchronization is the use of shared variables acting as valid-flags for exchanged data.

4.1.4.3 Exception handling

Often, the occurrence of a certain event requires that a behavior is interrupted immediately, prohibiting the behavior from further processing. This is called an exception. The behavior, to which the control will be transferred in such an event, is called an exception handler.

Exceptions can be divided into two groups, *abortion* and *interrupt*, as illustrated in Figure 4.2. In the case of abortion, the current behavior is terminated immediately and the exception handler will finish the execution. In the case of an interrupt, the control is transferred only temporarily to the handler. As soon as the interrupt handler terminates, the control is transferred back to the interrupted behavior which can resume its execution.

Typical examples of such exceptions are resets and interrupts in standard computer systems.

4.1.4.4 Timing

Although many computational models do not explicitly contain timing, there is often a need to include detailed timing information in the system specification. This is

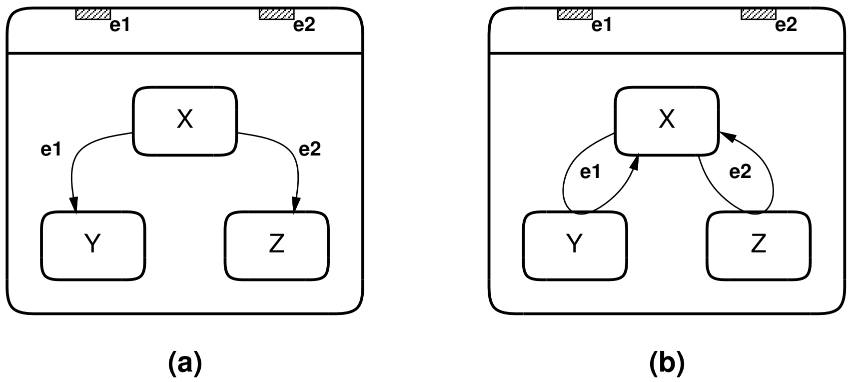

Figure 4.2: Exception handling: (a) abortion, (b) interrupt.

particularly true for real-time systems where the tasks have to be executed within the specified time periods.

Timing can be specified either exactly or in form of minimum or maximum constraints. For embedded systems, timing is typically measured in units of nanoseconds.

In general, a timing relation between two events can be described by a 4-tuple $T = (e1, e2, min, max)$, where the event $e1$ preceeds the event $e2$ by at least min time units and at most max time units. Such timing relations can be used for both timing delays and timing constraints.

Such timing information is especially important for describing parts of the system which interact with the environment according to a predefined protocol. In this case, the protocol defines the set of timing relations between signals, which both communicating parties have to respect. Such protocols are typically described graphically in form of timing diagrams.

4.1.4.5 State transitions

In order to model finite state machines, for example the FSM, FSMD and PSM models, explicit state transitions have to be supported. Such systems are often best conceptualized as having various modes, or states, of behavior. For example, a traffic-light controller might incorporate different modes for day and night operation and for the status of the traffic light itself.

In systems with various states, the transitions between these states typically occur in an unstructured manner. Such arbitrary transitions are similar to the use of `goto` statements in programming languages.

Transitions between states can be triggered by the detection of certain events or certain conditions. Depending on the actual FSM model, actions can be associated with each transition, and each particular state can have a behavior or computation associated with it.

4.1.5 Orthogonality

In addition to the requirements listed in the previous sections, there are additional goals and objectives for system-level languages. For example, an obvious objective is that a language is easy to understand.

Another important goal is the orthogonality of the concepts because this significantly simplifies the development of the tools working with the language. More specifically, it is desirable that all the concepts listed in Section 4.1.4 are organized in an orthogonal way. In other words, these concepts should be implemented independently from each other.

VHDL can serve as a counter example. In VHDL, signals incorporate synchronization, data storage and timing. This makes it very hard to identify for which purpose a particular signal is actually used, and thus an efficient implementation is hardly possible.

It should be noted that orthogonality implies minimality. If the concepts are organized in an orthogonal manner, only a minimal set of constructs is required.

4.2 Language Comparison

A fair amount of work has already been done in research about languages. However, much of previous work has focused on either languages for software design (programming languages) or languages for hardware simulation (hardware description languages). It can be expected that software languages are not suitable for describing hardware, and vice versa. Despite this, several system-level design approaches are using traditional languages, such as C, C++ and VHDL, for modeling embedded systems, as described in the introduction.

In this section, some of the traditional languages are analyzed and compared against the set of required concepts discussed in the previous sections. More specifically, C [X3/90], C++ [X3/97], Java [AG96], VHDL [IEEE93], Verilog [TM91], HardwareC [KM90], StateCharts [Har87], and SpecCharts [NVG91] are compared. In addition, SpecC [ZDG97b], which is described in the sections following this one, is included as well.

Figure 4.3 summarizes the results of the analysis[1]. For each language, it is shown

[1] Similar tables with language comparisons can be found, for example, in [GVN$^+$94], [JRV$^+$97] and [Nie98].

	C	C++	Java	VHDL	Verilog	HardwareC	Statecharts	SpecCharts	SpecC
Behavioral hierarchy	○	○	○	○	○	○	●	●	●
Structural hierarchy	○	○	○	●	●	●	○	○	●
Concurrency	○	○	◐	●	●	●	●	●	●
Synchronization	○	○	◐	●	●	●	●	●	●
Exception handling	◐	●	●	○	●	○	◐	●	●
Timing	○	○	○	●	●	◐	◐	◐	●
State transitions	○	○	○	○	○	○	●	●	●
Composite data types	●	●	●	●	◐	○	○	●	●

○ not supported ◐ partially supported ● supported

Figure 4.3: Comparison of language features

which requirements it supports and which are missing. Note that some concepts are only partially supported by some languages, as indicated by the half-filled circle. Please note also, that such a classification is only a rough characterization of a language. However, it indicates quite well which problems a language incorporates if it is considered for system-level design.

In addition to the features discussed earlier, the support of composite data types, which is a typical software language property, has been included in the last row of the table. Composite data types are user-defined data types such as arrays and records. These are often not supported by hardware languages, but are definitely needed for modeling systems containing software portions.

As shown in Figure 4.3, all the traditional languages lack one or more of the requirements. Hence, these languages cannot be used without problems for modeling embedded systems. In order to model systems containing both hardware and software, new languages need to be developed.

The SpecC language [DZG98] has been proposed as a new language that supports all the required concepts, as shown in the last column of Figure 4.3. SpecC is described in detail in the following sections.

4.3 Foundation

Accepting the fact, that a new language needs to be developed in order to meet all the requirements of embedded systems design, it has to be determined how the new language is being built. More specifically, the new language can either be developed from scratch, or can be built based upon an existing language. While the first approach offers the advantage of total freedom in terms of syntax and semantics, the second approach can easily leverage knowledge that is already present in the given language. Because it is obviously beneficial not to 'reinvent the wheel' (and possibly making mistakes while doing so), this approach was chosen for the development of the SpecC language.

When starting from an existing language, the features of this language are inherited by the new language. Hence, it is desirable to select a language which contains no unwanted characteristics which then would have to be taken out. For example, no constructs should be inherited which are not synthesizable.

Usually, it is easier to add a missing concept to a language, than taking an unwanted feature out. A language extension also has the advantage that existing programs for the base language will usually still work without modification when used with the new language.

For the SpecC language, several languages were considered as starting point, including C, C++, and Java. Eventually, C, or more precisely ANSI-C [X3/90], was selected because of its maturity and its large amount of already existing code. Although both, C++ and Java, offer advanced software features not present in C, the C language is still the de-facto standard for software development.

It should be emphasized that with the selection of C all requirements for software design are already satisfied. Furthermore, there are no features in the C language which cannot be implemented in an embedded system since, in the worst case, everything can be implemented in software. However, the missing concepts required for hardware design have to be added. This is described next.

The following sections introduce the SpecC language based on ANSI-C. For a fully detailed description, please refer also to the SpecC Language Reference Manual [DZG98], which includes a formally defined SpecC grammar using `lex` and `yacc` notation.

4.3.1 Types and expressions

The SpecC language is a true superset of ANSI-C [X3/90]. In other words, every C program that follows the ANSI-C standard can be used without modification as a SpecC program. The only exception is that the newly introduced SpecC keywords cannot be used for identifiers such as variable names. A complete list of these keywords is included in [DZG98].

Types and expressions supported by SpecC are mostly inherited from the C language. SpecC supports all the standard basic types, such as `int`, `float`, `double`, etc., and all aggregate and composite types, such as pointers, arrays and records (`struct`, `union`), together with the traditional operations known in C. In addition to these, SpecC provides explicit support for boolean, event, and bit vector types, as described next.

4.3.1.1 Boolean type

Similar to C++, the SpecC language explicitly supports a boolean data type `bool` for the representation of truth values.

```
bool f(bool b1, int a)
{
    bool b2;

    if (b1 == true)
        { b2 = b1 || (a > 0);
        }
    else
        { b2 = !b1;
        }
    return(b2);
}
```

A boolean value can have only one of two values, `true` or `false`. As illustrated in the example above, boolean values are used to express the result of logical operations such as comparisons. In expressions, a boolean type is converted implicitly to the integer type `int` whenever necessary. In this case, `true` is converted to `1` and `false` becomes `0`.

4.3.1.2 Bit vector type

In order to model hardware, explicit support for bit vectors of arbitrary length is required. SpecC provides a built-in bit vector type `bit[l:r]` with arbitrary precision specified by left (`l`) and right (`r`) bounds.

A bit vector can be thought of as a parameterized type whose bounds are defined with the name of the type. SpecC semantics require that the left and right bounds of any bit vector are constant expressions which can be evaluated statically. Hence, the length of any bit vector expression is constant and known at compile time. It should be emphasized that this is a synthesis requirement which, for example, is missing in VHDL.

A bit vector is either `signed` or `unsigned` and can be used as any other integral type within expressions. For example, the type `bit[sizeof(int)*8-1:0]` is equivalent to the integer type `int`. Implicit promotion to integral types, such as `int`, `long`, or `double`, is automatically performed when necessary. Furthermore, automatic conversion, i. e. extension or truncation, is supported as with any other integral type. No explicit type casting is necessary.

Bit vector constants are noted as a sequence of zeros and ones immediately followed by a suffix `b` or `ub` indicating `signed` or `unsigned` bit vector constants, respectively.

```
typedef bit[3:0]          nibble_type;
nibble_type               a;
unsigned bit[15:0]        c;

void f(nibble_type b, bit[16:1] d)
{
    a = 1101B;                          // vector assignment
    c = 1110001111100011ub;
    c[7:4] = a;                         // slice assignment

    b = c[2:5];                         // bit vector slicing
    c[0] = c[16];                       // single bit access
    d = a @ b @ c[0:15];                // concatenation
    b += 42 + a * 12;                   // arithmetic
    d = ~(b | 10101010B);               // logic operations
}
```

As shown in the example above, a concatenation operation, noted as `@`, and a bit slice operation, noted as `[l:b]`, are supported in SpecC. Both operations can be applied to bit vectors as well as to any other integral type. In this case, the integral type will be treated as a bit vector of suitable length.

In addition, a bit access operation, noted as an array access `[b]`, is provided as a short-hand for accessing a single bit (`[b:b]`) in a bit vector. Please note that, in this case, it is not required that the bit selector `b` is a constant expressions which can be statically evaluated, since the length of the resulting bitvector is always `1` and, thus, synthesis is possible.

4.3.1.3 Event type

In SpecC, events serve as the mechanism that supports synchronization and exception handling.

Events are represented by variables of the built-in type `event`. An event does *not* have a value. Therefore, events cannot be used within any expressions.

Events are used exclusively in two cases. First, they can used with the `wait` and `notify` statements in order to specify the synchronization of concurrent behaviors. For example, the following code shows a very simple example which coordinates the access to a shared variable `d` with `send` and `receive` functions.

```
int      d;
event    e;

void send(int x)
{
    d = x;
    notify e;
}

int receive(void)
{
    wait e;
    return(d);
}
```

Synchronization in SpecC is explained in more detail later in Section 4.8.

The second case, in which events are used, is exception handling supported by the `try-trap-interrupt` construct, which is described in Section 4.9.

4.3.1.4 Time type

In order for the SpecC language to support timing, a time type is used. However, strictly speaking, time is not an explicit type. Moreover, time is an implementation dependent integral type. For example, the current SpecC implementation uses `long long int`, a 64 bit integer type, for the representation of simulation time.

The SpecC language supports timed and untimed behavior, as defined in [ZDG97b]. Typically, timed behavior is used to model hardware, and untimed behavior is used to model software for which the execution time is not known.

In timed program sections, the time type is used with the `waitfor` statement to represent exact timing, and with the `do-timing` construct to represent timing ranges. Both, `waitfor` and `do-timing`, are described later in Section 4.10.

For untimed program sections, a special time variable `delta` is provided. The `delta` variable is of type time and is measured in implementation dependent units (e. g. nanoseconds). During simulation, `delta` evaluates to the elapsed real-time spent for executing the current behavior. For example, `waitfor(delta)` can be used to advance the simulation time by the actual amount spent on the host machine. In other words, assuming a software portion of a system is to be implemented on

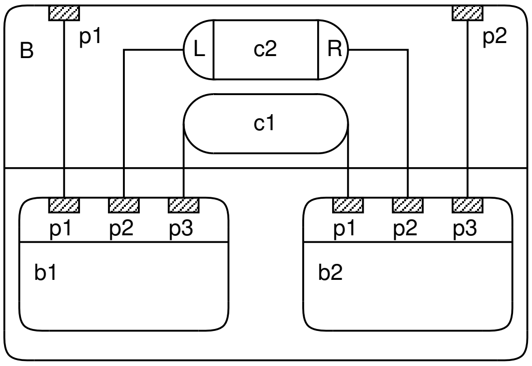

Figure 4.4: Basic structure of a SpecC model

the host machine, it can be synchronized easily with a simulated, timed hardware portion.

4.3.2 Statements and declarations

Similar to types and expressions, the majority of statements, declarations and definitions in the SpecC language are the ones inherited from C. These are assumed to be known and are not described in this work.

The statements and declarations, that were added to the C language, are described separately in the following sections. However, first the basic structure of a SpecC program is explained with a small example.

4.4 Basic Structure

As introduced in Chapter 2, a SpecC design model is captured as a hierarchical network of behaviors interconnected by channels with interfaces. The SpecC language reflects this model in a one-to-one fashion. Syntactically, a SpecC program consists of a set of `behavior`, `channel` and `interface` declarations.

A `behavior` is a class that can contain ports, component instantiations, and local variable and method definitions. Every behavior also has a public `main` method which specifies its functionality.

A `channel` is a class that encapsulates a set of local variables and methods. Hierarchical channels contain instantiations of child channels as well.

An `interface` class is used to declare the methods which are public in channels. Interface classes consist solely of method declarations. The associated method definitions are implemented in channels.

For example, the following SpecC description specifies the system illustrated in Figure 4.4:

```
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 specifies a behavior `B` consisting of two child behaviors `b1` and `b2`. The child behaviors are executing concurrently, specified by the `par` statement. Furthermore, `b1` and `b2` communicate via an integer variable `c1` and a channel `c2` which are connected to the ports of the child behaviors.

The SpecC constructs used in this example are described in detail in the following sections.

4.5 Behavioral Hierarchy

Behavioral hierarchy is the composition of child behaviors in time. In SpecC, child behaviors can either be executed sequentially or concurrently. Sequential execution can be specified by standard imperative statements, or as a finite state machine (FSM) model with explicit state transitions. On the other hand, concurrent execution is either parallel or pipelined.

4.5.1 Sequential execution

Syntactically, behavioral hierarchy is specified in the `main` method of the composite behavior. For sequential execution, the `main` method can either consist of an imperative program calling the child behaviors in a specific order, or of an explicit FSM in which the child behaviors take the role of states.

4.5.1.1 Imperative program

In the simplest case, child behaviors are executed in a fixed sequential order. For example, a behavior `B` consisting of three sequentially executed child behaviors can be specified as follows.

```
behavior B;

behavior B_seq(void)
```

```
{
   B     b1, b2, b3;

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

In a more general case, a conditional control-flow can be specified in a straightforward manner by use of standard C statements, such as `if-then-else`, `for`, and `while`. However, this is not a recommended modeling style since the mixture of programming statements with child behavior calls is difficult to analyze and thus aggravates the use of automated refinement tools. Such a model represents the case (h) in Figure 2.5 discussed earlier in Section 2.3.

In order to clearly specify a conditional, sequential control flow among child behaviors, the FSM model should be preferred.

4.5.1.2 Finite state machine

The SpecC language provides the `fsm` statement to specify finite state machines (FSMs) with explicit state transitions. Both Mealy and Moore type FSMs can be modeled with the `fsm` construct.

```
behavior B;

behavior B_fsm(in int a, in int b)
{
   B     b1, b2, b3;

   void main(void)
   {
     fsm{ b1: { if (b < 0)  break;
                if (b >= 0) goto b2;
              }
          b2: { if (a > 0)  goto b1;
                goto b3;
              }
          b3: { break;
              }
        }
   }
};
```

As shown in the example above, the `fsm` construct specifies a list of conditional state transitions among states which are represented by instantiated child behaviors. A state transition is a triple $\langle current\_state, condition, next\_state \rangle$, where *current_state* and *next_state* take the form of labels denoting child behavior instances. The *condition* expression determines whether the transition is valid.

The execution of a `fsm` construct starts with the execution of the behavior that is listed first in the transition list (`b1`). Once this behavior has finished, its state transition determines the next behavior to be executed. The conditions of the transitions are evaluated in the order they are specified and, as soon as one condition is `true`, the behavior specified after the `goto` statement is started. A `break` statement terminates the execution of the `fsm` construct.

Please note that the body of the `fsm` construct does not allow arbitrary statements. The SpecC syntax limits the state transitions to well-defined triples. This ensures that the `fsm` construct can be easily analyzed and refined by automated tools.

4.5.2 Concurrent execution

In SpecC, concurrent execution is either parallel or pipelined.

4.5.2.1 Parallel execution

Parallel execution of behaviors is specified with the `par` construct, as shown in the following example.

```
behavior B;

behavior B_par(void)
{
   B      b1, b2, b3;

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

Every statement in the compound statement block following the `par` keyword forms a new thread of control and is executed in parallel. The execution of the `par` statement itself completes when each thread of control has finished its execution. In other words, the `par` construct *forks* the control flow into a set of parallel threads which are *joined* again when the `par` statement is completed.

The example shows the behavioral hierarchy of three child behaviors `b1`, `b2` and `b3` which are executed in parallel. The parent behavior `B_par` will terminate as soon as all three children have completed their execution.

Note that for simulation on a sequentially executing host, the `par` construct is not really executed in parallel. Instead, the scheduler, which is part of the SpecC simulation library, executes one thread at a time and decides when to suspend and when to resume a particular thread depending on the simulation time and synchronization points.

4.5.2.2 Pipelined execution

The SpecC language provides explicit support for the specification of pipelines. Pipelined execution is a special form of concurrent execution. Similar to the `par` construct, pipelined execution is specified with a `pipe` construct, as shown in the following example.

```
behavior B(in int p1, out int p2);

behavior B_pipe(in int a, out int b)
{
   int          x;
   piped int    y;
   B            b1(a, x),
                b2(x, y),
                b3(y, b);

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

Each statement in the compound statement block after the `pipe` keyword forms a new thread of control. The set of control threads is then executed in a pipelined fashion. The `pipe` statement itself implies an infinite loop of execution and thus never finishes.

In the example, the child behaviors `b1`, `b2` and `b3` form a three-stage pipeline of behaviors. In the first iteration, only `b1` is executed. When `b1` completes, the second iteration starts and `b1` and `b2` are executed in parallel. In the third and every following iteration, all three child behaviors are executed in parallel.

Note that such an execution scheme could also be specified by iterated use of the `par` construct. However, in addition to the execution order, the `pipe` construct supports explicitly buffered communication between the pipeline stages which otherwise is difficult to specify and typically is not recognizable for automated refinement tools.

To specify buffered communication, the special storage class `piped` is used for variables connecting two pipeline stages. A variable with a `piped` storage class can be thought of as a variable with two storages. A write access to such a variable always writes to the first storage. A read access, on the other hand, reads from the second storage. The contents of the first storage are shifted to the second storage whenever a new iteration starts in the `pipe` construct.

In the example, a standard variable `x` connects the first pipeline stage (`b1`) with the second (`b2`). This variable is not buffered, in other words, every access to `x` from stage 1 is immediately visible in stage 2. On the other hand, the variable `y` connecting the second (`b2`) and the third stage (`b3`) is specified as `piped`. A value computed by behavior `b2`, that is stored in `y`, will be available for processing by `b3` in the next pipeline iteration when `b2` already produces new data.

Note that the `piped` storage class can be specified n times defining a variable with n buffers. This can be used to transfer data over n stages synchronously with the pipeline.

4.6 Structural Hierarchy

Structural hierarchy is represented in form of a hierarchical block diagram where the blocks have ports and are interconnected via communication channels. In SpecC, these blocks are called *behaviors*.

4.6.1 Behaviors

A SpecC behavior is an object for the specification of active functionality. Typically, behaviors are used to encapsulate computation. In terms of structure, a behavior has ports through which it can communicate with other behaviors.

Syntactically, a SpecC behavior is specified by use of a `behavior` declaration or definition. A `behavior` definition is a class that consists of a set of ports, a set of local variables and methods, and a mandatory `main` method. If the behavior is a composite behavior, a set of child behavior instantiations is included as well. For example, the following specifies a simple leaf behavior `B`.

```
behavior B(in int p1, out int p2)
{
  int  a, b;

  int f(int x)
  {
    return(x * x);
   }

  void main(void)
  {
    a = p1;    /* read data from the input port */
    b = f(a);  /* compute */
    p2 = b;    /* write data to the output port */
   }
};
```

Except for the `main` method, which is public, all local methods and variables in the behavior are private. In other words, a behavior resembles a black box whose contents are not visible from the outside[2].

Local variables and methods, such as `a`, `b`, and `f` in the example above, can be used to conveniently program the functionality of the behavior. Similar to the `main` function in a C program, the `main` method of a behavior is the root of the behaviors execution. It is called whenever an instantiated behavior is executed and its completion determines the completion of the behavior.

A SpecC program starts with the execution of the `main` method of the root behavior. The root behavior is identified by its name which is defined as `Main`. Usually, the behavior `Main` is a composite behavior resembling the test bench for the specified system. In this test bench, the top behavior, that specifies the actual system, is then instantiated. Please note that `main` and `Main` are names which need to be recognized by automated tools. However, these names are not keywords of the SpecC language.

A behavior *declaration* consists of the behavior name and the declaration of its ports. For a behavior *definition*, the behavior body is required. For example, a declaration for the behavior defined above is as follows.

```
behavior B(in int p1, out int p2);
```

A behavior is compatible with another behavior if the number and the types of their ports match. Compatibility of behaviors is important for the reuse and replacement

[2]By use of interfaces implemented by a behavior, it is possible to make selected local methods of the behavior public. Since this is rarely necessary, it is ignored in this context. Please refer to the SpecC Language Reference Manual [DZG98] for further information.

of components ("plug-and-play"). Please note that a behavior declaration is sufficient to determine compatibility. The behavior body is not required.

4.6.2 Netlists

Structural connectivity among components in a block diagram is typically represented by connectors and wires. In SpecC, connectors are represented by ports and wires by variables. In order to specify connectivity, the variables are then mapped onto the ports as part of the behavior instantiation.

Ports are defined with the declaration of the behavior, very much like arguments to functions are defined in a function declaration. A port can be of any SpecC type and includes a port direction as a type modifier. A port direction is either `in`, `out` or `inout`, and is handled as an access restriction to that port. Inside a class, an `in` port allows only read-access, and an `out` port only allows write-access. An `inout` port can be accessed in either way. When connecting ports, the port types and port directions must be compatible.

Port mapping lists are used to specifies the connectivity of the ports, as shown in the following example.

```
behavior B1(in event clk, out int p1, out bit[15:0] p2);

behavior B2(in event clk, in int p1, in bit[31:0] p2);

behavior B(in event clk, in bit[31:0] p1)
{
   int          i;
   bit[15:0]    b;

   B1   b1(clk, i, b);
   B2   b2(clk, i, p1[31:16] @ b);

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

In the example, two child behaviors `b1` and `b2` are instantiated in the behavior `B`. The three ports of `b1` are connected to the clock input port `clk` of `B`, the wire `i` and the internal bus `b`, respectively. Similar, `b2` is connected to `clk` and `i` as well.

SpecC also supports bus splitting in port mapping lists. Concatenated bit slices are used to represent sliced busses. In the example, this is demonstrated with the

32 bit wide port `p2` of `b2`. It is wired to the upper half of the incoming bus `p1` of `B` and the internal bus `b`.

4.7 Communication

In addition to netlists, which essentially allow communication through shared variables as described in the last section, the SpecC language supports a much more powerful concept for communication, namely channels and interfaces.

4.7.1 Channels

A SpecC channel is an object designed for the specification of complex communication. Typically, a channel encapsulates a (possibly hierarchical) communication protocol. In contrast to behaviors, channels are passive objects. In other words, channels serve as container for common methods used for communication. These methods are made available to be used by behaviors so that these can communicate.

Syntactically, a channel is specified by use of a **`channel`** declaration or definition, very similar to the **`behavior`** construct. A channel definition is a class, that consists of a channel declaration and a channel body, which contains a set of local variables and methods. In case of a hierarchical channel, child channel instantiations are part of the channel body as well.

Like behaviors, channels can have ports. For channel ports, the same semantics apply for channels as described earlier for behaviors.

However, much more important than ports are the interfaces of a channel, which are listed after the ports in the channel declaration. The interfaces determine the set of public methods which are provided by the channel. Interfaces are described in the following section.

By default, the local variables and methods defined in a channel are private, in other words, they cannot be accessed from outside the channel. However, the methods that are declared as implemented interfaces, are public and may be used by behaviors to perform communication via the channel.

Similar to behaviors, the compatibility of channels is required when a channel is to be replaced with another one. A channel is compatible with another channel, if the number and the types of the channel ports, and the list of the implemented interfaces, match.

4.7.2 Interfaces

Interfaces represent the missing link between behaviors and channels. As shown in the following example, an interface is a class which specifies the set of public methods implemented in a channel.

```
interface I
{
   void send(int x);
   int receive(void);
};

channel C implements I
{
   int  data;

   void send(int x)
      {
        data = x;
       }
   int receive(void)
      {
        return(data);
       }
};
```

The example specifies a channel `C` that provides a simple communication protocol via an encapsulated integer variable. The interface `I`, which the channel `implements`, contains the declarations of the public methods `send` and `receive`.

Interfaces are used to connect behaviors with channels in such a way that both, the behaviors and the channels, are easily exchangeable with compatible replacements. Interfaces essentially enable the *"plug-and-play"* feature of the SpecC language.

For example, consider two behaviors, `b1` and `b2`, which communicate via an instance of the channel `C` declared above.

```
behavior B1(I p1)
{
   void main(void)
      { int x;
        ...
        p1.send(x);
       }
}

behavior B2(I p1)
{
   void main(void)
      { int y;
```

```
14             ...
15             y = p1.receive();
16          }
17 }
18
19 behavior B(void)
20 {
21     C      c1;
22     B1     b1(c1);
23     B2     b2(c1);
24
25     void main(void)
26         {
27             ...
28          }
29 };
```

In the example, both behaviors `B1` and `B2` have ports of interface type `I`. Because channel `C` implements the interface `I`, the ports of `b1` and `b2` can be mapped to the channel `c1`. This way, `b1` and `b2` can communicate via the `send` and `receive` methods.

Now, if another channel `C2` is available with the same interface `I`, i. e.

```
channel C2 implements I;
```

then the protocol specified with channel `C` can be switched to the protocol provided by channel `C2` simply by replacing line 21 with

```
C2     c1;
```

Note that neither the replaced channels nor the connected behaviors have to be modified for this change. Please note also that the same easy replacement is possible for the behaviors `B1` and `B2`.

It should be mentioned that some communication protocols require the use of call-back functions. In such a case, some methods specified in a channel need to call-back methods provided by the behavior that initiated the communication. In order to support this, the SpecC language allows interfaces for behaviors as well. In addition, a keyword `this` is provided for a behavior to be able to identify itself. Please refer to [DZG98] for further documentation.

4.8 Synchronization

In order to allow controlled cooperation among concurrent executing behaviors, a synchronization mechanism is required. In SpecC, the built-in type `event` serves as the basic unit of synchronization, as stated in Section 4.3.1.3. To specify synchronization, events are used with the `wait`, `notify` and `notifyone` statements which all take a list of events as arguments.

A `wait` statement suspends the current behavior from execution until one of the events specified with the `wait` statement is triggered by another behavior. The execution of the waiting behavior then resumes.

The `notify` statement triggers all specified events so that all the behaviors waiting on one of these events can resume their execution. If no behavior is waiting on the triggered events at the time of the `notify` statement, the notification is ignored.

The `notifyone` statement acts similar as the `notify` statement. However, `notifyone` allows only one behavior from the set of currently waiting behaviors to resume its execution.

For example, the following code specifies a channel `C2` that can be used as a replacement for the channel `C` presented in the previous section.

```
channel C2 implements I
{
    int     data;
    bool    valid = false;
    event   wakeup;

    void send(int x)
      {
        data = x;
        valid = true;
        notify wakeup;
       }
    int receive(void)
      {
        while(!valid)
            { wait(wakeup);
             }
        valid = false;
        return(data);
       }
};
```

Compared to the primitive channel `C` on page 99, the channel `C2` uses the synchronization statements `wait` and `notify` to prevent the reading of uninitialized data. It also avoids that the same data is read multiple times. In other words, this channel ensures that the consumer always receives valid data.

4.9 Exception Handling

The SpecC language provides support for both types of exceptions discussed in Section 4.1.4.3, namely interrupt and abortion. The occurence of such exceptions is represented by events. The `notify` statement introduced in the previous section is used again to trigger such events.

In order for exceptions to be handled during the execution of a behavior, the behavior has to be made sensitive to a set of events. In SpecC, this is specified with the `try` construct, as shown in the following example.

```
1  behavior B0;
2  behavior B1;
3  behavior B2;
4
5  behavior B(in event IRQ, in event RST)
6  {
7      B0    b0;
8      B1    b1;
9      B2    b2;
10
11     void main(void)
12        {
13          try { b0.main(); }
14              interrupt (IRQ) { b1.main(); }
15              trap (RST)      { b2.main(); }
16        }
17 };
```

In the example, the behavior `B` consists of three child behaviors `b0`, `b1` and `b2`. The execution of behavior `B` will `try` to execute `b0` and, if no exception occurs, the completion of `b0` will also terminate the execution of `B`. However, if one of the events `IRQ` or `RST` occurs while the child behavior `b0` is executing, the execution will be interrupted or even aborted.

4.9.1 Interrupt

An interrupt is specified with the `interrupt` keyword as shown in line 14 in the example. The events, which will trigger a specific interrupt, are specified as arguments, i. e. `IRQ`.

If the event `IRQ` occurs during the execution of `b0`, the behavior `b0` will be stopped immediately in its execution and the interrupt handler `b1` will be started to service the interrupt. After `b1` has completed its execution, the control is transferred back to behavior `b0` which can resume its execution right from the point where it was stopped.

4.9.2 Abortion

Abortion is specified with the `trap` keyword. This also is followed by a list of events, i. e. `RST`, that will trigger the abortion, as shown in line 15 in the example.

If the event `RST` is notified while behavior `b0` is executing, it will be terminated immediately and the control is transferred to `b2` which will take over the execution. In contrast to an interrupt, `b0` will not regain control after `b2` is completed. Instead, the behavior `B` will terminate.

4.10 Timing

As discussed earlier, the notion of time is an important requirement for specification languages. Typical timing information includes the execution time or delay of components, and timing constraints for the system performance or communication protocols.

The SpecC language supports both types of timing specification discussed in Section 4.1.4.4, namely exact timing and timing ranges.

4.10.1 Exact timing

Exact timing, such as delay or execution time, is specified by use of the `waitfor` statement. The required time value is given in form of an argument and must be of the integral time type introduced in Section 4.3.1.4.

The semantics of the `waitfor` statement are as follows. Whenever a `waitfor` statement is executed, the current behavior is suspended from further execution for the specified simulation time. Any concurrent running behaviors will then be executed until they are suspended as well, due to `waitfor` or `wait`. Once all active behaviors are suspended, the simulation time will be increased such that the behaviors with the least amount of waiting time can resume their execution.

Please note that the simulation time is only increased by use of the `waitfor` statement. All other statements in the SpecC language execute in zero time.

4.10.2 Timing ranges

In order to specify timing constraints, timing ranges are supported in SpecC. A timing range is specified as a 4-tuple $T = \langle L_1, L_2, T_{min}, T_{max} \rangle$, where L_1 and L_2 are specific points in time. The time period between L_1 and L_2 is limited to a minimum of T_{min} and a maximum of T_{max} time units.

Syntactically, the `range` statement is provided for such timing ranges and L_1 and L_2 take the form of labels. Furthermore, T_{min} and T_{max} can be left unspecified, indicating the values $-\infty$ and $+\infty$, respectively. For example, the statement

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

specifies at a time period of at least 10 but not more than 20 time units between the labels `l1` and `l2`. On the other hand,

```
range(l3; l4; 0; );
```

simply states that the statements specified at label `l4` must not be executed before the statements at `l3`.

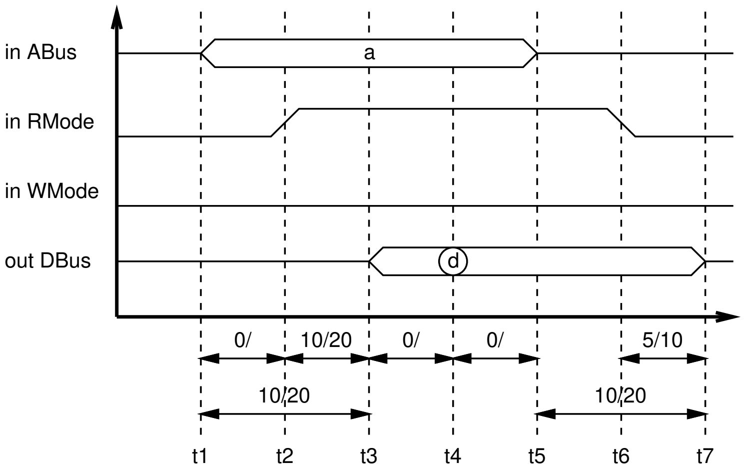

Figure 4.5: Timing diagram example: SRAM read protocol.

Timing ranges are most useful for the specification of *timing diagrams*. Consider, for example, the timing diagram of the read protocol of a static RAM, as shown in Figure 4.5. When reading a word from the SRAM, the address of the requested data is supplied with the address bus `ABus`. Then, the read operation is selected by setting `RMode` to high and `WMode` to low. After the specified time period, the requested value can finally be accessed from the data bus `DBus`. The timing constraints throughout this protocol are explicitly specified in form of annotated arcs in the timing diagram. All these constraints must be satisfied for a successful read access.

In SpecC, it is straightforward to capture such a timing diagram. The diagram shown in Figure 4.5 can be specified as follows.

```
bit[7:0] ReadByte(bit[15:0] Address)
{
    bit[7:0]        MyData;

    do    { t1: { ABus = Address;
                  waitfor(2);
                }
```

```
            t2 : { RMode = 1; WMode = 0;
                   waitfor(12);
                   }
            t3 : { waitfor(5);
                 }
            t4 : { MyData = DBus;
                   waitfor(5);
                 }
            t5 : { ABus = 0;
                   waitfor(2);
                 }
            t6 : { RMode = 0; WMode = 0;
                   waitfor(10);
                 }
            t7 : {
                 }
         }
   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);
         }
   return(MyData);
}
```

The `do-timing` construct, as shown in this example, is used to encapsulate a timing diagram representation. In the `do` part, the value changes at specific points in time are specified as labeled assignment statements. The range constraints are then listed in the following `timing` block.

The execution semantics of a `do-timing` construct are basically the same as for any sequence of compound statements. The labeled statements are simply executed in the order specified.

However, the attached timing constraints are validated during the execution of the construct by the simulation run-time system. A typical simulator will maintain a list of time stamps when executing a timing diagram. For each label, its execution time will be noted. Then, when the execution of the `do` block is completed, these time stamps are used to check whether the specified `range` constraints hold. Any violation of the constraints should be reported to the user.

The current implementation of the SpecC simulator, for example, will, by default, generate a run-time error message for each violated range constraint and then abort the simulation. However, this behavior can be overwritten by the user[3].

The range check performed by the simulator, makes it necessary to use `waitfor` statements within the timing diagram, as shown in the example. Without such `waitfor` statements, the specified timing constraints would not hold and, thus, the construct would fail its execution. Please note that the `waitfor` statements only specify *one* instance out of a typically infinite set of legal time periods.

4.11 Persistent Annotation

For the purpose of practicality in use with a set of separate tools, the SpecC language offers support for persistent annotation. Persistent annotation allows to attach any type of constants to any named symbol in a SpecC program. This annotation mechanism eliminates in many cases the need for separate files exchanged between subsequent tools working on the same design.

More specifically, persistent annotation can be used for convenient information interchange between the tools working with a shared SpecC design description. For example, an estimation tool can easily annotate its results with each behavior in the design so that these estimation results are available for use in an exploration or synthesis tool that is called afterwards. Moreover, such annotations are also available to the user.

The semantics of persistent annotations are out of the scope of the SpecC language. In particular, annotations do not change the execution semantics of a SpecC program. As such, they can be seen of as a special type of comments in a SpecC description.

Syntactically, the `note` declaration specifies persistent annotations, as shown in the following example.

```
/* C style comment, not persistent */
// C++ style comment, not persistent

note Author       = "Rainer Doemer";
note Date         = "Fri Dec 10 09:52:07 PST 1999";

const int x       = 42;
note x.Bits       = sizeof(x) * 8;

behavior B(in int a, out int b)
```

[3] In order to overwrite the default behavior for handling time constraint violations, a function called `_scc_range_check` needs to be defined by the user. If this function is present in the SpecC program, it will be called instead of the default handler.

```
11 {
12     note Version   = 1.2;
13
14     void main(void)
15     {
16         l1: b = 2 * a;
17             waitfor(10);
18         l2: b = 3 * a;
19
20         note NumOps   = 3;
21         note l1.OpID = 1;
22         note l2.OpID = 3;
23     }
24 };
25 note B.Area = 12030;
```

The SpecC language allows comments in the source code in form of C++ syntax. More specifically, comments are either enclosed by `/*` and `*/` delimiters, or start with `//` and last up to the end of the line, as shown with lines 1 and 2 in the example. Comments are simply ignored by the compiler, thus, they are not persistent.

The `note` declaration attaches a persistent note to the specified symbol, label or user-defined type. Such notes are named and their value is a constant or constant expression that can be evaluated at compile time.

There are two ways to define an annotation. First, a note can be attached to the current scope, such as global notes (lines 4 and 5 in the example) and notes at classes (line 12). Second, the annotated object can be named explicitly. In the example, this style is used to define the notes at variable `x` (line 8), the labels `l1` and `l2` (lines 21, 22), and the behavior B (line 25).

4.12 Library Support

Similar to the library and package concept provided in VHDL, the SpecC language supports the incorporation of pre-compiled design libraries into the specification description. This simplifies the handling of complex component libraries and also speeds up the compilation.

Syntactically, the `import` declaration specifies the inclusion of a binary library into the current design. This is also called binary import. In addition, the `#include` construct inherited from the C language supports the inclusion of source code (non-binary) files. An example of both constructs is shown next.

```
#include <stdio.h>
#include <stdlib.h>

```

```
import "Interfaces/I1";
import "Interfaces/I2";
import "Channels/PCI_Bus";
import "Components/MPEG_II";
```

An `#include` declaration is evaluated in a preprocessing step. The C preprocessor, which can be used without modification for SpecC programs as well, simply replaces the `#include` construct with the contents of the named file.

Similar, the `import` declaration efficiently incorporates pre-compiled, binary SpecC files. Any SpecC source description can be pre-compiled into a binary file with the SpecC compiler. Such files are typically named with the suffix `.sir`, indicating that these files contain the SpecC Internal Representation (SIR). SIR files can also be used to transfer designs in an efficient way between separate tools. The SpecC Internal Representation is described in more detail in Section 5.2.

4.13 Summary

Built on top of ANSI-C, the SpecC language is designed for the executable specification of embedded systems. To the well-known set of requirements for software languages, specific constructs needed for hardware design have been added.

SpecC is synthesizable. Every construct supported by the language has at least one straightforward implementation in either software or hardware.

Furthermore, the SpecC language supports modularity in form of both, behavioral and structural hierarchy. SpecC also satisfies the requirement of completeness. It provides support for all requirements for system-level design, namely concurrency, hierarchy, communication, synchronization, exception handling and timing, as discussed in Section 4.1.

It must be emphasized that the SpecC language provides *orthogonal constructs* for these orthogonal concepts. In other words, the identified, independent concepts are implemented with independent constructs in a one-to-one fashion. This allows to model embedded systems clearly and unambiguously.

The orthogonality also allows minimality. The SpecC language covers the complete set of system concepts with a minimal set of constructs. This makes the language easy to learn and easy to understand.

Last, but not least, it should be emphasized that the SpecC language has gained acceptance in the industry. Recently, SpecC has been proposed as a standard system-level language for the adoption in industry by some of Japan's top-tier electronics and semiconductor companies [CGC+99].

4.14 Possible Extensions

The SpecC language has been proven to work for system-level design. Several examples have already been successfully specified, simulated, and refined, as listed in Appendix B. However, this experience with the real use of the SpecC language has also shown that minor adjustments and some extensions are desirable to make system-level design even easier and more convenient. These issues, which could be implemented in a future version of the SpecC language, are addressed briefly in this section.

4.14.1 Fine tuning

Events, which are used for synchronization and exception handling, are currently only supported as plain, non-aggregate types. The reason for this is that events do not have a value and therefore cannot be used in expressions. However, it is desirable to support arrays and records of events. This could, for example, be introduced by allowing event expressions which can be evaluated at compile time and solely consist of access operations to arrays and records.

In particular for data stream processing applications, such as the vocoder described in Appendix B.6, it is desirable to pass sub-arrays through ports of behaviors and channels. Currently, this is only supported for bit vectors in form of bit slices. An equivalent scheme for general arrays can only be specified by passing pointers to sub-arrays through the ports. Such pointer-arithmetic could be easily avoided if the language provides a specific construct for this case.

The `pipe` statement in its current form never finishes. In other words, it contains an implicit endless loop and thus cannot be used in a nested form. An extension to this construct could, for example, allow the flushing of the pipeline after a specified number of iterations.

The persistent annotation of a SpecC program is currently limited to constant values. This could easily be extended to allow general expressions.

4.14.2 Operator overloading

Operator overloading, as supported for example by VHDL and C++, is desirable for the specification of operations such as vector additions and matrix multiplications, because it makes the source code easier to read. In addition, it allows experiments with the arithmetic precision used in computations. For example, saturated operations could be used instead of the default, non-saturated arithmetic.

Since operator overloading is currently not supported by the SpecC language, explicit function calls must be used for such cases. Operator overloading could easily be added to the SpecC language in very much the same way as C++ added this feature to the C language.

4.14.3 Object orientation

In a similar way, the SpecC language could also be extended to become object oriented. Object oriented features, such as object inheritance, could be easily applied to the SpecC behaviors, channels and interfaces. The implementation of inheritance for these classes in a C++ style would be straightforward.

4.14.4 Templates

The concept of templates, such as provided in C++, also would be applicable to SpecC. However, maybe a restricted form would be sufficient. For example, an equivalent for the VHDL `generate` and `generic` constructs would serve most purposes.

Chapter 5

The SpecC Design Environment

The SpecC approach presented in the previous chapters has been implemented in the SpecC design environment which is described in the following sections.

First, an overview about the SpecC design environment and its tools and libraries is given. Then, the major system components of the SpecC release 2.0.4 are described, which have been implemented by the author of this work. These components include the central design representation, called SpecC Internal Representation, the SpecC compiler, a profiler and a tool set.

5.1 Overview

The SpecC design environment has been built according to the methodology presented in Chapter 3. As shown in Figure 5.1, the SpecC tools reflect the design and validation flow shown earlier in Figure 3.1.

The tool flow starts with the design capture by use of the SpecC editor. The SpecC editor, called *VisualSpec* [IG98], is a graphical editor for SpecC models. VisualSpec allows to capture and modify a design by use of block diagrams, connectivity tables, hierarchy displays and flow charts. Only leaf behaviors and channels are specified in textual form in the SpecC language by use of a standard text editor.

VisualSpec also includes the graphical user interface (GUI) of the SpecC design environment. The GUI allows to call and control the SpecC tools directly from the graphics. Since design models can be captured, compiled, and executed very quickly, VisualSpec can also be seen as a rapid prototyping environment [AIG99] based on the SpecC approach.

Throughout the SpecC design environment, the design models are represented by the *SpecC Internal Representation* (SIR). The SIR is a complex data structure used internally by all SpecC tools to maintain the design models. The SIR is also a

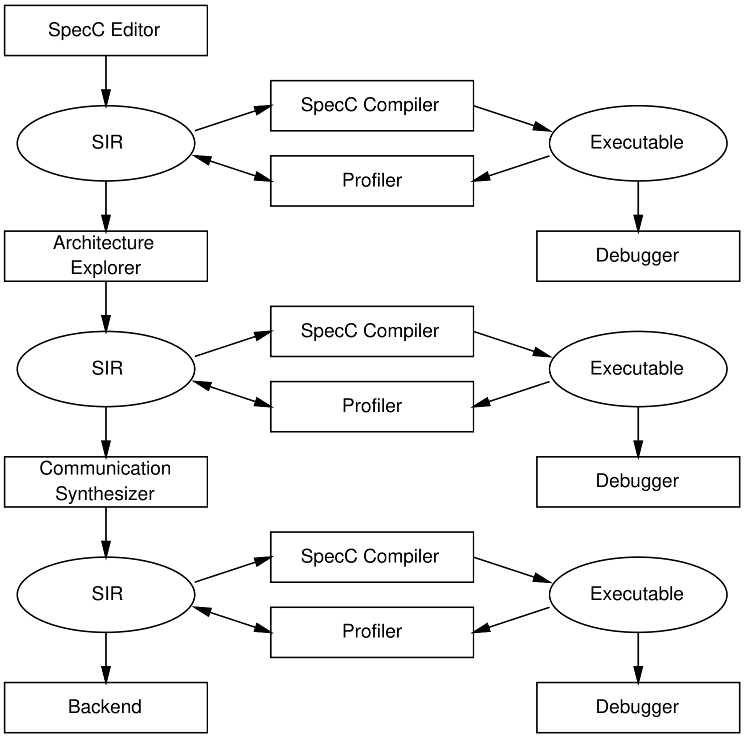

Figure 5.1: The SpecC design environment

binary file format, equivalent to SpecC source code stored in a text file. Section 5.2 describes the SIR in more detail.

The *SpecC compiler* is primarily used to compile SpecC design models into executable programs. As shown in Figure 5.1, the SpecC compiler can be used at any stage in the design flow to create an executable file for simulation. Furthermore, the SpecC compiler is also used to convert design files from their textual representation into SIR files, and vice versa. The SpecC Compiler is described in Section 5.3.

For simulation, the SpecC compiler links the executable file with the *simulation library*. The simulation library maintains the event queue and the simulation time during simulation. It also takes care of concurrent threads and their synchronization according to the execution semantics of the SpecC language. In other words, it implements the SpecC simulator.

Once an executable file has been created, the design can be simulated simply by running the SpecC program on the host computer. In case of problems, a standard *debugger* can be attached to the program. With the debugger, SpecC programs can be executed step by step, break points can be set, and data values can be inspected easily by the designer.

The *SpecC profiler* can be used to obtain run-time information about a design. In particular, during the execution of the simulation model, branching probabilities are obtained by the profiler by use of counters inserted into the design model. The branching probabilities are then back-annotated to the design model so that they can be used by the estimators, for example.

The synthesis flow is implemented by three main tools according to the SpecC methodology. First, the *architecture explorer* refines the specification model of the design into the architecture model, as discussed in Section 3.4. The architecture explorer itself consists of several smaller tools, including *estimators* for software and hardware, an *allocator* that determines the system architecture, a *partitioner* that computes and performs the architecture mapping, and a *scheduler* that sequentializes the behaviors assigned to processors.

The second major refinement tool is the *communication synthesizer* which takes the SIR file produced by the architecture explorer and performs communication synthesis as described in Section 3.5.

The generated communication model, in form of a SIR file, is then passed on to the *back end*. In the back end, specific compilers for each of the selected processors are called to implement the software portion of the system. Also, automatic synthesis tools are run for each custom hardware component, generating the final implementation model of the design.

5.1.1 SpecC release 2.0.4

The SpecC design environment consists of a large set of complex tools. Some of these tools, in particular the major refinement tools, architecture explorer, communication synthesizer and the back end, are, at the time of this writing, still under active development and have not been released yet. On the other hand, the tools for the specification capture and the validation flow have been released and are already in evaluation and use in industry and academia.

While the graphical editor VisualSpec [IG98] and the integrated prototyping environment [AIG99] are commercially developed and distributed, the tools for the SpecC validation flow have been made freely available on the world-wide web[1] (WWW).

The components of the SpecC system, which have been developed and implemented by the author of this work, are included in the public SpecC release 2.0.4. Table 5.1 lists the components of the release 2.0.4, along with the author and the

[1] The SpecC web pages are online at `http://www.ics.uci.edu/~specc/`. The most recent SpecC system can be downloaded from `http://www.ics.uci.edu/~specc/download.html`.

Source component	Author	Lines of code	Size [kB]
System setup	R. Dömer	3251	88.6
SpecC Internal Representation	R. Dömer	57522	1466.3
Bit vector library	A. Gerstlauer	2992	74.9
Simulation library	J. Zhu	14002	274.8
SpecC compiler	R. Dömer	13390	346.8
SpecC profiler	R. Dömer	2549	63.7
SpecC tool set	R. Dömer	5401	143.4
Design examples	SpecC team	6326	131.1
Total		105433	2589.6

Table 5.1: Source components of the SpecC release 2.0.4

size of the source files for each of the components of the SpecC system[2].

The main components developed by the author of this work, namely the SpecC Internal Representation, the SpecC compiler, the profiler and the tool set, are described in the following sections.

5.2 SpecC Internal Representation

The SpecC Internal Representation (SIR) is the common design representation in the SpecC design environment. All tools in the SpecC system use the SIR to read, write, store, maintain and modify the SpecC design models.

The SIR is three-fold. First, it is a binary file format for designs specified with the SpecC language. Second, it is a complex data structure with a well-defined Application Programming Interface (API). Third, it is provided as a shared library for use by any SpecC tool developer.

The motivation for the development of the SIR is based on the fact that the design models used in the SpecC design methodology are all represented by the SpecC language. Each tool working with a design model needs procedures for input, access and output of the model. Since these procedures are essentially the same for every tool, a shared library can be used to implement the required functions.

The benefit of the SIR as a common representation is that new tools can be developed very quickly since all functions dealing with the design representation are already prepared. There is no need any more to develop and implement these functions, which otherwise would require a significant amount of time. With the SIR,

[2] Recently, the version 2.0.5 of the SpecC system has been released. In addition to the components of release 2.0.4, the new version includes a set of tools for static system-level scheduling [CG99].

the SpecC tool developer can focus solely on the algorithm of the tool, knowing that the design representation and its access have already been taken care of.

In following sections, the SIR file format, the SIR data structure and the SIR API are briefly described. Then, the benefit of quick tool development with the SIR is demonstrated by the implementation of the SpecC profiler.

5.2.1 SIR File format

SpecC design models are stored in binary files, called SIR files. The SIR file format is an external representation of the internal SIR data structure. By use of SIR files, design models can be easily passed from one SpecC tool to another, without the need for a special interface between the tools. Also, every tool can read the output of every other tool so that, technically, refinement tools can be applied to a design model in any order.

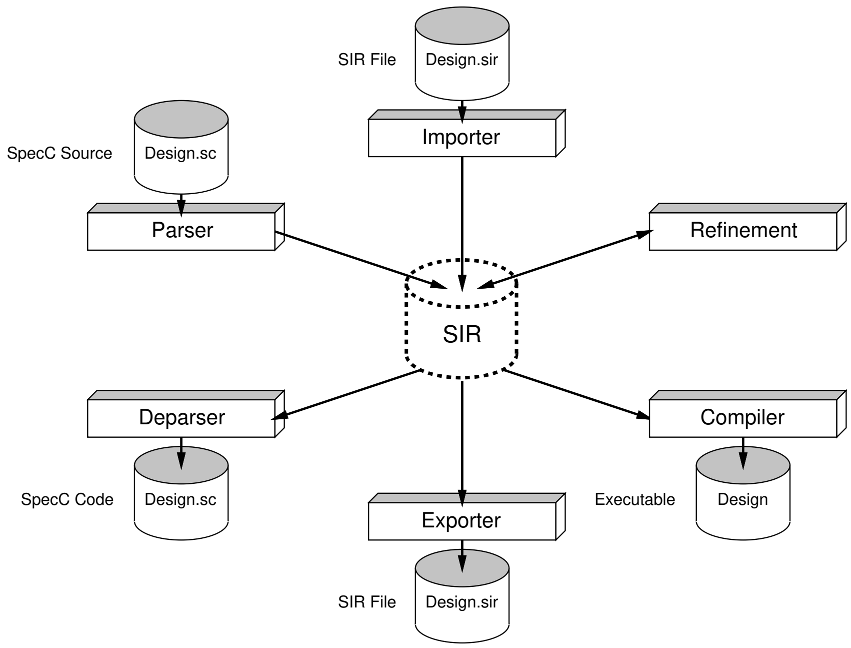

Figure 5.2: Design representation with the SIR

Figure 5.2 shows the different formats and the conversions between these formats for a design model in the SpecC system. In this star topology, the internal SIR is the central representation.

Initially, a specification model of any design is given in form of SpecC source code, typically stored in a file with suffix `.sc`. This textual representation is read by the *parser*, generating the internal SIR data structure. From the internal data structure, a binary SIR file can be created by use of the *exporter*. Typically, such a SIR file has the extension `.sir`.

The SIR file format is then used by all refinement tools. Each tool reads the SIR by use of the *importer*, performs its refinement on the internal data structure, and finally generates a new SIR file with the help of the exporter.

For inspection or textual modifications by the user, a binary SIR file can also be converted into a readable text file. The *deparser* creates SpecC program code from the internal representation, which, after any modification, can be translated back into the SIR by use of the parser.

Please note that the functionality of the parser, deparser, importer and exporter is part of the SIR implementation, whereas the boxes *refinement* and *compiler* are implemented as separate tools. However, the compiler, whose program flow is described later in Section 5.3, can be instructed to only perform the file conversions shown in Figure 5.2, instead of the default function to generate an executable file from SpecC source code.

5.2.2 SIR library

From the point of view of a programmer, the SpecC Internal Representation is a shared library that implements a complex data structure.

The SIR library is provided as a binary, shared library which can be linked to any tool developed for the SpecC system. In addition to the binary library, a set of C++ header files is provided. The header files contain the declarations of the functions and classes implemented by the SIR library.

The data structure implemented by the SIR library consists of a hierarchy of C++ classes. The organization of these classes, forming a hierarchical graph of objects, is included in Appendix C. However, for fully detailed information about the SIR data structure, its classes and methods, please consult the reference documentation [Döm98, Döm99]. In these documents, all SIR classes are listed and described in detail with their data members and API methods. In addition, the source code of example programs is listed which use the SIR API to build, modify and store SpecC design models.

5.2.3 Application Programming Interface

The SpecC Internal Representation offers a comprehensive Application Programming Interface (API) to the SIR data structure. The SIR API is embedded in the four interface *layers* to a SpecC design, as shown in Figure 5.3.

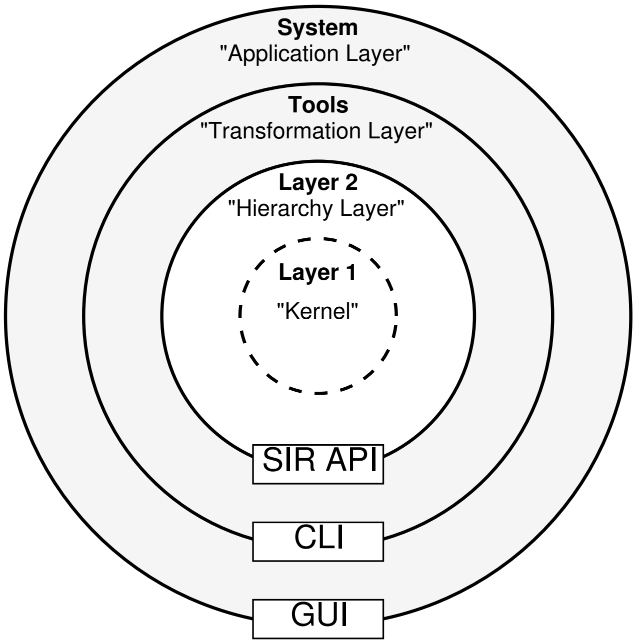

Figure 5.3: SIR Application Programming Interface

At the highest level, the so-called *application layer*, a Graphical User Interface (GUI) is used for the interaction with the user. In the SpecC design environment, this is implemented with the SpecC editor.

Alternatively, the SpecC tools can be used at the *transformation layer*. The transformation layer offers a textual interface, called Command Line Interface (CLI), to the SpecC tools in form of shell commands. For the advanced user, such shell commands allow the use of scripting languages to work on SpecC designs.

The API of the SpecC Internal Representation is shown with the white inner circles in Figure 5.3. For access to the in-memory representation, the SIR API offers two interface layers, namely the hierarchy layer and the kernel layer.

5.2.3.1 Kernel layer

The SIR kernel, as the innermost design representation, represents the lowest level of abstraction. The design model is represented basically as a parse tree created from the SpecC language description. Although symbol and type tables are maintained by the kernel, there is no representation of connectivity or any hierarchical relations among the symbols.

The use of kernel API methods requires detailed knowledge about the internals of the SIR data structure. No semantic or syntactic error checking is performed. It is in the responsibility of the user to correctly perform memory allocation and deallocation when inserting or removing objects. The user is completely in charge of maintaining the consistency of the data structure, such as pointers, links, etc.

Because of these difficulties, the direct use of the SIR kernel API should be avoided. Instead, the API of the hierarchy layer can be used which is built on top of the SIR kernel.

5.2.3.2 Hierarchy layer

For the SpecC tool developer, the hierarchy layer provides a safe API for the maintenance and refinement of SpecC design models. As the name indicates, the hierarchy layer explicitly represents hierarchical relations between the objects. The behavioral and structural hierarchy of the SpecC design model is reflected in the data structure in a one-to-one fashion.

The API of the hierarchy layer offers convenient methods for the whole data structure that guarantee the consistency of the design representation even in the case of errors. In other words, the hierarchy layer ensures that the design model is a syntactically and semantically valid SpecC model at any time.

The hierarchy layer also simplifies transformations on the data structure significantly. In addition, memory allocation and deallocation are performed automatically with the creation and deletion of objects.

5.2.4 Experiment

In order to demonstrate the value of the SIR for the quick development of new tools in the SpecC design environment, the following experiment has been conducted. The development and implementation of a set of tools for the SpecC system has been timed. The tools chosen for this experiment use the SIR library for design input, modification and output. Therefore, a short implementation time for the tools is expected, since the time for the implementation of the functions provided by the SIR can be saved.

5.2.4.1 Example application

As an example application, a set of simple profiling tools has been selected. The profiling tools are well-suited for this experiment, as they represent simple refinement tools which read a design model and create a modified version of the model. Also, the tasks of the tools are simple enough, so that not much time needs to be spent on the development and implementation of the algorithms.

In particular, four profiling tools have been implemented, whose tool flow is shown in Figure 5.4.

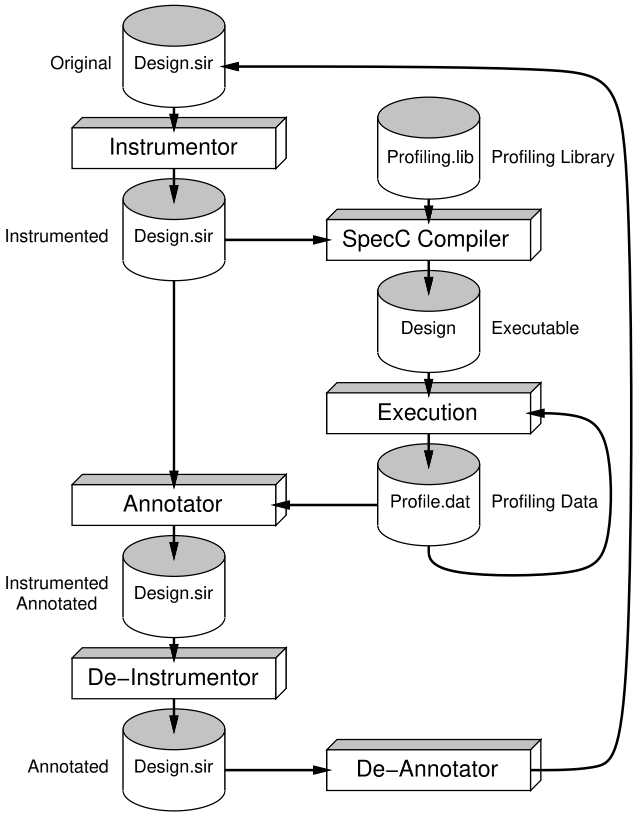

Figure 5.4: Program flow of the SpecC profiling tools

First, the task of the *instrumentor* is to insert counters into in the design model so that the execution of the methods and functions in the design is profiled when the design is simulated. In addition, the instrumentor inserts function calls which read the initial counter values in the beginning and write the final counter values out into a file at the end of the simulation. The functions for reading and writing of the profile values are provided by a *profiling library* which will be linked to the executable file by the SpecC compiler.

The second profiling tool is the *annotator* which will take the counter values obtained after the simulation and back-annotate them to the design model. As a result, every function and method in the design model will be annotated with the number of its executions.

In order to complete the set of profiling tools, two tools are needed which undo the changes of the instrumentor and the annotator. It is the task of the *de-instrumentor* to take out all counters and function calls inserted by the instrumentor. Similar, the *de-annotator* removes any counter annotations inserted by the annotator.

It should be noted that the four tools implemented in this experiment have been later combined into the profiler that is part of the standard SpecC distribution. The SpecC profiler is described in Section 5.4.1.

5.2.4.2 Results

The development and implementation times for the four tools, including the profiling library, are shown in Table 5.2.

Task	Time	Lines of code
Specification	3 h, 28 min	259
Profiling library	1 h, 32 min	75
Template	0 h, 59 min	354
Instrumentor	1 h, 54 min	124
De-Instrumentor	1 h, 15 min	99
Annotator	1 h, 6 min	99
De-Annotator	0 h, 19 min	42
Total	10 h, 33 min	1052

Table 5.2: Development and implementation of the profiling tools

Most of the development time was spent for the detailed specification of the four tasks. In particular, this includes the manual generation of code fragments which show the exact changes to be performed by the tools.

Since all four tools have a similar program flow, consisting of reading, modifying and writing, a program template was developed first. The template then was used as a starting point for the four programs[3].

Table 5.2 shows that all four tools have been developed, implemented and tested in a very short time. In fact, the complete set of all four profiling tools has been developed within one working day.

This result clearly shows the value of the SpecC Internal Representation. Without the SIR, the implementation of the profiling tool set would have required much more time.

[3] In order to obtain the actual size of a program, the lines of code written for the program template need to be added to the lines of code listed for the particular tool.

5.3 SpecC Compiler

The SpecC compiler, called `scc`, is the main tool in the validation flow of the SpecC methodology. The main purpose of the SpecC compiler is to generate an executable program for simulation from a design model. However, the SpecC compiler also serves as a converter between the different SpecC file formats, as mentioned earlier.

The program flow of the SpecC compiler is shown in Figure 5.5. By default, the SpecC compiler reads SpecC source code and generates, after several intermediate steps, an executable file. This default flow starts at the top of the graph and goes straight down to the bottom. The compiler can also be instructed to follow any other paths in the graph, performing different tasks, i. e. file conversions or only partial compilation.

The generation of a simulation model from source code in the SpecC language is performed in five steps. First, the source code is processed by the *preprocessor* which performs header file inclusion and other preprocessing directives in the code. Because the SpecC language contains no special preprocessor commands other than those defined by the C programming language, a standard C preprocessor is used for this task.

Second, the preprocessed code is read by the *parser* which builds the SpecC Internal Representation in the memory and, at the same time, performs syntax and semantic checking.

In order to create executable code, a C++ program is generated in the next step by the *translator*. The generated C++ program consists of two files, a header file with variable, function and class declarations, and a main file, containing the implementation of the declarations.

The generated program is then compiled by a standard C++ compiler into binary object code. Finally, the *linker* creates the executable program, combining the compiled object code with the SpecC simulation library and any other system libraries.

It should be emphasized that the SpecC compiler takes special care of debugging support when creating the C++ program from a SpecC model. As a result, any standard C/C++ debugger can be used to debug SpecC programs. In other words, the SpecC *debugger* is implemented by any standard debugger provided on the simulation host.

The debugging support of the SpecC compiler is achieved through two features. First, the C++ program is generated in such a way, that it reflects the original SpecC program line by line. Thus, each line of SpecC code has a corresponding line of generated C++ code. In addition, SpecC constructs are implemented by C++ constructs following a one-to-one mapping. For example, behaviors and channels are implemented by C++ classes, bit vectors are represented by C++ templates, and

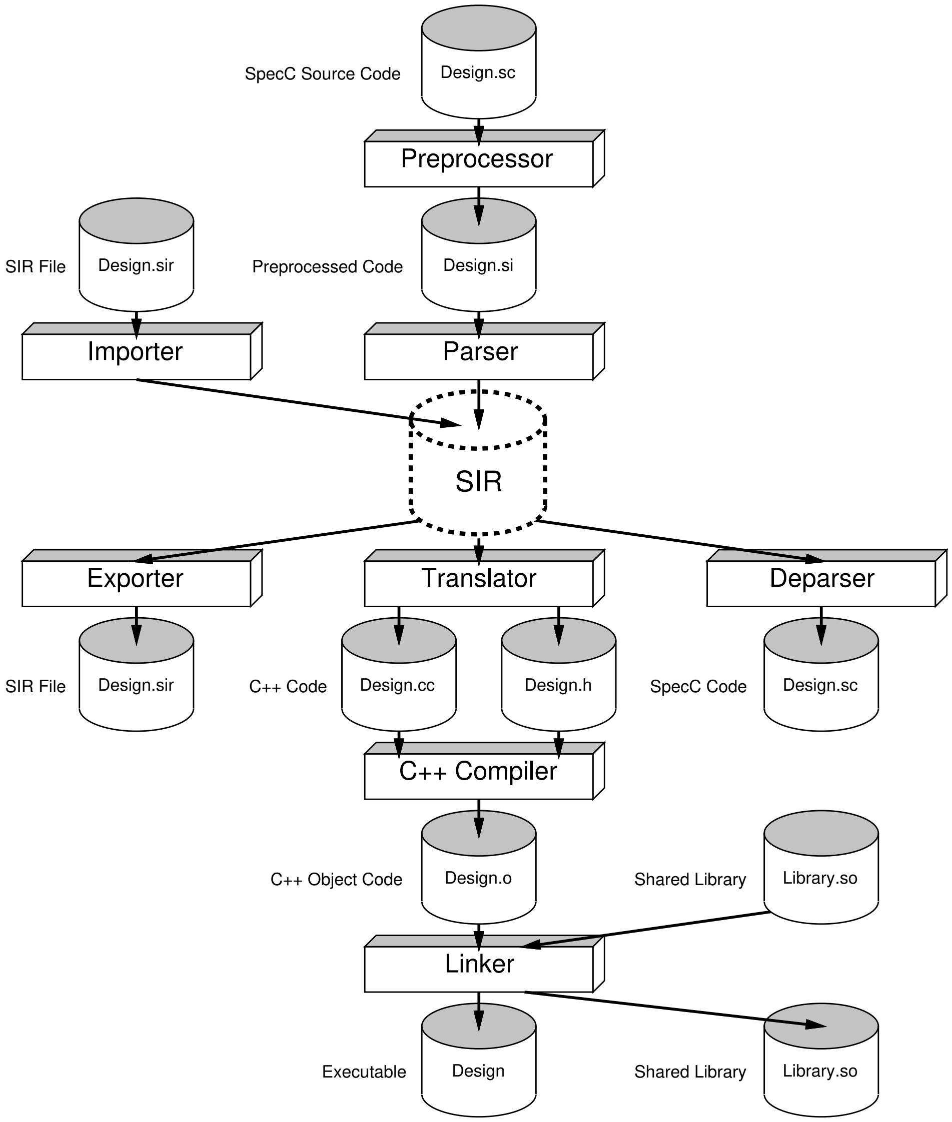

Figure 5.5: Program flow of the SpecC compiler

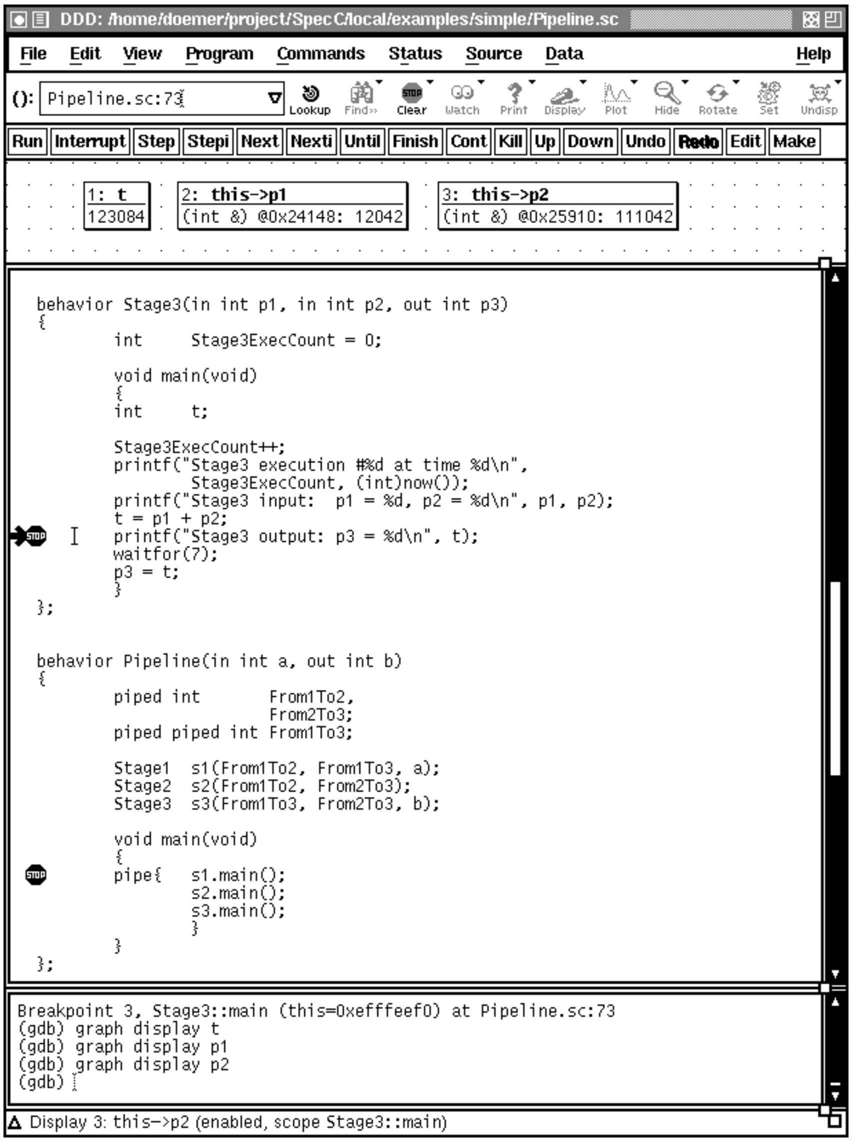

Figure 5.6: Standard debugger use for SpecC programs

statements like `par` and `pipe` are implemented by function calls to the simulation library.

Further, `line` directives are inserted into the generated C++ program, linking the program code with the SpecC source. As a result, any tool processing the C++ program will refer to the original SpecC code. For example, error and warning messages issued by the C++ compiler will point to the line in the SpecC source where the problem originated from.

Finally, if the generated executable program is run by a source level debugger, the debugger will display the original SpecC program in the source code window. As an example, Figure 5.6 shows the debugger `ddd` running the SpecC pipeline example that is part of the SpecC distribution.

5.4 SpecC Refinement Tools

The program flow of typical SpecC refinement tools is shown in Figure 5.7.

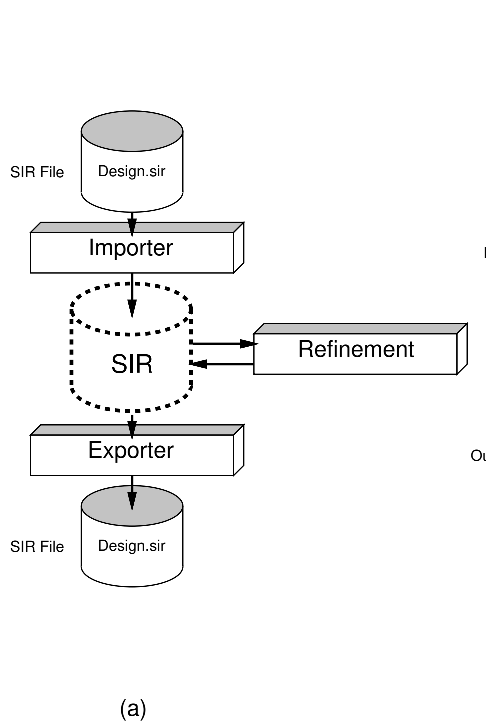

Figure 5.7: Program flow of typical SpecC refinement tools: (a) modification of the SIR, (b) creation of a new SIR from the input SIR.

A refinement tool inputs a design model from a SIR file, performs its refinement on the internal representation by use of the SIR API, and finally creates a new SIR file for the refined design model. The refinement itself can be performed by modification of the input SIR, as shown in Figure 5.7(a), or by creating a new output SIR from data in the input SIR, as shown in Figure 5.7(b).

Please note that, in both cases, a new refinement tool can be based on a significant amount of implementation that is already existing, since the importer, the exporter and the internal data structure with the API, are provided by the SIR library. The refinement tool developer can pay his full attention to the algorithms of the refinement task.

An initial set of simple refinement tools has been developed by the author of this work. These tools are briefly described in the next two sections.

5.4.1 SpecC profiler

The SpecC profiler has been developed based on the four profiling tools described earlier in Section 5.2.4.1. The SpecC profiler, which is part of the standard SpecC distribution, combines the four tools into one single program, but still follows the profiling flow described in Figure 5.4.

Since the profiling tool set has been described already, a further description of the SpecC profiler is redundant. For more details, however, please refer to the profiler manual which is listed in Appendix A.2.

5.4.2 SpecC tool set

The SpecC tool set consists of six utility programs which simplify the work with binary SIR files. Manual pages for these tools can be found in Appendix A.3.

The SpecC tool set includes the following tools.

- `sir_delete` allows to delete named objects from a SIR file.
- `sir_list` lists the objects contained in a SIR file with their type and classification.
- `sir_note` allows to attach and remove arbitrary annotations at objects in a SIR file.
- `sir_rename` allows to rename any named objects in a SIR file.
- `sir_strip` strips off line numbers and source file names from SIR files.
- `sir_tree` displays the behavioral hierarchy in a SIR file as a tree of behaviors and channels.

Chapter 6

IP Protection in the SpecC System

As discussed in the introduction, it is in the strong interest of IP providers to protect their intellectual property from being used without permission or being reverse-engineered. In particular, external IPs require technical measures for protection.

In order to protect hard IPs, the IP provider typically keeps the final implementation of the components in-house. Only simulation models of the IPs with different levels of accuracy are made available to the system integrator. For soft IPs, on the other hand, complete synthesizable models are needed by the system integrator. In both cases, these models (still) contain implementation and algorithm details of the IP which the IP vendor does not want to reveal to his customers. Therefore, the models are usually provided in binary format without source code. For example, many VHDL or Verilog simulators allow to precompile the description of an IP into object code, so that the source code is hidden, but the model is still simulatable [KB98].

Such an approach is well-known for software reuse and software protection. Software components usually consist of a set of public function and variable declarations whose implementation is supplied in form of a precompiled library. When producing an executable program, this library is integrated with the compiled code by the linker. All the necessary information to use such a software package is contained in the declaration of the API and the accompanying documentation. The actual implementation is hidden from the user in the object code and therefore protected.

In the SpecC system, IP protection is based on this software approach [DG00]. IP components are provided in form of a public interface declaration, specified in SpecC source code, and a linker library, containing the secret implementation supplied in binary object code.

However, special care has to be taken to make sure that an IP component cannot be reverse-engineered from the data made available. The following sections describe this problem and the solution taken in the SpecC system.

6.1 Public IP Declaration

As described in Section 2.5, IP components are modeled as behaviors or channels, depending on whether they contain computation or communication. The model of an IP assumes that the internals of these behaviors and channels are unknown.

Syntactically, the SpecC language distinguishes the *declaration* and the *definition* of behaviors and channels. A declaration only specifies the ports and interfaces, whereas a definition also contains the actual implementation. Thus, IP components can be naturally represented by a behavior or channel class, which is declared, but not defined.

6.1.1 Behavior IP

Computational IP components are specified as behavior declarations. A typical declaration consists of the name of the behavior and the number and the type of its ports. For example:

```
behavior IP1(in  int          p1,
             in  bit[255:0]   p2,
             out bit[127:0]   p3);
```

This specifies an IP component `IP1` with three ports, `p1`, `p2` and `p3`. `p1` and `p2` are input ports of integer and bit vector type, respectively, and `p3` is a 128 bit wide output port. Since no behavior body is defined, this declares the component `IP1` as a black box whose internal structure is unknown. Please note, however, that this declaration is sufficient in order to instantiate a component of type `IP1` in a design.

Since the internals of such IPs are unspecified, it is necessary for the IP provider to supply additional information, for example estimation data, together with the IP declaration. This can be done easily with annotations. For example:

```
note IP1.Version  = 1.2;
note IP1.Area     = 118000;
note IP1.ExecTime = 42.5;
note IP1.Power    = 0.32;
```

6.1.2 Channel IP

Channels can be used to specify communication IP, for example, proprietary communication protocols. A typical channel declaration consists of the name of the channel and the list of the implemented interfaces which have to be defined first.

For example, a channel IP2, which implements two interfaces with send and receive methods for bytes and words of data, can be defined as follows.

```
typedef bit[ 7:0] byte;
typedef bit[63:0] word;

interface I1
{
  void SendByte(byte B);
  byte ReceiveByte(void);
}

interface I2
{
  void SendWord(word W);
  word ReceiveWord(void);
}

channel IP2 implements I1, I2;
```

Again, this IP declaration does not reveal anything about the actual implementation of the protocol, but still allows to instantiate and use channels of IP2 type. Also, the channel IP2 and its interfaces I1 and I2 could be annotated in the same way as the IP component IP1 in Section 6.1.1.

It should be mentioned that, as defined in Chapter 4, the SpecC language allows ports and interfaces for both, behaviors and channels. The assumption, that behaviors have ports and channels have interfaces, is made in this chapter simply for easier understanding. Please note that this makes no difference to the applicability of the IP protection mechanism discussed in this chapter.

6.2 Secret IP Implementation

As mentioned before, the implementation of an IP behavior or IP channel is supplied as a precompiled library. In order to build such a library, the IP provider first specifies the IP implementation (or an accurate simulation model) as a class definition according to the IP declaration. Then, this SpecC source code will be compiled by the SpecC compiler in order to create the library. For example, for the behavior IP1 in Section 6.1.1, a shared library `libIP1.so` will be created.

However, the generation of such a library is not trivial because of the way behaviors and channels are implemented in the SpecC system.

6.2.1 Implementation problem

From the source code, the SpecC compiler first generates intermediate C++ code. Then, this C++ code can be compiled by a standard C++ compiler in order to produce the shared library required for the IP.

In the SpecC compiler, behaviors and channels are implemented as C++ classes, and behavior and channel instances are naturally represented by C++ objects. Among other reasons, which are beyond the scope of this chapter, this implementation was chosen because it keeps the generated code very similar to the original SpecC code and, thus, significantly simplifies source-level debugging of SpecC programs, as explained in Section 5.3.

For example, the following code defines a behavior `B` which consists of two child behaviors `b1` and `b2` connected by a channel `c1` and a variable `v1`. Right next to the SpecC code, a fragment of the generated C++ code for this behavior definition is shown.

```
behavior B(
        in   int p1,
        out  int p2)
{
  int     l1;
  C       c1;
  B1      b1(p1, l1, c1);
  B2      b2(p2, l1, c1);

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

```
class B : public _behavior
{ int     &p1;
  int     &p2;

  int     l1;
  C       c1;
  B1      b1;
  B2      b2;

  public:
  void main(void);

  B(int &p1, int &p2);
  virtual ~B(void);
};
```

In C++, in order to instantiate a class, the size of the class must be known so that sufficient memory can be allocated for the new object before the constructor of the class is called to initialize the memory. While the constructor is provided in the class itself, the memory must be allocated by the instantiator. C++ semantics [ES90] enforce that a class is defined (not just declared) before it can be instantiated. This ensures that the size of the required memory is known when an object of a class is created.

In the case of an IP component, which is supplied in a library, the size of the class still must be known by the user code. Therefore, in the C++ user code, a class declaration as in SpecC, is not sufficient. Instead, a class definition is required. This is a problem for the IP user because he does not know the internals of the IP class and thus cannot create a proper class definition.

6.2.2 Implementation solution

The problem can be solved if the size of the class is known. With this information, the IP user can create a *pseudo class* which only contains known contents and leaves enough space for the secret internals. In particular, this pseudo IP class consists of the known ports, the public interfaces and sufficient space reserved for the secret parts of the IP.

For example, a pseudo class for the behavior `B` discussed above can be defined as follows.

```
behavior B(                     class B : public _behavior
        in  int p1,             { int    &p1;
        out int p2)               int    &p2;
{
  int     l1;                     char   Reserved[X];
  C       c1;
  B1      b1(p1,l1,c1);
  B2      b2(p2,l1,c1);

  void main(void)                 public:
  {                               void main(void);
   par{ b1.main();
        b2.main(); }              B(int &p1, int &p2);
  }                               virtual ~B(void);
};                              };
```

In this pseudo class, the dummy array `Reserved[X]` replaces the internal IP components `l1`, `c1`, `b1`, and `b2`. The size `X` of the reserved array must be equal to (or greater than) the size of all the replaced components.

Please note that such a class replacement is highly compiler dependent because the C++ language leaves some freedom for the implementation of classes [Str97]. Therefore, when this approach is implemented, it must be integrated with the compiler being used.

With this solution, the IP component can be used just as any other component, given that the reserved size `X` is provided along with the component declaration and the IP library.

The value of `X` can be computed by the IP provider from the IP implementation. The reserved size basically is the sum of the sizes of the local variables, the instantiated channels and child behaviors, plus any implementation dependent overhead.

More formally, the size of an IP class C is computed as

$$\text{sizeof}(C) = X_{public} + X_{secret}$$

where

$$\begin{aligned} X_{public} &= \sum_{p \in Ports(C)} \text{sizeof}(p) + \sum_{i \in Interfaces(C)} \text{sizeof}(i) \\ X_{secret} &= \sum_{l \in Locals(C)} \text{sizeof}(l) + \sum_{c \in Channels(C)} \text{sizeof}(c) \\ &+ \sum_{b \in Behaviors(C)} \text{sizeof}(b) + \Delta \end{aligned}$$

Here, Δ represents the implementation dependent size needed for base classes, data alignment, etc.

Please note that, although the equation is recursively defined, it can be easily computed by the SpecC compiler because language semantics require that the sizeof() operator can always be evaluated at compile time.

6.3 Integration with the SpecC compiler

The approach for IP protection described in this chapter has been implemented and integrated with the SpecC compiler `scc`, which was presented in Section 5.3.

In order to support IP, the SpecC compiler has been extended with an *IP mode* (enabled by option `-ip`) which changes the behavior of the exporter, the deparser, the translator and the underlying C++ compiler and linker (please refer to Figure 5.5 on page 122).

In IP mode, the compiler recognizes special annotations (`scc_Public`) which the user attaches to behaviors and channels to mark them as IPs with public ports and interfaces. All objects not marked public will be treated as secret implementation by the compiler and will be hidden in the output.

In particular, the exporter and the deparser will only generate code for the public objects. All other objects will be omitted. From the implementation of an IP, the IP provider can use this to automatically generate the files describing the public interfaces of the IP.

Furthermore, when these public files are generated, the behavior and channel declarations of IP components will be automatically annotated with the reserved IP size (`scc_ReservedSize`), as discussed in Section 6.2. This annotation will later be used by the IP user as the value `X` in the IP pseudo classes, which are generated by the compiler when the IP component is instantiated.

The compilation flow is also affected by the IP mode. When generating C++ code, the SpecC compiler ensures that only objects marked public will have external linkage. In other words, all non-public objects will have internal linkage and are therefore not visible outside the file scope.

Furthermore, in IP mode, the underlying C++ compiler and the linker are instructed to create a shared library instead of an executable file. In the library, all symbols internal to the IP are stripped off. This ensures that the symbol table in the library is minimal and does not reveal any internal methods of the IP.

In summary, using the IP mode, the IP provider can automatically create the public IP interface and the IP library while being sure that no information about the secret implementation will be available to the IP user. On the other hand, the IP user can simply include the annotated interface declarations in his design and use the IP components just as his own behaviors and channels by linking his executable file against the provided IP libraries.

6.4 Experiments and Results

The IP support of the SpecC system has been successfully tested with a set of design examples. First, a simple example using different RT level components as IPs is presented. Then, the SpecC IP protection scheme is applied to several industrial-size examples at the system level.

6.4.1 RT level IP examples

As the first experiment, a generic adder, specified at the gate and the RT level, has been modeled as an IP component. For three different bit widths, namely 8, 16 and 32 bits, adder components have been created as a set of public IP declarations and shared libraries (see also Section B.2).

Adder example	Internal components	Reserved size
RTL model, 8 bit	1	12
RTL model, 16 bit	1	12
RTL model, 32 bit	1	16
Gate model, 8 bit	65	2428
Gate model, 16 bit	131	5020
Gate model, 32 bit	261	10052

Table 6.1: RT level IP examples

For each generated adder, Table 6.1 shows the number of the hidden, internal components and the minimum reserved size `X`. It is obvious that the RTL models are much less complex than the models composed of logic gates.

Please note that, in order to not reveal the complexity of the IP implementation through these numbers, the IP provider is free to choose any number greater than the minimum size computed by the compiler. For example, the reserved size 12000 works well for all the adders.

Using the IP-enabled SpecC compiler, a public interface and a shared library have been automatically created to allow the adders being used as IP components. For example, the public interface generated for the 32 bit adder is shown next.

```
/////////////////////////////////////////////////////////
// SpecC source code generated by SpecC V2.0.4
// Design: ADD32_GTL
// File:   ADD32.sc
// Time:   Thu Jun 17 15:46:30 1999
/////////////////////////////////////////////////////////

behavior ADD32( in bit[0:0] c_in,
                in bit[31:0] a,
                in bit[31:0] b,
                out bit[31:0] s,
                out bit[0:0] c_out);

note ADD32.BitWidth = 32;
note ADD32.scc_ReservedSize = 10052u;

/////////////////////////////////////////////////////////
```

6.4.2 System level IP examples

Four system-level designs have also been modeled as IP components. The examples consist of two controller components, namely an elevator controller and a traffic light controller (see Section B.4), and two data compression IPs, namely a JPEG encoder (see Section B.5) and a GSM vocoder (see Section B.6).

Table 6.2 shows the characteristics of the IP models. Again, the number of internal components hidden in the IP, and the reserved size `X` for each IP are listed.

Considering the complexity of these designs (for example, the GSM vocoder consists of about 13.000 lines of SpecC source code [GZG+99]), these results show that the IP approach implemented in the SpecC system works very well with large IP models at the system level.

Although the system level components are internally much more complex, the

IP example	Internal components	Reserved size
Elevator controller	91	4248
Traffic light ctrlr.	24	892
JPEG encoder	4	2728
GSM vocoder	84	12020

Table 6.2: System level IP examples

public IP interface is as simple as the adder interface shown earlier. For example, the interface declaration of the GSM vocoder IP is shown next.

```
/////////////////////////////////////////////////////
// SpecC source code generated by SpecC V2.0.4
// Design: GSM_Vocoder
// File:   GSM_Vocoder_public.sc
// Time:   Tue Jun 22 10:42:36 PDT 1999
/////////////////////////////////////////////////////

behavior GSM_Vocoder(
                    in bit[12:0] Sample,
                    out unsigned bit[243:0] Frame,
                    in bool DTX_Mode,
                    out unsigned bit[5:0] DTX_Ctrl,
                    in event NewSample,
                    out event FrameReady );

note GSM_Vocoder.Comment = "GSM EFR speech encoder";
note GSM_Vocoder.Version = "GSM 06.60";
note GSM_Vocoder.scc_ReservedSize = 12020u;

/////////////////////////////////////////////////////
```

Chapter 7

Conclusion

The increasing complexity of SOC design requires higher design effort, more efficient tools and new methodologies. Due to market pressures, increasing the design time is not an option.

System-level design reduces the complexity of the SOC design process by raising the level of abstraction. In addition, system-level design takes advantage of the reuse of pre-designed, complex components, called IPs. In order to enable the reuse of IP components, IP must become an integral part of the system design methodology. In particular, IP reuse must be supported by the design language, the design models, the methodology, and the tools used in the design process.

In this work, the SpecC approach to system-level design with explicit support of IP reuse has been presented. The SpecC approach is based on an IP-centric design model, an IP-centric design methodology, and the SpecC language which has been specifically developed for the purpose of embedded systems design.

7.1 Contributions

The contributions of this work are summarized in the following sections.

7.1.1 IP-centric model

The SpecC model meets the goals and requirements of system-level design. It is suitable to represent abstract properties of the intended system in early stages of the design process, as well as specific and detailed design characteristics later in the implementation.

As described in Chapter 2, the SpecC design model consists of a hierarchical network of behaviors and channels. In this model, computation and communication are clearly separated in the way that the behaviors contain the computation and functionality, whereas the channels encapsulate the communication in the system. This separation is essential in order to support IP reuse.

The support of IP is a major benefit of the SpecC model. IP components are integrated in a SpecC design model the same way as any other components in the system. Moreover, IP components can be easily inserted or replaced in the system model, at any time in the design process. In other words, the SpecC model is IP-centric, as it allows "plug-and-play" with IP components.

For design specification with SpecC, modeling guidelines have been set up in Chapter 2. Following these guidelines will ensure that a design model is well-defined. A well-defined SpecC model will work well with the SpecC tool set, since it is synthesizable, supports IP, and in particular meets the requirements of the SpecC methodology.

Well-defined composite behaviors, supporting sequential, concurrent, pipelined, FSM-style and exceptional execution, are organized hierarchically, forming structural and behavioral hierarchy in the system model. At the lowest level in the hierarchy, the leaf behaviors, specified as arbitrary algorithmic programs, and the IP behaviors, whose internal composition is hidden, represent the smallest indivisible units in the model.

Communication is modeled by use of explicitly connected shared variables, or by channels, which also support hierarchy. Channels encapsulate the communication protocols, hiding the details of the communication, while providing an abstract, high-level interface to the connected behaviors. This encapsulation mechanism is exploited specifically for the wrapper concept used with IPs.

7.1.2 IP-centric methodology

Based on the modeling guidelines defined for SpecC design models, the SpecC design methodology has been presented in Chapter 3. The SpecC methodology is IP-centric and features a set of well-defined design models and well-defined refinement tasks, which transform an abstract, executable specification of the design into a detailed implementation.

The SpecC methodology consists of a horizontal validation flow, allowing simulation, estimation and analysis at any abstraction level. In addition, the vertical exploration and synthesis flow refines the initial design specification in several steps into a final implementation architecture ready for manufacturing.

In particular, the synthesis flow is based on four well-defined models, namely the specification model, the architecture model, the communication model, and the implementation model.

The specification model is the most abstract model in the design flow. It contains an accurate description of the final implementation only in terms of the functionality. The next model, the architecture model, adds the structure of the final system to the model, so that it accurately reflects the target architecture. Then, the communication model mirrors the communication performed in the final system in an bit-exact and cycle-exact manner. Finally, the implementation model refines the internal structure of the components in the model, allowing clock-cycle accurate simulation of the implemented design.

The four models clearly specify the input and output of the tasks in the design flow. In other words, the four models serve as a detailed specification for the tools in the SpecC design environment. This applies in particular to architecture exploration and communication synthesis, which have been described with their intermediate refinement steps by use of detailed examples.

Architecture exploration includes the traditional tasks of architecture allocation, hardware/software partitioning and system-level scheduling. After the target architecture has been selected, architecture exploration maps the specification model onto the allocated architecture by assigning behaviors to processing elements, variables to memories and channels to the system busses.

Then, communication synthesis refines the architecture model into the communication model, performing protocol selection, transducer insertion and protocol synthesis. Finally, the back end utilizes behavioral synthesis and software compilation to create the implementation model, providing a clear hand-off for design manufacturing.

Since the reuse of IP is integrated with the design flow, the SpecC design methodology is IP-centric. It supports the easy insertion and replacement of IP components, allowing quick design space exploration.

The SpecC methodology promises a large productivity gain and a significant reduction of design time and design costs, due to less and smaller iterations in the design process. With the SpecC methodology, the designers can focus on the design space exploration, making design decisions based on their experience. The tedious and error prone refinement tasks with the design models are performed automatically by the tools.

7.1.3 SpecC language

In Chapter 4, the requirements and objectives for system-level design languages have been discussed and identified. A language suitable for the design of embedded systems must be executable and synthesizable. Further, it must completely support software and hardware-specific concepts. More specifically, in addition to the well-known software concepts, hardware-specific concepts are required, including behavioral and structural hierarchy, concurrency, timing, synchronization, exception handling and state transitions. Finally, all these concepts should be represented in an independent and straightforward manner.

A set of traditional languages has been examined and compared against these requirements and goals. Since none of these languages satisfies all the requirements, a new language, called SpecC [DZG98], has been proposed. The SpecC language has been targeted specifically to support the identified concepts needed in embedded systems design.

The SpecC language, which has been described in detail in Chapter 4, has been developed and implemented. Compared to the set of traditional languages, SpecC is the only language that supports all the required concepts. Also, when compared to recent system-level languages, SpecC turns out to be a superior specification and modeling language[1].

Built on top of the ANSI-C language, the de-facto standard for software development, SpecC inherits the benefits of a popular and successful software programming language. Moreover, since SpecC is a true superset of C, a large library of already existing algorithms can immediately be used. Also, the similarity with C makes it easy to learn and easy to understand for everyone familiar with the C language.

SpecC combines the features found in software and hardware design, as it is based on a software language and adds all concepts needed for hardware models. In particular, the SpecC language contains special constructs to represent the needed hardware concepts, including communication, concurrency, hierarchy, synchronization, exception handling, state transitions and timing.

The SpecC language provides a complete set of constructs which, at the same time, is also minimal. SpecC maps the modeling concepts onto independent language constructs in a one to one fashion. As a result, SpecC precisely covers the unique requirements for embedded systems design in an orthogonal manner.

The SpecC language also encourages the reuse of IP. Directly following the IP-centric SpecC model discussed in Chapter 2, the SpecC language features "plug-and-play" support for IP components.

In summary, the contribution of this task is the development and implementation of a new specification and modeling language, called SpecC, which precisely covers the requirements for the design of embedded systems. The SpecC language satisfies all the requirements and goals, as is executable and synthesizable, and supports all hardware- and software-specific concepts needed for modeling embedded systems.

[1] A comparison of the SpecC language with the Scenic approach, which has recently been renamed to SystemC, can be found in [DG98]. Further, a comparison with VHDL+, an extension of VHDL, can be found in [GZG98].

7.1.4 SpecC design environment

The SpecC methodology and the SpecC language have been implemented in the SpecC design environment, which has been described in Chapter 5. The SpecC design environment consists of a set of CAD tools for system validation, analysis, and synthesis, integrated in a graphical user interface (GUI).

Since the SpecC design environment, the SpecC models, the SpecC language and the SpecC methodology have been developed concurrently and consistently, they represent a coherent system. The SpecC language matches the SpecC model, and the implemented programs reflect the SpecC methodology. All the components forming the SpecC design environment are designed and tuned for the specific requirements and goals of system-level design.

The validation flow of the design environment has been implemented with the SpecC release 2.0.4. This release has been made freely available on the world-wide web (WWW) and is currently in evaluation and use in academia and industry. The release includes the tools developed by the author of this work, in particular, the SpecC compiler, the SpecC Internal Representation, a profiler and a tool set.

7.1.4.1 SpecC Internal Representation

The SpecC Internal Representation (SIR) is the central design representation used by all SpecC tools for input, output, access and modification of SpecC design models.

The SIR is a complex data structure, embedded in a comprehensive, well-defined and well-documented API. As such, the SIR provides an abstraction layer above the specific details of the SpecC language.

The benefit of the SIR as a common design representation is that new tools can be developed very quickly, which has been proven with the implementation of a set of profiling tools. The SIR provides all required functions to access the design model. Without the SIR library, the development and implementation of such functions would require a significant amount of time. With the SIR, the SpecC tool developer can focus solely on the algorithms of his task.

In conclusion, the SIR and its API provide a solid base for the quick development of new tools for the SpecC design environment.

7.1.4.2 SpecC compiler

The SpecC compiler is the main tool in the validation flow of the SpecC methodology. Its main purpose is the generation of an executable simulation model from a SpecC design model, at any stage in the design flow. The SpecC compiler also serves as a converter between the different file formats used in the SpecC design environment.

Together with the SpecC simulation library, the SpecC compiler essentially satisfies the requirement of executability for any SpecC design model. Hence, it enables dynamic validation and analysis of the design model, simply by execution on the host computer.

The SpecC compiler provides special support for debugging and profiling. As a result, any standard C/C++ debugger can be used for debugging SpecC programs, furnishing the SpecC simulation with single-stepping, break points, and data inspection capabilities.

Finally, the SpecC compiler has been extended to provide automatic IP protection, as summarized in the next section.

7.1.5 IP protection

IP reuse and IP protection have been implemented in the SpecC design environment. In particular, the SpecC compiler has been extended in order to support the recognition, the use and the generation of IP components.

For IP protection, the SpecC compiler allows the automatic creation of public IP interfaces and secret IP libraries from the IP source code. Using the implemented IP mode, the IP provider can automatically create the public interface and the IP library, being sure that no information about his secret implementation will be available to the IP user. On the other hand, the IP user can simply include the IP interface declaration in his design model and use the IP component just as any other behavior or channel. For simulation, the IP user simply links his executable file against the provided IP library.

With the SpecC IP protection, any IP is fully protected against reverse-engineering, and the use of IPs is just a matter of "plug-and-play".

7.1.6 Experience

Using the SpecC design environment, the IP-centric methodology has been successfully applied to several designs of industrial size.

In Appendix B, a set of example designs is listed, which have been modeled according to the SpecC modeling guidelines, and have been specified with the SpecC language. After successful compilation, simulation and debugging, the SpecC methodology has been manually applied to a subset of the examples, generating detailed implementation models.

As a result, the SpecC approach has been proven with real-world examples, including a JPEG encoder [CPC+99] and a GSM vocoder [GZG+99].

7.1.7 Impact

As of today, the SpecC approach is evaluated and already in use in academia and industry. The SpecC methodology and the SpecC language have gained wide acceptance, in particular, in the industry.

Recently, the SpecC language has been proposed as a standard system-level language for adoption in industry by some of Japan's top-tier electronics and semiconductor companies [CGC+99].

In conclusion, the SpecC approach presented in this work has a significant impact on the future of SOC design and the deep sub-micron era.

7.2 Future Work

In addition to support and maintenance of the current SpecC design environment, future work will focus on the SpecC language and the implementation of the SpecC synthesis flow.

7.2.1 SpecC language

The experience with the real use of the SpecC language has shown that minor adjustments and some extensions are desirable in order to make the language more convenient. These issues, which have been outlined in Section 4.14, need to be addressed in a possible new release of the SpecC language.

At the same time, future work will emphasize on the standardization of the SpecC language.

7.2.2 Synthesis flow

For the synthesis flow, efficient algorithms need to be developed and implemented in order to support the system designer with the refinement of the design models.

In particular, the tasks of architecture exploration and communication synthesis require research on their algorithms, and the implementation of automated tools.

Appendix A

SpecC Users Manual

For quick reference, the manual pages of the SpecC programs and tools, developed and implemented for this work, are listed in the following sections.

A.1 SpecC Compiler `scc`

NAME

scc – SpecC Compiler

SYNOPSIS

scc *–h*

scc *design* [*command*] [*options*]

DESCRIPTION

scc is the compiler for the SpecC language. The main purpose of **scc** is to compile a SpecC source program into an executable program for simulation. Furthermore, **scc** serves as a general tool to translate SpecC code from various input to various output formats which include SpecC source text, SpecC binary files in SpecC Internal Representation format, and other compiler intermediate files.

Using the first command syntax as shown in the synopsis above, a brief usage information and the compiler version are printed to standard output and the program exits. Using the second command syntax, the specified *design* is compiled. By default, **scc** reads a SpecC source file, performs preprocessing and builds the SpecC Internal Representation (SIR). Then, C++ code is generated, compiled and linked into an executable file to be used for simulation. However, the subtasks performed by **scc** are controlled by the given *command* so that, for example, only partial compilation is performed with the specified *design.*

On successful completion, the exit value 0 is returned. In case of errors during processing, an error code with a brief diagnostic message is written to standard error and the program execution is aborted with the exit value 10.

For preprocessing and C++ compilation, **scc** relies on the availability of an external C++ compiler which is used automatically in the background. By default, the GNU compiler **gcc/g++** is used.

ARGUMENTS

design specifies the name of the design; by default, this name is used as base name for the input file and all output files;

COMMAND

The *command* has the format - *suffix1* 2 *suffix2,* where *suffix1* and *suffix2* specify the format of the main input and output file, respectively. This command also implies the compilation steps being performed. By default, the command –sc2out is used which specifies reading a SpecC source file (e.g. design.sc) and generating an executable file (e.g. a.out) for simulation. All necessary intermediate files (e.g. design.cc, design.o) are generated automatically.

Legal command suffixes are:

sc SpecC source file (default: *design.sc)*

si preprocessed SpecC source file (default: *design.si)*

sir binary SIR file in SpecC Internal Representation format (default: *design.sir)*

cc C++ simulation source file (default: *design.cc)*

h C++ simulation header file (default: *design.h)* *cch* both, C++ simulation source file and C++ header file (default: *design.cc* and *design.h)*

o linker object file (default: *design.o)*

out executable file for simulation (default: *design);* however, with the –ip option, a shared library will be produced (default: *libdesign.so)*

OPTIONS

–v | *–vv* | *-vvv* increase the verbosity level so that all tasks performed are logged to standard error (default: be silent); at level 1, informative messages for each task performed are displayed; at level 2, additionally input and output file names are listed; at level 3, very detailed information about each executed task is printed;

–w | *–ww* | *-www* increase the warning level so that warning messages are enabled (default: warnings are disabled); four levels are supported ranging from only important warnings (level 1) to pedantic warnings (level 4); for most cases, warning level 2 is recommended (–ww);

–g enable debugging of the generated simulation code (default: no debugging code); this option disables optimization;

–O enable optimization of the generated simulation code (default: no optimization); this option disables debugging;

–ip enable intellectual property (IP) mode; when generating a SIR binary or SpecC text file, only declarations of symbols marked public will be included (the public interface of an IP is created); when generating C++ code, non-public symbols will be output so that they will be invisible outside the file scope; when compiling or linking, the compiler and linker are instructed to create a shared library instead of an executable file (creation of an IP simulation library);

–sl suppress source line information (preprocessor directives) when generating SpecC or C++ source code (default: include source line directives);

–sn suppress all annotations when generating SpecC source code (default: include annotations);

–i input file specify the name of the input file explicitly (default: *design.suffix1);* the name '-' can be used to specify reading from standard input;

–o output file specify the name of the final output file explicitly (default: *design.suffix2);* the name '-' can be used to specify writing to standard output;

–D do not define any standard macros; by default, the macro __SPECC__ is defined automatically (it is set to 1); furthermore, implementation dependent macros may be defined; this option suppresses the definition of all these macros;

–Dmacrodef define the preprocessor macro *macrodef* to be passed to the preprocessor;

–U do not undefine any macros; by default, few macros are undefined automatically (in order to allow C/C++ standard header files to be used); this option is implementation dependent;

–Uundef undefine the preprocessor macro *undef* which will be passed to the preprocessor as being undefined; the macro *undef* will be undefined after the definition of all command-line macros; this allows to selectively suppress macros from being defined in the preprocessing stage;

–I clear the standard include path; by default, the standard include path consists of the directory $SPECC/inc; this option suppresses the default include path;

–Idir append *dir* to the include path (extend the list of directories to be searched for including source files); include directories are searched in the order of their specification; unless suppressed by option –I, the standard include path is automatically appended to this list; by default, only the standard include directories are searched;

–L clear the standard library path; by default, the standard library path consists of the directory $SPECC/lib; this option suppresses the default library path;

–Ldir append *dir* to the library path (extend the list of directories to be searched for linker libraries); the library path is searched in the specified order; unless suppressed by option –L, the standard library path is automatically appended to this list; by default, only the standard library path is searched;

–l when linking, do not use any standard libraries; by default, the standard libraries libbit, libsim, libqt, and libprof are used for linking the executable file; this option suppresses linking against theses standard libraries;

–llib pass *lib* as a library to the linker so that the executable is linked against *lib;* libraries are linked in the specified order; unless suppressed by option –l, the standard libraries are automatically appended to this list; by default, only standard libraries are used;

–P reset the import path; clear the list of directories to be searched for importing binary files; by default, only the current directory is searched; this option suppresses this standard import path;

–Pdir append *dir* to the import path (extend the list of directories to be searched for importing binary files); import directories are searched in the order of their specification; unless suppressed by option –P, the standard search path is automatically appended to this list; by default, only the standard import path is searched;

–xpp preprocessor_call redefine the command to be used for calling the C preprocessor (default: "g++ -E -x c %p %i -o %o"); in the specified string, every occurence of %p will be replaced with a preprocessor option; additional options will be appended; also, %i and %o will be replaced automatically with the actual input and output filename, respectively;

–xcc compiler_call redefine the command to be used for calling the C/C++ compiler (default: "g++ -c %c %i -o %o"); in the specified string, every occurence of %c will be replaced with a compiler option; additional options will be appended; also, %i and %o will be replaced automatically with the actual input and output filename, respectively;

–xld linker_call redefine the command to be used for calling the linker (default: "g++ %i -o %o %l"); in the specified string, every occurence of %l will be replaced with a linker option; additional options will be appended; also, %i and %o will be replaced automatically with the actual input and output filename, respectively;

–xp preprocessor_option pass an option directly to the C/C++ preprocessor; for every %p in the preprocessor call (see above), an option has to be specified (default: none);

–xc compiler_option pass an option directly to the C/C++ compiler; for every %c in the compiler call (see above), an option has to be specified (default: none);

–xl linker_option pass an option directly to the linker; for every %l in the linker call (see above), an option has to be specified (default: none);

ENVIRONMENT

The environment variable SPECC is used to determine the home directory of the SpecC system where SpecC standard include files and SpecC system libraries are located.

ANNOTATIONS

The following SpecC annotations are recognized by the compiler:

scc_ReservedSize for external behaviors and channels (IP components), this indicates the size reserved in the C++ class for internal use; the annotation type is unsigned int; if found at class definitions, this annotation is checked automatically for reasonable values; for IP declarations, the

	annotation can be created automatically with the –ip option;
scc_Public	for global symbols, this annotation indicates whether the symbol is public and will be visible in a shared library; the annotation type is bool; this annotation only is recognized with the –ip option;

VERSION

The SpecC compiler **scc** is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

SEE ALSO

gcc(1), **g++**(1), **sprof**(l), **sir_tools**(l)

BUGS, LIMITATIONS

Semantic type checking of certain expressions is not fully implemented.

A.2 SpecC Profiler `sprof`

NAME

sprof – SpecC Profiler

SYNOPSIS

sprof *-h*

sprof *command* [*options*] *design_in design_out*

DESCRIPTION

sprof is the profiler of the SpecC system. Profiling of SpecC programs consists of three phases. First, a design is instrumented by the profiler with a set of counters. These counters are incremented by counting statements which the profiler inserts at the beginning of each function and class method. Also, the main method of the behavior Main is instrumented with a function call to the profiling run-time library, so that profiling is enabled when the design is simulated.

Second, an executable profiling model of the design is created with the SpecC compiler. Each time the profiling model is executed, the number of executions for each function and each method are counted. The profiling counters are stored in a file, called profile of the design. This file is read whenever the execution of a profiling model starts and is written when the execution ends.

The third profiling phase consists of back-annotation of the counter values from the profile to the design and de-instrumentation of the design. This is also performed by the SpecC profiler.

Using the first command syntax shown in the synopsis above, a brief usage information including the profiler version is printed to standard output. Using the second command syntax, the profiling task specified with *command* is performed. For all tasks, **sprof** reads the SpecC design file specified with *design_in,* performs the specified task and then writes the modified design into a new file specified with *design_out.* Both design files are binary files containing the SpecC Internal Representation (SIR) of the design. The SpecC compiler **scc** may be used to convert the binary SIR files into readable source code (and vice versa).

On successful completion, **sprof** returns the exit value 0. In case of errors, an error code with a brief diagnostic message is written to standard error and the program execution is aborted with the exit value 10.

COMMAND

The profiler is controlled by the given *command* which is one of *+i* , *–i* , *+b* , *–b* .

+i instrument the design with counters and counting statements for profiling;

–i de-instrument the design; remove all inserted profiling counters and counting statements;

+b back-annotate the counter values from the profile to the instrumented design in form of annotations;

–b remove the back-annotated profile from the design (remove all profiling annotations);

OPTIONS

–a when back-annotating (command *+b),* add the counter values from the profile to the current annotated values (default: current profiling annotation must not exist);

–h print a short usage and version information and then quit;

–v enable verbosity mode; all tasks performed are logged to standard error;

–i input file specify the input SIR file explicitly; the name '-' can be used to specify reading from standard input (default: *design_in* with suffix .sir);

–o output file specify the output SIR file explicitly; the name '-' can be used to specify writing to standard output (default: *design_out* with suffix .sir);

–p profile specify the file name for the profile explicitly (default: specc_profile);

ARGUMENTS

design_in specifies the name of the input design; by default, this name is used as base name for the input file;

design_out specifies the name of the output design; by default, this name is used as base name for the output file;

ANNOTATIONS

The following SpecC annotations are recognized by the profiler:

sprof_Instrumented a global annotation of type bool, indicating that the design has been instrumented by the profiler;

sprof_Profiled a global annotation of type bool, indicating that the design already has been profiled;

sprof_ExecCountIndex for every function or method, this annotation indicates the index of its counter in the global counter array; the annotation type is unsigned int; this annotation is used only in an instrumented design;

sprof_ExecCount for every function or method, this annotation specifies the number of executions during profiling; the annotation type is unsigned int; this annotation in created as the result of profiling;

VERSION

The SpecC profiler **sprof** is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

SEE ALSO

scc(l), **sir_tools**(l)

BUGS, LIMITATIONS

Advanced profiling features such as support for call graphs, etc. are not supported. However, standard C profiling tools can be used instead.

A.3 SpecC Tool Set

For the SpecC system, several tools have been developed and implemented, which directly work with binary SIR files. With these tools, it is not necessary to convert given SIR files to text files in order to look up information about their contents or to apply simple changes.

A.3.1 `sir_delete`

NAME

sir_delete – part of the SpecC SIR tool set

SYNOPSIS

sir_delete [*options*] *design* [*object_name...*]

DESCRIPTION

sir_delete allows to delete objects in a SIR file. A SIR file is a binary file containing the SpecC Internal Representation of a design. **sir_delete** reads the SIR file specified with *design* and deletes all objects specified with the *object_name* list. When done, **sir_delete** writes back the modified *design* into the same file, unless the –i or –o options are used.

On successful completion, the exit value 0 is returned. In case of errors, an error code with a diagnostic message is written to standard error and the program execution is aborted with the exit value 10. In this case, no output is produced, in other words, the specified *design* is left unchanged.

ARGUMENTS

design specifies the design to work with; if no –i or –o options are specified, the suffix '.sir' will be appended to this name in order to obtain the SIR file to read and write, respectively;

object_name specifies the symbol to be deleted; for global symbols, *object_name* is simply the symbol name; for class members and methods, *object_name* is the class name followed by a '.' and the member or method name; for local symbols in functions and methods, the same syntax is used, the symbol name follows after a '.' appended to the function or method specifier;

OPTIONS

–h prints a short usage and version information and then quits;

–i input file specifies the name of the input file explicitly; the name '-' can be used to specify reading from standard input;

–o output file specifies the name of the output file explicitly; the name '-' can be used to specify writing to standard output;

–v enables verbosity mode; all actions performed are logged to standard error;

VERSION

The SpecC SIR tool set is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

SEE ALSO

scc(l), **sir_list**(l), **sir_note**(l), **sir_rename**(l), **sir_strip**(l), **sir_tree**(l)

BUGS, LIMITATIONS

sir_delete can only delete global symbols and local symbols at class level or function/method level. Symbols defined locally within compound statements or user-defined types are not accessable due to the limited syntax used for *object_name.*

A.3.2 `sir_list`

NAME

sir_list – part of the SpecC SIR tool set

SYNOPSIS

sir_list [*options*] *sir_file* [[*options*] *sir_file...*]

DESCRIPTION

sir_list lists the contents of one or more SIR files. A SIR file is a binary file containing the SpecC Internal Representation of a design. For each specified *sir_file*, **sir_list** reads the SIR file and prints a list of the global and local symbols contained in the file to standard output, along with additional information depending on the *options* given. The symbols are listed in alphabetical order.

On successful completion, the exit value 0 is returned. In case of errors, an error code with a diagnostic message is written to standard error and the program execution is aborted with the exit value 10.

ARGUMENTS

sir_file specifies the SIR file whose contents will be listed; if *sir_file* does not exist, the suffix '.sir' will be appended; the name '-' can be used to specify reading from standard input;

OPTIONS

–a enables printing of all symbol lists (equivalent to +BCDFINPSV);

–c lists behaviors, channels and interfaces only; this ist the default (equivalent to +BCI –DFNPSV);

–h prints a short usage and version information and then quits;

–l specifies a long listing; for each symbol, a set of flags (as defined below) is listed;

–r recursively lists the contents of behaviors, channels and interfaces (instantiated behaviors and channels, local variables and methods);

–t prints the type information with each symbol;

–v specifies verbosity mode; for each sub-list, a section header is printed;

–x includes external definitions in the lists; external definitions are declarations of functions, classes without body and variables of storage class extern;

+B | *–B* specifies whether to include or exclude the list of behaviors;

+C | *–C* specifies whether to include or exclude the list of channels;

+D | *–D* specifies whether to include or exclude the design name;

+F | *–F* specifies whether to include or exclude the list of functions;

+I | *–I* specifies whether to include or exclude the list of interfaces;

+N | *–N* specifies whether to include or exclude the list of annotations for each listed symbol;

+P | *–P* specifies whether to include or exclude the list of imported files;

+S | *–S* specifies whether to include or exclude the list of source files;

+V | *–V* specifies whether to include or exclude the list of variables;

FLAGS

With the –l option, a set of flags is printed with each symbol. From the flags, the symbol class, the storage class and the symbol classification can be determined; the flags are defined as follows:

symbol type is one of [BCDFINPSVbcfv], indicating behavior (B), channel (C), design (D), global function (F), interface (I), annotation (N), import file (P), source file (S), global variable (V), behavior instance (b), channel instance (c), class method (f), or class variable (v);

storage class is intern or extern (one of [ix]), indicating internal definition (i), or external declaration (x);

classification is one of [acefhilnoprstwx], indicating for behaviors: concurrent (c), FSM (f), leaf (l), pipeline (p), sequential (s), exception (t), external (x), or other (o); for channels: leaf (l), hierarchical (h), wrapper (w), external (x), or other (o); for interfaces: internal (i) or external (x); otherwise storage class: auto (a), extern (e), none (n), register (r), static (s), typedef (t), or piped (p followed by the number of pipe stages);

VERSION

The SpecC SIR tool set is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

SEE ALSO

scc(l), **sir_delete**(l), **sir_note**(l), **sir_rename**(l), **sir_strip**(l), **sir_tree**(l)

BUGS, LIMITATIONS

sir_list can only list global symbols and symbols at class level (with option –r). Symbols and annotations defined locally within compound statements or user-defined types are not included.

A.3.3 `sir_note`

NAME

sir_note – part of the SpecC SIR tool set

SYNOPSIS

sir_note [*options*] *design* [*object_name*] [*annotation...*]

DESCRIPTION

sir_note allows to annotate objects in a SIR file. A SIR file is a binary file containing the SpecC Internal Representation of a design. **sir_note** reads the SIR file specified with *design* and, when done, writes back the modified *design* into the same file, unless the -i or -o options are used.

sir_note annotates the object specified with *object_name* or, if no *object_name* is specified, annotates the *design* itself with global annotations. For each *annotation* that is specified, **sir_note** attaches, modifies or removes the annotation, depending on whether such an annotation already exists and a new value is specified.

On successful completion, the exit value 0 is returned. In case of errors, an error code with a diagnostic message is written to standard error and the program execution is aborted with the exit value 10. In this case, no output is produced, in other words, the specified *design* is left unchanged.

ARGUMENTS

design specifies the design to work with; if no –i or –o options are specified, the suffix '.sir' will be appended to this name in order to obtain the SIR file to read and write, respectively.

object_name specifies the symbol to be annotated; for global symbols, *object_name* is simply the symbol name; for class members and methods, *object_name* is the class name followed by a '.' and the member or method name; for local symbols in functions and methods, the same syntax is used, the symbol name follows after a '.' appended to the function or method specifier;

annotation specifies the new annotation to be attached to the specified object; syntactically, *annotation* is composed of the name of the note followed by an assignment character ('=') and optionally the new value; for the value, the standard SpecC syntax for constants applies; if no new value is given, the specified annotation will be removed;

OPTIONS

–h prints a short usage and version information and then quits;

–i input file specifies the name of the input file explicitly; the name '-' can be used to specify reading from standard input;

–o output file specifies the name of the output file explicitly; the name '-' can be used to specify writing to standard output;

–v enables verbosity mode; all actions performed are logged to standard error;

VERSION

The SpecC SIR tool set is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

SEE ALSO

scc(l), **sir_delete**(l), **sir_list**(l), **sir_rename**(l), **sir_strip**(l), **sir_tree**(l)

BUGS, LIMITATIONS

sir_note can only annotate global symbols and local symbols at class level or function/method level. Symbols defined locally within compound statements or user-defined types are not accessable due to the limited syntax used for *object_name.*

A.3.4 `sir_rename`

NAME

sir_rename – part of the SpecC SIR tool set

SYNOPSIS

sir_rename [*options*] *design_in design_out* [*object_name new_name*] [*object_name new_name...*]

DESCRIPTION

sir_rename allows to rename objects in a SIR file; a SIR file is a binary file containing the SpecC Internal Representation of a design. **sir_rename** reads the SIR file specified with *design_in* and generates a modified design in a new SIR file specified with *design_out.* For each pair *object_name* and *new_name,* **sir_rename** renames the specified object to the new name.

On successful completion, the exit value 0 is returned. In case of errors, an error code with a diagnostic message is written to standard error and the program execution is aborted with the exit value 10. In this case, no output is produced.

ARGUMENTS

design_in specifies the input design; if no -i option is specified, the suffix '.sir' will be appended in order to obtain the SIR file to read;

design_out specifies the output design; if no -o option is specified, the suffix '.sir' will be appended in order to obtain the SIR file to write;

object_name specifies the symbol to be renamed; for global symbols, *object_name* is simply the symbol name; for class members and methods, *object_name* is the class name followed by a '.' and the member or method name; for local symbols in functions and methods, the same syntax is used, the symbol name follows after a '.' appended to the function or method specifier.

new_name specifies the new name of the object; *new_name* must be a legal SpecC identifier; also, further semantic restrictions apply;

OPTIONS

–h prints a short usage and version information and then quits;

–i input file specifies the name of the input file explicitly; the name '-' can be used to specify reading from standard input;

–o output file specifies the name of the output file explicitly; the name '-' can be used to specify writing to standard output;

–v enables verbosity mode; all actions performed are logged to standard error;

VERSION

The SpecC SIR tool set is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

Copyright (c) 1998, 1999 CECS, University of California, Irvine.

SEE ALSO

scc(1), **sir_delete**(1), **sir_list**(1), **sir_note**(1), **sir_strip**(1), **sir_tree**(1)

BUGS, LIMITATIONS

sir_rename can only rename global symbols and local symbols at class level or function/method level. Symbols defined locally within compound statements or user-defined types are not accessable due to the limited syntax used for *object_name.*

A.3.5 `sir_strip`

NAME

sir_strip – part of the SpecC SIR tool set

SYNOPSIS

sir_strip [*options*] *sir_file...*

DESCRIPTION

With **sir_strip,** source location and import file information can be stripped from a SIR file. A SIR file is a binary file containing the SpecC Internal Representation of a design. **sir_strip** reads the specified SIR file, removes the source file and import file entries from the design data structure and writes the SIR file back, thus, reducing the file size.

Please note that the stripped information cannot be restored without access to the original source files. Therefore, stripping is recommended for binary files which are to be distributed without source code.

On successful completion, the exit value 0 is returned. In case of errors, an error code with a diagnostic message is written to standard error and the program execution is aborted with the exit value 10. In this case, no output is produced, in other words, the specified *sir_file* is left unchanged.

ARGUMENTS

sir_file specifies the SIR file to be stripped; if the specified file does not exist, the suffix '.sir' will be appended to the file name; the name '-' can be used to specify reading from standard input and writing to standard output, thus working as a filter;

OPTIONS

–h prints a short usage and version information and then quits;

–i disables stripping of import file entries; only source location information is removed;

–s disables stripping of source file entries; only import file information is removed;

–v enables verbosity mode; all actions performed are logged to standard error;

VERSION

The SpecC SIR tool set is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

Copyright (c) 1998, 1999 CECS, University of California, Irvine.

SEE ALSO

scc(l), **sir_delete**(l), **sir_list**(l), **sir_note**(l), **sir_rename**(l), **sir_tree**(l)

BUGS, LIMITATIONS

None.

A.3.6 `sir_tree`

NAME

sir_tree – part of the SpecC SIR tool set

SYNOPSIS

sir_tree [*options*] *sir_file* [*class_name...*]

DESCRIPTION

sir_tree graphically lists the instantiation hierarchy of behaviors and channels contained in a SIR file. A SIR file is a binary file containing the SpecC Internal Representation of a design. **sir_tree** reads the specified SIR file and prints the tree of behavior and channel instances to standard output, along with additional information depending on the options given.

If there are no class names specified, **sir_tree** automatically determines the root behaviors and channels and prints a tree for each of them. Otherwise, **sir_tree** prints a tree for each specified class name in the given order.

ARGUMENTS

sir_file specifies the SIR file whose contents will be displayed; if *sir_file* does not exist, the suffix '.sir' will be appended; the name '-' can be used to specify reading from standard input;

class_name specifies the name of a behavior or a channel whose instantiation tree will be printed;

OPTIONS

–b graphically displays the branches of the tree; otherwise, by default, simple tabulators will be used for tree indentation;

–f prints a flattened tree, in other words, no indentation will be used;

–h prints a short usage and version information and then quits;

–l specifies a long listing; for each behavior or channel, a set of flags (as defined below) is listed;

–t prints the type information with each behavior and channel;

–B excludes behaviors from being displayed;

–C excludes channels from being displayed;

FLAGS

With the –l option, a set of flags is printed with each behavior and channel. From the flags, the class type, the storage class and the class classification can be determined; the flags are defined as follows:

class type is one of [BC], indicating behavior (B) or channel (C).

storage class is intern or extern (one of [ix]), indicating internal class with known body (i), or external class with unknown body (x).

classification is one of [cfhlopstwx], indicating for behaviors: concurrent (c), FSM (f), leaf (l), pipeline (p), sequential (s), exception (t), external (x), or other (o); for channels: leaf (l), hierarchical (h), wrapper (w), external (x), or other (o).

VERSION

The SpecC SIR tool set is version 2.0.4.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

SEE ALSO

scc(l), **sir_delete**(l), **sir_list**(l), **sir_note**(l), **sir_rename**(l), **sir_strip**(l)

BUGS, LIMITATIONS

None.

Appendix B

SpecC Design Examples

Numerous design examples have been developed and successfully been used with the SpecC system, including a discrete cosine transformation (DCT) [AG98], an ATM packet filter [KZG97], a JPEG encoder [CPC$^+$99], and a GSM vocoder [GZG$^+$99]. A set of selected examples is presented in the following sections.

B.1 Tutorial Examples

The set of *tutorial* examples, as listed in Table B.1, is part of the SpecC standard distribution[1]. These small examples demonstrate specific features of SpecC and can serve as a tutorial for the SpecC language. Since all ten examples are complete and fully functional, they can be compiled with the SpecC compiler and simulated "out-of-the-box".

- `Adder.sc` describes a simple 8 bit adder built from logic gates.
- `Behaviors.sc` lists the types of SpecC behaviors as described in Section 2.3.
- `BitVectors.sc` demonstrates the use of SpecC bit vectors as defined in Section 4.3.1.2.
- `Callback.sc` contains a call-back communication between a sender and a receiver as mentioned in Section 4.6.1.
- `DataTypes.sc` lists the basic data types supported by SpecC as specified in Section 4.3.1.

[1] In the SpecC distribution, these examples can be found in the `examples/simple/` directory.

Example	Behaviors	Channels	Lines of code
`Adder.sc`	7	0	165
`Behaviors.sc`	8	0	113
`BitVectors.sc`	6	0	143
`Callback.sc`	3	1	231
`DataTypes.sc`	1	0	113
`FSM.sc`	9	0	168
`HelloWorld.sc`	1	0	23
`Notes.sc`	2	1	127
`Pipeline.sc`	6	0	132
`Timing.sc`	3	1	245

Table B.1: SpecC tutorial examples

- `FSM.sc` describes a clock-driven finite state machine as discussed in Section 4.5.1.2.
- `HelloWorld.sc` contains the famous "Hello World!" example in SpecC.
- `Notes.sc` demonstrates the use of annotations as described in Section 4.11.
- `Pipeline.sc` contains a three-stage pipeline design as presented in Section 4.5.2.2.
- `Timing.sc` demonstrates the specification of timing diagrams as discussed in Section 4.10.2.

More detailed information on these examples is contained in the distribution of the SpecC system.

B.2 Library Example

In order to demonstrate library management and IP support with SpecC, the so-called *library* example was developed. This example is also part of the SpecC standard distribution[2]. Using adders as example components, the example shows, how components from a library of gates can be composed and made available as IP components.

Please note that this example demonstrates library and IP issues at low abstraction levels, the gate and RT level. This is done only for the purpose of using

[2]In the SpecC distribution, this example can be found in the `examples/library/` directory.

well-known components, namely adders composed of gates, so that the design itself does not need any explanation. For the library and IP issues, the same principles and characteristics apply to all levels of abstraction.

The library example resembles the following scenario: An IP vendor develops a set of IP adder components based on his own (or somebody elses) gate library. In order to sell these components, he creates the public interfaces and ports of the components using the IP mode of the SpecC compiler. The IP provider also generates two simulation libraries, one RT-level library for fast simulation, and one gate-level library, which accurately models the components behavior. Furthermore, the IP provider develops a test bench for the implemented components, as well as for the IPs, in order to validate the correct functionality.

Library example	Behaviors	Lines of code
Gate library	6	95
Adder library	11	365
Test bench	2	118
Total	19	578

Table B.2: Library example

The library example consists of a total of 19 different behaviors, as shown in Table B.2.

Components	Gate level	RT level
Adder, 8 bit	65	1
Adder, 16 bit	131	1
Adder, 32 bit	261	1

Table B.3: Composition of IP library components

The example IP library consists of a total of six adder models, as shown in Table B.3. The three RT level models consist of a single behavior instance each, whereas the three gate level models are composed of a large set of gates.

The composition of the gate level adders can be illustrated by use of the hierarchy tree. The (shortened) hierarchy tree[3] of the 8 bit adder model is shown below. For more information, please consult the source code of the example.

[3]The hierarchy tree was created with the SIR tool set: `sir_tree -blt Adder/ADD08_GTL.sir`

```
B i s   behavior ADD08
B i s   |------ FA fa0
B i s   |       |------ HA ha1
B i l   |       |       |------ AND2 and
B i l   |       |       \------ XOR2 xor
B i s   |       |------ HA ha2
B i l   |       |       |------ AND2 and
B i l   |       |       \------ XOR2 xor
B i l   |       \------ OR2 or1
B i s   |------ FA fa1
B i s   |       |------ HA ha1
B i l   |       |       |------ AND2 and
B i l   |       |       \------ XOR2 xor
B i s   |       |------ HA ha2
B i l   |       |       |------ AND2 and
B i l   |       |       \------ XOR2 xor
B i l   |       \------ OR2 or1
B i s   |------ FA fa2
 ...   ...       ...
B i s   \------ FA fa7
B i s           |------ HA ha1
B i l           |       |------ AND2 and
B i l           |       \------ XOR2 xor
B i s           |------ HA ha2
B i l           |       |------ AND2 and
B i l           |       \------ XOR2 xor
B i l           \------ OR2 or1
```

B.3 Communication Examples

Two different communication schemes are demonstrated by the examples shown in Table B.4. The first example consists of a sender and a receiver component which communicate via a noisy bit channel. In order to account for transmission errors, Forward Error Correction (FEC) is applied.

Communication	Behaviors	Channels	Lines of code
Send & Receive, FEC	12	3	711
Client & Server, FIFO	5	2	271

Table B.4: Communication examples

The second example models a client-server communication where the server executes requests from the client in FIFO order. Both examples are contained in the

SpecC standard distribution[4]. For more details, please refer to [GZG98].

B.4 Controller Examples

Two controller models were developed as examples for control-dominated systems, as shown in Table B.5. The first example resembles a central elevator controller for three elevators in a building with ten floors. The second example specifies a controller for a traffic light at a road junction. Again, both examples are contained in the SpecC standard distribution[5].

Controller	Behaviors	Channels	Lines of code
Traffic light	28	0	527
Elevator	16	3	2035

Table B.5: Controller examples

B.5 JPEG Encoder

As an example for multi-media applications, a JPEG picture encoder was modeled with the SpecC language [CPC+99]. Figure B.1 shows the JPEG encoder embedded in its test bench.

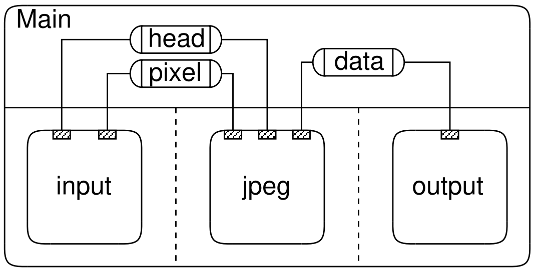

Figure B.1: JPEG encoder with test bench

The encoder component `jpeg` reads the header and pixel information of a photo by use of the channels `head` and `pixel`, respectively. It then encodes the picture and sends the generated bit stream out via the channel `data`.

[4]In the SpecC distribution, these examples can be found in the `examples/fec/` and `examples/fifo/` directories, respectively.

[5]In the SpecC distribution, these examples can be found in the `examples/elevator/` and `examples/tlc/` directories, respectively.

The SpecC model of the JPEG encoder with the test bench consists of a total of 7 different behaviors and 10 channels, as shown in Table B.6.

JPEG encoder	Behaviors	Channels	Lines of code
JPEG	4	7	1123
Test bench	3	3	341
Total	7	10	1464

Table B.6: JPEG encoder example

Internally, the JPEG encoder is composed of four concurrent behaviors which perform data handling, DCT, data quantization, and Huffman encoding. These four tasks communicate via internal channels. This structural composition of the system can be illustrated with the hierarchy tree[6], which is shown next.

```
B i o   behavior Main
B i l   |------ Input input
B i c   |------ Jpeg jpeg
B i l   |       |------ DCT dct
B i l   |       |------ HandleData handledata
B i l   |       |------ HuffmanEncode huffmanencode
B i l   |       |------ Quantization quantization
C i l   |       |------ cSyncBlock d_q_ch
C i l   |       |------ cSyncInt ddone
C i l   |       |------ cSyncBlock h_d_ch
C i l   |       |------ cSyncInt hddone
C i l   |       |------ cSyncInt hdone
C i l   |       |------ cSyncBlock q_h_ch
C i l   |       \------ cSyncInt qdone
B i l   |------ Output output
C i l   |------ cSyncByte data
C i l   |------ cSyncInt header
C i l   \------ cSyncByte pixel
```

B.6 GSM Vocoder

As an industrial-strength application, a GSM enhanced full rate speech encoder, also called GSM vocoder, was modeled and successfully simulated with the SpecC system [GZG+99].

The GSM vocoder is used in wireless, digital telecommunication for highly efficient speech compression. With the GSM encoder, speech data is sampled at a rate

[6]The hierarchy tree was created with the SIR tool set: `sir_tree -blt tb.sir`

of 8 kHz and packed into frames of 160 samples with 13 bit precision. Each frame is then encoded into 244 bits resulting in a compression rate greater than 8.

GSM vocoder	Behaviors	Lines of code
Coder	67	12382
Test bench	4	606
Total	71	12988

Table B.7: GSM vocoder example

As shown in Table B.7, the specification model of the vocoder consists of a total of 71 different behaviors, specified in about 13000 lines of SpecC source code.

The complex structural composition of the GSM vocoder is shown as a hierarchy tree[7] as follows.

```
B i o   Main
B i l   |------ arg_handler_exec
B i o   |------ coder_exec
B i o   |       |------ coder_12k2_exec
B i o   |       |       |------ codebooks_exec
B i o   |       |       |       |------ adap_codebook_exec
B i l   |       |       |       |       |------ convolve_exec
B i l   |       |       |       |       |------ enc_lag6_exec
B i o   |       |       |       |       |------ find_targetvec_exec
B i l   |       |       |       |       |       |------ CN_excitation_gain
B i l   |       |       |       |       |       |------ residu_1
B i l   |       |       |       |       |       |------ residu_2
B i l   |       |       |       |       |       |------ syn_filt_1
B i l   |       |       |       |       |       \------ syn_filt_2
B i l   |       |       |       |       |------ g_pitch_exec
B i o   |       |       |       |       |------ imp_resp_exec
B i l   |       |       |       |       |       |------ syn_filt_1
B i l   |       |       |       |       |       \------ syn_filt_2
B i c   |       |       |       |       |------ par_weight_exec
B i l   |       |       |       |       |       |------ weight_1
B i l   |       |       |       |       |       \------ weight_2
B i l   |       |       |       |       |------ pitch_fr6_exec
B i l   |       |       |       |       |------ pred_lt_6_exec
B i l   |       |       |       |       \------ q_gain_pitch_exec
B i o   |       |       |       |------ inno_codebook_exec
B i l   |       |       |       |       |------ build_cn_code_exec
B i o   |       |       |       |       \------ codebook_exec
B i s   |       |       |       |               |------ code_10i40
B i l   |       |       |       |               |       |------ build_code
```

[7]The hierarchy tree was created with help of the SIR tool set: `sir_tree -bl testbench.sir`

```
B i l   |       |       |       |               |       |------ cor_h
B i l   |       |       |       |               |       |------ cor_h_x
B i l   |       |       |       |               |       |------ q_p
B i l   |       |       |       |               |       |------ search10i40
B i l   |       |       |       |               |       \------ set_sign
B i l   |       |       |       |               |------ filter_c
B i l   |       |       |       |               |------ filter_h
B i l   |       |       |       |               |------ g_code
B i l   |       |       |       |               |------ upd_res
B i l   |       |       |       |               \------ upd_target
B i o   |       |       |       \------ update_exec
B i o   |       |       |               |------ ex_syn_upd_sh_exec
B i l   |       |       |               |       |------ excitation_exec
B i l   |       |       |               |       |------ syn_filt_exec
B i l   |       |       |               |       \------ upd_mem_exec
B i l   |       |       |               \------ q_gain_code_exec
B i o   |       |       |------ lp_analysis_exec
B i s   |       |       |       |------ find_1
B i l   |       |       |       |       |------ autocorrelation
B i l   |       |       |       |       |------ lag_windowing
B i l   |       |       |       |       \------ levinson_durbin
B i s   |       |       |       |------ find_2
B i l   |       |       |       |       |------ autocorrelation
B i l   |       |       |       |       |------ lag_windowing
B i l   |       |       |       |       \------ levinson_durbin
B i l   |       |       |       |------ int_lpc2_exec
B i l   |       |       |       |------ lsp_1
B i l   |       |       |       |------ lsp_2
B i l   |       |       |       |------ no_speech_upd_exec
B i f   |       |       |       |------ q_plsf_and_intlpc_exec
B i l   |       |       |       |       |------ int_lpc_exec
B i l   |       |       |       |       |------ q_plsf_5_exec
B i l   |       |       |       |       \------ update_lsps_exec
B i f   |       |       |       \------ vad_lp_exec
B i l   |       |       |               |------ TX_dtx_exec
B i l   |       |       |               |------ VAD_computation_exec
B i l   |       |       |               |------ nodtx_setflags_exec
B i l   |       |       |               \------ nop_exec
B i o   |       |       |------ open_loop_exec
B i o   |       |       |       |------ ol_lag_estimate
B i l   |       |       |       |       |------ minmax_1
B i l   |       |       |       |       |------ minmax_2
B i l   |       |       |       |       |------ periodicity_update
B i l   |       |       |       |       |------ pitch_openloop_1
B i l   |       |       |       |       \------ pitch_openloop_2
B i l   |       |       |       |------ residual
B i l   |       |       |       |------ syn_filter
B i l   |       |       |       |------ weight_ai_1
B i l   |       |       |       \------ weight_ai_2
B i l   |       |       \------ shift_signals_exec
```

```
B i o   |       |------ post_process_exec
B i l   |       |       |------ cn_encoder_exec
B i l   |       |       |------ prm2bits_12k2_exec
B i l   |       |       \------ sid_codeword_encoder_exec
B i o   |       \------ pre_process_exec
B i l   |               |------ encoder_homingframe_test_exec
B i l   |               |------ filter_and_scale_exec
B i l   |               \------ ser2par_exec
B i l   |------ monitor_exec
B i l   \------ stimulus_exec
```

Appendix C

SpecC Internal Representation

The SpecC Internal Representation (SIR) is a file format and a data structure. The organization of the SIR data structure is described in the following sections.

For more detailed information, such as the contents of each particular SIR class, please refer to [Döm99].

C.1 SIR graph

The internal representation of a SpecC design is a complex data structure, which can be viewed as a graph. The nodes of the graph are represented by C++ class objects, whereas the edges are represented by C++ pointers.

The nodes in the SIR graph are of different type. For example, a node representing a behavior declaration is of type `SIR_Behavior`, whereas nodes representing statements and expressions are of type `SIR_Statement` and `SIR_Expression`, respectively. For each type, a C++ class defines the data members and API methods available for the node. These SIR class declarations are listed in detail in [Döm99].

Furthermore, the nodes in any SIR graph can be classified into two groups, called *levels*. The nodes at level 1 contain all basic data contained in a SIR file, whereas the level 2 nodes represent a higher-level view of the SIR data. In other words, the SIR classes at level 1 contain all the information the SpecC language can express, whereas the level 2 classes offer an additional, more abstract view of that information. For example, the behavioral hierarchy in a SpecC program, which is not directly visible at level 1, is represented explicitly at level 2 (by the classes `SIR_Behavior`, `SIR_BhvrInst`, etc.). Level 2 classes are built automatically on top of the level 1 classes. As such, they rely on the data stored at level 1.

Figure C.1 lists the classes of SIR level 1, whereas the level 2 classes are listed in Figure C.2.

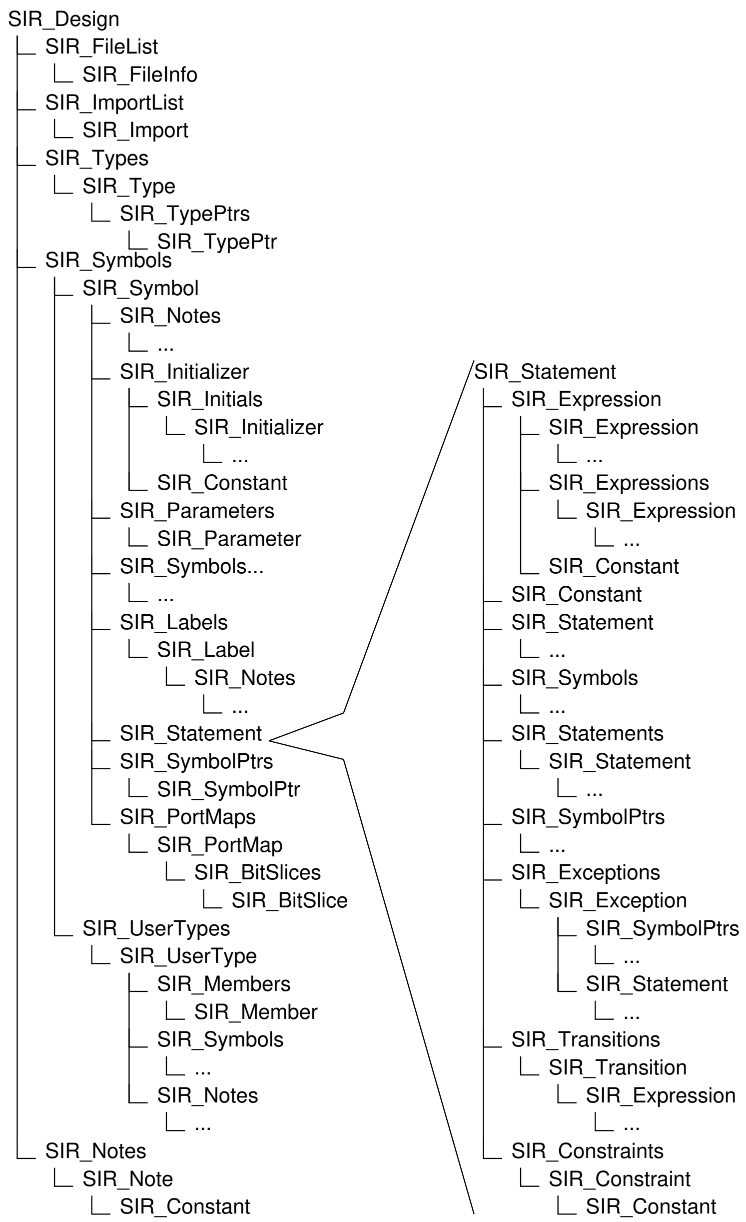

Figure C.1: Generic SIR design tree of level 1 classes

```
SIR_Design
|__ SIR_Behaviors
|    |__ SIR_Behavior
|         |__ SIR_Ports
|         |    |__ SIR_Port
|         |__ SIR_PortVars
|         |    |__ SIR_PortVar
|         |__ SIR_ImplIfs
|         |    |__ SIR_ImplIf
|         |__ SIR_BhvrInsts
|         |    |__ SIR_BhvrInst
|         |__ SIR_ChnlInsts
|         |    |__ SIR_ChnlInst
|         |__ SIR_Variables
|         |    |__ ...
|         |__ SIR_Functions
|              |__ ...
|__ SIR_Channels
|    |__ SIR_Channel
|__ SIR_Interfaces
|    |__ SIR_Interface
|         |__ SIR_Functions
|              |__ ...
|__ SIR_Variables
|    |__ SIR_Variable
|__ SIR_Functions
     |__ SIR_Function
          |__ SIR_Arguments
          |    |__ SIR_Argument
          |__ SIR_ArgVars
          |    |__ SIR_ArgVar
          |__ SIR_Variables
               |__ ...
```

```
SIR_Channel
|__ SIR_Ports
|    |__ SIR_Port
|__ SIR_PortVars
|    |__ SIR_PortVar
|__ SIR_ImplIfs
|    |__ SIR_ImplIf
|__ SIR_BhvrInsts
|    |__ SIR_BhvrInst
|__ SIR_ChnlInsts
|    |__ SIR_ChnlInst
|__ SIR_Variables
|    |__ ...
|__ SIR_Functions
     |__ ...
```

Figure C.2: Generic SIR design tree of level 2 classes

The edges in the SIR graph, representing relations among the nodes, can also be classified into two groups, which will be called *pointers* and *links*. Although all edges are implemented as standard C++ pointers, it is important to distinguish these two in the SIR data structure.

A *pointer* represents a containment relation of two objects. For example, a compound statement contains a list of statements. Therefore, there exists a pointer from the compound statement object to the header of the statement list. There is a pointer from the header of the list to the elements of the list as well.

A *link* represents a loose connection between two objects, which does not imply any containment. For example, expressions and symbols have a link to a node representing their type.

C.2 Design Trees

The classification of SIR nodes into two levels and the separation between pointers and links allows to view the SpecC data structure as a generic tree. The SIR graph becomes a tree, if the edges classified as links are ignored and only pointer edges are counted, building the arcs between the nodes. Such a graph is called a *design tree*.

Using the level classification for the nodes, the two generic SIR design trees are shown in Figure C.1 (level 1) and Figure C.2 (level 2). The roots of both trees are represented by an object of class `SIR_Design`, which is the only class belonging to both levels.

For level 1, the root object contains a list of source files (`SIR_FileList`), a list of imported binary files (`SIR_ImportList`), the global type table (`SIR_Types`), the global symbol table (`SIR_Symbols`), and an optional list of global annotations (`SIR_Notes`).

For level 2, a design consists of a list of behaviors (`SIR_Behaviors`), a list of channels (`SIR_Channels`), a list of interfaces (`SIR_Interfaces`), a list of global variables (`SIR_Variables`), and a list of global functions (`SIR_Functions`).

In both cases, the lists then can contain list elements, which again can contain objects, and so on.

The design trees are used mainly for two purposes. First, whenever some sort of traversal is performed over the SIR data structure, the traversal is done on the design trees. All iterators provided by the classes operate on the design tree only. They follow all pointers, but never follow a link. For example, when reading or writing a SIR file, it is the level 1 design tree[1] that is traversed in depth-first-search (DFS) order. This ensures that each object exists exactly once in the SIR file.

[1]SIR files only contain data from level 1 classes. Since all level 2 classes can be constructed automatically from the level 1 classes, there is no need to store them in a SIR file.

Second, many methods offered by the classes operate not only on the object itself, but also on the subtree below. For example, all `Delete()` methods behave this way. When, for example, a behavior is deleted, all its local variables and functions, including their contents, are deleted as well. In particular, when the root node of a design is deleted, all the memory occupied by the SIR data structure for this design is freed.

C.3 Base Classes

In order to keep the amount of source code for the SIR data structure implementation minimal, base classes are used whenever the same functionality is provided by different standard classes.

Almost all classes in the design trees are derived from the template classes `SIR_List` or `SIR_ListElem`. `SIR_List` represents a double-linked list containing objects of class `SIR_ListElem`.

For level 1, all classes are derived from class `SIR_Unit` which provides basic services for binary input and output. Furthermore, almost all level 1 classes are based on class `SIR_Node`, which allows to store source code location information, such as the file name and the line number, with each object.

For level 2, almost all classes are derived from class `SIR_Definition` which provides basic support for creation, deletion and renaming of objects. Furthermore, behaviors and channels are based on class `SIR_Class`. Finally, behavior and channel instances are derived from class `SIR_Instance`.

C.4 Error Handling

An important issue in program design is error handling. Errors during program execution must be detected and handled in a well-defined way. It is not acceptable to ignore error conditions, nor to simply abort the program when an error is detected.

This is true in particular for libraries that are to be linked with a larger program. Errors occurring in any library function must be detected and reported to the main program, which solely can decide whether to report the error to the user, and whether to handle and go on with the error, or to abort the program execution. Also, it is important that, even in error conditions, all data structures are being kept in a clean and well-defined state.

In general, error conditions can be classified into several categories. For example, there are *warnings*, *recoverable errors*, and *fatal errors*.

In terms of error handling, errors can be detected and handled locally in a program module, can be reported to the caller, or can be taken care of globally.

As an example for the latter, an out-of-memory condition is best handled globally, so that standard program modules can just assume to always have enough memory available.

In the SpecC Internal Representation, error handling is based on the conventions and functions defined in the header file `GL_Global.h`. The SIR automatically takes care of out-of-memory conditions. Every allocation and deallocation of dynamic memory is handled here. If no memory is available, the program is aborted with an error message, since out-of-memory is a fatal error condition.

For recoverable errors, the SIR reserves a set of error codes which identify each particular error. More specifically, the SIR uses the error codes in the range from `SIR_ERROR_BASE` up to `SIR_ERROR_BASE` + `SIR_ERROR_RANGE`. With this scheme, each error condition in a SpecC program can be uniquely identified and handled in the right way. As a special case, the no-error condition `NO_ERROR` is defined as 0.

With the SIR, errors are reported in two different ways. First, a library function may return an error code directly as its return value. In this case, the return value is either `NO_ERROR`, or one error code from the set of numbers reserved for the SIR.

For library functions returning pointers, the second method is used. In case of an error, the function returns `NULL`, indicating an error condition. The actual error code can then be obtained from the global variable `SIR_Error` which is exported by the SIR.

In order for a main program to report errors to the user in a suitable manner, error codes must be combined with a descriptive error message. In most cases, such a message can only be generated by the library which detects the error condition. Because of this, the SIR provides a function `SIR_ErrorText` which takes an error code as argument and returns a character string describing the error.

Bibliography

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

[AG98] G. Aggarwal, D. Gajski. *Modeling Guidelines for ASIC Reuse*. Technical Report ICS-TR-98-03, University of California, Irvine, Mar. 1998.

[AIG99] D. Araki, T. Ishii, D. Gajski. "Rapid Prototyping with HW/SW Codesign Tool". In *Proceedings of the IEEE Engineering of Computer Based Systems Symposium and Workshop*, Mar. 1999.

[APR+96] M. Aubury, I. Page, G. Randall, J. Saul, R. Watts. "Handel-C Language Reference Guide". Oxford University Computing Laboratory, Aug. 1996.

[Arn99] G. Arnout. "C for System Level Design". In *Conference Proceedings of Design, Automation and Test in Europe*, Munich, Germany, Mar. 1999.

[BFS95] A. Balboni, W. Fornaciari, D. Sciuto. "Tosca: A Pragmatic Approach to Co-Design Automation of Control-dominated Systems". In *Hardware/Software Co-Design*. Edited by M. Sami, G. De Micheli. Kluwer Academic Publishers, 1995.

[BG92] G. Berry, G. Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation. In *Science of Computer Programming*, vol. 19, no. 2, 1992.

[BGJ+97] 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, Apr. 1997.

[BHS91] F. Belina, D. Hogrefe, A. Sarma. *SDL With Applications From Protocol Specification*. Prentice Hall, 1991.

[BS99] M. Birnbaum, H. Sachs. "How VSIA Answers the SOC Dilemma". In *IEEE Computer*, Jun. 1999.

[CG99] E. Chang, D. Gajski. *SpecC System-level Static Scheduling.* Technical Report ICS-TR-99-23, University of California, Irvine, May 1999.

[CGC+99] A. Cataldo, R. Goering, P. Clarke, Y. Hara. "Japanese propose system-level lingua franca". In *Electronic Engineering Times*, CMP Media, New York, Nov. 15, 1999.

[CGH+93] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli. "A Formal Specification Model for Hardware/Software Codesign". In *Proceedings of the International Workshop on Hardware/Software Codesign*, IEEE, Oct. 1993.

[CHM+99] P. Coste, F. Hessel, P. Le Marrec, Z. Sugar, M. Romdhani, N. Zergainoh, A. Jerraya. "Multilanguage Design of Heterogeneous Systems". In *Proceedings of the International Workshop on Hardware/Software Codesign*, IEEE, May 1999.

[COB95] P. Chou, R. Ortega, G. Borriello. "The Chinook Hardware/Software Co-Synthesis System". In *International Symposium on System Synthesis*, Cannes, France, Sep. 1995.

[CPC+99] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao, D. Gajski. *Design of a JPEG Encoding System.* Technical Report ICS-TR-99-54, University of California, Irvine, Nov. 1999.

[CSN98] F. Chan, M. Spiller, R. Newton. "WELD – An Environment for Web-Based Electronic Design". In *Proceedings of the Design Automation Conference*, San Francisco, 1998.

[DBB99] M. Dalpasso, A. Bogliolo, L. Benini. "Specification and validation of distributed IP-based designs with JavaCAD". In *Conference Proceedings of Design, Automation and Test in Europe*, Munich, Germany, Mar. 1999.

[DG98] R. Dömer, D. Gajski. *Comparison of the Scenic Design Environment and the SpecC System.* Internal Report, University of California, Irvine, May 1998.

[DG00] R. Dömer, D. Gajski. "Reuse and Protection of Intellectual Property in the SpecC System". Regular paper accepted for the *Asia and*

South Pacific Design Automation Conference 2000, Yokohama, Japan, Jan. 2000.

[Döm98] R. Dömer. *The SpecC Internal Representation.* Internal Technical Report, University of California, Irvine, June 1998.

[Döm99] R. Dömer. *The SpecC Internal Representation, SpecC V2.0.3.* Internal Technical Report, 2nd edition, University of California, Irvine, Jan. 1999.

[DGZ98] R. Dömer, D. Gajski, J. Zhu. "Specification and Design of Embedded Systems". In *Informationstechnik und Technische Informatik*, it+ti magazine 3/98, Oldenbourg Verlag, Germany, June 1998.

[DH89] D. Drusinsky, D. Harel. "Using Statecharts for Hardware Description and Synthesis". In *IEEE Transactions on Computer Aided Design*, 1989.

[DZG98] R. Dömer, J. Zhu, D. Gajski. *The SpecC Language Reference Manual.* Technical Report ICS-TR-98-13, University of California, Irvine, Mar. 1998.

[EHB93] R. Ernst, J. Henkel, T. Benner. "Hardware-Software Cosynthesis for Microcontrollers". In *IEEE Design and Test*, Vol. 12, 1993.

[ES90] M. Ellis, B. Stroustrup. *The annotated C++ Reference Manual.* Addison-Wesley, 1990.

[GAC+98] D. Gajski, G. Aggarwal, E. Chang, R. Dömer, T. Ishii, J. Kleinsmith, J. Zhu. *Methodology for Design of Embedded Systems.* Technical Report ICS-TR-98-07, University of California, Irvine, Mar. 1998.

[Gaj97] D. Gajski. *Principles of Digital Design.* Prentice Hall, 1997.

[GCM92] R. Gupta, C. Coelho., G. De Micheli. "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components". In *Proceedings of the Design Automation Conference*, Anaheim, 1992.

[GDW+91] D. Gajski, N. Dutt, C. Wu, Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1991.

[GDZ99a] D. Gajski, R. Dömer, J. Zhu. "IP-centric Methodology and Design with the SpecC Language". In *System Level Synthesis*, edited by A. Jerraya, J. Mermet. Kluwer Academic Publishers, May 1999.

[GDZ99b] D. Gajski, R. Dömer, J. Zhu. "IP-centric Methodology and Specification Language". In *Distributed and Parallel Embedded Systems*, edited by F. Rammig. Kluwer Academic Publishers, Sep. 1999.

[GK83] D. Gajski, R. Kuhn. "Guest Editor's Introduction: New VLSI Tools". In *IEEE Computer*, Dec. 1983.

[GKL99] A. Ghosh, J. Kunkel, S. Liao. "Hardware Synthesis from C/C++". In *Conference Proceedings of Design, Automation and Test in Europe*, Munich, Germany, Mar. 1999.

[GL97] R. Gupta, S. Liao. "Using a Programming Language for Digital System Design". In *IEEE Design & Test of Computers*, IEEE, 1997.

[GM96] R. Gupta, G. De Micheli. "A Co-Synthesis Approach to Embedded System Design Automation". In *Design Automation for Embedded Systems*, vol. 1, no. 1-2, 1996.

[GVN93] D. Gajski, F. Vahid, S. Narayan. "SpecCharts: A VHDL Front-End for Embedded Systems". Technical Report ICS-TR-93-31, University of California, Irvine, June 1993.

[GVN+94] D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[GZD97a] D. Gajski, J. Zhu, R. Dömer. *The SpecC+ Language.* Technical Report ICS-TR-97-15, University of California, Irvine, Apr. 1997.

[GZD97b] D. Gajski, J. Zhu, R. Dömer. *Essential Issues in Codesign.* Technical Report ICS-TR-97-26, University of California, Irvine, June 1997.

[GZD97c] 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.

[GZG98] A. Gerstlauer, S. Zhao, D. Gajski. *VHDL+/SpecC Comparisons – A Case Study.* Technical Report ICS-TR-98-23, University of California, Irvine, May 1998.

[GZG+99] A. Gerstlauer, S. Zhao, D. Gajski, A. Horak. *Design of a GSM Vocoder using SpecC Methodology.* Technical Report ICS-TR-99-11, University of California, Irvine, Feb. 1999.

[Ham99] S. Hamilton. "SRC: Taking Moore's Law Into the Next Century". In *IEEE Computer*, Jan. 1999.

[Har87] D. Harel. "Statecharts: a Visual Formalism for Complex Systems". In *Science of Computer Programming*, 8, 1987.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HE97] J. Henkel, R. Ernst. "A Hardware-Software Partitioner Using a Dynamically Determined Granularity". In *Proceedings of the Design Automation Conference*, Anaheim, 1997.

[IAJ94] T. Ismail, M. Abid, A. Jerraya. "COSMOS: A Codesign Approach for Communicating Systems". In *Proceedings of the International Workshop on Hardware/Software Codesign*. IEEE, 1994.

[ICL97] ICL Inc. *Extensions to VHDL for System Specification: VHDL+ Version 3.0*. ICL, Manchester, United Kingdom, Nov. 1997.

[IEEE87] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE Std. 1076-1987, IEEE, 1987.

[IEEE93] IEEE. *IEEE Standard VHDL Language Reference Manual, Revision 1993*. IEEE Std. 1076-1993, IEEE, 1993.

[IEEE96] IEEE. *Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Std. 1364-1996, IEEE, 1996.

[IG98] T. Ishii, D. Gajski. "Visual Specification Environment: An Authoring Tool for Embedded Systems Co-Design". In *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies*, Sendai, Japan, Oct. 1998.

[ITU92] ITU. *Recommendation Z.100: Specification and Description Language SDL*, volume X.R25-X.R32. ITU, 1992.

[JDK+97] A. Jerraya, H. Ding, P. Kission, M. Rahmouni. *Behavioral Synthesis and Component Reuse with VHDL*. Kluwer Academic Publishers, 1997.

[JKH99] A. Jantsch, S. Kumar, A. Hemani. "The Rugy Model: A Conceptual Frame for the Study of Modelling, Analysis and Synthesis Concepts of Electronic Systems". In *Conference Proceedings of Design, Automation and Test in Europe*, Munich, Germany, Mar. 1999.

[JRV+97] A. Jerraya, M. Romdhani, C. Valderrama, P. Le Marrec, F. Hessel, G. Marchioro, J. Daveau. "Languages for System-Level Specification and Design". In *Hardware/Software Co-Design: Principles and Practice.* Edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.

[KB98] M. Keating, P. Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs.* Kluwer Academic Publishers, 1998.

[KG98] J. Kleinsmith, D. Gajski. *Communication Synthesis for Reuse.* Technical Report ICS-TR-98-06, University of California, Irvine, Feb. 1998.

[KKR94] G. Koch, U. Kebschull, W. Rosenstiel. "A Prototyping Environment for Hardware/Software Codesign in the COBRA Project". In *Proceedings of the International Workshop on Hardware/Software Codesign*, IEEE, 1994.

[KL93] A. Kalavade, E. Lee. "A Hardware/Software Codesign Methodology for DSP Applications". In *IEEE Design and Test*, Sep. 1993.

[KLM+98] A. Kahng, J. Lach, W. Mangione-Smith, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, G. Wolfe. "Watermarking Techniques for Intellectual Property Protection". In *Proceedings of the Design Automation Conference*, San Francisco, 1998.

[KM90] D. Ku, G. De Micheli. "HardwareC – A Language for Hardware Design, Version 2.0". Technical Report CSL-TR-90-419, Stanford University, Apr. 1990.

[KZG97] J. Kleinsmith, J. Zhu, D. Gajski. *ATM Modeling Example for SpecGen Evaluation.* Technical Report ICS-TR-97-47, University of California, Irvine, Oct. 1997.

[Lie97] C. Liem. *Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications.* Kluwer Academic Publishers, 1997.

[LM87] E. Lee, D. Messerschmidt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing". In *IEEE Transactions on Computers*, 1987.

[LMD94] B. Landwehr, P. Marwedel, R. Dömer. "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming". In *Proceedings of the European Design Automation Conference*, 1994.

[LP97] 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.

[LS96] E. Lee, A. Sangiovanni-Vincentelli. "Comparing Models of Computation". In *Proceedings of the International Conference on Computer Aided Design*, San Jose, 1996.

[LSS99] L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich. "Models of Computation for Embedded System Design". In *System Level Synthesis*. Edited by A. Jerraya, J. Mermet. Kluwer Academic Publishers, 1999.

[LTG97] S. Liao, S. Tjiang, R. Gupta. "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment". In *Proceedings of the Design Automation Conference*, Anaheim, 1997.

[Mar93] P. Marwedel. *Synthese und Simulation von VLSI-Systemen*. Hanser Verlag, Germany, 1993.

[MG95] P. Marwedel, G. Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[MGK97] 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.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[Mic99] G. De Micheli. "Hardware Synthesis from C/C++ Models". In *Conference Proceedings of Design, Automation and Test in Europe*, Munich, Germany, Mar. 1999.

[Nie98] R. Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, 1998.

[NVG91] 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.

[ÖBE+97] 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.

[Pet62] C. Petri. *Kommunikation mit Automaten*. Dissertation, Bonn, Germany, 1962.

[RJB98] J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Ros97] W. Rosenstiel. "Prototyping and Emulation". In *Hardware/Software Co-Design: Principles and Practice*. Edited by J. Staunstrup, W. Wolf. Kluwer Academic Publishers, 1997.

[RVB$^+$96] 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.

[Sch99] S. Schulz. "Towards A New System Level Design Language – SLDL". In *System Level Synthesis*. Edited by A. Jerraya, J. Mermet. Kluwer Academic Publishers, 1999.

[SIA97] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors*. SEMATECH, 1997.

[SK$^+$99] R. Seepold, A. Kunzmann (editors), et. al. *Reuse Techniques for VLSI Design*. Kluwer Academic Publishers, 1999.

[SM98] L. Semeria, G. De Micheli. "SpC: Synthesis of Pointers in C: Application of Pointer Analysis to the Behavioral Synthesis from C". In *Proceedings of the International Conference on Computer Aided Design*, 1998.

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

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

[VRD$^+$97] 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.

[WL95] R. Wilson, M. Lam. "Efficient Context-Sensitive Pointer Analysis for C Programs". In *Proceedings of the Conference on Programming Languages Design and Implementation*, June 1995.

[Wol97] 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.

[X3/90] X3 Secretariat. *Standard – The C Language*. X3J11/90-013, ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association, Washington, 1990.

[X3/97] X3 Secretariat. *Draft Standard – The C++ Language*. X3J16/97-14882, Information Technology Council, Washington, 1990.

[YMS+99] J. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, R. Newton. "The JavaTime Approach to Mixed Hardware-Software System Design". In *System Level Synthesis*. Edited by A. Jerraya, J. Mermet. Kluwer Academic Publishers, 1999.

[YW97] T. Yen, W. Wolf. *Hardware-software Co-synthesis of Distributed Embedded Systems*. Kluwer Academic Publishers, 1997.

[ZDG97a] J. Zhu, R. Dömer, D. Gajski. *Syntax and Semantics of the SpecC+ Language*. Technical Report ICS-TR-97-16, University of California, Irvine, Apr. 1997.

[ZDG97b] J. Zhu, R. Dömer, D. Gajski. "Syntax and Semantics of the SpecC Language". In *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies*, Osaka, Japan, Dec. 1997.

[ZMD99] Y. Zorian, E. Marinissen, S. Dey. "Testing Embedded Core-Based System Chips". In *IEEE Computer*, Jun. 1999.

Glossary

ALU	Arithmetic Logic Unit
ALAP	As Late As Possible
ANSI	American National Standards Institute
API	Application Programming Interface
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BIST	Built-In Self-Test
CAD	Computer-Aided Design
CDFG	Control Data Flow Graph
CECS	Center for Embedded Computer Systems
CFG	Control Flow Graph
CFSM	Codesign Finite State Machine
CLI	Command Line Interface
CSP	Communicating Sequential Processes
DCT	Discrete Cosine Transformation
DFG	Data Flow Graph
DFS	Depth First Search

DMA	Direct Memory Access
DSP	Digital Signal Processor
EDA	Electronic Design Automation
EOF	End Of File
FEC	Forward Error Correction
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSMC	Finite State Machine with Coprocessors
FSMD	Finite State Machine with Datapath
GSM	Global System for Mobile communication
GUI	Graphical User Interface
HCFSMD	Hierarchical Concurrent Finite State Machine with Datapath
HDL	Hardware Description Language
HLS	High Level Synthesis
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineering
IP	Intellectual Property
ISA	Instruction Set Architecture
ISO	International Standards Organisation
ITU	International Telecommunication Union
JPEG	Joint Photographic Experts Group
MILP	Mixed Integer Linear Programming
MPEG	Motion Picture Expert Group

PCI	Peripheral Component Interconnect
PE	Processing Element
PSM	Program State Machine
RAM	Random Access Memory
ROM	Read Only Memory
RPC	Remote Procedure Call
RT	Register Transfer
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SDF	Synchronous Data Flow
SDL	Specification Description Language
SIA	Semiconductor Industry Association
SIR	SpecC Internal Representation
SLD	System-Level Design
SLDL	System-Level Design Language
SOC	System On Chip
SPW	Signal Processing Work system
SRAM	Static Random Access Memory
SRC	Semiconductor Research Corporation
SW	Software
TIMA	Techniques of Informatics and Microelectronics for computer Architecture
TU	Technical University
UC	University of California
UML	Unified Modeling Language

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
VLIW	Very Large Instruction Word
VME	Versa Module Eurocard
VSIA	Virtual Socket Interface Alliance
WWW	World Wide Web

Index