Efficient Protocol Design Flow for Embedded Systems

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Dipl.-Inf.
Daniel Dietterle

geboren am 10. Februar 1978 in Eberswalde-Finow

Gutachter: Prof. Dr.-Ing. Rolf Kraemer

Gutachter: Prof. Dr.-Ing. Jörg Nolte

Gutachter: Prof. Dr.-Ing. Adam Wolisz

Tag der mündlichen Prüfung: 25. Februar 2009

Abstract

It is predicted that, in the next years, wireless sensor networks could be massively deployed in a wide variety of application areas, such as agriculture, logistics, automation, or infrastructure monitoring. An extremely low power consumption, high dependability, and low cost are common requirements for sensor nodes in all these applications. This can be achieved only by tiny, power-efficient microcontrollers and communication systems integrated on a single chip.

Formal description techniques, such as SDL (Specification and Description Language), are suitable to formally prove properties of models designed in these languages. Code generators facilitate the automatic transformation of SDL models into software implementations, while preserving the properties of the model and, thus, achieving high system dependability. The implementations consist of the translated state machine behavior and, additionally, require a run-time environment for model execution.

The objective of this work was to investigate an integrated design flow for embedded systems, which should allow the development of efficient and dependable system implementations from abstract SDL specifications. In this thesis, concepts for minimal SDL run-time environment have been devised and realized by an example implementation.

Not only pure software implementations should be considered, but starting from these also the hardware/software (HW/SW) partitioning of the system should be supported. For this purpose, a cosimulation framework that allows the coupling of an instruction set simulator (ISS) with a functional SDL simulation has been investigated and prototypically implemented within the scope of this thesis.

By shifting functionality to dedicated hardware components it is possible to take computational load from the microcontroller and to decrease the overall energy consumption by reducing the clock frequency and lowering the supply voltage. Due to the use of SDL, the design flow lends itself particularly to the implementation of communication protocols, and is limited to applications with soft real-time requirements.

For an SDL-based design flow targeted to resource-constrained embedded systems, concepts and real implementations of minimal SDL run-time environments were lacking. Available software tools, indeed, enable the transformation of SDL models into C code, however for an efficient implementation, an integration into existing real-time operating systems (RTOS) for small microcontrollers is essential.

A prototypical implementation of a run-time library for the Reflex RTOS has been created to validate our general concepts. It is about 30 % faster and consumes less than half of the program memory compared to the operating system independent run-time environment of the tool vendor Telelogic. For simple SDL models, the application requires in total less than 8 kbytes program memory and 1 kbyte RAM.

For the evaluation of design alternatives that realize different hardware/software partitionings, instruction set simulators are particularly suitable. They facilitate the identification of performance bottlenecks of the HW/SW system.

Test stimuli are required in order to measure the performance and response time of systems under design. The development of an environment that generates such test signals can be a laborious task. Thus, it is reasonable, especially in the design of protocols, to use an SDL simulation of a communication network to generate these test stimuli. Such an SDL model already exists and is the basis for the implementation. The protocol implementation simulated by the ISS then becomes part of the network simulation. An efficient coupling of SDL simulations with instruction set simulators had to be investigated, and a solution is presented in this thesis.

Based on the general concepts, a cosimulation framework for the ISS TSIM for the LEON2 processor was realized by the author. The joint SDL and instruction set simulation is very fast, which could be demonstrated by connecting a software implementation of the complex IEEE 802.15.3 medium access control (MAC) protocol with an SDL simulation of a network consisting of four devices. The real execution time for 10 seconds of simulation time amounted to just 50 seconds.

The overall design flow was validated by means of a HW/SW implementation of the IEEE 802.15.3 wireless MAC protocol. The author designed a complete SDL model of the protocol and integrated it into Reflex. By using our cosimulation environment for the TSIM simulator, the model was partitioned into hardware and software. For the hardware part, a dedicated protocol accelerator was designed by the author. This hardware component was integrated on a single chip with the LEON2 processor and, finally, manufactured.

It could be shown that the presented methodology enables the design and implementation of efficient HW/SW systems. Consequently, it can be applied to the development of dependable and energy-efficient wireless sensor nodes and other embedded systems.

Keywords: Model-based design, protocol engineering, cosimulation, IEEE 802.15.3.

Zusammenfassung

Es wird vorausgesagt, dass in einigen Jahren eine riesige Menge von drahtlos kommunizierenden Sensorknoten in den verschiedensten Anwendungsgebieten, etwa der Landwirtschaft, Logistik, Automatisierung oder der Überwachung von Infrastruktur, Einzug halten könnten. Diesen Geräten ist gemeinsam, dass sie einen äußerst geringen Stromverbrauch, hohe Zuverlässigkeit und geringe Kosten aufweisen müssen. Dies ist nur mit kleinen, Strom sparenden Mikrocontrollern und auf einem einzigen Chip integrierten Kommunikationssystemen erreichbar.

Formale Beschreibungssprachen, wie SDL (Specification and Description Language), eignen sich dazu, Eigenschaften von in dieser Sprache beschriebenen Modellen formal zu beweisen. Durch Code-Generatoren wird die automatische Umsetzung von SDL-Modellen in eine Software-Implementation unterstützt, welche die Eigenschaften des Modells erhalten soll und somit eine hohe Zuverlässigkeit des Systems erreicht. Neben der Umsetzung des Zustandsmaschinenverhaltens wird auch eine Laufzeitumgebung zur Ausführung benötigt.

Die Zielstellung dieser Arbeit war es, einen durchgängigen Entwurfsprozess für eingebettete Systeme auf der Basis von SDL zu untersuchen, der es erlaubt, von abstrakten Spezifikationen in SDL zu effizienten und zuverlässigen Systemimplementationen zu gelangen. Es wurden neue Konzepte für minimale Laufzeitumgebungen erarbeitet und beispielhaft umgesetzt.

Es sollte jedoch nicht nur die Generierung von reinen Software-Implementationen betrachtet werden, sondern von diesen ausgehend auch die Hardware/Software- (HW/SW-) Partitionierung der Systeme unterstützt werden. Zu diesem Zweck wurde im Rahmen dieser Arbeit ein Cosimulations-Ansatz zur Kopplung eines Befehlssatzsimulators (Instruction Set Simulator, ISS) mit einer abstrakten SDL-Simulation untersucht und prototypisch implementiert.

Durch die Verlagerung von Funktionen in eigens dafür entworfene HW-Komponenten ist es möglich, Last vom Mikrocontroller zu nehmen und den Gesamtenergieverbrauch des eingebetteten Systems durch das Absenken der Taktfrequenz und Versorgungsspannung zu verringern. Wegen der Verwendung von SDL eignet sich der Entwurfsprozess besonders für die Implementierung von Kommunikationsprotokollen und ist auf Anwendungen mit weichen Echtzeitanforderungen beschränkt.

Für eine SDL-basierte Entwurfsmethodik ausgerichtet auf extrem ressourcenbeschränkte eingebettete Systeme fehlten bislang Konzepte und tatsächliche Imple-

mentationen von minimalen SDL-Laufzeitumgebungen. Verfügbare Werkzeuge erlauben zwar die Übersetzung von SDL-Modellen in C-Code, für eine effiziente Implementation ist jedoch die Integration in vorhandene Mikrocontroller-Echtzeitbetriebssysteme (RTOS) erforderlich.

Eine prototypische Implementation einer Laufzeitbibliothek für das RTOS Reflex wurde entwickelt, um die allgemeinen Konzepte zu validieren. Verglichen mit der betriebssystem-unabhängigen Laufzeitumgebung des Werkzeugherstellers Telelogic ist diese Implementation um ca. 30 % schneller und benötigt weniger als die Hälfte des Programmspeichers. Bei kleinen SDL-Systemen benötigt die gesamte Applikation weniger als 8 kB Programmspeicher und 1 kB RAM.

Zur Bewertung von Entwurfsalternativen, die unterschiedliche HW/SW-Partitionierungen realisieren, eignen sich besonders Befehlssatzsimulatoren. Diese erlauben die taktgenaue Simulation der Ausführung von Programmen auf einem Prozessor, häufig auch von Modellen eigener HW-Komponenten, und sind um Größenordnungen schneller als reine HW-Simulationen. Mit ihnen lassen sich zeitkritische Teile eines HW/SW-Systems identifizieren.

Um Reaktionen des zu evaluierenden Systems hervorzurufen und dessen Performance messen zu können, werden Teststimuli benötigt. Die Entwicklung einer Umgebung, welche solche Testsignale erzeugt, kann sehr aufwändig sein. Daher ist es naheliegend, gerade im Bereich der Entwicklung von Protokollen, die Teststimuli durch die Simulation eines Kommunikationnetzes in SDL zu erzeugen. Ein solches SDL-Modell liegt ja bereits der Implementierung zu Grunde. Die im ISS ausgeführte Protokollimplementierung wird dann Teil der Netzwerksimulation. Die möglichst effiziente Kopplung von SDL-Simulationen mit dem ISS musste untersucht werden und wurde im Rahmen dieser Arbeit gelöst.

Basierend auf den allgemeinen Konzepten wurde eine Cosimulation mit dem ISS TSIM für den LEON2-Prozessor vom Autor realisiert. Die gemeinsame Simulation ist sehr schnell, was anhand der Kopplung einer SW-Implementation des sehr komplexen MAC-Protokolls IEEE 802.15.3 mit einer SDL-Simulation eines aus vier Stationen bestehenden Netzes nachgewiesen wurde. Die reale Simulationszeit für 10 Sekunden simulierte Zeit betrug gerade einmal 50 Sekunden.

Der gesamte Entwurfsprozess wurde anhand einer HW/SW-Implementierung des im Standard IEEE 802.15.3 festgelegten drahtlosen Medienzugriffsprotokolls validiert. Dazu wurde vom Autor ein komplettes SDL-Modell des Protokolls entwickelt, dieses in Reflex integriert und in einem HW/SW-Codesign-Prozess partitioniert. Dabei wurde die Cosimulationsumgebung mit dem TSIM-Simulator verwendet. Für die HW-Partition wurde vom Autor ein Protokollbeschleuniger entworfen. Dieser wurde gemeinsam mit dem LEON2-Prozessor auf einem Chip integriert und gefertigt.

Somit wurde nachgewiesen, dass die vorgestellte Methodik geeignet ist, um effiziente HW/SW-Systeme zu entwerfen und zu implementieren. Sie kann folglich zur Entwicklung von zuverlässigen und Strom sparenden drahtlosen Sensorknoten und anderen eingebetteten Systemen angewendet werden.

Acknowledgements

I would like to express my gratitude to my supervisor Prof. Rolf Kraemer for giving me the opportunity to conduct my research work at the IHP and for his enormous support throughout the last years.

A special thanks to Prof. Peter Langendörfer for the scientific discussions and guidance in the phase of writing the manuscript.

Many people from IHP's Systems Department contributed to or supported me in my work, without their help this thesis would not have been possible. I would like to thank Dr Irina Babanskaja, Jerzy Ryman, and Dr Kai Dombrowski for contributing to the design of the IEEE 802.15.3 MAC protocol and its validation. The model architecture is largely inspired by previous work done by Klaus Tittelbach-Helmrich, whom I would also like to thank for his efforts to apply this model to a communication system in the 60 GHz band and the valuable feedback that led to many improvements.

Gerald Wagenknecht put a lot of effort into designing the first version of the tight integration library for Reflex. His work and our discussions together with Dr Jean-Pierre Ebert led us to a deeper understanding of the output of the CAdvanced code generator and not to get lost in the dozens of macros. I very much appreciated the support from the developers of Reflex, in particular from Karsten Walther from BTU Cottbus and my colleague Marcin Brzozowski, on questions related to this new operating system.

I owe special gratitude to a number of colleagues who made it possible that at the end of many years of research and implementation work the results from the application of our novel design flow to the IEEE 802.15.3 MAC protocol implementation have been turned into an ASIC and can be demonstrated on a prototyping board: Dr Zoran Stamenkovic, Goran Panic, and Gunter Schoof for the design and synthesis of the LEON2 processor system; Silvia Hinrich, Brigitte Cheuffa-Tchako, Peter Dähnert, and Christoph Wolf for testing the chip; Jens Lehmann, Jörg Domke, and Horst Frankenfeldt for designing and assembling the PCB.

Many thanks to Mike Turi and Jonathan Cree from Washington State University for proofreading the manuscript.

I received tremendous support from my friends and family, your love and motivation have created the confidence that is needed to complete such a huge task. So, this thesis belongs also to all of you. Thank you for being there for me at all times!

Contents

List of Tables

List of Figures

Man muss ins Gelingen verliebt sein, nicht ins Scheitern.
Ernst Bloch

Chapter 1

Introduction

The purpose of this chapter is to set the scene for the presentation of our research work on an efficient protocol design flow for embedded systems and its application for the implementation of the IEEE 802.15.3 medium access control (MAC) protocol.

First, background on embedded systems is provided, characteristics of protocol design as opposed to the design of other applications are presented, and the role of an integrated design flow is highlighted.

Having introduced the main challenges in the design and implementation of protocols for embedded systems, we state our scientific contributions that address these problems. This thesis shall be useful as a guideline for engineers facing these challenges.

Concluding this chapter, an overview on the structure of the thesis is given.

1.1 Scope of the thesis

Digital systems can be classified in general-purpose information processing systems and application-specific systems. Personal computers (PCs) are a good example of general-purpose systems. They are programmable and support a wide range of applications.

Application-specific systems, on the other hand, are designed to fulfill a very specific task, for instance to control the operation of a washing machine. It is typical for such systems to be contained within a larger environment. Therefore, application-specific systems are often referred to as *embedded systems*.

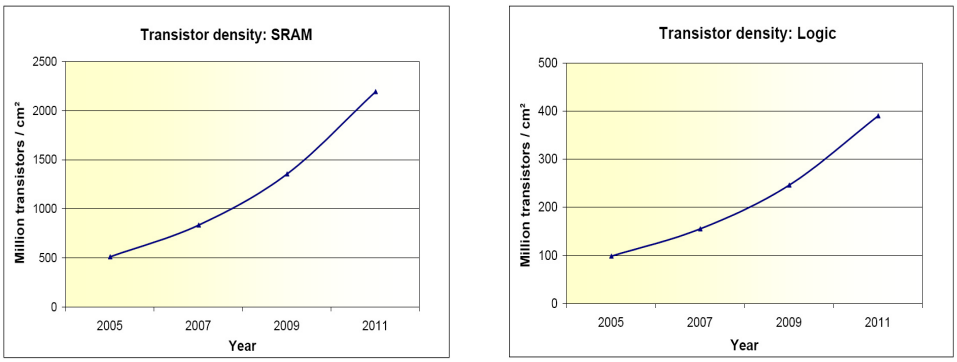


Figure 1.1: Estimated growth of transistor density in SRAM and logic cells according to the ITRS roadmap 2005 [ITR05]

In the past decades, embedded systems have replaced many non-computing systems, such as car window openers, and are now ubiquitous. Their wide adoption has become economically feasible by the mass production of digital hardware components. Since embedded systems are often part of another product that may be sold by the millions, they must be very cheap. The price of an integrated circuit is determined by its die size; the less area a chip consumes the cheaper it is. So, for economic reasons, embedded systems have only a fraction of the computing and memory resources available that a modern PC has built in.

Another aspect that is important for many embedded system applications is the need for *low power consumption*. Since a high clock frequency and a large chip size lead to increased dynamic and static power dissipation, embedded system designers must design systems that perform the required task using the lowest clock frequency and smallest chip size possible.

Moore's Law, which states the observation that the number of transistors on integrated circuits doubles roughly every 18 months, still holds true today and is reflected by the current ITRS roadmap [ITR05] that predicts the future development of semiconductor technology. Consequently, more transistors can be integrated on the same area (cf. Fig. 1.1) for roughly the same price. This leads to a *growing complexity* of future embedded computing systems.

A further trend that can be observed and is predicted for the future is that more and more electronic devices will be *networked*. The Wireless World Research Forum (WWRF) assumes that by the year 2017 there will be one thousand wireless devices at service for each individual. Already now, new applications making use

of wireless personal area networks (WPAN) and wireless sensor networks (WSN) are emerging. Application areas include, but are not limited to, home automation and security, personal health care, logistics, traffic control, process and factory automation, agriculture.

Communication protocols define rules for the interconnection of communication endpoints [Kön03]. Protocols are therefore the basis for the realization of computer networks, in general, and for wireless sensor networks, in particular. They can be *implemented* in hardware, in software, or as a mix between the two. The implementation method influences the efficiency and other parameters, such as the flexibility to introduce later protocol extensions or bug fixes. However, protocols are *designed* on an abstract level without having any specific implementation method in mind.

What makes protocol design for embedded systems special in contrast to application development in general? Even though one may find applications that exhibit similar features as protocols, there are a number of reasons to treat protocol design as a special case. Protocol behavior is often controlled by the expiration of *timers*, for instance to set a limit on the time to wait for a response from a communication partner. In the next section, we will address the issue of real-time requirements specifically.

Furthermore, there are *complex interactions* between one or more protocol instances at communication partners that can be interleaved with each other. This complexity, paired with the time dependency, causes a vast number of possible protocol states and protocol runs. In any case, it must be guaranteed that the protocol instance does not end up in a deadlock and that from each state that has been reached any other state can be reached again, i.e. that there are no livelocks. The required robustness and correctness, especially within the context of embedded systems that must operate reliably for months or years without maintenance, makes a tailored design approach, that is methods and tools, necessary.

1.2 Problem statement

Challenges in current embedded systems design methodologies According to the HiPEAC roadmap on embedded systems [V+06], the increased system complexity makes new tools for verification and testing necessary. Up to the year 2009, an aggregated annual growth rate for design and test automation tools of

nearly 20% is predicted. The current de facto situation in the semiconductor industry is that the validation of embedded systems and complex SoCs may consume up to 80% of the total design cost and time [MED03]. This can potentially prevent the embedded system product under development from being successful on the market. Hence, a main challenge in the electronic design automation (EDA) field is to provide new methods that can enable the rapid validation of embedded systems leading to low-cost devices [V+06]. The aforementioned vision of the WWRF can only be reached with appropriate tool support.

System-level design languages have the potential of shortening product development cycles by providing validation and system simulation at an early design phase, as reported in [V+06]. For this reason, an increase of projects that rely on system-level design languages has been predicted. Similarly, a recent study [Cel05] by Celoxica on system design trends has revealed that there is currently a transition from paper-based system specifications to electronic specification languages such as UML, Matlab or C taking place in the industry. Model-based design and the use of hardware/software partitioning tools are not yet common, though they promise to speed up the development process and reduce the number of design errors.

Not only models and architectures are important for developing new implementations within a short time, but also a tailored design flow and corresponding tools that support various steps of the design flow. As outlined in the previous section, reliability and energy efficiency must be addressed in the design of embedded systems.

SDL-based protocol design flow for embedded systems Specification and Description Language (SDL) is a popular high-level language to formally specify system behavior. In particular, it has been successfully applied to protocol engineering and has attracted a lot of attention from the research community of this field. SDL not only allows the simulation of models, but also their formal verification. The purpose of verification is to identify and eliminate incorrect behavior, such as deadlocks or livelocks, that may occur under a particular sequence of events.

In the past, a number of tools [IAJ94], [ÖBE+97], [DMTS00] have been proposed that support an integrated design flow from high-level SDL specifications to system implementations. It is possible to automatically transform SDL spec-

ifications into executable software and even generate hardware designs from the specification by special-purpose compilers. These approaches will be discussed in Chapter 3 in more detail.

In this work, the author has focused on adapting and extending the protocol engineering design flow based on SDL to the requirements of deeply embedded systems. The motivation for choosing SDL was its suitability to formal verification as well as mature tool support for simulation and implementation.

The existing approaches, however, are not targeted specifically to embedded systems and the tight processing and memory requirements of, for instance, 16-bit microcontrollers with less than 64 kbytes of memory. Rather, the previous work has focused on high-performance communication processors or rapid prototyping systems.

The adaptation of the SDL-based design methodology for communication protocols to the requirements of embedded systems demands for solutions to the research problems listed below.

- General concepts for an efficient use of real-time operating systems and embedded systems hardware platforms as *run-time environments* for SDL models have to be investigated.
- Mechanisms for connecting an *instruction set simulator* with an SDL simulator have to be studied.

These items will be motivated and further explained in the remainder of this section by discussing the limitations of the current approaches. The feasibility of the new concepts has to be shown experimentally. This is achieved by

- a prototypical implementation of an SDL run-time environment for a target operating system, its validation and performance comparison with existing approaches, and
- a proof of concept for the cosimulation framework by realizing an interface between a particular instruction set simulator and Telelogic's SDL simulator.

Figure 1.2 shows the integrated SDL-based design flow. The novel elements that have to be introduced to target resource-limited embedded systems are highlighted. They comprise an optimized run-time environment for embedded systems (denoted as tight integration library in the figure) and a cosimulation framework.

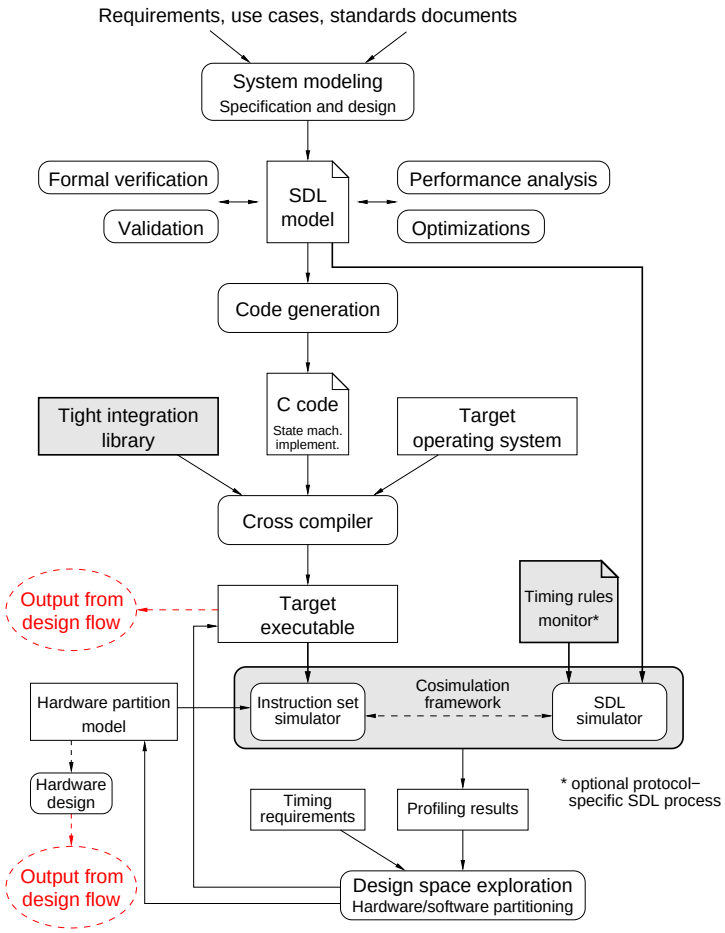


Figure 1.2: Overview of the complete design flow starting from an initial SDL model to a hardware/software implementation.

For resource-constrained embedded systems, designers typically choose operating systems with a very small memory footprint and, consequently, a very limited set of features and available services. TinyOS [LMP+05], Contiki [DGV04], or Reflex [WN07] are examples of the most commonly used operating systems for wireless sensor nodes, a particular kind of deeply embedded system. One of the challenges in targeting SDL specifications to such operating systems is to map SDL concepts to the available mechanisms of the operating system. Furthermore, the features of the hardware platform, such as programmable timers, must be exploited as efficiently as possible.

So far, no general framework for targeting SDL models to operating systems

for deeply embedded systems has been presented. Our objective was to close this gap by investigating concepts for a mapping that has a small memory footprint and little run-time overhead. We base our work upon the output of the very successful commercial code generator CAdvanced from Telelogic, as the generated code facilitates the integration with a target operating system.

With our conceptual ideas and their realization as a software framework for lightweight SDL run-time environments, it should be easily possible to create integration models for different real-time operating systems and hardware platforms. The performance of system implementations that are developed on the basis of our concepts will be significantly better than with the standard approach which uses a generic SDL run-time environment.

Embedded systems may consist of general-purpose microcontrollers and dedicated hardware blocks. Since the dedicated hardware is designed for a specific purpose, its performance is typically several magnitudes higher than that of microcontrollers if hardware parallelism can be exploited and required operations are not directly supported by the instruction set of the processor. Hence, a mixed hardware/software implementation may provide the best tradeoff between the required flexibility, which can be ensured by the use of software, and performance. By using dedicated hardware for time-critical and processing-intensive tasks, the general-purpose processor has less strict performance requirements and its clock frequency could be reduced. This way, the energy efficiency of the system is increased. This is particularly important for battery-powered devices.

Hardware/software codesign tools support the designer in finding optimal system partitionings with respect to user-defined constraints such as real-time behavior, power consumption or resource usage. Some tools allow the automatic generation of hardware from high-level specifications. Such a tool, which uses annotated SDL as system specification language, was proposed by Muth [Mut02] for rapid prototyping applications. The designer has to specify which SDL processes shall be generated as hardware based on their computational complexity and timing requirements. A worst-case real-time analysis is conducted by the tool and checked against the requirements. This way, the designer can be sure that a chosen partitioning satisfies the timing specification. We consider this work as the most advanced integrated design flow based on SDL.

The granularity of mapping complete SDL processes to either hardware or software is rather coarse in this approach. It causes significant overhead in terms

of hardware area and signal exchange delay, because of required buffer space for hardware signal queues and the time needed to transfer signal parameters into the buffers. For this reason, we discard the automatic generation of hardware from SDL specifications from our design flow. Instead, we prefer a fine-grained partitioning of SDL processes into hardware and software. The manual design of dedicated hardware and its interfaces to the software partition promises to yield more efficient systems, however increases the design effort.

During design space exploration, i.e. the process where the designer investigates different system architectures and hardware/software partitionings, usually a profiling of the system is performed. The objective of the profiling is to obtain information about the execution time and bottlenecks of the current design alternative. The profiling information can be gathered either by using a real execution of the system, which may require significant design effort to create such a system, or by simulations. Simulations can be conducted with more or less accuracy depending on the chosen system model. Naturally, the closer the profiling results reflect the reality, the better decisions can be made in the design exploration phase.

Our work particularly addresses the design flow for communication protocols. In this case, the designer must analyze whether a given hardware/software implementation model of the protocol interacts with its environment, i.e. the upper and lower protocol layers, according to the timing specification, and analyze what parts of the implementation model are bottlenecks. This can be concluded from a static worst-case execution analysis or, as stated above, by simulating and profiling the design. A worst-case analysis might yield too pessimistic results and, hence, lead to an implementation which is overdesigned. While such an approach is necessary for hard real-time systems, it is likely to require more resources and computing power for the average case.

A short note on real-time behavior with respect to wireless communication protocols: A computer system that must react on external stimuli and produce a result within a certain time is called a *real-time system*. If missing a deadline could possibly cause catastrophic consequences, this computer system is said to have *hard* (or safety-critical) real-time requirements. Otherwise, it is called a *soft* real-time system.

In wireless communications, loss of messages over the wireless channel is quite common due to the presence of noise and interference. Therefore, other devices will not see a difference between missing a deadline in one protocol instance or a

transmission error. Protocols are designed to tolerate and recover from transmission errors. Even if a message is occasionally sent late and interferes with another message, this will only lower the performance, but not break the system. If an application with hard real-time constraints shall relies on a wireless communication system, the application designer must account for the occasional loss of messages and provide a fail-save operation. Hence, the application and the protocol implementation could just as well be in two different domains, where the protocol implementation has less strict requirements with respect to timing.

Therefore, the design flow presented in this thesis targets soft real-time embedded systems only. Still, the designer shall be supported in identifying where the implementation fails to meet the specified timing constraints and what are the bottlenecks. This is an important requirement for the hardware/software partitioning process.

Obtaining profiling information from system simulations is a viable and better suited approach than a static worst-case analysis, in the area of communication protocol design, as long as there are no hard real-time requirements. The performance of implementation models can be estimated by different methods.

One possibility is to annotate transitions in SDL processes with a user-defined duration. Additionally, fixed process scheduling and signal exchange overhead can be included. The performance of the SDL system is then measured by simulating the model and advancing the simulation time according to the timing annotations. Since these annotations are only more or less accurate estimations, the performance results obtained from the simulations may not correspond well to the real execution times.

An improvement would be to use the real execution time of the SDL model on the host computer as a basis and scale the simulation results to the computational performance of the target embedded system microcontroller. Again, the confidence in the accuracy of the performance measurements is low due to the different instruction set architectures, effects caused by caches, etc. In particular when simulating systems with external components that trigger interrupts, accuracy can be low.

The most accurate profiling results can be obtained by simulating the execution of a hardware/software system on the target processor. A tool that allows such kind of performance measurements is an instruction set simulator. They are freely or commercially available for many microcontrollers. Though the simulation runs

require more time than the previously described methods, the profiling results are very accurate, especially for cycle-true simulations.

When simulating a particular communication protocol implementation, there must be a *test bench* that creates stimuli for the system under test. The test bench reflects the behavior of the peer communication entities as well as the lower and upper protocol layers.

The development of a comprehensive test bench for a communication protocol implementation is a time-consuming task. This development time can be saved by reusing the SDL model of the communication system. In order to validate the protocol functionality before starting the implementation phase, designers typically create a simulation environment that can instantiate and interconnect several protocol entities in a network model. This can be easily done in SDL by introducing models for the lower protocol layers and a physical network. Stimuli for a simulation of the protocol *implementation model* can be directly obtained from the simulation of the network model by using the SDL signals exchanged with a particular protocol instance.

Ideally, these signals could be used as an input for the protocol implementation model simulated by an instruction set simulator (ISS). The signals generated as output by the implementation model could be directly introduced back into the SDL simulation of the communication network. This way, an interactive *cosimulation* run can be performed. As a result, profiling results for the implementation model are gathered, and it can be checked whether the model satisfies all timing requirements of the protocol.

The concept of integrating an instruction set simulation of a protocol implementation into the functional network simulation in SDL is shown schematically in Fig. 1.3

To support the designer in detecting all cases where the implementation model did not respond to a received stimulus in time, i.e. missed a real-time requirement, the author has created the concept of a *timing rules monitor*. This is an SDL process that receives all SDL signals that are sent as stimuli to the implementation model and all of its responses, which are also SDL signals. The designer may specify a set of rules describing the acceptable behavior of the implementation model. The reason for the introduction of the timing rules monitor is the weak semantics of SDL with respect to specifying real-time requirements as will be outlined in more detail later in this thesis. The only way to observe a deviation from the acceptable

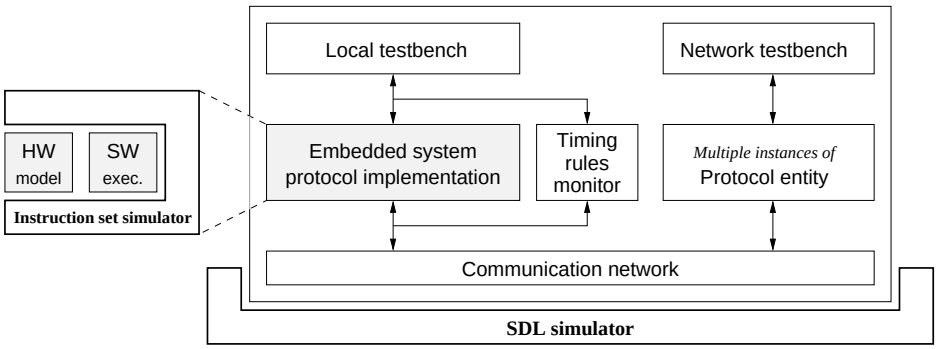


Figure 1.3: The instruction set simulation of the target system including models of the hardware partition is embedded in the overall network simulation based on SDL.

timing behavior without the timing rules monitor would be to notice differences in the protocol runs. Such differences, however, are difficult to spot, especially when long test benches are used and, possibly, communication failures leading to timeouts are part of the normal behavior in the simulated protocol run.

To our knowledge, the cosimulation of SDL specification models and implementation models in an ISS has not been treated before. Therefore, mechanisms for an efficient coupling, which also preserve the semantics of SDL, have to be investigated. Furthermore, based on the general considerations, a prototypical implementation is needed to prove the validity of the concept and to serve as a tool in the design flow.

Validation of our work The SDL-based protocol design flow has been validated by applying it to an implementation of the IEEE 802.15.3 medium access control protocol [IEE03a]. This implementation is part of a generic wireless communication platform, which has an IEEE 802.15.3-compliant physical layer offering data rates between 11 and 55 Mbit/s. The MAC protocol can be considered as very complex. Among other features, it provides an asynchronous and isochronous data service.

From a practical point of view, we had to answer the question if and how it would be possible to implement such a complex MAC protocol for a battery-powered sensor node. In tackling this problem we had the freedom to develop protocol software as well as to design dedicated hardware as a protocol accelerator.

As will be reported later in Chapter 7, our concepts for an embedded systems design methodology and the tools implementing these concepts supported the hardware/software design process and led to an efficient system-on-chip implementation of the above mentioned MAC protocol. This example shows that the methodology can also be successfully applied to other protocols that are typically used within the area of wireless sensor networking or embedded systems. They will likely have lower complexity and data rates compared to IEEE 802.15.3. Examples of such protocols are S-MAC [YHE02], T-MAC [vDL03], IEEE 802.15.4 [IEE03b] or Bluetooth [IEE05].

1.3 Contributions

The author has addressed the design challenges presented in the previous section by

- creating an integrated design flow that is based on SDL,
- studying concepts for an efficient tight integration layer, i.e. an SDL run-time environment, for the CAdvanced code generator from Telelogic,
- investigating general problems of connecting an instruction set simulator with an SDL simulation,
- implementing the theoretical concepts for an efficient run-time environment and using the real-time operating system for extremely resource-constrained devices Reflex [Nol09] as an example,
- implementing a cosimulation framework that couples the Telelogic SDL simulator with the LEON2 instruction set simulator TSIM, and
- successfully applying the methodology and newly developed tools to the design and implementation of the IEEE 802.15.3 MAC protocol.

Throughout the proposed design flow, an initial SDL model is gradually refined and optimized. Our work builds upon mature software tools and provides links between them to create an integrated design flow. For performance analysis and exploration of design alternatives, a concept for coupling the high-level SDL simulator with an instruction set simulator has been elaborated. This approach

shortens the protocol development time by reusing the original SDL model as a test bench for the hardware/software system simulation.

In order to validate our design methodology it has been applied to a hardware/software implementation of the IEEE 802.15.3 MAC protocol for a low-power wireless communication platform. For this purpose, an SDL model for this protocol has been designed. Due to the complexity of the protocol, an architecture for the model and suitable abstractions had to be found in a first step. After identifying all SDL processes with their specific responsibilities and services provided to other processes, the complete protocol functionality has been modeled in SDL.

Following the methodology, the author used the CAdvanced code generator from Telelogic to generate C code from the MAC protocol model. Our run-time environment for the Reflex operating system has been used to obtain a first all-software implementation of the protocol on the basis of the generated C code.

The author partitioned the automatically generated and optimized implementation model into hardware and software parts with the help of the cosimulation framework. Finally, a protocol accelerator was designed in VHDL.

With this design effort, which has been achieved by the author of this thesis, we have proved that our approach can serve as a general guideline and template for future embedded systems protocol implementations. The software layer to integrate SDL models into Reflex and the cosimulation framework can be directly reused for other systems designed using SDL. Both can be easily adapted to other operating systems or instruction set simulators.

Being a sophisticated and complex protocol, the IEEE 802.15.3 MAC protocol includes various features and functionalities, such as contention access and reserved time slots, that are also present in similar wireless MAC protocols. Therefore, the SDL model of the protocol and its architecture can serve as a blueprint for other MAC protocol designs. Similarly, the architecture of the protocol accelerator, that has been designed to perform time-critical and processing-intensive tasks, can be adapted to other protocol implementations. The SDL model and the protocol accelerator are important results of this thesis and will be thoroughly presented. Our theoretical concepts and design results have been presented at several international conferences, and the architecture of the protocol accelerator has been submitted as a patent.

1.4 Overview

The thesis is structured in nine chapters. In Chapter 2, the wider context of our work is presented. It includes background on typical protocol design and implementation methods, an overview on common MAC protocols for wireless personal area networks, as well as a general overview on the design methodology for embedded systems.

In Chapter 3 we highlight alternative protocol design methodologies that have been proposed in the literature and discuss their advantages and weaknesses. Chapter 3 concludes the introductory part of the thesis.

Chapter 4 presents the concepts of our novel contributions to an integrated protocol design flow for embedded systems. Prototypical implementations of two important building blocks of our design flow, the tight integration layer for Reflex and the cosimulation framework, are then covered in detail in Chapters 5 and 6, respectively.

Results from the design of the IEEE 802.15.3 MAC protocol are summarized in Chapter 7. The main results, which are also templates for similar design tasks, are our high-level SDL model and the protocol accelerator architecture.

We present a critical discussion of our work and an outlook on future work in Chapter 8. Finally, the thesis is concluded in Chapter 9.

Chapter 2

Design of Embedded Communication Systems

Similar to the way how humans communicate with each other using languages and rules of conversation, computer communication systems require the definition of messages and of the order in which they may be exchanged. These definitions constitute (communication) protocols. Protocols have been standardized to achieve interoperability between devices of different manufactures. The standards do not prescribe an implementation method, they are abstract syntactical and functional descriptions.

Protocol engineering is a discipline of computer science which deals with the modeling and specification of communication systems and protocols. It also puts special emphasis on quality and, therefore, formal languages have been developed that facilitate verification as well as methods for the test of protocol implementations. Protocol engineering also comprises efficient implementation models for protocol specifications.

Significant research effort has been devoted in the academia and EDA industry to the development of methodologies for embedded systems and system-on-chip (SoC) design. Similar to the protocol engineering area, system specification languages and design tools were proposed to improve the quality of the design and to accelerate the product development cycle. Many researchers are working on tool support for the computer-aided exploration of design alternatives and the translation of high-level behavioral system descriptions to low-level physical representations.

The work presented in this thesis can be treated as a contribution to bring together the established protocol engineering methodology with the tools and methods developed for embedded system and SoC design. This concerns, in particular, hardware/software partitioning. Our methodology enables the design and implementation of power-efficient, severely resource-limited and wirelessly networked devices and sensor nodes.

The chapter is structured into two main sections that introduce the main results and concepts of the protocol engineering and hardware/software codesign disciplines.

2.1 Protocol engineering

Communication protocol entities provide one or more services to other protocol entities or applications and may use services from other protocols for this purpose. The International Standards Organization (ISO) has defined an Open Systems Interconnection basic reference model [ISO94] (OSI reference model) that structures communication functionality in seven protocol layers—from the physical layer up to the application layer—as depicted in Fig. 2.1. This layering approach has been adopted with great success as it provides a common basis for the coordination of standards development, breaks the complexity of communication systems, and facilitates modular implementations where building blocks may come from different vendors.

Protocol entities interact with their peer entities at the communication partners. This interaction follows strict rules. A communication protocol is a convention that specifies possible temporal sequences of interactions between the involved protocol entities and defines the format of the messages that are exchanged [Kön03]. These messages are known as protocol data units (PDU). By defining the structure of PDUs in the protocol, it is ensured that there is a common interpretation of the data units among the peer entities.

Even though there is a plethora of different communication protocols, one can find recurring mechanisms in their design. Section 2.1.1 highlights some of the most commonly used protocol mechanisms, such as connection management, error control and flow control. We will then introduce popular wireless MAC protocols for short-range communication in Section 2.1.2. The methods and languages employed by designers to specify protocol behavior will be covered in Section 2.1.3.

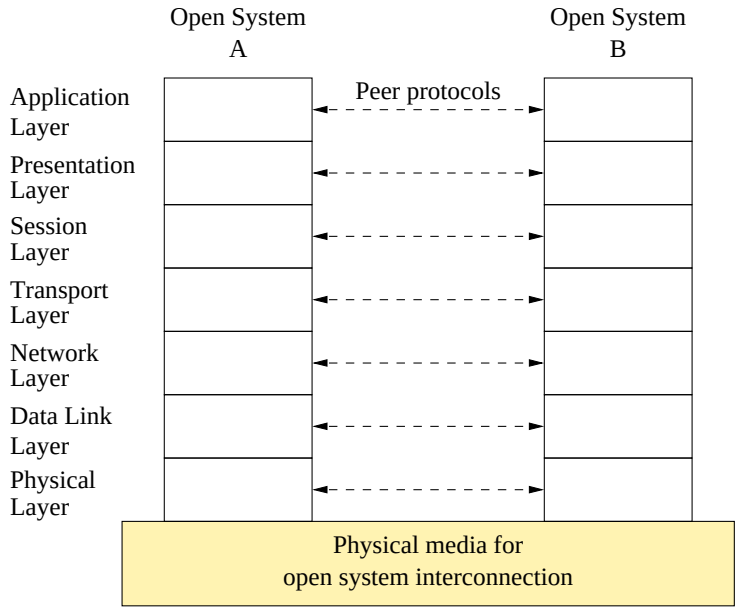


Figure 2.1: Seven layer OSI reference model and peer protocols [ISO94]

Finally, Section 2.1.4 will cover the development process for communication protocols with emphasis on verification, implementation, and test.

2.1.1 Protocol mechanisms

When comparing the behavior of different protocols, it becomes apparent that they are often composed of similar temporal sequences and logical functions, also known as protocol mechanisms or protocol functions. We will present some of the most common protocol functions in this section. This will provide a deeper insight into the diversity and complexity of protocol operation.

Connection management

In telecommunications, one can distinguish between connection-oriented and connectionless protocols. Connection-oriented protocols require the establishment of a logical connection between the communication partners before data transfer can be started. With connectionless protocols, on the contrary, data transfer is initiated without prior connection setup. A connection can be viewed as a shared state between the peer protocol entities and as an agreement about the communica-

tions relationship, whereas connectionless protocols are also described as stateless. Transmission Control Protocol (TCP) is a well-known example of a connection-oriented protocol while Internet Protocol (IP) is a connectionless protocol.

Connection management as a protocol mechanism refers to all the activities necessary for setting up, maintaining, and terminating a connection. In the connection establishment phase, the involved protocol entities often negotiate parameters, such as the quality of service (QoS) of the connection. During the data transfer phase, when the connection is already established, there might be service disruptions in the lower protocol layers. To prevent any loss of user data and provide an uninterrupted service, the protocol may try to resynchronize with the communication partner or reassign resources in the network.

For the termination of a connection one has to consider that there still may be data packets belonging to the connection traveling in the communications network. There is the possibility of deferring the termination until all packets have been delivered or timed out, or to release the connection abruptly. When reusing connection identifiers of old connections for new ones too soon, there is the chance that old packets that arrive late will be related to the new connection.

Error control

Error control is one of the most important and commonly used protocol functions. Since the physical media and intermediate relay nodes in communication networks cannot guarantee error-free forwarding of PDUs, protocols need to be able to detect and correct transmission errors in order to provide a reliable data transfer service. It is not only that individual bits of the transmitted data may be corrupted, but messages are sometimes lost or duplicated. The error control techniques that are typically found in protocols are reviewed below.

For error *detection* purposes, protocols often utilize *check sums*. This means that the bits of the PDU are used to calculate an error check sequence by means of a mathematical algorithm, for example modulo-16 addition. The error check sequence becomes part of the PDU. Its receivers can run the same algorithm on the PDU. If individual bits have been altered in transmission, there is a high probability that the check sum calculated over the received data does not match the received check sum. Thus, the receiver is able to detect transmission errors.

Cyclic redundancy check (CRC) algorithms are commonly used in lower protocol layers because of their ability to detect burst errors and their simple hardware

implementation. With *forward error correction (FEC)* coding it is possible to recover the original bit stream even in the case of transmission errors (provided their number does not exceed a certain limit) by utilizing redundancy information in the bit stream.

If a protocol entity has detected a transmission error or loss of a PDU, *automatic repeat request (ARQ)* mechanisms are commonly used to initiated the retransmission of the lost PDU. ARQ mechanisms rely on the transmission of (positive or negative) acknowledgment (ACK) messages by the receiver and on the detection of timeouts at the sender.

Stop-and-Wait is the simplest form of ARQ. In this scheme, the sender waits for a positive acknowledgment from the receiver before sending out the next PDU to that receiver. If the sender does not receive the acknowledgment within a fixed time or receives a negative ACK (in some protocols), it will retransmit the message. During the time when the sender is waiting for an ACK, data transfer cannot proceed.

The *Go-Back-N* scheme offers a more efficient use of bandwidth. It allows the sender to transmit data without receiving an ACK up to a maximum window size (number of not yet acknowledged PDUs). The receiver acknowledges the reception of all PDUs up to a *sequence number* given in the ACK message. If a transmission error occurs and the ACK from the receiver is not received in time, the sender will retransmit all PDUs starting from the first one that has not been acknowledged, even if the receiver has already correctly received some of them.

An alternative to the Go-Back-N scheme is *Selective Repeat*. It provides the possibility to selectively repeat only those PDUs that had transmission errors and also operates with a transmission window. Selective Repeat requires the receiver to provide buffer space for received PDUs in order to deliver the messages in the correct sequence.

The operation of the three presented ARQ schemes is visualized in Fig. 2.2 by means of Message Sequence Charts (MSC) showing an example data transfer and the use of timers.

Not only data PDUs are subject to transmission errors, but also acknowledgment PDUs. If an ACK is lost, the sender will retransmit the data PDU. Hence, the receiver must detect when it has received a duplicate. Again, *sequence numbers* can be used for this purpose. The sender must use the same sequence number for the retransmitted PDU as for the original one.

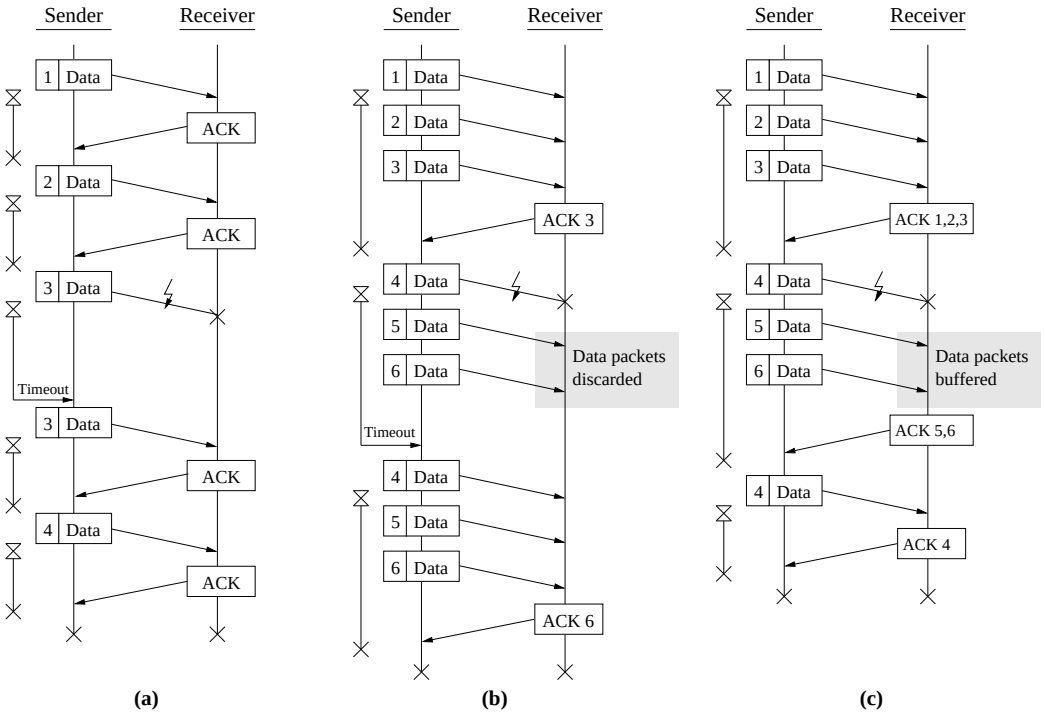


Figure 2.2: Illustration of the three ARQ schemes Stop-and-Wait **(a)**, Go-Back-N **(b)**, and Selective Repeat **(c)**.

Sequence numbering can be considered a separate protocol mechanism. As with any numbers represented in computers, their range is limited by the amount of bits used to represent their value. To reduce protocol overhead, only few bits are normally used for the encoding of sequence numbers and so it must be considered in the design of protocols that sequence numbers will be reused after a certain number of PDUs have been transmitted.

Time monitoring is essential for error control. It is used for timeout-driven retransmissions, but also to detect if the communication peer is still active. This becomes more important in networks with dynamic topologies, for example wireless networks where nodes may go out of communication range of each other.

Flow control

Telecommunication networks are often heterogeneous networks. This means that the communication nodes differ with respect to their capabilities and performance.

There may be nodes that are not capable to process data at a rate at that other nodes transmit data, or there is not enough free buffer space at the receiver node to hold received messages.

In order to avoid overloading of receiving nodes by excessive data transmissions, flow control is applied. With this protocol function, the rate at which nodes transmit PDUs is adjusted. There are two basic flow control mechanisms: window-based and rate-based flow control.

Window-based flow control is an end-to-end protocol mechanism where the receiver grants the sender a certain amount of data—the transmission window—to be transmitted before the sender must wait for a window update from the receiver. The well-known *sliding window protocol* is part of TCP. In this variant, the window size is negotiated when a TCP connection is established. With each acknowledgment PDU the receiver implicitly allows the sending protocol entity to transmit more data, up to the previously negotiated window size, which is a fixed number of bytes.

This window-based scheme has some disadvantages in networks with very high data rates, because the window buffer can be filled within a very short time. The relatively long transmission latency that occurs from the receiver to the sender will slow down the sender and lead to bursty traffic. An alternative approach that alleviates these problems is to allocate a desired transmission rate on all links of the path from source to destination through the network, at connection setup time. This is then called rate-based flow control, but is not described here further.

Coding and decoding of PDUs

Coding and decoding of PDUs is a protocol function that is performed locally at each protocol entity and, therefore, is not visible in the interactions between protocol entities. Its role is to create properly formatted PDUs (conforming to the protocol syntax) from user data and additional protocol control information.

Conversely, received PDUs must be parsed and decoded to extract the information conveyed over the network. The structure of PDUs, which is defined by the protocol, determines how efficiently the coding and decoding can be implemented in software using standard microprocessors. For this reason, some new protocols, like IPv6, use 32-bit aligned fields. On the other hand, PDUs are formatted as terse as possible with the least number of bits to reduce protocol overhead, increase bandwidth efficiency, and lower the energy required for transmission and reception.

The last point is particularly important in networks with battery-powered devices such as wireless sensor networks.

Assembly and disassembly of PDUs

Service data units (SDU) passed for transmission from an application or higher protocol layer do not always fit in size into the PDU structure of a protocol. In this case, the SDU must be split over multiple PDUs. This process is called *segmentation*, the inverse function at the peer entity is called *reassembly*.

It is also possible to collect multiple smaller SDUs to form a single PDU. This is done to reduce protocol overhead as fixed per-PDU overhead is required only once. Hence, the efficiency of the protocol can be increased. The corresponding protocol functionality at the sender and at the receiver is termed blocking and unblocking, respectively.

2.1.2 Wireless medium access control protocols

Wireless communications have experienced an enormous growth in the last 10 to 15 years. This development has been driven by the wide deployment of mobile phone networks and wireless local area networks (WLAN). Users enjoy the feeling of being anytime and anywhere connected, whereas WLANs helped to avoid costly and annoying wiring in homes or large buildings.

The trend in wireless communications goes into the direction of sensor and actuator networks for a variety of application areas: building and industrial automation, logistics, retail industry, automotive and transport, telemedicine, etc. One of the biggest challenges in the design of such networks is *energy efficiency*, as many of the connected devices will be battery-powered or use energy-scavenging techniques to avoid wiring, which is costly and not always feasible. Therefore, only a limited amount of energy will be at the disposal of the devices for communication.

The wireless channel is a shared medium. Similarly to shared wired media, a multiple access scheme must be in place in order to control the organized access of all devices to the channel. Common multiple access schemes are time division multiple access (TDMA), frequency division multiple access (FDMA), code division multiple access (CDMA), space division multiple access (SDMA), and combinations of them.

In FDMA, CDMA, and SDMA the transmitters can use a frequency, code

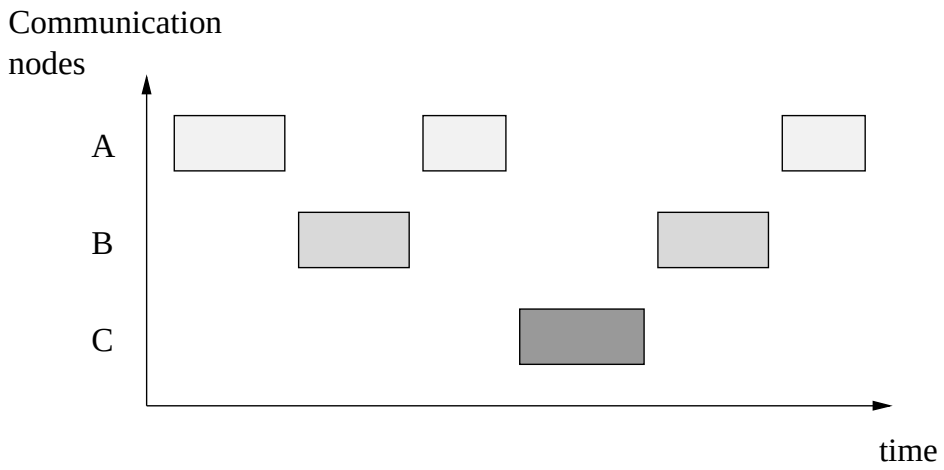


Figure 2.3: Communication nodes share the wireless channel in the TDMA scheme. Each node may access the channel, i.e. transmit data, exclusively in a time slot in some predefined or dynamic order. All nodes use the same frequency band.

sequence, or region exclusively and are separated from each other. With TDMA, the channel is granted for a period of time in the complete frequency band and space exclusively to a single transmitter (Fig. 2.3). TDMA is especially suited for wireless sensor networks because of its algorithmic simplicity, limited signal processing demands, and flexibility to changing topologies. We will, therefore, focus on the TDMA scheme in the remainder of this section.

Medium access control is a sublayer of the data link layer (layer 2) in the OSI reference model (cf. Fig. 2.1). The medium access control protocol is responsible for implementing the multiple access scheme in a protocol-specific way. The choice of the MAC protocol heavily influences how much energy the device spends on communication. The major sources of energy waste that can be influenced by the the MAC protocol will be briefly covered in the following. From these observations, conclusions are drawn for the design of energy-efficient wireless MAC protocols.

A wireless transceiver consumes energy when transmitting, receiving, and also when just listening on the channel for incoming data. The order of magnitude for all three tasks is roughly the same [FN01] for current short-range wireless communication systems like WLAN. The reason for this is the relatively high power consumption of the radio frequency (RF) transceiver circuitry, no matter if the circuitry is actually in transmission, receive, or listening mode. The RF

transceiver must provide a carrier frequency in the 2.4 GHz band and have some filters and amplifiers powered on, in all three modes.

Therefore, the highest energy efficiency is achieved when transmitters and receivers are exactly synchronized and operate in transmission or reception mode only when required, with an output power level that is just sufficient for an error-free transmission. Obviously, these ideal conditions cannot be achieved in practice. The main causes of energy loss in wireless communications are described in the following.

Idle listening refers to the situation that one or more devices are listening for data, but there is no transmitter sending any data. Similarly, *overemitting* describes a situation where a node transmits data and no other device is receiving the message. Another source of energy misuse is *overhearing*. In that case, a receiver consumes power to receive and decode a message, only to find out that it is not the destination and it has to discard this message. Moreover, *collisions* of packets lead to an increased energy consumption, as these packets have to be retransmitted. The amount of *protocol overhead* in the form of control information conveyed in each message influences energy efficiency of the protocol.

Finally, the used data rate and transmission power level are important parameters that determine how much energy per bit is spent in communication. A high data rate reduces the time required for transmission, but may also require a higher signal-to-noise ratio and hence higher transmission power for an error-free transmission.

The design of the wireless MAC protocol heavily influences how energy-efficient the communication system will be, that is what sources of energy waste will occur and how frequently. As examples of popular MAC protocols employed in wireless sensor networks, the following section will introduce the design principles of S-MAC and T-MAC, and the IEEE 802.15.3 MAC protocol designed for WPANs, since it has been used to validate our design methodology and novel ideas.

S-MAC

The sensor-MAC (S-MAC) protocol has been proposed by Ye *et al* [YHE02] to address the specific requirements of multi-hop wireless sensor networks. Such networks consist of a large number of distributed nodes that cooperate to perform a common task, such as environmental monitoring.

Each wireless sensor node has one or more sensors, embedded processor and

low-power radio, and is normally battery operated. Due to the, possibly, large network size, recharging of batteries is not feasible in many intended application scenarios. Therefore, energy efficiency is the major design goal for wireless sensor networks in order to prolong the lifetime of the network. Typically, the amount of energy spent to communicate a single bit can be used to perform hundreds of microcontroller instructions on a node. This requires a new MAC protocol approach which significantly reduces the communication overhead compared to, for instance, the IEEE 802.11 MAC protocol, where nodes spend a lot of energy for idle listening.

The basic scheme of the S-MAC protocol is a periodic listen and sleep cycle. Each node goes into sleep mode for some time, and then wakes up and listens to see if any other node wants to send data to it. While sleeping, nodes consume only a fraction of the power when being active. The ratio of the active communication period is called duty cycle. A low duty cycle increases network lifetime at the expense of higher latency and lower responsiveness of nodes. WSN applications, however, usually do not require a level of responsiveness that must be provided, for example, in WLANs.

Neighboring nodes must synchronize their wake-up schedules in order to communicate. For this purpose, nodes broadcast information about their wake-up and listen interval to all nodes in their communication range in a periodic SYNC packet. This way, virtual clusters are formed in the WSN where all nodes in a cluster share the same schedule.

The S-MAC protocol uses a contention-based access scheme in the listen period. An RTS/CTS handshake is placed before any data transmission to address the hidden terminal problem.

The hidden terminal problem refers to a situation where two devices cannot sense the frame transmission of the other, however there is a region where both transmission ranges overlap and a receiver could detect frames from both devices. In this case, even if both devices that wish to transmit a frame sense the channel as idle before actually starting the transmission, a frame collision may occur at a receiver in the region of overlapping transmission ranges.

The RTS/CTS handshake is a mechanism to mitigate the effects of the hidden terminal problem: Before the actual frame transmission starts, the transmitter device sends a short, so-called RTS (Ready-To-Send) frame to the receiver. When received correctly, the receiver responds immediately with a short CTS (Clear-To-

Send) frame. If a collision occurs during the RTS frame transmission, the CTS frame will not be sent and the handshake is initiated once more. The advantage of this method is that only a short frame was affected by the collision, thus reducing the amount of wasted energy.

S-MAC has been implemented on the well-known Berkeley motes, a family of sensor nodes from UC Berkeley that run the TinyOS embedded operating system [HSW+00]. The low complexity of the protocol and the moderate physical layer data rates used by the Berkeley motes allowed a pure software implementation. The software has been written manually in the nesC language, a dialect of the C programming language, and amounts to roughly 3000 lines of code.

T-MAC

Timeout-MAC (T-MAC), an improvement of S-MAC, has been proposed by van Dam *et al* [vDL03]. In contrast to S-MAC, T-MAC operates with a fixed period (615 ms) of the listen and sleep cycle and uses a time-out mechanism to dynamically adapt the end of the listen period. If a node does not detect an incoming message or collision within a timeout value (15 ms) after the last transmitted frame, it assumes that no neighbor wants to communicate with it and goes to sleep.

The adaptive duty-cycle allows T-MAC to automatically adjust to fluctuations in network traffic. The down-side of T-MAC's rather aggressive power-down policy, however, is that nodes often go to sleep too early: when a node s wants to send a message to r, but looses contention to a third node n that is not a common neighbor, s must remain silent and r goes to sleep. After n's transmission finishes, s will send out an RTS to sleeping r and receive no matching CTS, hence, s must wait until the next listen cycle to try again.

IEEE 802.15.3 MAC protocol

Similar to the well-known IEEE 802.11 standard [IEE99], IEEE 802.15.3 comprises specifications for the MAC and physical layers. This standard provides data rates from 11 to 55 Mbit/s at distances of greater than 70 meters to enable wireless personal area networking [IEE03a]. In contrast to IEEE 802.11 WLAN and to the previously introduced protocols, it provides QoS to support multimedia applications.

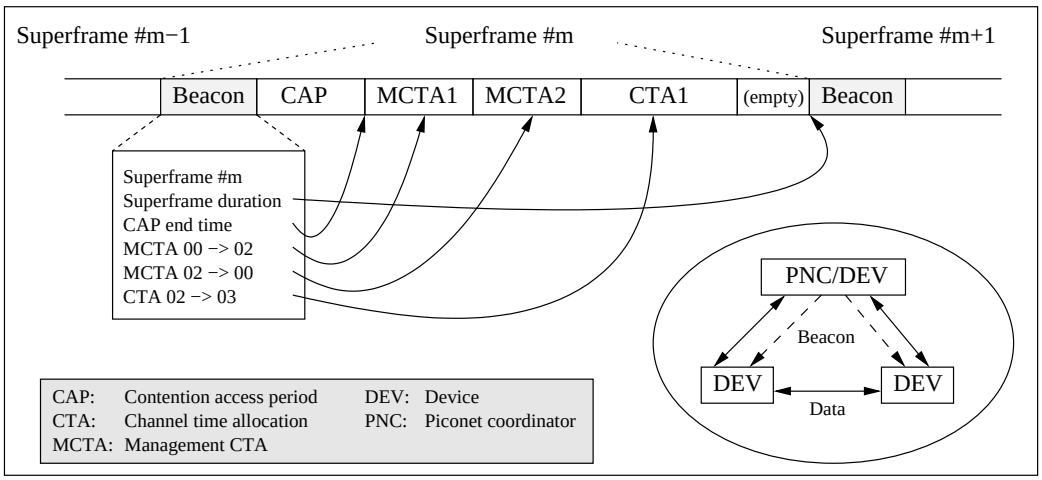


Figure 2.4: Superframe structure, beacon contents, and basic network topology defined in the IEEE 802.15.3 standard.

In each IEEE 802.15.3 wireless network[1] there is one so-called *piconet coordinator (PNC)* that is responsible for managing the associated devices and granting them channel access time. Synchronization in the network is maintained by the PNC broadcasting beacon frames. All devices must be in communication range of the PNC. The beacon contains information about the piconet and, most importantly, a list of reserved time slots indicating their position relative to the start of the beacon frame, their duration, as well as the sender and receiver. Devices may communicate directly in a peer-to-peer fashion.

The time interval between two consecutive beacon frames is termed *superframe*. The superframe duration is a fixed time interval, such that a strict time synchronization between the devices is maintained. This is very useful to support isochronous data services and to allow devices to sleep for a couple of superframes without having to go through a long resynchronization period.

The protocol allows an optional contention access period (CAP) following directly after the beacon. It resembles the same behavior as the distributed coordination function (DCF) of the IEEE 802.11 MAC protocol. The superframe structure and basic topology of the IEEE 802.15.3 standard are illustrated in Figure 2.4.

The protocol functionality defined in the standard is so extensive and complex,

[1]The term *piconet* is used in the standard to denote a set of logically associated devices belonging to the same network and having a common coordinator.

especially for the PNC, that a list of the main features must suffice to become a feeling for what a challenge and potential for errors the protocol design presents. The protocol includes: scanning for available piconets, starting a new piconet or synchronizing on an existing one, contention-based and TDMA-based channel access, error control and ARQ, association and disassociation, managing asynchronous channel time requests, managing[2] isochronous streams, managing power save sets, scheduling time slots considering available channel time and the power save mode of devices, beacon generation and parsing, encryption and decryption, etc.

The most processing-intensive and timing-critical functionality is directly related to the channel access mechanism, namely CRC calculation and check, acknowledgment generation, frame transmission of the right frame at an exact point in time, as well as the AES algorithm for encryption and decryption.

The ARQ schemes applied by this MAC protocol are a timeout-driven[3] Stop-and-Wait for asynchronous data and command frames, and Selective Repeat as an option for isochronous streams. The latter is termed delayed ACK in the standard. Window-based flow control is realized by these delayed ACK frames, as well. The receiver of streaming data indicates how many data frames may be sent before a window update has to be requested.

We have modeled the complete IEEE 802.15.3 MAC protocol in SDL. It serves as an example of the application of our protocol design flow for embedded systems. The SDL model will be explained in some detail in Chapter 7. In that chapter we will also present the full implementation of the IEEE 802.15.3 MAC protocol including a hardware accelerator design.

2.1.3 Specification and design of communication protocols

This section shall provide an overview on languages and tools commonly used for protocol engineering. Protocol specification and design languages shall fulfill a number of goals:

An *unambiguous description* of protocol behavior is required for specification, implementation, and test. In contrast to natural language descriptions, which leave room for interpretation, a formal language captures protocol behavior and its data types in a precise way. Formal specification of protocols has the potential

[2] creation, modification, and termination

[3] An immediate acknowledgment frame must be transmitted 10 μs after the end of the received frame.

of revealing inconsistencies or missing definitions at a very early stage in the design process. This way, the cost and development time for protocols can be reduced significantly. As a guide for protocol implementers, the IEEE Standards Association, for instance, supplements its standards with SDL models of the specified protocols in some cases. Finally, an unambiguous protocol specification can be used to derive test cases to test a protocol implementation.

In the same line, *formal verification* of protocols becomes increasingly important due to the inherent complexity and concurrency of communication protocols and, at the same time, high demands on reliability and dependability. Formal verification is a strict mathematical approach towards proving certain properties of a specification. Therefore, protocol specification languages should lend themselves to formal verification.

Last but not least, specification languages shall support a *simple transition to protocol implementations*. Firstly, this saves development time by avoiding duplicate design effort and, secondly, ensures that the verified properties of the protocol specification are transferred to the implementation model.

The major difficulty lies in designing a specification language that is expressive enough to satisfy the protocol designer's needs, while formal verification approaches are based on simple and mathematically tractable models [ZHT93]. Various formal description techniques (FDT) for the specification and design of distributed and reactive systems have been proposed in the literature, and some of them have gained wider application in industrial projects.

SDL [ITU02] and Estelle [ISO][DB89] are two popular protocol specification languages, which are based on finite state machine models. LOTOS (Language of Temporal Ordering Specification) is a theoretical framework based on algebraic concepts that originate from Milner's Calculus of Communicating Systems (CCS) [Mil80] and Hoare's Communicating Sequential Processes (CSP) [Hoa85]. All three specification languages have become international standards.

Other FDTs make use of Petri nets, process algebras, or temporal logic. They have received more attention in the design and verification of concurrent and reactive systems where design errors could have direct catastrophic consequences. Such hard real-time systems are not addressed by this thesis (see also the discussion in Section 1.2).

In the following section, the specification language SDL shall be presented in more detail. Thereby we intend to motivate the use of SDL in our design flow,

introduce its language concepts and provide a background that later chapters will build upon. The introduction of other popular protocol specification languages and FDTs would go beyond the scope of this thesis. We refer the reader to [Hog89] for an overview on Estelle and LOTOS, and to [Pop06] for an introduction to Message Sequence Charts. The design or extension of a protocol specification language is outside the focus of this work.

Specification and Description Language (SDL)

History Originally addressing the specification of telecommunication systems, SDL has evolved into a language that is suitable for the specification of any reactive, distributed system. Studies and a first, small standard have been produced by the CCITT (now ITU-T) already in the 1970s, when the need for a high-level, unambiguous description of telephony systems became apparent [RS82]. The standard formally defines the semantics of the language and has been updated every four years. Today, SDL includes extensions for object-oriented modeling and design.

Modeling of behavior SDL is a *constructive* FDT, which means that it is used to develop abstract protocol models. The execution of the abstract model specifies the behavior of the communicating protocol entities. In contrast, *descriptive* FDTs, for which Temporal Logic is an example, only express properties, that must be fulfilled by the protocol, by means of logic formulas.

Communicating extended finite state machines are the basis for behavioral description in SDL. With finite state machines (FSM) it is possible to model behavior by means of a set of states and transitions from one state to another one. These transitions are triggered by an external input and can have associated output actions.

As an example, a protocol description could introduce states such as *Connected*, *Data transfer*, and *Disconnected* with transitions between these states that are initiated by higher layer requests or the reception of messages from a peer protocol entity. Transition actions could be the output of a service primitive to the higher protocol layer or a message to the peer entity.

Often, the protocol state machine must keep additional information such as sequence numbers of received and transmitted PDUs. Since such variables can take many different values, this would lead to a large number of states if each

variable assignment would be represented by a new state. Therefore, *extended* FSMs allow the use of variables in addition to states.

The behavioral description in SDL is composed of multiple concurrent extended FSM models. Each state machine is encapsulated in a *process* [4]. Processes communicate with each other by the exchange of *asynchronous signals*, remote procedure call, and shared variables. Asynchronous signals may carry any number of data parameters.

Each process has got its own signal input queue, which is of infinite size. Signals sent to a process are first stored in that queue, until they are consumed. The signals are consumed when the process is not currently performing a transition, i.e. the FSM is waiting in a state. Then, the first signal in the queue is fetched and will start the transition that is associated with the current state of the process and the input signal type. If no such transition is specified, the signal is simply discarded, unless the designer explicitly specified that this signal type has to be saved in the queue. In both cases, the next signal in the queue will be processed.

Processes start with an initial transition that is not triggered by an input signal. The first transition usually performs all required initializations and ends in a state of the process FSM. The behavior of transitions can be complex. SDL allows to model the control flow similar to imperative programming languages, i.e. it provides conditional statements, loops, sequential statements, procedure calls, etc. A transition may also contain signal output to other processes. Transitions are executed until the control flow reaches a next state of the FSM. SDL has got a graphical and a textual representation. Because of the expressive nature of the language, it is easy to learn and has thus become a popular tool for protocol specification.

A simplified example of the modeling of behavior within an SDL process is given in Fig. 2.5. It shows a state machine that consists of the states *Disconnected*, *ConnConfirm*, and *Connected* with three transitions between them. The transitions are triggered by the following input signals: *Conn.req* (request from higher layer), *ReceivePDU* (indication from lower protocol layer), and T (timeout signal). The figure also shows the initial transition, declarations, the task symbol, a procedure call, the symbol for a conditional statement (choice), and the output of signals.

[4] With SDL-2000, the term *agent* was introduced to denote all active components of an SDL specification.

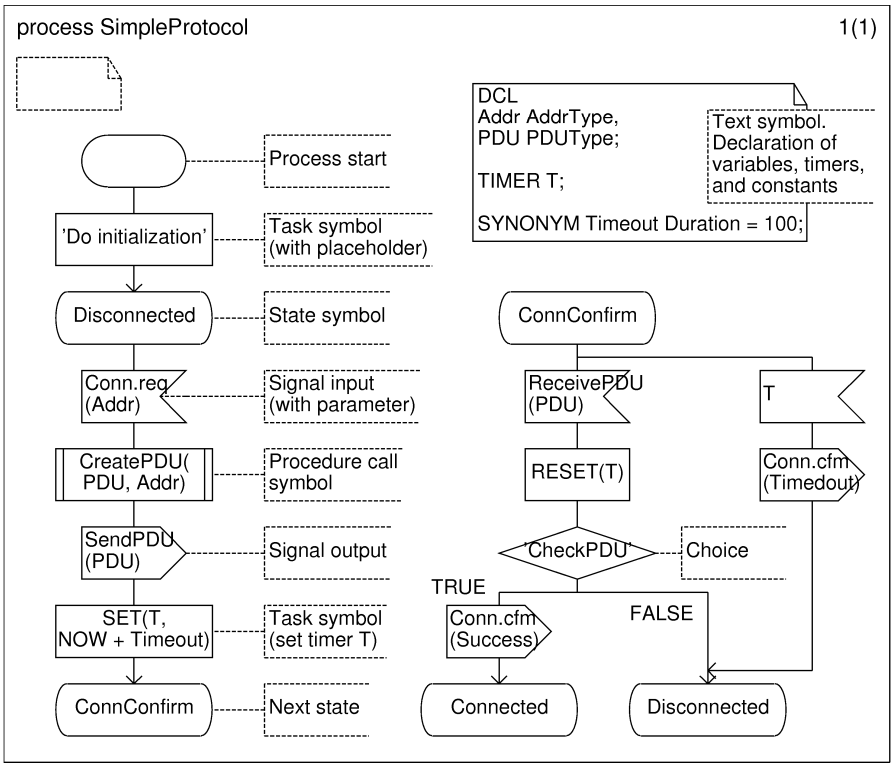


Figure 2.5: Fragment of a simplified protocol specification as part of an SDL process.

Timers The specification of timing behavior is an important aspect of protocol modeling. SDL provides the concept of *timers* for this purpose. Timers are local to SDL processes and can be set from any transition in that process. When the specified time interval has elapsed, a timer input signal is placed in the signal queue of the process. Like any other signal, it can be consumed when the process execution is in a state and has got a transition that is triggered by this timer signal. The same timer may also be reset by the process, which has the effect of deactivating the timer and removing the timer signal from the input queue, in case the time interval has already elapsed. The use of timers in an SDL process is illustrated in Fig. 2.5, as well.

Since SDL is an abstract specification language, the execution of transitions does not consume time, nor is it possible to annotate transitions with an execution time. When simulating an abstract SDL model, time advances only after all

transitions have been executed. In this case, new transitions are enabled only by external signals or when a timer expires.

In implementations derived from SDL specifications, the execution of transitions takes real time. Real-time systems require an action or response from the system within a bounded time interval. The SDL timer mechanism seems to be appropriate to specify exactly such behavior. However, due to the asynchronous nature of the timer mechanism, it cannot be guaranteed that the process executes the transition associated with the timer signal exactly (or even with a bounded delay) after the timer was set. There are three unpredictable delays before this transition is triggered [Leu95]:

- The timer signal is created and placed in the input queue of the timed process only some time after the timer expired.
- There may be other signals in the queue that have arrived earlier than the timer signal and that still need to be consumed by the process.
- Even if the timer signal is the only signal in the input queue, the process may still be executing a transition, from which it cannot be interrupted. Hence, an unknown time will pass before the process has reached the next state and is able to react on the timer signal.

In summary, SDL is very useful to prove liveness properties, i.e. that some action will be executed *eventually*, but is ill-suited to specify real-time constraints, i.e. that an action will be executed within a bounded delay.

A question that is often encountered in protocol design and implementation is whether the implementation is fast enough and meets its timing requirements. The design flow should provide additional support for the protocol engineer to identify timing bottlenecks, as this is not directly possible with SDL.

Structure of SDL specifications An SDL specification is a formal description of both the architecture and the behavior of a system. Executable specifications are hierarchically structured. The top-level entity is called *system*. The system diagram is a container for lower-level structural entities, called *blocks*, that themselves may contain other blocks or *processes*. SDL processes capture the behavioral description of a specification, the blocks are used to provide structure and for initialization purposes. Connections for communication between blocks and processes must be specified, they are called *channels*.

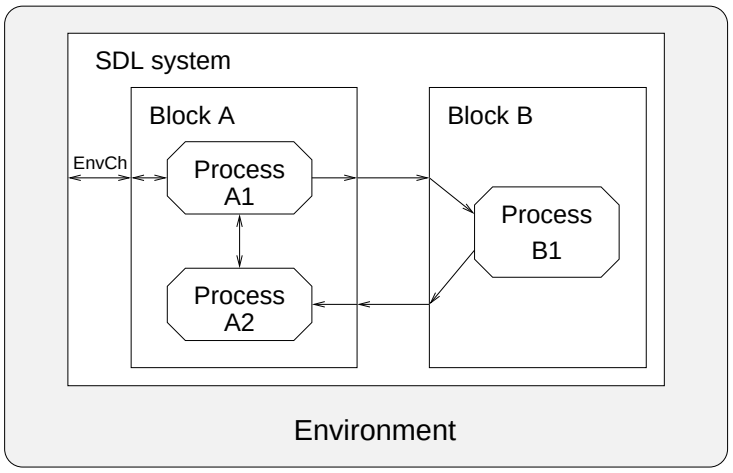


Figure 2.6: Illustration of an SDL system embedded in its environment.

An SDL system is embedded in an *environment*, from which it may receive and send signals. This is shown in Fig. 2.6. The system specification in this simple example is composed of two blocks, *A* and *B*. Block *A* contains two processes *A1* and *A2*, block *B* only process *B1*. Signal channels between the processes and blocks as well as the environment (*EnvCh*) are shown. Names and signal lists are omitted for simplicity.

SDL specifications may also contain *packages*. Packages are collections of reusable components, such as data and signal type definitions, procedures, block and process types. References of these types can be instantiated in the system specification.

Formal semantics With SDL-2000 a new formal semantics of the language was introduced. The old semantics suffered from a very extensive description using a combination of the Meta IV and CSP languages, and was not executable. The new formal semantics was designed to be more practical.

It consists of two parts: static and dynamic semantics. The *static semantics* describes the SDL syntax and all language elements. SDL is a very rich language, therefore a transformation of language constructs to an abstract core language is part of the static semantics of SDL. By means of rewrite rules, the SDL syntax tree is transformed into an abstract syntax tree (AST).

The *dynamic semantics* associates with each SDL specification, represented as

an abstract syntax tree, a particular multi-agent real-time *Abstract State Machine* (ASM) [FHvLP00]. Intuitively, an ASM consists of a set of autonomous ASM agents cooperatively performing concurrent machine runs. The behavior of ASM agents is determined by ASM programs, each consisting of a set of transition rules, which define the set of possible runs.

There are strong structural similarities between SDL systems and ASMs. For instance, ASM agents will be introduced for SDL agents, agent sets, and channel segments. ASM agents can be created during the system initialization phase as well as dynamically, which allows, e.g., to directly represent dynamic process creation in the underlying model. The execution of a system starts with the creation of a single ASM agent for the SDL unit "system". This ASM agent then creates further ASM agents according to the substructure of the system as defined by the specification, and associates an ASM program with each of them.

ASM programs are determined by the kind of the SDL unit modeled by a given ASM agent. Following an abstract operational view, behavior is expressed in terms of SDL abstract machine runs. The SDL abstract machine is independent of a particular SDL specification.

Formal semantics of a specification language are important to be able to analyze and prove properties of specifications. In order to reason about properties of an implementation derived from a correct, abstract specification in SDL, the semantics of SDL must be preserved in the transformation from abstract specification to implementation.

Tool support Formal languages are often used to communicate between design engineers. During implementation, the specifications are taken as input and the product is described in a programming language. A key feature is that SDL can be used for specification, design and implementation, thus avoiding errors introduced when converting between different languages for different design phases.

The availability of tool support for modeling, simulation, automatic code generation, and verification has contributed to the success of SDL as a system specification language. The most popular SDL tools are the (commercial) integrated development environments Telelogic TAU [Tel06] and Cinderella SDL [Cin07]. More details on tool support for the protocol design flow are presented in Sect. 2.1.4.

2.1.4 Protocol development

This section shall provide a rough overview on the protocol development process. A more in-depth presentation and discussion of specific design approaches is left to Chapter 3.

The protocol development process bears some similarities to the way how software is developed. There are, however, also some additional elements, which are specific to protocol engineering, and that shall be illustrated briefly before we begin with a systematic presentation of the development tasks.

Communication protocols are used to form networks of potentially heterogeneous devices. This means that the same protocol is implemented and running on diverse hardware/software platforms, i.e. on different processors and, possibly, using different operating systems. All these specific implementations are derived from a single protocol specification and must adhere to it in order to facilitate communication between these devices. This property is called *interoperability*.

Hence, the protocol specification must be unambiguous and abstract, i.e. implementation-independent. As introduced in the previous section, formal description techniques have been applied for this purpose. Since the protocol development is relying on the specification as the basis for implementations, the verification of the protocol specification is a vital step. Thereby, certain properties of the specification, such as the presence of deadlocks, livelocks or unreachable statements, are checked.

Protocol specifications often contain options and implementation-dependent decisions. Therefore, any protocol implementation must not only be tested for conformance with the specification, but also for interoperability with other implementations of the same protocol.

The phases of a typical protocol development process have been described by König [Kön03] and are depicted in Fig. 2.7. The figure shows the activities, their results and interdependencies. In the following, we will present the main objectives of these phases, methods and tools that are commonly used, as well as open research problems. Those development phases that are most relevant within the scope of this thesis are treated in more detail.

Requirements analysis Like any other engineering task, the development of a communication service and a protocol that provides this service starts with an analysis of the requirements it must fulfill. This analysis is often driven by use cases

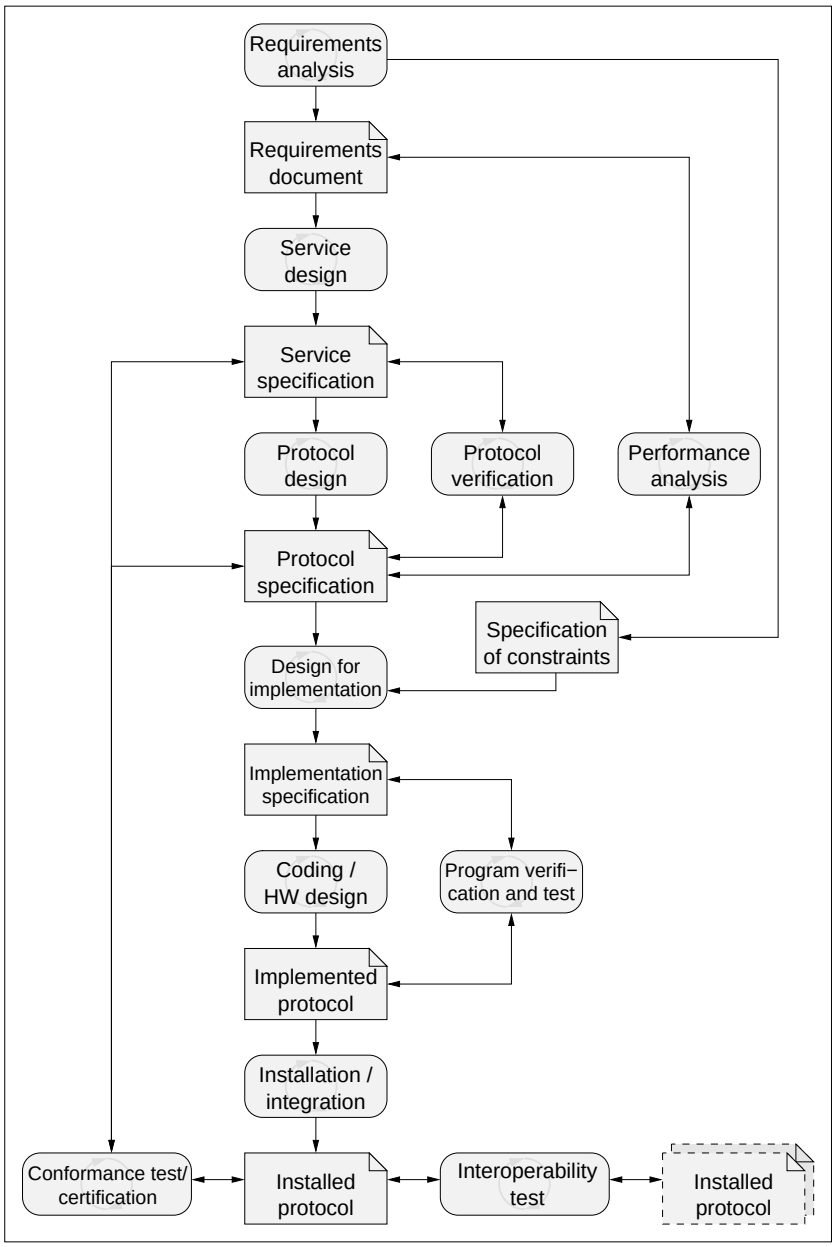


Figure 2.7: Protocol development phases and their results based on [Kön03].

or application scenarios and, typically, leads to informal descriptions of system requirements. In the software development process, the Unified Modeling Language (UML) has been found useful to specify requirements formally. In [dV02] the UML-based specification of requirements of real-time systems has been proposed.

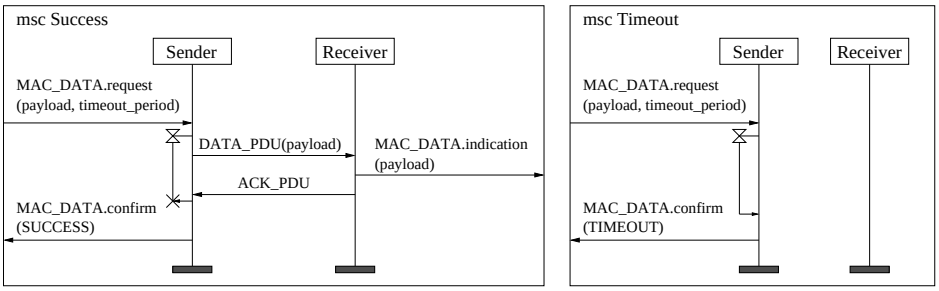


Figure 2.8: Two exemplary Message Sequence Charts as part of a service specification.

Service design The goal of the service design activity is to create the service specification. This specification shall clearly define the interactions of the communication service with its environment. The communication service may receive and send service primitives through service access points (SAP).

The service designers specify the set of primitives and their parameters. They also describe when these primitives may be issued to the communication service and what effects to expect. That could be primitives sent out by the local service provider or at the communication partners some time after issuing the initial service primitive.

The service specification is taken as a reference document for the following development process, in particular the design of the protocol. Because of this high importance, formal descriptions are often used to capture the service specification unambiguously. This specification, and also the protocol specification, are abstract in the sense that they are implementation-independent, to facilitate implementations on heterogeneous platforms.

Message Sequence Charts (MSC) [ITU04] is a standardized, formal language to specify the interactions between communicating entities in a graphical and textual form. MSCs have become very popular to capture possible communication sequences. With the standard MSC-2000, data types were introduced to the language. MSC is therefore a convenient tool for service design.

In Fig. 2.8, two MSCs are presented that show the possible effects of a service primitive requesting the transmission of data. In the first case, left in the figure, the transmission is successful, while in the other case a timeout occurred.

Protocol design In the protocol design phase it is specified *how* the service will be provided by the protocol. Some of the most common protocol mechanisms have been presented already in Sect. 2.1.1. In addition to the behavioral description, the format of protocol data units (PDU) exchanged with peer protocol entities is precisely defined. The specification must also cover exceptional cases such as how to handle invalid service primitives or received PDUs. Besides this, the specification must be unambiguous and shall be implementation-independent.

For this reason, as already introduced in Sect. 2.1.3, formal description techniques (FDTs) have been applied for protocol specification. SDL is one of the most popular formal specification languages due to its expressiveness, formal semantics, and wide tool support. However, many standards bodies provide only plain (English) textual specifications for their protocols, that are sometimes supplemented by formal descriptions in order to resolve ambiguities, but are far from being complete. Many of today's protocol implementations in the internet domain are developed without the use of formal methods. This is caused by the need to deliver products within a short time and the fact that FDT tools often require more time for the development process and create less efficient code.

In the literature, systematic approaches to protocol design have been proposed. According to [PS91], protocol design approaches can be classified into two main categories: *synthetic* and *analytic* methods. The synthetic design methodologies aim to generate designs that are correct by construction, while the analytic methods iterate a sequence of (re)design, analysis, error detection and correction, and may lead to incomplete and erroneous designs. Even though synthetic design approaches promise to be less time-consuming and introduce less errors, up to now there is no mature design methodology due to the lack of practical tools and the diverse nature of communication protocols [Kön03].

In [WFGG04], a design approach that introduces SDL design patterns and advocates the composition of protocols from micro-protocols is presented. Design patterns express successful solutions to common design problems in the object-oriented software development process and can be considered as reusable micro-architectures that contribute to an overall system architecture [GHJV93]. Similarly, the SDL design pattern library shall provide configurable building blocks for protocol design in SDL. As an example, a well-known mechanism found in reliable systems—a watchdog and a heartbeat—has been turned into the two reusable SDL design patterns Watchdog and Heartbeat.

Micro-protocols resemble to a large extent the various protocol mechanisms discussed in Sect. 2.1.1, such as error control, flow control, etc. Unfortunately, protocols are often much more complex and exhibit interdependencies between their mechanisms such that it is difficult to apply this design paradigm in its purest form in practice. However, the SDL model for the IEEE 802.15.3 MAC protocol that we have developed (cf. Sect. 7.1) was designed with reusability in mind by encapsulating independent protocol functions in SDL processes and by applying a layered design approach.

Protocol verification This stage in the development process shall assert if the protocol specification actually provides the communication service it was designed for, and if the specification fulfills certain general properties. Such properties are, for instance, that there are no states in which the protocol is stuck (*deadlock-free protocol*) or from which it cannot reach any other state (*livelock-free protocol*). Protocol verification is necessary because the employed design methodologies do not guarantee that the specification provides the intended service, let alone that it fulfills the safety properties.

The protocol specification serves as a reference for the following design phases and potentially many different implementations. It is well known that errors that have not been detected in the specification cause escalating costs for detecting and correcting them during implementation, test, or even in production. This is the reason why formal verification techniques have first been introduced commercially to hardware design [Kur97] and why formal verification has attracted a lot of research work in the past decades. Exhaustive implementation tests are not feasible due to the complexity of distributed systems and the limited available time for testing, therefore even carefully tested systems still contain residual errors. Verification, on the other hand, has the objective of proving that the mathematical model of a system has the desired properties.

Protocol verification is based on formal models with clearly defined semantics. There are two general approaches towards formal verification: model checking and theorem proving [HS96].

Model checking is an automatic procedure which takes as inputs the formal model and the specified properties that shall be checked. It produces an output whether or not the properties hold. There are many different model checking techniques, and the properties are often expressed by means of temporal logic

formulas, such as LTL or CTL. The model checker analyzes all possible sequences of protocol behavior. This is only possible for finite models. Due to the high complexity and the use of variables with a large range of values in the model, model checking suffers from the so-called state explosion problem. Approaches to reduce the state space to be held in memory and careful modeling enable the verification of not just trivial models.

The other approach—automated theorem proving—is based on reasoning about the model. By logical inference, certain properties about the model shall be deduced. Theorem proving may require the interaction of the user to cope with the complexity of the model. There were also attempts to employ theorem proving with the goal of reducing the state space for a subsequent model checker run [HS96].

Protocol verification tools for the most popular formal description techniques, such as SDL or LOTOS, are available. The commercial tool set Telelogic TAU SDL Suite [Tel06], for instance, contains at least basic support for a reachability analysis of the SDL model, which helps to identify design errors early in the process. Other formal verification approaches based on SDL have been reported in the literature [BDHS00] [MIJ03]. They make use of model checkers such as SPIN [Hol97] (for the formal language PROMELA), the IF tool set, or CADP [GLM02] (based on LOTOS) and require a conversion of the SDL model into the respective internal representation suitable for the model checker. The translation of SDL models into Petri Net representations and their verification has also been considered [FDT95] [AHV03].

Performance analysis While the goal of protocol verification is to check functional correctness of the specification, performance analysis considers the so called *non-functional properties* of the protocol. These properties capture, for instance, the timing behavior, achievable throughput and latency, the protocol behavior under varying load conditions or in the presence of communication errors, or the required resources such as buffer space.

Some of these values are difficult to obtain on an abstract specification level, since the target platform as well as implementation-specific parameters have an influence on the protocol performance. Therefore, performance analysis is per se a task that is carried out throughout the development process. However, the earlier it is applied, the easier it is to make changes to the protocol.

In [HHM98] stochastic process algebras and a corresponding extension of the

specification language LOTOS were presented. The mathematical model behind the communicating processes are Markovian processes. Markovian processes are useful to model queueing systems. They allow to model the probability of state transitions and to analyze the system at its equilibrium point. Similar approaches are based on Timed Automata or Timed Petri Nets, which allow to annotate transitions or places with stochastic, minimum or maximum times.

Another common approach for performance testing is to simulate the protocol specification and provide a simulation environment which can be used to vary the delay of messages, introduce different load models or exceptional cases such as the loss of messages. The protocol can then be analyzed under these conditions. Drawbacks of this method are that it is slow, since it requires complete simulation runs, and that it is limited to the modeled and simulated conditions. However, it can be set up with little effort and serves as a protocol validation at the same time. Examples for this approach are the SDL-based tools PerfSDL [Mal99] and QUEST [HHL+01].

Implementation design The objective of the implementation design activity is to provide a mapping of the abstract protocol specification to the target platform. Based on this high-level design document the actual implementation is developed. For this purpose, typically a number of implementation-specific protocol options must be defined and any non-determinisms must be resolved.

In the protocol engineering literature, protocols were mostly considered as software products. Consequently, implementation design needs to consider the integration of the protocol implementation into the operating system, particularly the mapping to operating system processes and ways of interacting with the rest of the system [Kön03]. For mixed hardware/software implementations the designer is confronted with the task of partitioning the protocol functionality into hardware and software and to define the architecture and interfaces of the hardware blocks. When presenting our cosimulation approach we will discuss this topic in more detail.

In particular for the implementation of multi-layer communication systems, two basic models of how to use operating system (OS) processes have been described [Svo89]. They are called server model and activity-thread model. They both assume that FSMs are used to describe the protocol behavior. In Fig. 2.9, their basic principles are shown schematically.

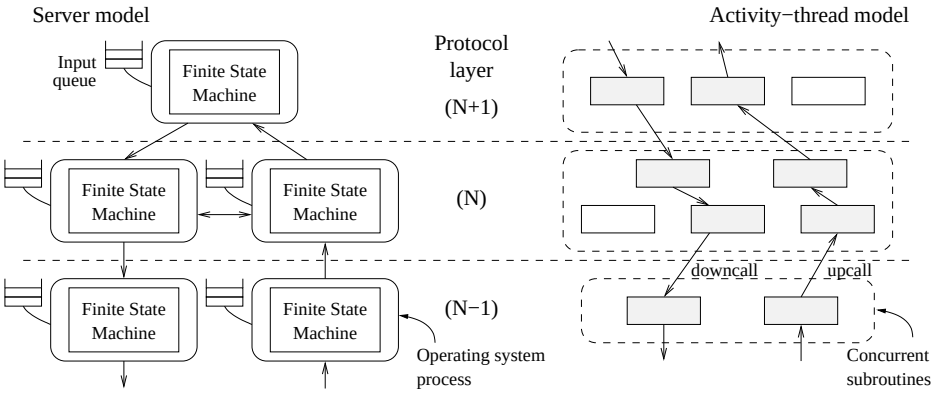


Figure 2.9: Implementation models for multi-layer communication systems: server model (*left*) and activity-thread model (*right*).

In the *server model*, one OS process is used per state machine in the specification. When an input signal is sent to this state machine, the corresponding OS process is activated and executes the transition depending on its current state and the received input. Finally, the process is waiting for new input. This is a very simple principle, however it has the disadvantage that for each new input, i.e. service request or received PDU, a process switch will occur. Generally, such a context switch is a time-consuming operation and has an adverse effect on the performance. Furthermore, buffer space to queue input signals while the process is not able to consume them must be allocated.

In contrast to the server model, the *activity-thread model* [Cla85] avoids frequent context switches, as the processing of input events is handled by procedure calls. Each protocol layer or state machine is implemented as a collection of subroutines that may be executed concurrently. The subroutines contain the transitions related to a single input event. If a transition creates an output event for another FSM, the corresponding procedure is called. This way, the execution path passes through the protocol layers without the need for a context switch. Both directions are possible: *downcalls* from a higher layer service user and *upcalls* when a PDU is received. A careful design is required to enable concurrency and handle possible cyclic dependencies between the procedure calls. This additional effort pays off, since the activity-thread model can achieve a higher efficiency than the server model.

The automatic transformation of abstract specifications to implementations

is supported for many formal specification languages. In this case, many design decisions are already fixed by the tool. The server model is often used as the process model for automatic transformations. An example for this is the code generator CAdvanced from Telelogic. The efficiency of manual implementations, however, is not achieved by such tool-generated implementations without further optimizations.

Implementation The implementation phase comprises coding and in some cases hardware design. For these development tasks, specific methods and tools, such as code review, implementation test, and hardware simulation, which are originating from the software engineering and hardware design areas, are applied.

The abstract specification does not define *how* the concepts of timers, interprocess communication etc. are to be realized. This must be taken care of by the implementers, who have to find a solution to provide these services, possibly by relying on functionality provided by the operating system.

Any protocol implementation must fulfill the specification. To verify if a given implementation indeed conforms to the specification, additional tests are required. This will be discussed further below. Naturally, these tests cannot cover all possible protocol runs and every scenario due to the behavioral complexity of the implementation and the possible interactions with peer protocol entities. Therefore, even tested implementations may still contain errors that take years before they are discovered.

An approach to avoid these errors is to transform the specification by an automatic process into an implementation. As already stated above, these kind of implementations have not yet reached the performance of manual designs, mainly because the tools apply only syntactical transformations. However, it is a fast and easy solution to create working prototypes from a verified protocol specification. Tools have been developed that do not only create software prototypes, but also hardware components can be generated from, for example, SDL specifications for prototyping purposes [DSH99].

A typical software implementation of the server model, which is commonly used for prototyping from state machine-based formal specifications, consists of an operating system process for each state machine and an associated input signal queue. The process executes a loop: if there is an input in the queue, it is consumed and, depending on the state and the event type, the corresponding transition is

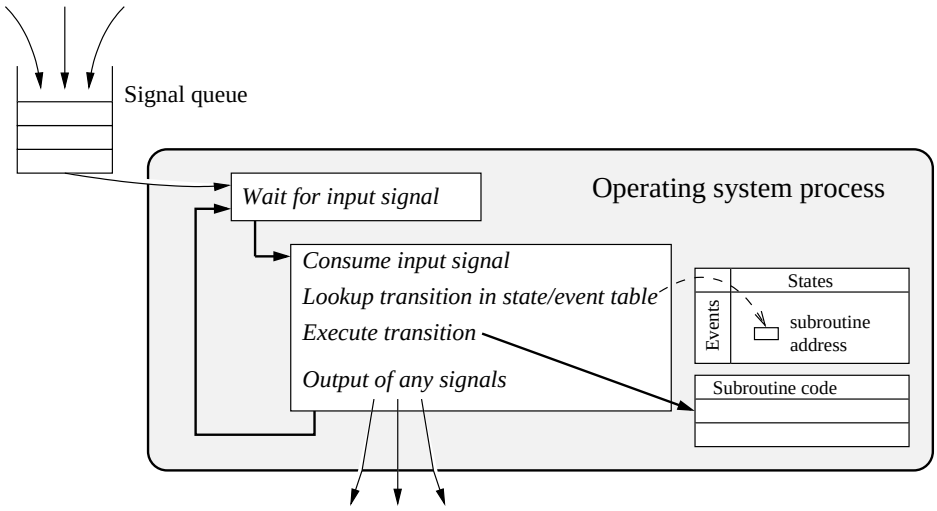


Figure 2.10: Possible software implementation of the server model using a table lookup to select the transition to be executed.

performed (cf. Fig. 2.10) [KK96]. Then, the process is waiting for the next input. The operating system is responsible for managing the queue and scheduling the process.

The choice of the right transition can be programmed either by means of nested *switch* statements or through a table lookup and following subroutine call. In protocols that manage multiple simultaneous connections, it is possible to instantiate a new state machine process for each new connection, or to handle all connections by a single process. In the latter case, the process needs to maintain separate sets of local variables for each connection, but does not require additional OS resources.

Integration The completed protocol implementation is finally integrated with the target platform. In particular, interfaces to other protocol layers or hardware have to be designed and methods for the operating system or applications to interact with the communication protocol have to be provided. This can be realized by means of an API (application programming interface), i.e. a procedural interface, or via a buffered interface. In the latter case, the communication with the protocol implementation is asynchronous.

Conformance test The purpose of protocol conformance testing is to check the capabilities and behavior of an implementation against the protocol specification. It is a *black-box test*, i.e. only the externally observable behavior of an implementation is examined. The conformance test can be part of a certification process that is conducted by a test laboratory to provide a product with a certification label.

Protocol testing has been an area of intense research activity. A great number of scientific contributions has been published in the proceedings of the international conference series TestCom [Tes].

The International Standardization Organization (ISO) has issued a set of standards relating to conformance testing. These are the Conformance Testing Methodology and Framework (CTMF) [ISO91], which was later complemented by the Formal Methods in Conformance Testing (FMCT) [ISO97] standard. Besides defining a number of terms and the processes for developing test suites and conducting tests, the standards also define the test notation Tree and Tabular Combined Notation (TTCN). It allows to describe test cases unambiguously, i.e. the test cases codify which responses are expected by the system-under-test as a reaction on certain stimuli. TTCN is the only standardized test notation. It is maintained by ETSI and currently available in its version TTCN-3.

Apart from the manual specification of test cases, the derivation of test cases from formal protocol specifications, e.g. state machine-based descriptions or temporal logic specifications, was studied. We will not get into an in-depth discussion of conformance testing, as this topic is complementary to our work.

Interoperability test The conformance of an implementation to a protocol standard does not yet guarantee that a trouble-free communication with alternative implementations of the same standard is possible. Reasons for this are differently implemented protocol options which may limit compatibility or any implementation-specific details not defined in the standard, such as timer values or specific algorithms, that could impede the interoperability of two implementations.

Therefore, implemented protocols must be tested in real networks with implementations of different vendors, in order to demonstrate their interoperability. In case that this was not successful, the cause has to be investigated and the implementation has to be altered, or, where appropriate, networks have to be configured

in such a way that only interoperable systems communicate with each other.

2.2 Hardware/software codesign

The functionality of many of today's electronic devices and embedded systems is realized by a mix of software as well as dedicated hardware[5].

As an example, mobile phones are typically based on a high-performance low-power CPU such as an ARM11 core, which performs higher layer protocol processing and a large variety of user applications. Additionally, the GSM, GPRS or UMTS baseband processing in the physical layer is implemented by a number of hardware macros for each communication standard and software running on a digital signal processor (DSP) for layer 1 control [Ram07]. Analog circuitry is used in the radio transceiver for transforming the baseband signals into analog signals in the desired frequency band and vice versa. Short-range radio communication interfaces, such as Bluetooth or NFC, in the mobile phone as well as additional functionalities, such as image processing from a built-in camera, are likewise realized by a mix of hardware and software.

Hardware/software codesign denotes the integrated design of systems that consist of hardware and software parts. Algorithms that allow an exploration of different design alternatives and provide estimations about their related costs form an essential part of codesign methodologies.

Hardware/software systems, of course, must be functionally correct and satisfy all timing requirements. However, beyond that they should require an as small as possible chip area and consume as little power as possible, while at the same time being flexible to future changes or extensions. These design objectives partly contradict each other, so that an acceptable and optimal tradeoff must be found by the designers. Hardware implementations can be very efficient in terms of power consumption and chip area, but often require a much longer design time than software and, once manufactured and unlike software, are impossible to change without an expensive redesign of the chip.

As shown in the introduction of this thesis, system complexity is continuously growing caused by the persistent trend of miniaturization in semiconductor technology. At the same time and aggravating the situation, product development

[5] in contrast to general-purpose hardware components, such as standard microprocessors on which the software is running

cycles and the time-to-market are becoming shorter. Wrong design decisions in early development stages, i.e. the high-level system design, are particularly time-consuming and costly to repair [Dol00],[KV04]. Therefore, tools that support the architectural design of systems and help to cope with the growing complexity are gaining more and more importance.

This section shall highlight the state-of-the-art of hardware/software codesign, irrespective of the specific focus on communication protocol design of our work. First we look at the architectural hardware components and standard software, such as operating systems for embedded systems, that today's systems are composed of. After that we look at ways to describe system behavior on different abstraction levels. Existing methods for an optimal mapping of system functionality specified in an abstract manner to hardware and software, and to estimate the quality of the partitioning, will be discussed afterwards. Finally, we present a tool that supports an integrated hardware/software codesign flow as an example.

2.2.1 Architectural components of hardware/software systems

Hereafter, we will present different implementation options for hardware/software systems. System functionality implemented in software is running on processors, while dedicated integrated circuits can be designed to realize hardware functionality. Typically, a mix of processor cores, dedicated hardware, and even reconfigurable hardware structures can be found in state-of-the-art complex systems, that consist of several millions of transistors on a single chip.

Software Software implementations require a processor on which they are executed. A characteristic of processors is their programmability, which means that they execute a sequence of instructions, the program. The instructions are stored in memory and must be fetched by the processor before execution. The programmability allows different kind of applications to be run on the processor, thereby creating great flexibility as program memory in most systems can be easily updated.

A number of different processor types and architectures have been designed to specifically target their application domains. *General-purpose processors* are typically found in desktop PCs and workstations. They have a very general instruction set, thus supporting a broad range of applications, from word processing to image or video processing and scientific computations. Modern high-performance

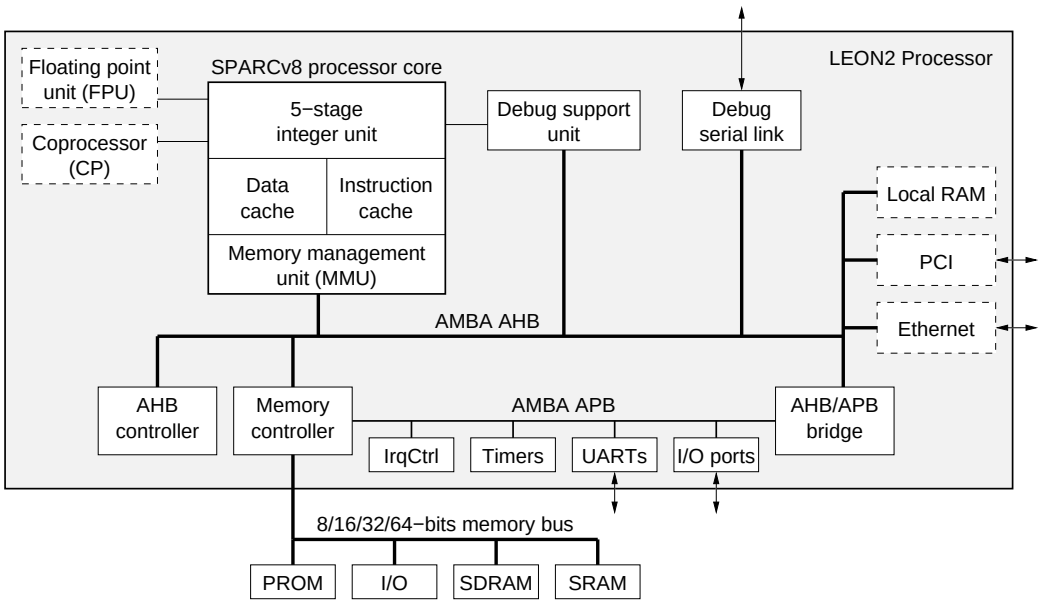


Figure 2.11: Architecture of the LEON2 processor as an example of a 32-bit microcontroller (adapted from [Gai05]).

general-purpose processors feature many optimizations to speed up processing, such as instruction pipelining, instruction-level parallelism (superscalar architecture), or branch prediction. A hierarchical memory architecture consisting of a small register file integrated with the processor, medium-sized caches located close to the processor, and large and slow external memories was introduced to ensure on average a high throughput of the processor.

Microcontrollers and *DSPs* are designed for specific application domains. Microcontrollers are equipped with a number of on-chip peripheral components, such as timers, interrupt controller, analog-to-digital converters, communication ports and general-purpose input/output ports to support embedded control applications. The architecture of the 32-bit LEON2 processor [Gai05] is shown as an example in Fig. 2.11. It is based on a SPARC-compatible processor core and features a number of peripheral modules (interrupt controller, timers, general-purpose input/output ports) and communication interfaces (Ethernet, PCI, UART).

DSPs have instruction sets that optimally support mathematical computations that are common in signal processing applications. Such computations are, for example, vector or matrix operations or Fourier transformations. For this purpose,

DSPs typically possess multiply-accumulate units, perform address calculations and loop counters in parallel to instruction processing, and have arithmetic instructions that operate on multiple subword operands in parallel.

Application specific instruction set processors (ASIP) constitute a further specialization. As the name suggests, ASIPs are designed for a particular application rather than application domain. To cut design costs, ASIPs often have less features than fully-fledged general-purpose processors, but their instruction set is optimized to increase the performance for the targeted class of applications. They are the right choice when DSPs or general-purpose processors are too slow for the intended application and a large degree of flexibility, that characterizes software implementations, must be maintained.

A crucial point for the use of DSPs or ASIPs in hardware/software systems is the availability of efficient compilers to generate optimized code for the processors from a high-level programming language such as C. It is a challenge to identify how programs written in a high-level programming language can be mapped in the most efficient way to the application-specific instructions of an ASIP and to take full advantage of any parallelism in the processor architecture. Because of these difficulties, hand-optimized assembler routines are often delivered in software libraries that can then be used by application programs written in C.

The simultaneous and automatic design of application-specific processors and compilers for their instruction sets has been a major focus of research in the hardware/software codesign area. The LISA (Language for Instruction Set Architectures) processor design platform [LIS07] [HML03], for instance, allows to generate software tools, such as assembler, linker, C compiler, and simulator, from an architecture and instruction set specification, and creates a synthesizable hardware description of the corresponding ASIP.

Operating systems are another important building block of hardware/software systems. Their main purpose is to provide abstractions from the underlying hardware, i.e. to provide device drivers, and to facilitate the concurrent execution of different software tasks by task scheduling and providing synchronization primitives. More advanced operating systems also contain a sophisticated memory management, communication protocol implementations, or support for real-time applications.

Operating systems facilitate the reuse of application code across different hardware platforms, hide the designer from hardware-dependent details, and allow the

development of software based on communicating sequential processes without the need to implement all the inter-process communication and scheduling schemes from scratch.

Even for the memory- and processing-limited microcontrollers often used in wireless sensor network applications, very lightweight operating systems that provide only a minimum set of functionality have been developed. TinyOS [LMP+05] is currently the most popular one of this kind in the academic research community, though it has also got some drawbacks. Other examples include Contiki [DGV04] and Reflex [Nol09] [WKSN08], which will be presented in more detail in a Sect. 5.2.1.

Hardware In contrast to the instruction-set architectures presented in the previous section, the hardware implementation options that will be discussed in the following are not programmable. Dedicated hardware implementations, or integrated circuits, are designed and optimized for a specific task. Since they do not deal with instructions, the required functionality is realized by a network of logic blocks.

Field programmable gate arrays (FPGA) and *mask programmable gate arrays (MPGA)* are particular implementation variants of integrated circuits. They have a regular structure of identical logic cells with an extensive, configurable interconnection network between them. The configuration of the connections and thereby of the circuit's functionality can be customized either at production time, i.e. by using an adequate mask (MPGA), or afterwards (in the field) by means of programmable configuration memory (FPGA). Programmable gate arrays can be freely configured for any required logic functionality, thus significantly reducing the cost of systems produced in low volume. Due to the vast number of interconnections, the maximum processing speed (clock frequency) is lower and the power consumption is higher compared with optimized, non-configurable circuits. Still, the processing throughput can be much higher than what is achievable with DSPs or general-purpose processors because of the larger degree of functional parallelism that can be achieved.

Non-configurable hardware implementations are called *application-specific integrated circuits (ASIC)*. They offer the least flexibility and have the highest production costs compared with all the implementation options mentioned above. However, they can be tuned to consume the least chip area and power consump-

tion and, therefore, are the technology of choice for large volume production or for systems with most stringent performance and low-power consumption requirements. The design of ASICs can be based on automatic synthesis methods from a high-level hardware description language like VHDL or Verilog. In that case, standard logic cells, such as inverters and flip-flops, or macro blocks are placed to provide the required functionality. This design style is known as semi-custom design. In full-custom design, the designers work on the layout level and optimize the physical structure of transistors and the location of connections and vias. This design method is very laborious, that's why it is used only for the most critical parts of a circuit, while semi-custom design and the advent of EDA tools made the design of today's complex systems with many millions of transistors possible.

Reconfigurable hardware architectures, in which the pre-designed atomic logic functions have a much higher granularity than in FPGAs, combine flexibility and the performance of ASICs. Such macro cells could be arithmetic operations (adders, multipliers), shift registers, or building blocks for a discrete Fourier transformation. These reconfigurable architectures are a relatively new research topic and have applications, for instance, in fast data-path architectures. They allow to customize an integrated circuit to a variety of algorithms to implement different communication standards, but do not cause as much overhead as FPGAs. Configuration memory is also reduced since, at a higher level of abstraction, there are less possibilities for customization.

An example for the combination of different hardware and software components on a single chip is the Pleiades platform for reconfigurable computing [AZW+02]. It is based on a general-purpose *control processor* which has the sole purpose of configuring a set of so-called *satellite processors* and a *communication network* between them, as shown in Fig. 2.12. The satellite processors could be DSPs, reconfigurable data paths, address generators, but also memory. A *configuration bus* is used by the control processor to program the desired functionality. An evaluation [ASI+98] based on the computation of vector dot products, which is heavily used by speech coding applications, showed that the performance, in terms of delay and energy consumption, of a chip that was designed around the Pleiades architecture and could be programmed by means of 11 configuration registers, was significantly better than equivalent implementations based on a general-purpose processor (StrongARM), DSPs (TMS320C2xx, TMS320LC54x), or an FPGA.

Summarizing the overview given so far, in Fig. 2.13 the discussed hardware

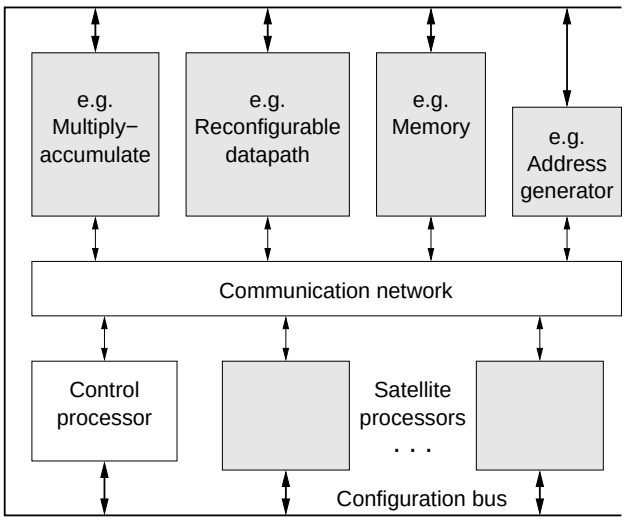


Figure 2.12: The Pleiades architecture template (adapted from [AZW+02]).

and software implementation variants are compared with respect to the level of flexibility they offer, once the system is manufactured, and the achievable performance, in terms of power consumption and processing speed. This comparison shall serve only as a general guideline, and real performance figures may vary for different applications. Furthermore, the cost factor is not included in the figure. Which choice turns out to be most cost-efficient depends on the number of chips sold.

It should not be forgotten to account for the time required to design a functionally correct system—software development, in general, tends to be much less time-consuming than logic design. Last but not least, the level of experience of the design team and the available IP cores determine to some degree what target architecture comes into consideration for a hardware/software system.

2.2.2 System modeling

Models are abstract representations of a physical reality. System models are created to be able to reason about certain properties of the system's behavior, to communicate among a group of people about the system, and to serve as a specification for the design process that will lead to a physical implementation of the system, which is compliant with the model. In hardware/software codesign, they shall be particularly suitable for the exploration of different design alternatives.

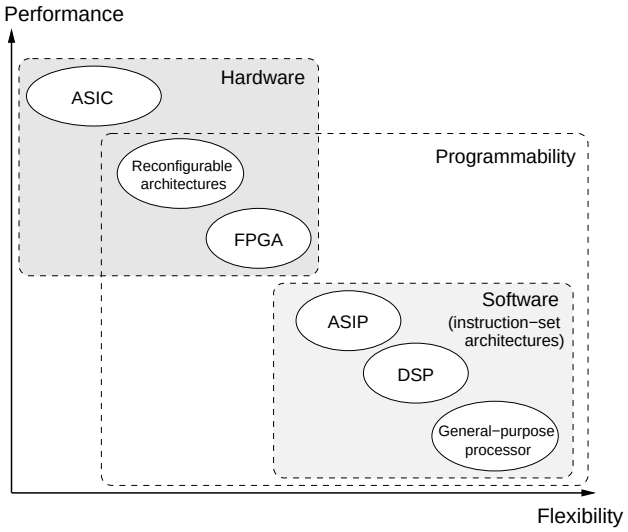


Figure 2.13: Comparison of the hardware and software target architecture components with respect to flexibility and relative performance.

A characteristic of embedded systems, in contrast to the rest of the computing world, is that they interface with the physical world. This can be via inputs that sense real-world phenomena or via outputs that control the behavior of actors, such as step motors, fuel injection, or pumps. When capturing physical phenomena we are used to describe them as *continuous-time* models, for instance in the form of a set of differential equations. When dealing with (digital) computing systems that interact with the physical world one must resort to *discrete-time* models. System-level modeling methodologies for embedded systems must bridge these two domains [HS07].

Different kinds of abstractions and languages have been proposed and applied in the past to ease the modeling of certain properties of systems. These approaches are also known as *models of computation*. Well-known examples are Synchronous Data Flow (SDF) [LM87], Communicating Sequential Processes (CSP) [Hoa78], Finite State Machines (FSM), or Synchronous/Reactive (SR) [HP85].

Each model of computation has an associated semantics to determine the system functionality unambiguously. The semantics can be stated in a denotational or operational manner. *Denotational semantics* use algebraic objects to express the meaning of the modeling language, while *operational semantics* define the behavior of a model as the execution of this model by an abstract machine. Generally, it is

not straightforward or even possible to transform behavioral specifications across models of computation while preserving the original semantics. There are also differences in the suitability to generate physical implementations from system-level descriptions, which is simply not the main purpose of some models of computation.

A particular challenge in system-level modeling is to capture functional as well as non-functional requirements. *Functional requirements* relate to the system behavior, i.e. the correct processing of inputs and showing the expected responses. Typically, such requirements can be easily formulated with mathematical equations. *Non-functional requirements*, on the other hand, describe constraints on the design or implementation of a system, such as the timing properties (maximum delay, jitter), power consumption, cost, necessary memory and processing resources, etc. It is often impossible to properly map these requirements to implementation-level models in the process of refining a system-level model [HS07].

In recent years, *model-based design* has gained wide popularity in systems design driven by maturing tool support. Tool vendors offer a broad range of basic modules that can be composed by the designer to create complex behavior. The basic modules stem from different models of computation. For instance, the controller of a digital baseband processor could be specified as a finite state machine while the data path is designed as a synchronous dataflow graph. It is also possible to refine a top-level system diagram in a stepwise process. Often, an integrated development environment allows to simulate, analyze, and to automatically transform the design into an implementation. Simulink [Mat07] is a well-known tool representing the model-based design philosophy.

Major shortcomings of the model-based design approach that are often cited in the literature (cf. [SVDN07], [HS07]) are its lack of separation between the functional and architectural models making it impossible to explore different architecture options, and the missing support for handling non-functional requirements, such as timing, in model transformations.

Independence between the functional system description and the target architecture is the objective of the *platform-based design* methodology [CCH+99], [KMN+00]. A hardware platform is considered as a microprocessor-based architecture that can be rapidly extended and customized for a range of applications. Likewise, a software layer that provides abstractions to make use of the different parts of the hardware platform, that is the programmable cores and memory system via the (real-time) operating system, and the I/O and commu-

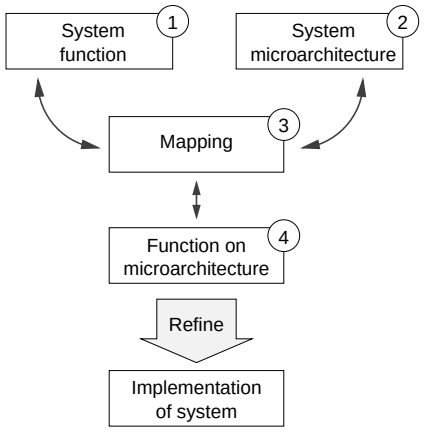


Figure 2.14: Separation of concerns in the platform-based design methodology, adapted from [KMN+00].

nication subsystems via device drivers and network connections, is referred to as software platform.

In platform-based design, the functional and architectural models are specified separately from each other. In a mapping process, the functional modules are mapped to resources of the system platform. For example, an algorithm could be performed by an operating system task or implemented on a DSP or ASIC. By comparing different mappings of the system functionality onto the architecture space, the most optimal solution can be selected. Finally, in a refinement process, the implementation of the system is derived. This design flow is depicted in Fig. 2.14.

The main objectives behind the platform-based design methodology are

- to shorten the development time for complex systems by facilitating reuse of models and IP cores,
- to optimize system performance by conducting an efficient design space exploration, and
- to enable the cooperation of design teams and provide standardized interfaces for component suppliers.

It is generally recognized that platform-based design has the potential of bringing great improvements in the design of complex SoCs and changing the traditional

system design methodology. In the last couple of years and still ongoing, there are research activities on standardizing system-level specification languages, for instance SysML [Sys07] or MARTE [MAR07] as new profiles to UML 2.0. Appropriate tools supporting the new methodology have to follow suit.

In some application domains, such as the automotive or avionics industries, that are characterized by the high complexity and dependability requirements of their systems, designed by independent design teams from different organizations, efforts to define common platforms with well-defined interfaces can be observed [SVDN07]. A good example is the AUTOSAR ((AUTomotive Open System ARchitecture) consortium of companies working in the automotive electronics domain [AUT08].

2.2.3 Hardware/software partitioning

The objectives of hardware/software partitioning are to select a system architecture consisting of the components described in the previous section 2.2.1 and to map the functionality to the architecture components in such a way that the system performance is sufficient and all other design constraints are met. The architecture design space and the combinatorial possibilities for the mapping are immense. Therefore, the design space must be limited and good heuristics must be found to achieve this task in a reasonable time frame. Often, an architecture is fixed by the designers, and tools are used to find an optimal partitioning of the system functionality.

Automatic tools need to be able to assess what effect their design decisions have on the quality of the final product. This means to check if all timing requirements (performance) have been met and at what cost (number of components, energy consumption, required chip area etc.) this was achieved.

We will first present common approaches for partitioning algorithms and will then come to techniques that provide sufficiently accurate estimates of the quality of a design without the need to actually implement the system.

System partitioning The partitioning problem is tackled by creating models of the system architecture and functionality. Graphs are a common means to capture dependencies and communication links between components. The *architecture graph* reflects the computing and communication resources, such as processors, ASICs or busses, of a system architecture.

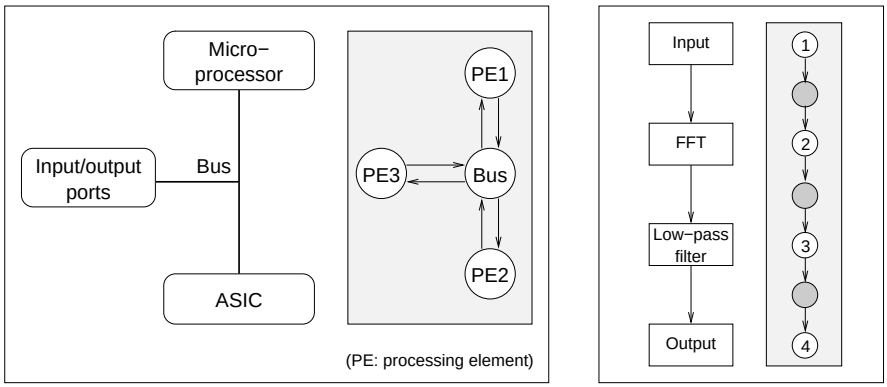


Figure 2.15: Example of a system architecture and corresponding architecture graph (left), and the problem graph for a frequency filtering application (right).

The *problem graph*, on the other hand, represents functional and communication objects as its nodes. The edges of the graph show temporal dependencies between the objects. The system behavior can be described on different abstraction levels. Nodes could represent, for instance, complete tasks in a coarse-grained specification, or single operations or boolean logic functions in a fine-grained model.

The general partitioning problem can be described as finding an allocation, binding, and a processing schedule. *Allocation* refers to the selection of appropriate hardware resources, *binding* to the mapping of functional objects to the allocated resources, and the *schedule* determines the order of sequential processing of tasks on a processor.

To illustrate the introduced concepts with a simple example, Fig. 2.15 (left-hand side) shows the architecture and corresponding architecture graph of a typical hardware/software system consisting of a microprocessor, an ASIC, input/output ports, and a bus connecting these components. The right-hand side of the figure shows a coarse-grained model and the corresponding problem graph of an application that calculates the frequency spectrum of an input signal, filters low frequencies, and produces output. In the problem graph, communication nodes have been introduced between the tasks.

The partitioning problem is known to be $\mathcal{NP}$-hard [Kal95]. In other words, all possible combinations for an implementation (allocation, binding, and schedule) have to be considered and the one that provides the required performance and has the least associated costs is selected. Defining a suitable cost metrics is another

challenge in codesign research. Since it is generally not feasible to study all possible implementations for even low-complexity systems, most often *heuristics* are applied to guide in the selection of an acceptable partitioning [AAMO05].

The partitioning algorithms proposed in the literature can be classified into constructive and iterative approaches. *Constructive* partitioning algorithms take one object at a time and group it with another object or group of objects. The grouping is driven by a so-called closeness function, which indicates the similarity between two objects and, thus, expresses a metrics for grouping them together, i.e. mapping the objects into the same partition. At the end of the process, the system partitioning is completed. *Iterative* partitioning algorithms, on the contrary, start with an initial system partitioning and try to improve it by migrating objects between the partitions until reaching an optimal solution. Recently, also knowledge-based algorithms that apply problem-specific knowledge, for instance represented by a set of rules have been discussed in the literature [AAMO05].

Simulated Annealing [KGV83] is a probabilistic optimization technique that is not only applied to system partitioning but in general to combinatorial problems where no efficient algorithms are known and a cost function can be defined. Its main concept is borrowed from physics. When a heated flux is slowly cooled down and becomes solid, the material arranges in a configuration with the least energy level. Applied to the partitioning problem, the algorithm works as follows. Starting from an initial partitioning, for instance an all-software or all-hardware implementation, objects are picked and moved into another partition at random. In order to avoid being stuck in local minima, partitionings that have higher costs are accepted with a probability that is dependent on the current temperature and the costs. The temperature is gradually decreased until reaching an equilibrium low-energy (or low-cost) state. Due to the random nature of the algorithm, it may take a long time until a good partitioning is found. Choosing an appropriate cost function and temperature scheduling is a non-trivial task [LVL03].

Performance and cost estimation Efficient design space exploration requires the ability to estimate the quality of design alternatives. It is, in general, economically not feasible to fully implement or prototype a system for comparison purposes only, though rapid prototyping has the advantage of early *system validation*.

Estimation algorithms can be applied to systems described at different abstraction levels. System-level estimation techniques typically have lower accuracy than

those based on lower-level specifications. Being able to produce realistic estimates is only one important criterion for estimation algorithms. Similarly important is the aspect of *fidelity*, which expresses that comparisons gained by estimation hold true in reality, i.e. that a system design that was estimated to be better than another indeed leads to a preferable implementation.

The quality (or cost) of a design is often expressed in terms of its performance characteristics: power or energy consumption, production costs, or other metrics, such as testability. Combining all different cost measures to a single quantity that is suitable for meaningful comparisons is not straightforward. In the following, we show how the mentioned metrics can be obtained from system specifications.

Relevant *performance metrics* are typically throughput, latency, and jitter. Throughput refers to the rate at which the system is able to permanently process a stream of data. Latency describes the initial delay before an event causes an effect. Jitter s a measure for processing or communication variations. Throughput and latency can be estimated by relating the processing speed offered by an architecture to the number of instructions or operations required by an algorithm. For instruction-set processors, the average number of cycles per instruction can be determined from different kinds of benchmarking applications. At a given clock frequency, the processing time for a software function with known number of instructions can be estimated—provided that there are no loops and recursions or their number of iterations is predictable. Moreover, general-purpose processors often use caches, and the performance heavily depends on whether instructions and data can be found in the cache.

For hardware implementations, an estimate of the achievable clock frequency can be obtained by finding critical paths, i.e. the longest paths from a register output to the next register input.

Probabilistic algorithms and implementation-level knowledge about a system are required to produce good estimates for jitter. Therefore, this is a measure that is very hard to predict.

The *power consumption* of a design is determined by its clock frequency, the supply voltage and switching capacity. For the latter, one can make assumptions based on the number of transistors or gates in the design. With continuing miniaturization of semiconductor technology, static leakage current, which is proportional to the silicon area, takes an increasing share of the total dissipated power. Therefore, to estimate the system's power consumption, one has to estimate the

hardware complexity, required clock frequency, and the part of the design that is actively switching. The *energy consumption* can then be obtained by determining the time required for an operation and multiplying with the estimated power consumption.

Production costs are often considered to be proportional to the required chip area, which can be estimated by summing up the area for memories, processors, and other hardware. The hardware complexity is given by the number of registers, combinatorial blocks, and wiring. Memory resources required by software depend on the program size as well as data memory for variables and stack. The pin count of a chip also influences its cost and size.

2.2.4 Tools

In the following, we will present the current state of available codesign tools, open problems, and directions for their future development. A comprehensive overview and comparison is outside the scope of this thesis. More information, specifically on real-time control systems, can be retrieved from the ARTIST Network of Excellence website [ART08].

Metropolis Metropolis [Met07] is a design environment for complex electronic systems that supports simulation, formal verification, and synthesis [BWH+03]. It has been jointly developed by the University of California at Berkeley, Politecnico Torino, and Cadence Berkeley Laboratories. The development of Metropolis has been influenced by the POLIS approach [BCG+97] and bears similarities to modeling languages such as Ptolemy, SystemC, and SpecC.

Different models of computation can be combined in Metropolis. This facilitates the construction of heterogeneous systems, enables system-level specifications and their refinement, and promotes reuse of abstractions.

One of the main goals of the Metropolis environment is the separation of behavior from the system architecture. Metropolis allows to model functionality as well as computing and other resources of an architecture by means of the Metropolis metamodel, a language with a defined formal execution semantics.

The functionality of a system is described as a set of concurrent processes that execute independently and communicate with each other [ZDSV+06]. Processes communicate with each other by calling methods on ports. A port is associated with an interface that declares the set of methods associated with that port. Me-

dia are passive objects that implement interfaces. In this way, computation and communication are orthogonalized. Processes and media can be hierarchically composed into networks.

The architecture model serves two purposes: to offer services to the functional model and to specify the cost of providing these services. Each service is broken down into a sequence of events, and each event is annotated with a value representing its cost. The cost may be in terms of CPU cycles, time, power, etc.

Quantity managers can be part of the architecture model. They provide an abstraction for the use of limited resources, such as the access to a shared bus.

To evaluate the performance of a particular implementation, the functional model needs to be mapped to an architectural model. This is achieved by synchronizing execution events that occur in the behavioral model, for instance reading from or writing to a port, with events in the architecture domain.

The designer may try alternative architectures and mappings to come up with a preferred solution. System simulations expose the performance of a chosen mapping. For this purpose, the system modeled with Metropolis is converted to a SystemC [Sys05] representation which can be executed.

An automated design space exploration that examines a number of architectures and mappings is not integrated with Metropolis. Instead, the designer must choose relevant metrics to be traced and analyze the simulation results for the best design alternative.

System designers have the possibility to express legal executions with the help of constraints. The constraints are specified with temporal-logic formulas and can be formally verified thanks to the semantics of the metamodel. For instance, to mark the case of two producers writing to a shared medium simultaneously as illegal, one can specify a constraint expressing that each *begin-write* event may not be followed by another begin-write event until an *end-write* event happened (cf. Fig. 2.16). The model checker Spin [Hol97] can be connected as an external tool to verify if the specified behavior would break any constraints. The metamodel is first translated into a representation that can be analyzed by Spin for this purpose.

Figure 2.16 gives an example of the basic concepts of the metamodel. In the top-left corner, the functional model consisting of three processes—two producers and one consumer—connected by a shared medium is depicted. The constraint below that diagram expresses that write access to the medium must be exclusive. The diagram on the right-hand side represents the system architecture. Three

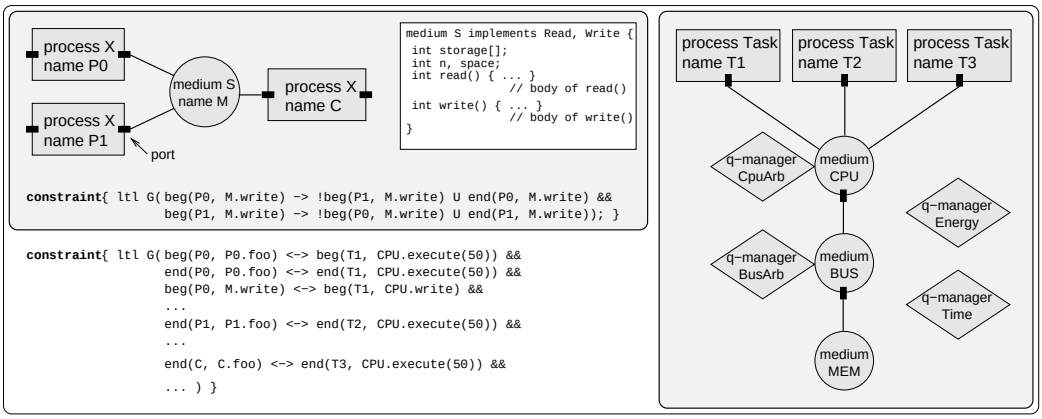


Figure 2.16: Mapping of the functional model (top-left) to an architecture (right) in Metropolis by means of constraints (bottom-left) that link events in the two models to each other.

tasks are using a single CPU, which is connected to a memory via a bus. Access to the CPU is governed by a quality manager. Similarly, bus access is managed by a bus arbiter. The consumption of time and energy is measured by two additional quality managers. The mapping of the processes in the functional model to the three tasks in the architecture is achieved with the constraints shown below the functional model in the figure.

An automatic synthesis technique called quasistatic scheduling (QSS) to schedule a concurrent specification on computational resources that provide limited concurrency can be applied. QSS considers a system to be specified as a set of concurrent processes communicating through FIFO queues and generates a set of tasks that are fully and statically scheduled, except for data-dependent controls that can be resolved only at runtime [BWH+03]. Metropolis provides library elements to model interprocess communication through FIFOs. A Petri net model of the part of the system model where QSS shall be applied to can be automatically generated. By analyzing this model, the schedule can be fixed and communication primitives are removed. The Petri net can be transformed back into the metamodel language. In this way, QSS can serve two purposes: functional verification and optimization of interprocess communication.

COSYMA COSYMA [ÖBE+97] is another example of a hardware/software codesign tool. It was developed in the 1990s at the Technical University Braun-

schweig. The focus of COSYMA was put on embedded systems design. Unlike Metropolis, it supports automatic design space exploration and hardware/software cosynthesis. Simulated annealing is used as a technique to find optimal partitionings. It starts with a situation where all functions are implemented in software. Functions are moved to hardware until all timing constraints are satisfied.

The system architecture is limited to a single processor with an attached application-specific coprocessor. Heterogeneous modeling styles and platform-based design are out of the scope of the COSYMA system.

Other tools In the last couple of years, several other system design methodologies and languages emerged. The primary reason for this development was the perception that with traditional design methods based on hardware description languages (HDLs) the complexity of future systems could not be handled anymore. Notable examples for this trend are the SystemC language [Sys05] and EDA tools based on it, and Simulink [Mat07]. They offer rich modeling libraries, are appealing to software and hardware design communities alike by embracing different models of computation, allow system simulation and automatic synthesis of a subset of their modeling languages.

Addressing specifically real-time control systems, another class of design tools is developing. Here we mention Jitterbug [LC02] and TrueTime [OHC07] that support analysis and simulation of a system's timing behavior. Another well-known class of tools for safety-critical real-time systems comes from TTTech [TTT07], which are based on the time-triggered architecture (TTA) [KB03].

Research challenges Many of the current challenges in the design of embedded systems arise from their growing complexity. As has been presented above, today's design methodologies develop into combining heterogeneous models of computation in a single representation. Reusable components with well-defined interfaces facilitate the bottom-up construction of large system. However, embedded systems implementations must not only be functionally correct, but also satisfy non-functional requirements, such as to deliver responses always in time, have a low power consumption, or fit into the available program memory. Expressing non-functional constraints in component descriptions in such a way that they support composability is one of the challenges for the future.

Reliable performance estimations with respect to timing behavior is difficult

to achieve without going into implementation-level details. This becomes even more severe with the adoption of more powerful microprocessors that make use of caches, pipelining, or speculative execution, thereby making accurate predictions about the run-time behavior impossible [ÅCH05]. This is particularly relevant for critical real-time systems as their design methodology is based on worst-case predictions. Following this methodology, critical system engineering allocates much more resources than needed for a best-effort system, this way increasing their costs dramatically [HS07]. Design methodologies that enable the separation of the two domains—critical and best-effort components—and that allow to share resources are still missing.

It has been found [HS07] that with the current design process, software is the most costly and least reliable part of embedded applications. This is due to the lack of rigorous techniques for embedded systems design. Research on testing and verification of functional as well as non-functional properties based on formal methods is part of the agenda in European embedded systems research projects [ART08]. Efficient high-level behavioral synthesis to automatically derive software and hardware implementations would be a step towards more reliable systems and could shorten the development time from specification to implementation.

Chapter 3

Related Work

We have given an overview on the current practice in protocol engineering as well as hardware/software codesign for embedded systems in the previous chapter. In the following, we will pursue the problem of combining existing protocol engineering techniques with design approaches for embedded systems. Our goal is to develop a methodology for efficient communication protocol implementation on tiny, resource-limited target platforms.

When developing embedded communication systems the objectives are to design efficient, correct protocols, to implement these protocols, and integrate them into the complete system. An integrated design methodology should support all steps in the design flow and enable short development times.

SDL (Specification and Description Language) has become a popular language for communication protocol design. Development tools, such as Telelogic TAU SDL Suite [Tel06], provide the ability to design, simulate, verify, implement in software, and test protocols. Extensive research has been conducted to improve the design flow, e.g. by specifying real-time constraints, developing more efficient implementation models or generating hardware implementations. We will discuss this work in the following section.

A number of protocol implementations derived from formal specifications using SDL have been described in the literature. These approaches, results and experiences will also be the topic of this chapter.

Other design methodologies that are not based on SDL are also conceivable. Candidates are, for instance, SystemC [Sys05] with a potentially simple refinement process to derive hardware and software implementations, Simulink [Mat07]

that provides an integrated simulation environment with other protocol layers and physical channel models, or Unified Modeling Language (UML), which is a very popular modeling technique. However, the advantages of an SDL-based design flow are the high-abstraction level and implementation-independence of SDL specifications and its formal semantics enabling protocol verification. Therefore, we will not elaborate on those alternative approaches.

3.1 System design with SDL

Specification and verification The most important concepts and abstractions of SDL have been presented already in Sect. 2.1.3. Due to its capabilities to formally specify data types, such as protocol data units, and behavior in an implementation-independent way, the language has been used in the standardization of communication protocols, for instance the IEEE standards 802.15.1 or 802.15.4.

Formal verification of SDL specifications has been described in a number of reports ([MIJ03], [BDHS00], [SS01], [RB98], [JG01]). In the latter publication, the authors presented their experiences from a verification experiment of the industrial protocol MASCARA (Mobile Access Scheme based on Contention and Reservation for ATM), a wireless medium access control protocol. The protocol has been specified using SDL, on a total of roughly 300 pages. This specification has been automatically translated into an equivalent IF specification for further analysis. The IF language [BFG$^+$99] has been defined as an intermediate representation for timed asynchronous systems. The main challenges for the verification of this large system were to reduce the complexity of the model and to analyze subsystems separately. The authors applied a mix of static and dynamic techniques for model checking and complexity reduction. In Fig. 3.1, the verification tool chain as presented in [JG01] is shown.

The model checker CADP [GLM02] has been used to analyze the labeled transition system. To demonstrate the usefulness of this approach, a number of generic properties, such as the lack of deadlocks, and expected behavior of the protocol was checked. To specify such properties, they have to be described as temporal logic formulas or, alternatively and more intuitive for the non-expert user, by means of finite automata for expressing labeled transition systems. Protocol behavior that was found to violate a specified property could be visualized with message

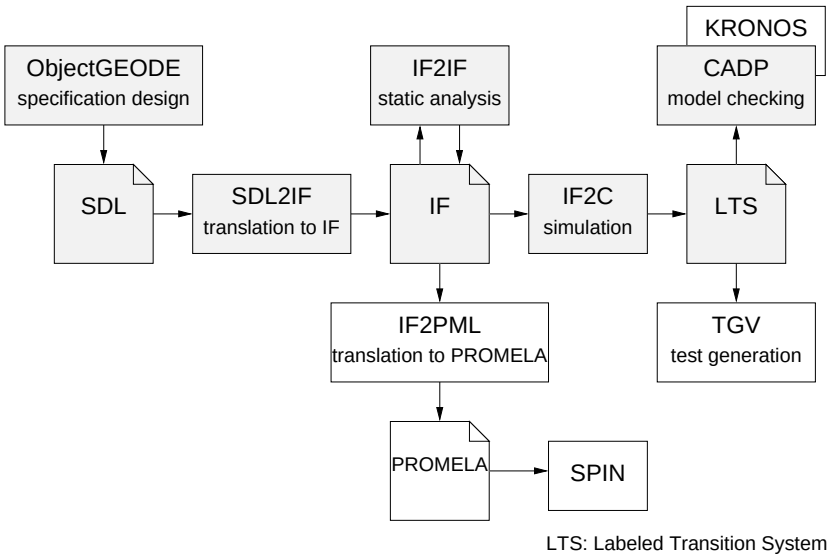


Figure 3.1: Tools and languages used for the verification experiment of the MASCARA protocol [JG01]. The shaded boxes mark the used verification flow, alternative tools and languages are also shown.

sequence charts. The specification of such properties directly from the SDL model was not possible.

Many researchers (cf. [Leu95], [FL98], [dW04], [Bou07]) have pointed out that the built-in concepts of SDL to specify *timing behavior* are insufficient to model real-time systems. The most severe limitations of the language in this respect were found to be the lack of concepts to specify deadlines, i.e. the time for the system to react on an event, communication delays over channels, different system clocks, and processing delays. SDL has abstract timing semantics, that is all system activities are performed without any delay and time advances only when all transitions have completed and a timer expired or an external event was triggered.

Several research groups have proposed extensions to SDL to express *real-time constraints* as well as processing and communication delays.

Fischer and Leue address in [FL98] the inadequacy of SDL for the specification of quality-of-service requirements. They note that, due to the language's asynchronous timer mechanism, systems satisfy the SDL specification even if they exceed the limits by an unspecified and even potentially unbounded amount of time. The most that can be expressed is that there is a *minimum* amount of time

that passes between the setting of the timer and the recognition of its expiry by the timed process. As a remedy, the authors introduce the concept of *complementary real-time specification*, which means that the semantic models of Metric Temporal Logic—a formal language similar to Computational Tree Logic—and SDL can be combined. For this purpose, the so-called Global State Transition System (GSTS) was defined, which serves as a common formal model for the interpretation of SDL specifications and temporal logic formulas. For a detailed presentation of this approach we refer to the literature.

Design and verification of real-time properties of SDL specifications has been studied by Bourgeois [Bou07]. The author introduced timing annotations to express assumptions about the run-time system and environment as well as timing requirements. Annotations are comments in the SDL model. The introduced modeling features to describe the temporal behavior of the specification comprise the specification of clocks, the duration of SDL processes, a notion of urgency of transitions, the definition of events and maximum available time between two events, as well as the behavior of communication channels and external inputs. The processing time is an estimation by the designer and has no relation to the real implementation. Unfortunately, the approach does not consider the influence of the run-time system with its scheduling and queueing delays. The annotated SDL specification is parsed and mapped to a timed automaton. This automaton can then be analyzed and verified with the UPPAAL tool [LPY97].

In [dW04], a methodology for simulation-based *performance analysis* of communication protocols based on a combination of SDL and UML 2.0 is presented. SDL is used as a means to specify protocol behavior. Since the language abstracts from the temporal properties of the execution environment and implementation details, these features are modeled and refined with the help of UML diagrams. The proposed methodology has been validated by an experimental scenario.

Codesign tools The design of a system encompasses its behavioral specification and the development of a system architecture on which the functionality is mapped. Already in the 1990s, a number of tools were developed that used SDL specifications as a high-level system description and support the design process by performing an exploration of different architectures and mapping the SDL model onto them. As examples from this generation of codesign tools, COSMOS [IAJ94] and CORSAIR [DMTS00] shall be briefly discussed.

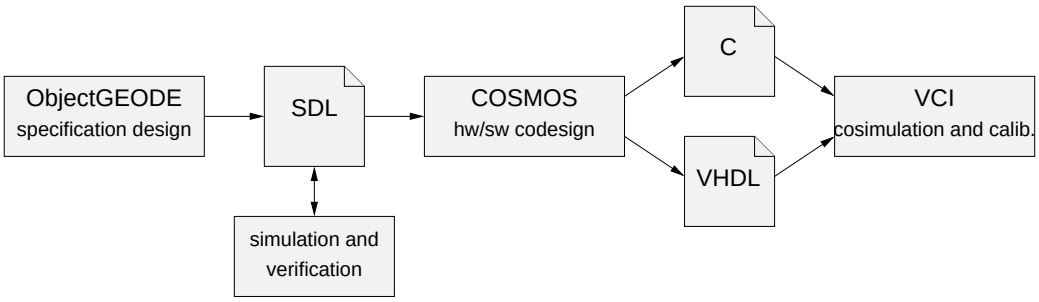


Figure 3.2: Hardware/software codesign flow based on the COSMOS tool.

COSMOS is a hardware/software codesign tool developed at the TIMA laboratories. A commercial product, ArchiMate [MdCP00], has been created based on the initial work. COSMOS uses SDL as an input language. Hardware/software partitioning is performed manually on the level of SDL processes. This means that the designer has the choice to map SDL processes onto a multiprocessor architecture [ZMDJ98].

Hardware processors, in other words ASICs, are then automatically generated and described in VHDL, while the remaining SDL processes are translated to C programs and executed on software processors. Additionally, interface components to connect the hardware and software processes are generated.

A cosimulation of the produced VHDL and C code is performed as the final step in order to validate the temporal properties of the design. The three tools used in the codesign flow are shown in Fig. 3.2. A drawback of this cosimulation approach is the rather long simulation time in the order of minutes or even hours in the case of more complex systems than described in [ZMDJ98].

COSMOS does not allow system partitioning with a finer granularity than SDL processes. This means that in order to fully take advantage of the efficiency gain of hardware implementations, the designer must structure the high-level specification with implementation details in mind. The efficiency of the generated hardware designs was reported to be poor [MHA+02], though these results were obtained with the ArchiMate tool. We will present other approaches towards hardware synthesis from SDL specifications in the following section on implementation synthesis.

The limitations of the COSMOS tool were tackled by the CORSAIR (Codesign and Rapid Prototyping System for Applications with Realtime Constraints) environment. It has been developed at the University of Erlangen-Nuremberg.

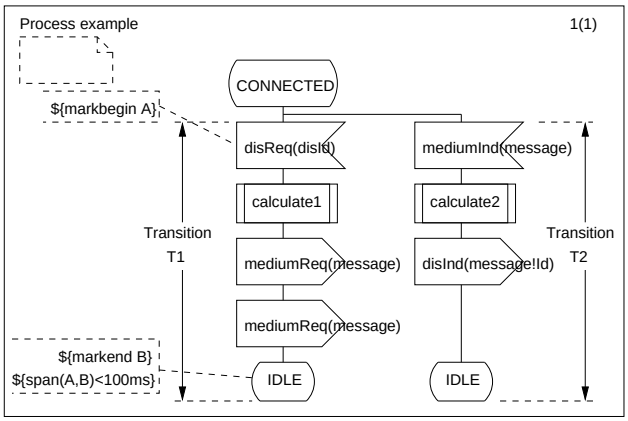


Figure 3.3: Modeling of timing constraints with SDL*, taken from [DMTS00].

Similarly to COSMOS, it is an integrated design tool which addresses mixed hardware/software systems and, likewise, their automatic synthesis. However, an extension of SDL, called SDL*, is used for system specification. This language incorporates timing- and implementation-related aspects of the system. An example that illustrates a timing annotation in an SDL* process is shown in Fig. 3.3.

These timing annotations in the specification are considered by the tool in the generation of a prototype implementation that meets the real-time constraints. The CORSAIR tool first creates a problem graph from the SDL* specification and, subsequently maps this graph to an architecture, consisting of processing nodes and interconnections built-up from library components. This scheme is depicted in Fig. 3.4. The optimization goals for this mapping process are a reasonable use of resources as well as meeting the timing constraints. The allocation, binding, and scheduling of resources are typical codesign tasks and have been presented already in Sect. 2.2.3 in general terms. The system architecture specification is also expressed in SDL*. The complete design flow, including system specification, architecture and implementation synthesis, is shown schematically in Fig. 3.5.

The synthesis of software, hardware, and interfaces is also supported by the tool and will be cover below. The goal of the design process with CORSAIR is a prototypical system realized on a multiprocessor platform that is connected to an FPGA board. The synthesized components are programmed on the target platform.

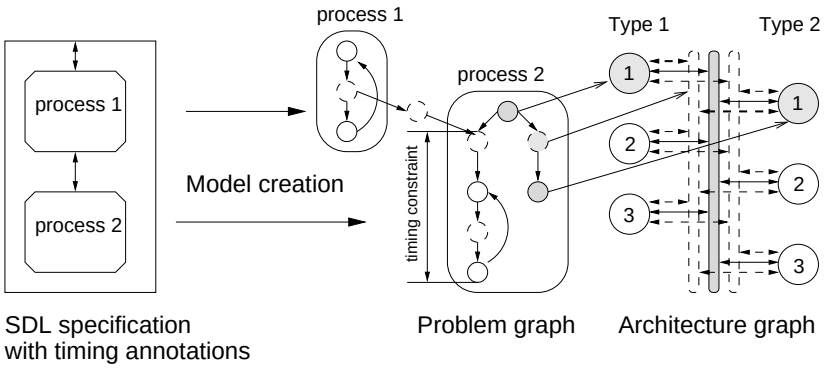


Figure 3.4: Generation of the problem graph from the SDL* specification and binding of architectural resources, from [DMTS00].

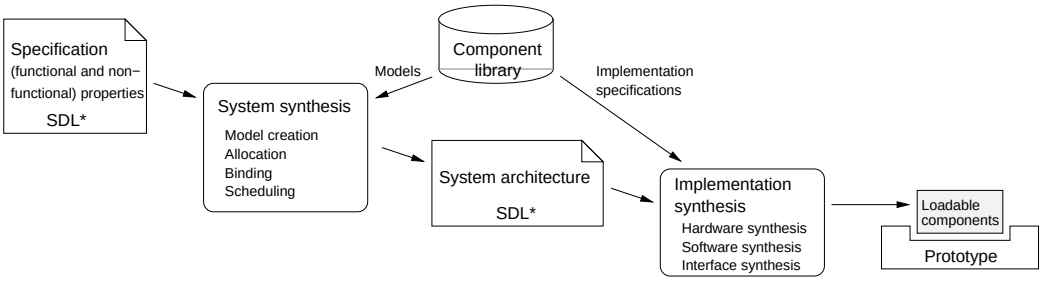


Figure 3.5: Codesign flow with CORSAIR [DMTS00].

Implementation synthesis In the following, we will present research work that addresses the transformation of abstract SDL specifications to implementations in concrete execution environments. The semantic equivalence of both, the specification and the *implementation model*, must be ensured during this transformation process, otherwise all previous theoretical examinations become obsolete. However, the fact that abstract SDL models operate with infinite queues and unlimited memory capacity contradicts the physical reality and shows that there are limitations when going from a theoretical model to a real implementation. An extension of the SDL semantics towards the use of bounded input ports was recently proposed by Gotzhein *et al* [GGK07].

In general, SDL specifications can be realized as mixed hardware/software systems. The extensive study of the problem of generating efficient implementations has led to a large number of optimization techniques. We will briefly present the

most important of these.

The generation of software implementations is already state-of-the-art and has been part of commercial SDL tools for a couple of years. In the 1990s and beyond, the automatic translation into hardware descriptions, i.e. VHDL code, was studied by a number of research groups. Tools for the generation of prototypical hardware implementations were presented, which to our knowledge did not find their way into successful commercial products. Besides code generation, embedding the SDL system into a run-time environment or operating system and the design of interfaces between hardware and software are important, as well.

A straightforward implementation model for SDL specifications, that maintains the semantics of the language, is the so-called *server model*, which was already introduced in Sect. 2.1.4. Each SDL process is realized by an asynchronous server—an entity with its own signal queue that processes the input signals one-by-one and communicates with other processes by placing asynchronous signals in their input queues. This model can be applied to software and hardware implementations alike. In the first case, all servers are executed concurrently under the scheduling regime of the operating or run-time system, while in the latter case true hardware parallelism can be exploited. The server model is used, for instance, by the Telelogic TAU code generator for software implementations or by the COSMOS tool [ZMDJ98] for VHDL generation. Disadvantages of this approach are the additional scheduling and context-switch overhead as well as the required resources to maintain signal queues.

An optimization that removes the mentioned overhead connected with the server model is the *activity thread model*. In this model, communication between SDL processes is synchronous, i.e. by procedure calls. Instead of sending a message to another process, the corresponding transition at the receiver process is called. Execution remains within the same thread, no context switch and signal buffers are required. The SDL specification must be analyzed to identify all possible chains of transitions that are triggered by an external event and end with a transition without signal output or a signal to the environment. Such a chain is called an activity thread.

The activity thread model can be efficiently applied to the implementation of simple protocol stacks where, basically, signals are exchanged only from higher to lower layers when transmitting, or in the opposite direction after the reception of a packet. More complex interaction patterns within the model may create additional

overhead since transitions that are part of multiple activity threads, i.e. are executed following different external events, will be multiplied unless a serialization is introduced [MF00]. This pertains to an implementation in hardware. For software implementations using the activity thread model, special care must be taken when there are interdependencies between processes or multiple signal outputs within a transition, in order to preserve the semantics of the model [HMTKL96], [Kön03]. Such a dependency analysis was presented by Leue and Oechslin [LO96], who proposed optimizations to enhance parallelism within an implementation.

Efficiency improvements have been achieved also by reducing the number of copy operations of large buffers, such as protocol data units, and by a technique called application-level framing. The first technique uses *references* to buffers, which can be easily passed. Application-level framing is a technique for handling protocol data units across protocol layers. Already at the highest protocol layer, the application layer, buffer space that is large enough to accommodate lower-layer headers is allocated. All layers simply add their control information at the right place in the buffer. Likewise, in the receive direction, each layer accesses the relevant part of the frame buffer.

Deriving hardware implementations from SDL specifications has been a focus of research work since the 1990s. Attempts to use SDL as a description language for synchronous digital systems, similarly to how SystemC today can be used, have been unsuccessful due to the lack of concepts to express synchronous behavior and data types for bit manipulation operations [MHA+02].

Of the many approaches towards generating hardware implementations from SDL specifications, most notably the work by Muth [Mut02], who alsocontributed to the CORSAIR system, shall be highlighted here.

The rapid prototyping design flow presented in [Mut02] starts with an initial SDL specification and a set of timing annotations that express the real-time constraints of the system. The objective is to generate a compliant hardware/software implementation on a rapid prototyping platform consisting of multiple high-performance general-purpose processors and a configurable I/O processor (CIOP). The CIOP consists of programmable hardware (Xilinx FPGA) and is connected via a PCI bus with the other processors of the platform.

A real-time analysis of the SDL specification uses the event stream model to model the possible occurrence of external events and timers. Deadlines for the processing of events can be specified. In the course of the real-time analysis,

also an estimation of the worst-case message queue depth is undertaken. This is important in order to preserve the semantics of the original SDL model and allocate resources efficiently.

SDL processes are manually mapped to processing units of the target architecture. The partitioning depends on their "timing requirements and computational complexity" [Mut02]. For the software part, Telelogic's CAdvanced code generator is used. This C code generator creates implementations based on the server model. A run-time system is required to provide all the necessary support functions such as timers, inter-process communication and interfacing with the environment, which are an implicit part of the SDL specification. In the described approach, the run-time system is realized by the free real-time operating system RTEMS.

For VHDL generation from SDL, different implementation techniques can be chosen: the server model as well as a serialized and parallel activity thread implementation. For the server model, the process-specific extended finite state machine is modeled in VHDL and extended with pre-designed components for signal queues and output ports.

In the case of an activity thread implementation, the chain of transitions originating from an initial external event is determined. Every activity thread is implemented as a separate VHDL process. Since multiple activity threads may access the same shared variables of a process, locks are inserted to protect these variables from concurrent access in a parallel implementation, or activity threads are serialized.

The resulting VHDL code is synthesized and mapped to a Xilinx FPGA. The hardware design process is shown in Fig. 3.6 together with an example SDL hardware partition. The macro processor m4 is used to replace certain operations, such as timer operations, by corresponding hardware blocks. In Fig. 3.7, the hardware architecture for the partition in Fig. 3.6 is depicted.

As already mentioned above, an execution environment for SDL specifications must provide services and mechanisms that are part of the semantic model of the language. Among those are the timer handling, process scheduling as well as queues for asynchronous messages. SDL tools do provide implementations of such *run-time environments*, which must enforce the semantics of the language. For performance reasons, it is also possible to map the aforementioned services to operating system mechanisms that may be anyway part of the embedded system and to get rid of a run-time environment supplied by the SDL tool vendor.

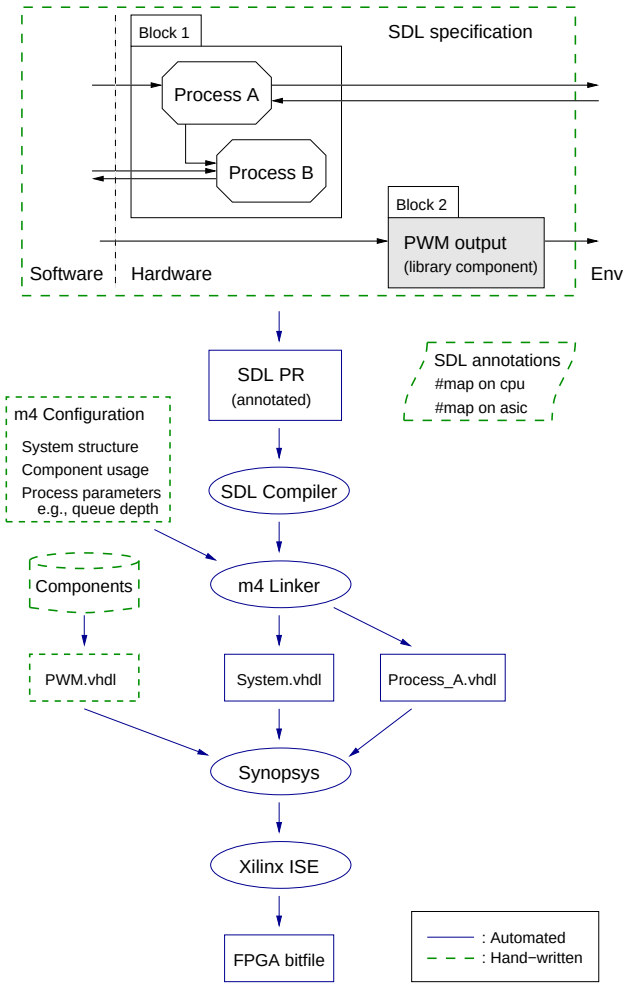


Figure 3.6: Hardware synthesis process for a rapid prototyping system as described by Muth [Mut02].

An aspect that is often overlooked is the fact that also the run-time environment and operating system together with the chosen mapping is subject to design errors. So, even if the SDL model is formally verified, its run-time environment may not be. This may compromise the trust in the correctness of the system.

A recent approach towards a real-time operating system that is designed using formal methods was reported in [VdJ07]. An effort to port this operating system, which is called OpenComRTOS, to be able to easily map SDL specifications onto it was announced.

All other software implementations of SDL specifications require trust that

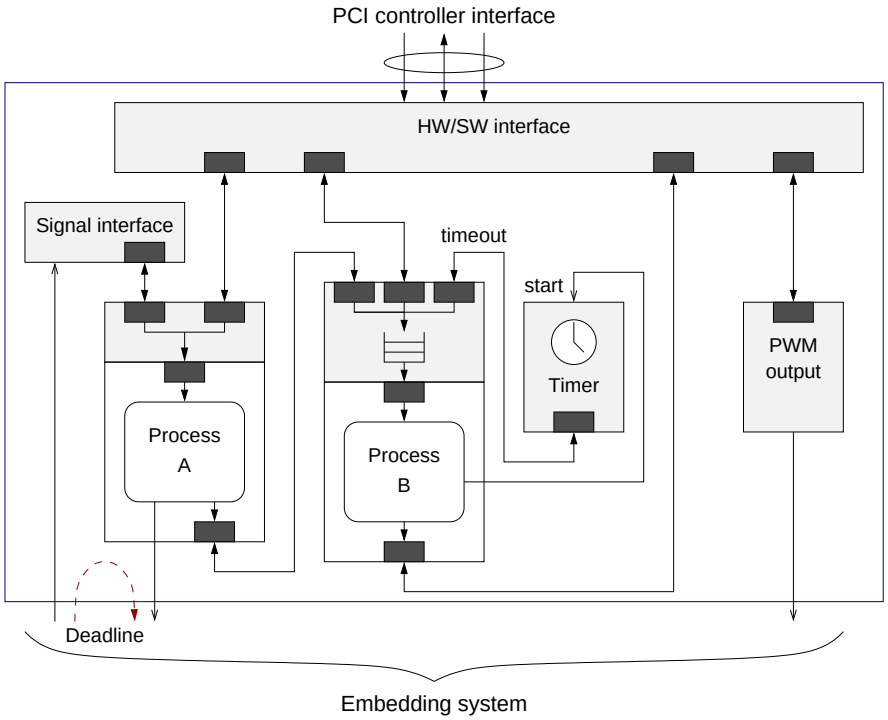


Figure 3.7: Hardware architecture generated by the REAR system presented in [Mut02] for the example SDL partition in Fig. 3.6.

the run-time system was thoroughly tested and most errors have been found by previous implementations. For a number of different operating systems, specific mappings—based on the output of the code generator—have been created. Telelogic, for instance, offers integration models for Neutrino, VxWorks, OSE Delta, and Nucleus+. Integration models for other platforms have been developed by various research groups.

Relation to own work Our embedded systems design flow proposed in this thesis is also based on SDL as high-level system description language. Many of the presented techniques, such as formal verification or performance analysis, are complementary to our approach and, hence, can be combined. We have particularly addressed the design of efficient run-time environments for embedded systems. Furthermore, we developed a cosimulation approach with an instruction set simulator to support hardware/software partitioning in the design space exploration phase.

The software synthesis methods, i.e. code generators from SDL specifications, are mature and industry-standard tools. Therefore, we base our design flow for embedded systems on the output of a successful commercial C code generator, namely CAdvanced which is part of Telelogic TAU SDL Suite [Tel06].

However, the operating system integrations available today are limited to a couple of real-time operating systems (Neutrino, VxWorks, OSE Delta, and Nucleus+) as mentioned above. Though these are designed for embedded systems, their complexity and memory requirements exceed the available resources of typical 16-bit microcontrollers, which shall be addressed by this thesis.

The automatically generated C code from CAdvanced contains many preprocessor macros facilitating an OS integration. Each run-time environment must define these macros such that the SDL concepts, for instance signals, processes and timers, can be mapped to the resources of the run-time environment or OS. Unfortunately, Telelogic does not provide a general framework for real-time operating system integration, but merely recommends to use the existing integrations as examples when designing a tight integration for a new OS.

Principle concepts for targeting extremely resource-limited devices and their operating systems are missing. Furthermore, to our knowledge, no SDL run-time systems have been described for operating systems such as TinyOS, Contiki or Reflex, which are suitable operating systems for the kind of deeply embedded devices that are the focus of this thesis.

Therefore, such general concepts had to be investigated, first. Additionally, a proof of feasibility and efficiency had to be presented. For this purpose, an integration library for an example operating system was realized. The author has chosen Reflex because it is designed in C++ and the realization of our concepts can be shown clearly in the software design.

This implementation serves at the same time as an important building block in the design flow. It can be used to generate efficient target executables for platforms where Reflex has been ported to, for instance the Texas Instruments MSP430, Freescale HCS12 or Atmel ATMega 128 microcontrollers, as listed in [Nol09].

Automatic hardware synthesis from a high-level SDL specification is an option that we excluded from our proposed embedded systems design flow due to the increased hardware complexity compared with a hand-optimized semi-custom design. A hardware compiler is an excellent tool for rapid prototyping as the design time is significantly reduced. However, for chips that must be very cheap and, in

order to achieve this, the silicon area should be as small as possible, manual design promises to give better results. As outlined above in the description of the CORSAIR system, which applies automatic VHDL generation, one source of overhead are mechanisms to buffer SDL signals as inputs to VHDL processes. Moreover, the translation of the finite state machines and SDL data types to hardware can most likely be more efficiently achieved by hand-optimized design. Efficient behavioral compilers from high-level languages are still a fundamental research topic today.

It is questionable whether the mapping of complete SDL processes to either hardware or software, as employed in the CORSAIR tool, is really an optimal solution. This coarse-grain partitioning would lead to the mapping of a complete process to hardware even if only one or two transitions, for instance a time-critical cyclic redundancy check (CRC) calculation, present an actual bottleneck in the implementation. Furthermore, the communication between the software and hardware representations of SDL processes via SDL signals is time-consuming as all signal parameters, including control information such as the sender process address, have to be passed. For these reasons, we propose a fine-grain, arbitrary partitioning of SDL processes and give the designer the freedom to use an optimized interface to the hardware partition.

The partitioning decision is made on the basis of simulations of the hardware/software system in our approach, whereas software models for the hardware can be used in place of real hardware designs in order to speed up the simulation. We do not consider it necessary to use tools for automatic design space exploration and selection of an optimal hardware/software partitioning, as these tools base their decision on automatically derived and not necessarily optimized hardware designs. We argue that the designer usually has a good knowledge about the system and its potential bottlenecks, and thus can easily select eligible architectures and suitable partitionings. However, the designer needs to be guided by the simulation results in order to estimate the performance impact of architectural decisions, identify bottlenecks in the design, and thus be able to improve the partitioning.

As outlined in detail in Sect. 1.2 and depicted in Fig. 1.3 on page 11, we consider an interactive cosimulation between an abstract SDL simulation integrated with an instruction set simulator that emulates a real system implementation as the most suitable approach particularly for communication systems design and implementation. This allows the reuse of the original SDL model, which was used

to derive a target software executable, in a communication network model and generate test stimuli for the emulated implementation model. The instruction set simulator gives reliable and accurate information about the system performance. This way, bottlenecks and unsatisfied timing requirements can be easily identified. We have not found any previous work that proposes a similar approach. Therefore, the author has developed his own concepts for a cosimulation and proved them experimentally.

Finally, as already mentioned, hardware synthesis from the SDL model is performed manually in our approach. This also includes the design of optimized interfaces to other hardware blocks and the software partition. The hardware design process and FPGA or ASIC synthesis is well supported by EDA tools.

3.2 Communication protocol implementations based on SDL

In this thesis, we present not only a design methodology for embedded communication systems, but show how it has been applied for a single-chip wireless MAC protocol implementation of the IEEE 802.15.3 standard. Wireless communication systems developed from SDL specifications have been reported in the literature. In order to consider our approach in the context of previous work we will closely study comparable designs that have been published. It should be noted that certainly many more systems were designed following proprietary SDL-based processes in companies, but results have hardly been made available.

Drosos *et al* describe in [DZM01] the design of an ARM-based processor for multi-mode, DECT- and GSM-capable, cellular phones. In particular, the authors present the design process for the MAC protocol of the DECT standard.

This MAC protocol has been implemented entirely in software. The development started from an SDL model, which was automatically translated into C by the CAdvanced code generator from Telelogic. In order to validate the protocol together with the target hardware platform, two steps were performed: a tight integration for the Virtuoso operating system and a simulation model for the ARMulator debugging system were created.

A tight integration approach was chosen because the SDL run-time system used in the light integration model from Telelogic does not allow preemption. With the tight integration approach, each SDL process is mapped to an OS task,

such that higher-priority processes can preempt those with lower priority. A tight integration library for the Virtuoso OS had to be developed, since Telelogic does not provide one. Since there is no generic template for targeting other operating systems this adaptation was a major result [DZM01]. Details on the design of the tight integration library were not published.

The ARMulator environment allows to simulate the ARM processor and its peripherals on the instruction level. It can be used for software benchmarking and hardware/software cross-development. The memory model of the ARMulator was extended to incorporate a behavioral model of the radio front-end and other system parts. The MAC protocol software interacts via a number of command requests with the radio transceiver. Events from the transceiver trigger processor interrupts. An environment task is responsible for creating SDL signals from these events and send them to the right processes.

System integration has been found to require little effort due to the previous validation by simulation and the specification of hardware interfaces and behavior [DZM01].

A very similar problem to our MAC protocol implementation was covered in Marko Hännikäinen's PhD thesis [Hän02]. The author addressed the design of a wireless communication system (TUTWLAN) with QoS capabilities. Like in our approach, a partitioning of the MAC protocol into hardware implemented on an FPGA board and software running on a DSP was elaborated.

In [HKHS00] the applied SDL-based high-level design is explained in detail. A commercial tool from Telelogic was used for editing, simulation, and code generation. The complete specification can be considered as complex with its 24 processes, 75 procedures, and altogether 96 different internal signal types. The author argues that simulation at early design phases helped to increase the quality of the design and saved time and cost [HKHS00].

The light integration model was selected for targeting the SDL specification on the DSP. There is no operating system required. The compiled C code consumes 490 kbytes of memory, which is a rather large amount for embedded system applications. Environment functions have been added to provide interfaces to the radio module and host computer. The SDL model itself is independent of the target platform.

A number of optimizations to increase the performance of the SDL implementation were proposed and their effects measured. These optimizations comprise the

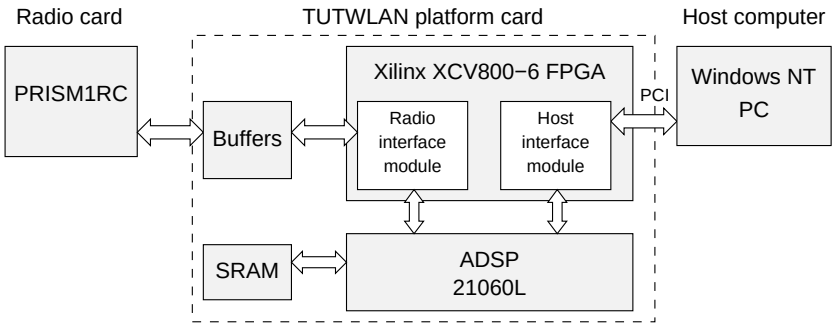


Figure 3.8: Target platform for the TUTWLAN system reported in [SHH02]. A MAC protocol hardware accelerator is part of the radio interface module implemented in the FPGA.

introduction of process priorities, explicit addressing of signals[1], use of efficient data types by avoiding array and string types, as well as algorithm implementations as external functions in C. The performance effects have been measured by simulating the complete SDL model on a host computer. It was not reported that a simulation for the target processor or real-time analysis was conducted.

A small number of tasks have been identified to be realized in hardware. It seems that the decision has been made on the basis of the designer's intuition and knowledge of the protocol's timing constraints. A validation method for these design decisions was not reported. The selected tasks for hardware implementation are the synchronization to the TDMA frame, data encryption, and CRC calculation [SHH02].

A hardware accelerator has been designed that provides a clear functional interface to the protocol software running on the DSP. The hardware accelerator consists of transmission and reception data paths, a transmission control block, and a status register with information on the received frame. For details on the implementation we refer to the article [SHH02]. The demonstrator platform for the WLAN system with QoS support is shown in Fig. 3.8. The FPGA and DSP are clocked at 40 MHz.

A hardware/software implementation of the IEEE 802.15.3 MAC protocol was presented in [HBB04]. As the previous two protocol implementations, the design

[1]When implicit addressing is used, the receiver of signals is determined by a time-consuming process from the structure (signal routes between processes) of the SDL model.

methodology employed by the authors starts with a high-level SDL specification. However, in this case no automatic transformation into a hardware or software implementation is applied. Instead, the system is re-implemented in SystemC. This new representation is synthesizable into hardware and can also be simulated. The SystemC code has been augmented by text outputs in order to trace the exchange of signals at the interface to its environment.

By means of equivalence checks of the recorded message sequence charts generated from the simulation of the abstract SDL model and the SystemC simulation traces, it could be determined if the hardware implementation correctly reflects the original model. Of course, this approach cannot give a proof for complete correctness since it relies only on the previously selected test cases.

Time-critical functions of the MAC protocol that have been identified to be implemented in hardware are immediate acknowledgment transmission and beacon frame decoding. In order to be able to send an acknowledgment, the integrity of a received frame must be checked first. Beacon frame decoding is required to extract information about the position of reserved time slots for transmission or reception.

The hardware accelerator architecture is shown in Fig. 3.9. The `Superframe Control` block decodes beacon frames and maintains a timer that indicates transmission and reception opportunities to the `Main Control` block. The latter initiates the transmission or reception process. The actual frame processing is performed by the `TX` and `RX Coordination` blocks. Additionally, there is an interface to the physical layer and a memory buffer for received frames and those that are to be transmitted. The software part of the MAC protocol interacts with the protocol accelerator through another interface component.

Hardware/software partitioning was not part of the presented design methodology, because the complete SDL model was translated into hardware. However, it outlines an interesting new approach: the SDL specification can be simulated and verified with the available tools and then automatically translated into a behavioral, i.e. non-synthesizable, SystemC model. In a stepwise refinement process parts of the model that should be realized in hardware are manually translated using the synthesizable subset of the SystemC language. This codesign approach would benefit greatly from the availability of operating system models in the SystemC framework. This feature has been announced for new releases of SystemC, but so far has not materialized.

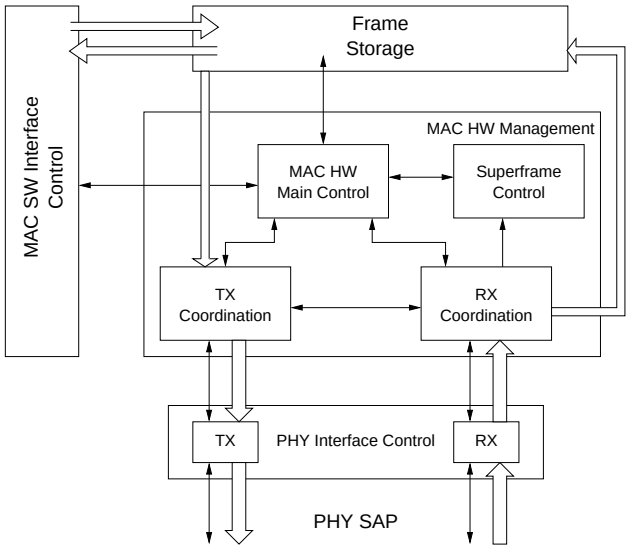


Figure 3.9: Hardware accelerator functional blocks for the implementation of the IEEE 802.15.3 MAC protocol reported in [HBB04].

Relation to own work The examples of protocol implementations presented in this section are results from projects in industry and academia. They document that SDL is used for the design and implementation of communication systems and show the typically employed design flow. The code generator CAdvanced is commonly used for software synthesis from SDL models.

However, the examples also show the lack of a general template and concepts for creating efficient SDL run-time environments for operating systems other than those already supported by Telelogic. If no operating system is used for the implementation and the light integration library from Telelogic is used, which provides a non-preemptive process scheduler, interprocess communication and support for timers, the code size of the target executable is larger and the performance lower than with a tight integration approach. The light integration scheduler is realized as an infinite loop where in each cycle it is checked whether the current system time has reached the expiration time of the first timer in the queue. If there is an SDL process that can be executed to process an input signal or timer, the process activity function containing the state machine implementation is called. The realization of the scheduler as an infinite loop has the drawback that the processor cannot be put into a low-power sleep mode waiting for the next timer interrupt

when there are no active processes.

The use of an instruction set simulator for performance analysis of the executable on the target hardware and even for hardware/software cross-development as reported in [DZM01] is a technique that has proven its effectiveness. We extend this approach by supporting the designer in specifying and detecting deviations from the timing requirements of an application or protocol and by reusing the SDL network model as a test bench that generates stimuli for the implementation model.

System-on-chip designs that contain an extremely low-power and resource-constrained microcontroller, hardware accelerators or coprocessors for tasks such as encryption or time-critical protocol functions, a wireless transceiver, on-chip SRAM and flash memory, as well as various peripheral interfaces have become available recently. Products from Texas Instruments featuring an 8051 microcontroller unit and low-power RF transceivers [Ins07] are excellent examples for such architectures. This demonstrates the validity and practical relevance of our focus on protocol implementations for embedded systems consisting of hardware and software parts. Our work, however, is not limited to single-chip solutions, but is just as well applicable to architectures consisting of a microcontroller connected to application-specific hardware via peripheral interfaces.

Chapter 4

Integrated Design Flow based on SDL

This chapter gives a conceptual overview of our SDL-based design methodology for embedded systems protocol design and implementation. Our design flow largely resembles existing approaches and makes use of mature tools.

However, as has been discussed in Sect. 1.2 and in the previous chapter, there are still gaps in the traditional design flow when targeting extremely resource-limited devices and in the support for hardware/software partitioning. In this chapter, we present our concepts for addressing the identified problems of creating efficient SDL run-time environments from real-time operating systems for deeply embedded systems and a cosimulation framework that allows the coupling of an instruction set simulator with an SDL simulator. We will not just state our new concepts, but provide the rationale for our decisions, discuss alternative approaches and their pros and cons. The prototypical implementation and validation of our concepts and tools that support the design flow, as well as design results for the IEEE 802.15.3 MAC protocol are then covered in the following chapters.

Compared to other approaches that use SDL merely as a tool for the specification of protocols and start a completely new software and hardware implementation effort without using the SDL model for synthesis, our solution saves development time and at a minimum allows the model to be automatically transformed into an all-software implementation. This way, the formally verified properties of the model are preserved in the implementation. The efficient run-time environment for deeply embedded systems reduces memory and processing overheads.

Finally, the simulation of the implementation model by an instruction set simulator gives highly accurate profiling information. Such an approach has been previously described, however, with our novel contribution of coupling the instruction set simulator with a functional SDL simulation of a communication network while reusing the original protocol model, we save development time for the test benches for the instruction set simulation. These benefits are hard to quantify and will vary between different system implementations.

The chapter is structured in three sections. First, we introduce the complete design flow from an initial SDL model to a hardware/software implementation for embedded systems. In Sect. 4.2, we investigate concepts for efficient SDL run-time environments, and address general questions concerning the interactive cosimulation of two SDL systems in Sect. 4.3.

4.1 General overview

We base our design flow for communication protocol design and implementation on the high-level language SDL. It is a popular and suitable language for this purpose due to

- its formal semantics, which enables to prove the functional correctness of SDL models,
- the high-abstraction level of the language, facilitating short development times and to focus on the behavioral aspects of the model,
- existing tool support for the simulation of models, thus being able to visualize protocol runs and efficiently debug the design,
- an immediate path towards software implementations supported by mature C code generators.

It should be noted, however, that the language exhibits a couple of drawbacks that must be addressed by the design methodology. Among those are

- the lack of specifying real-time constraints,
- the use of data types (unbounded strings and arrays) that are difficult to implement efficiently, and

- the missing support to express synchronous behavior as would be required for a direct translation to hardware descriptions.

The last point highlights that SDL uses only a single model of computation, namely extended finite state machines that communicate by the exchange of asynchronous signals.

The design flow, which has been sketched already in the introduction of this thesis, can be structured in different phases. In the following, we present these phases and motivate our decisions for particular tools and approaches.

System specification For the above mentioned reasons, we have chosen to use SDL for system specification. An initial SDL model can be derived, for instance, from a specification document describing a communications standard and written in a natural language such as English, or following a requirements analysis and definition of use cases. In contrast to a natural language description, the SDL model will unequivocally capture the abstract behavior of the system and still be implementation-independent.

The SDL specification is amenable to formal verification. For this purpose, a number of tools have been developed and can be applied as mentioned in Sect. 3.1. By means of simulations, the model can be validated and the performance of the protocol for various, on not yet specified parameter settings, for instance timer values, can be determined. All these activities complement our design flow. Results from the analyses are used to improve the model in a cyclic process as shown in Fig. 4.1.

In a next step, the SDL model is the basis for an automatic transformation into a software implementation. We have used the CAdvanced code generator from Telelogic [Tel06] for this purpose. In principle, this automatic process could yield highly efficient implementations if sufficient effort for analysis techniques is spent in the design of the code generator. Such optimizations have been proposed by a number of researchers, cf. [LO96], [LK99].

This potential is not fully exploited by commercially available tools. Therefore, taking into account the limitations of the chosen code generator, we manually extend the SDL model by replacing inefficiently handled concepts with more optimized, possibly implementation-dependent C code. We give examples for such optimizations, mainly concerning data types, addressing of signals, and buffer management, in the following Sect. 4.2. The development of a tailored SDL compiler

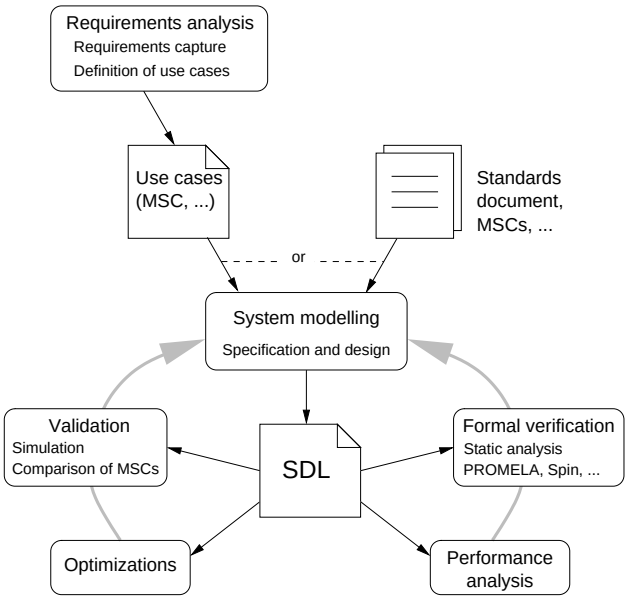


Figure 4.1: Specification and design phase of our proposed methodology. The SDL model is the starting point for the following transformation steps.

was out of the scope of this thesis.

Software synthesis Although it is conceivable to use SDL merely as system specification language and develop the software and hardware implementation from scratch, we argue that at least for the software implementation—which typically takes up the largest share of system functionality—the automatic translation to C code should be followed. This way, a previously verified model is the basis for the system under development. Furthermore, design time is shortened and needless re-implementation avoided.

Consequently, in our design methodology we obtain first an all-software system implementation by combining the SDL model translated into C code with an operating system and an integration library that provides an execution environment for the SDL model and is linked to operating system services. This is illustrated in Fig. 4.2.

There are two basic models for integrating the compiled SDL model with an operating system when using the CAdvanced compiler. They are called *light integration* and *tight integration*. The former maps the complete model to a single

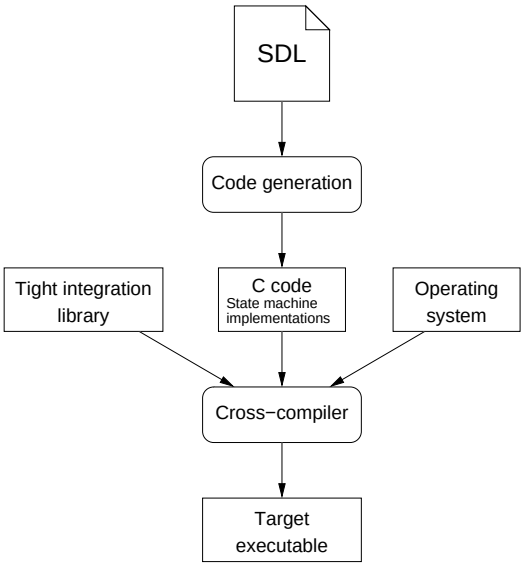


Figure 4.2: Software synthesis and integration with a target operating system.

OS process, while with the latter approach each SDL process is mapped to a corresponding OS process. The tight integration approach requires a more extensive and elaborate adaptation to the OS, but has two important advantages compared to the light integration approach:

- The execution of lower-priority processes can be preempted by those with a higher priority. This way, external signals may interrupt the current activity and be processed immediately in order to guarantee real-time constraints.
- General OS functions, such as scheduling or interprocess communication, have to be provided only once. In the light integration approach, an SDL run-time environment is required for this purpose, which increases code size.

The adaptation to the operating system has to be designed only once for each target operating system. Telelogic, the tool vendor of the CAdvanced code generator, recommends to use existing tight integration models as examples for targeting other operating systems. There is no general framework that would allow porting an existing integration model easily to a new operating system. The integration is realized by providing OS-specific definitions for macros which are part of the generated code.

In the next section, we discuss general concepts for the mapping of SDL models to operating systems for extremely resource-limited devices. As an example for a transformation of these concepts into a practical implementation we have created a prototypical tight integration library for the Reflex OS [Nol09]. This OS has also been used in our MAC protocol implementation that validates the complete design flow. We refer to Chapter 5 for a comprehensive presentation of the design of the tight integration model for Reflex.

Hardware/software partitioning Our design flow is laid out specifically for the development of soft real-time embedded systems consisting of hardware and software. A static real-time analysis as required for hard real-time systems is rejected in favor of system simulations. As already discussed in Sect. 1.2 this is due to the pessimistic estimations of worst-case timing behavior by static analysis tools, leading to overdesigned systems and, hence, inefficient implementations for the average case.

Behavioral hardware compilers from high-level languages are still lacking the efficiency of manual hardware design and have applications mainly in rapid prototyping. Therefore, we argue that the design space exploration and a mapping of functionality to hardware and software should be performed by hand. This is motivated by the fact that the design team usually has good knowledge about the complexity and probable bottlenecks of the system. By the way, this is also the approach taken by other design methodologies, for instance CORSAIR [DMTS00]. However, the effects of the design decisions must be analyzed afterwards.

Such an analysis must confirm that the timing requirements are met by the implementation model or indicate new performance bottlenecks that must be addressed in a new design space exploration cycle. This requires manually adapting the software partition, creating a new target executable, and designing models for the new hardware partition. With the new partitioning, another profiling and cosimulation cycle is started until the system reaches the performance targets. The hardware models serve as the specification for the following implementation phase. The design space exploration activity is shown schematically in Fig. 4.3.

System simulations can be used to evaluate the performance of a mixed hardware/software implementation. In order to obtain accurate timing information for a particular design alternative, we require a target executable, i.e. the software partition, and a model for the hardware partition. The initial partitioning

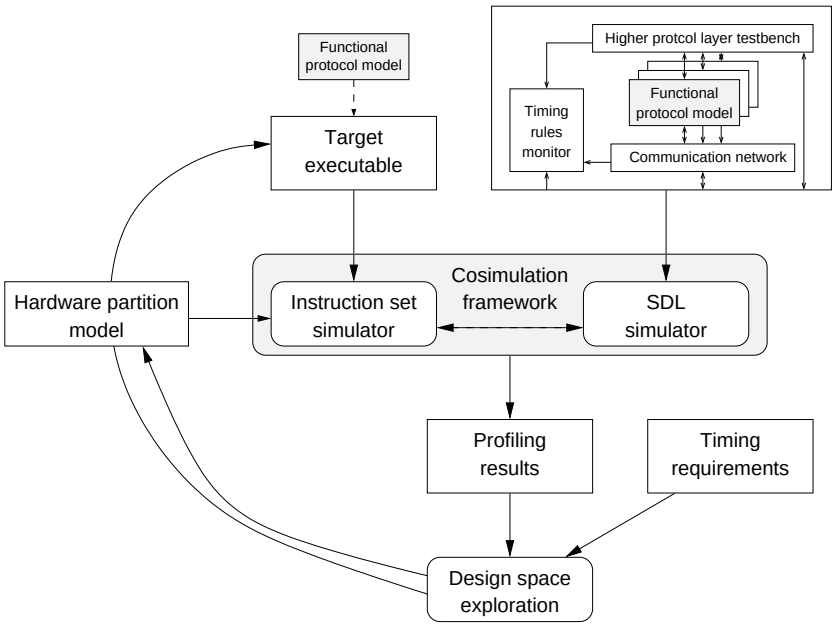


Figure 4.3: Starting from an initial software partition (*target executable*), an optimal hardware/software system is obtained in a cyclic process by making use of our cosimulation framework.

starts with an all-software system—the transformed SDL model integrated with the target operating system and compiled for the embedded system.

In our design flow, we rely on an instruction set simulator (ISS) for the target processor in order to emulate the real execution of the software and provide very accurate, i.e. cycle-true, profiling information. Instruction set simulators often allow the simulation of hardware models and their interactions with the software partition. An ISS is typically several orders of magnitude faster than a register transfer level (RTL)-level hardware simulation and still provides the same level of accuracy.

Communication protocol implementations, as well as other kinds of applications, need to interact with and receive messages from the local environment and remote communication partners. These entities provide stimuli at certain time intervals to the system-under-design and expect responses within certain deadlines.

We propose the coupling of an SDL simulator with the instruction set simulator. The major advantage of this approach is the reuse of the original, abstract SDL model and, thus, a shorter time of development. In most cases, a test en-

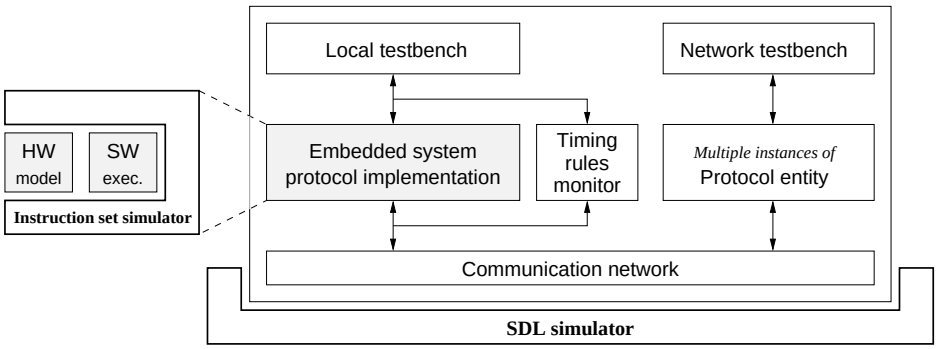


Figure 4.4: The instruction set simulation of the target system including models of the hardware partition is embedded in the overall network simulation based on SDL.

vironment for the protocol under development, including test benches, a model for the communication network and peer entities, is already created in the specification and design phase for validation purposes. As depicted in Fig. 4.4, the hardware/software system running on the instruction set simulator is integrated with the overall network simulation.

Additionally, a so-called timing rules monitor—a protocol-specific SDL process—can be included in the cosimulation in order to explicitly flag violations of timing requirements. The problem that makes such a monitor necessary is the missing support for the specification of real-time requirements in SDL. With the timing rules monitor it is possible to specify deadlines by which responses from the implementation model must have occurred after receiving certain stimuli. This SDL process makes the analysis of the simulation results easier for the designer, since it can be difficult to spot deviations from the acceptable behavior in the protocol run and automates the processing of trace outputs from the SDL simulation. The timing rules monitor receives copies of all SDL signals that are sent as outputs to the emulated implementation model and all signals sent by this model to its environment.

Interfaces between the two SDL models, one simulated by the ISS the other by the SDL simulator, must be provided, and the synchronization of the two simulation runs must be solved. Our concepts for this coupling are discussed in Sect. 4.3 in detail. Based on these concepts, we developed a software framework for an interactive cosimulation of the TSIM instruction set simulator for the LEON2

processor and the SDL simulator which is part of the Telelogic TAU SDL Suite. This prototypical implementation will be presented in Chapter 6.

The LEON2 processor is not a typical microcontroller for deeply embedded systems. The motivation for choosing this processor and the TSIM instruction set simulator was its availability when conducting this research work and the need for a processor that would support more than 64 kbytes of program memory in order to accommodate a software implementation of the rather complex IEEE 802.15.3 MAC protocol. It is important to note that our concepts for a cosimulation of an instruction set simulator and an SDL simulator are generally applicable, not only to the design of embedded systems. For this reason, we do not see that the relevance of our work suffers by validating our approach with the LEON2 instruction set simulator.

Implementation and test The hardware modules that have been identified in the codesign phase must be designed in a hardware description language. This also includes their interfaces to the software parts. In a following synthesis step, either a bitfile for an FPGA implementation or a chip layout for an ASIC are created. A mature tool chain for these tasks exists.

The hardware and software synthesis steps and the typically used tools are depicted in Fig. 4.5. The processor(s) and other hardware library components that are part of the platform are not explicitly shown. Naturally, the designed hardware components must be integrated into the processor subsystem.

A software interface for the hardware components must be developed, as well. This could be a device driver which provides access to registers and handles hardware interrupts. Together with all the sources for the target executable, the final executable is compiled for the target platform.

All components, such as processing and memory resources, where the software resides, are finally integrated on a single chip or a printed circuit board (PCB). The physical embedded system is now subject to further tests. In Sect. 2.1.4 the well-established protocol test methodologies, for instance conformance test and interoperability test, were presented.

Additionally, functional tests and production tests are carried out which are often supported by specific provisions in the design, such as a scan chain or debug registers. The design for testability is another current research topic. Certainly, input from this field could benefit our proposed design methodology. So far we do

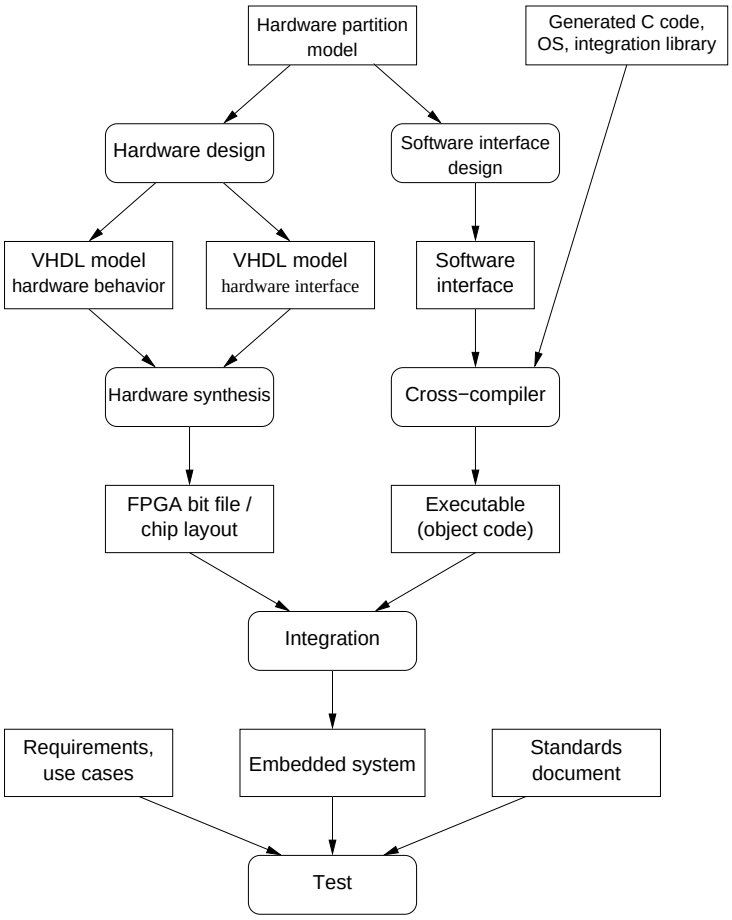


Figure 4.5: Sequence of the final steps in design flow: hardware and interface design, synthesis, integration, and test.

not directly address this topic in our work and leave it to the designer to choose the best available techniques to increase the testability of the embedded system.

4.2 SDL run-time environment for deeply embedded systems

Motivation and scope The automatic transformation of SDL models into executable software for embedded systems is an important aspect of our design methodology as it ensures that verified properties of the models are preserved by the implementation. For a software implementation, the state machine behavior

specified by the SDL processes must be translated, as well as a run-time system that provides all the implicit support functions, such as signal queues, process scheduling, and timer handling, is required.

In our design flow, the CAdvanced code generator from Telelogic was selected for software synthesis. We argue that the so-called tight integration approach is best suited to develop efficient applications that require process priorities and preemption. This code generator applies the *server model* for the transformation of SDL systems, that is each SDL process is represented by a concurrent task and has its own input signal queue. When an SDL signal is sent to another process, the signal is placed in the queue and processed only after the process was scheduled by the run-time environment. The server model was described in Sect. 2.1 in more detail.

The C code generated by CAdvanced from an SDL model contains macros in every place where a run-time system function must be inserted, for instance a signal output or setting a timer. For a general discussion of our concepts for an efficient run-time environment, the details of the code generator output are of less importance. For this reason, they will be outlined only in Chapter 5 where we present our integration library for the real-time operating system Reflex.

Any run-time system for the CAdvanced code generator must provide the following functionality:

- Management of the processes, including their signal queues.
- Management of signal buffers, i.e. signal allocation and deallocation.
- The timer mechanism and the current system time.
- An interface to the environment.

Tight integration libraries for target operating systems provide these support functions by making use of OS resources and services as illustrated in Fig. 4.6. Therefore, a specific mapping for each target operating system must be created. Telelogic provides such mappings only for a few operating systems. Recently, new operating systems have appeared that target extremely resource-limited and low-power microcontrollers, such as the MSP430 family from Texas Instruments with available program memory of 48 kbytes, a RAM size of 10 kbytes, and a clock frequency of maximum 8 MHz.

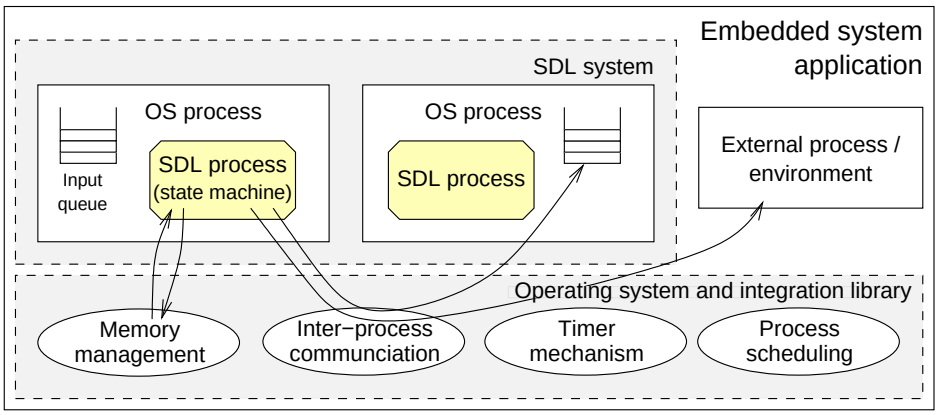


Figure 4.6: Software architecture for an embedded system application consisting of an SDL system, external (environment) process, and operating system. The tight integration library provides the links between SDL model and operating system.

Among the most popular of such operating systems are TinyOS [LMP+05], Contiki [DGV04], and Reflex [Nol09]. These operating systems provide only minimal services: simple schedulers, hardware drivers for input/output devices and timers. Dynamic memory management of objects on a heap is typically not included as it would blow up the code size significantly and is not strictly needed for the intended applications. The operating systems provide mechanisms to design applications based on an event-flow model, i.e. tasks are triggered by sending asynchronous events. This is very similar to the SDL model of computation and, therefore, facilitates an efficient mapping of systems specified in SDL to such an OS.

For these relatively new operating systems, no tight integration libraries are available for the CAdvanced code generator. In this section, we discuss general problems that must be addressed by any SDL run-time environment for deeply embedded systems. Among those are the allocation of memory for SDL signals or the handling of timers. We have selected Reflex as an example for a prototypical implementation of the design concepts. However, the implemented software library can be easily adapted to other operating systems.

Design objectives and limitations An obvious and primary requirement for any SDL run-time environment is to preserve the semantics of the language. Partly,

this requirement is already addressed by the code generator that must transform SDL models according to the semantic rules of the language. However, especially the handling of timers and input signals are problems related to the execution of SDL models and must be correctly implemented by the run-time environment. Input signals of SDL processes have to be processed in the order of their arrival, and also the queue of saved signals is organized according to the first in, first out (FIFO) scheme. After consuming a signal from the input queue, the save queue has to be traversed to determine how to handle the signals in this queue, before the next input signal can be consumed. For the correct handling of SDL timers, there are semantic rules concerning the operations used to set, reset, or retrieve the status of a timer. This will be covered in the discussion of our concepts for timer handling.

A high performance and low memory footprint of the target executable are further design goals that must be reached. Performance and memory overheads translate directly to a higher power consumption and increased resource requirements. Both are critical for deeply embedded systems. These are reasons why we decided to avoid dynamic memory management altogether. In other words, all memory that can be used by an implementation during its run-time must be pre-allocated at compile time and cannot be dynamically allocated from a heap. This is a decision that differentiates our concepts from previous approaches and tight integration examples. Its main advantages are that library code for dynamic memory management (`malloc`/`free`) is saved and that access to pre-allocated memory chunks is typically much faster. This can be seen from our comparisons with the light integration approach presented in Sect. 5.3. Below, we will go into the details of the memory management for SDL signal buffers.

However, by not allowing dynamic memory allocation from a heap, we exclude some features of SDL that cannot be used in modeling anymore, when targeting deeply embedded systems. This affects certain data types that have dynamic sizes, for instance strings or dynamic arrays, and the dynamic creation of process instances. If the designer uses such dynamic features of the language, the compilation will abort with an error, because we have defined the corresponding macros inserted by the code generator such that the C preprocessor stops with an error message indicating the unsupported feature.

Dynamic, and potentially infinite data types, must be replaced by finite data types with known size at compile time. This is not a severe limitation and can

be dealt with late in the design process when the abstract model is optimized. Infinite-sized data types are anyway only a theoretical concept. In every practical implementation there is a limit of the available memory. By forcing the designer to define upper limits for certain data types, this practical limitation is made explicit and leads the designer to include provisions for dealing with finite resources. Otherwise, the application could easily run out of memory and fail, since it was modeled under the precondition that there is always sufficient memory space available.

Dynamic process instances are, strictly speaking, not needed. The behavior of dynamic instances can be emulated easily with a single static instance of the process, which manages a set of process variables. Each dynamic process instance can then be represented by its own set of variables, including its current state, and an identifier. SDL signals sent to or from the static process instance would have to include the instance identifier in the parameter list. Though dynamic process instances can make system modeling easier in some cases, we do not really see their exclusion as a severe limitation, in particular when targeting resource-limited devices.

Additionally, we currently do not include support for services and procedures, in order to reduce memory overhead, though this support can easily be added. Alternatively, we recommend to use abstract data types and define operators for these types to model procedures that are defined outside SDL processes. Procedures inside SDL processes can be emulated with labels and jumps. This approach has been applied several times in the IEEE 802.15.3 MAC protocol model.

SDL allows three possibilities for specifying the receiver process of a signal. In a signal output statement, either the address of the receiver process, the name of a signal route that leads to the receiver, or nothing can be stated. In the last two cases, the receiver process is determined implicitly by the structure of the SDL model and the signal routes specifications between the processes. The code generator includes functions to calculate the receiver process if implicit addressing is used in the model. This approach incurs run-time and code size overheads. Therefore, we support only direct addressing in our run-time environment. This has no influence on the behavior that can be modeled in SDL, but merely demands additional coding effort from the designer to explicitly state the receiver process for each output signal.

Representation of SDL processes Since the CAdvanced code generator applies the server model for the transformation of SDL systems, a mapping of SDL processes to OS tasks is necessary. In principle, it would be possible to map multiple, or even all, SDL processes to a single OS task. However, this would require additional code for process scheduling within a task and, more importantly, would not allow preemption of processes within a single task. Therefore, our approach is to create one OS task for each SDL process instance. The OS scheduler is responsible for scheduling the processes according to their priorities or deadlines and whether they have got an input signal to process.

It is impossible to develop a software template for the run-time environment functions that is independent of a specific target OS. This is due to the different languages used in the design of the operating systems, for instance nesC, C, or C++. Therefore, we limit ourselves to the description of general concepts that can be adapted to a specific implementation. In particular, the exact representation of schedulable tasks differs widely in the studied operating systems.

Memory management for signal buffers Communication between SDL processes is realized by sending asynchronous signals that may carry any number of parameters. Hence, buffer space for storing signal parameters and other control information such as the sender's address is required. This memory buffer is occupied only during the lifetime of the signal, i.e. from the time when the sender process creates the signal and fills its parameters until the receiver consumes or discards the signal. An unsuccessful attempt to allocate sufficient memory for an output signal will lead to a system failure.

Memory management for signal buffers can be implemented in two different ways. There could be a single heap from which memory for all signals is allocated dynamically. Alternatively, pools of memory for each signal type can be preallocated statically and used only when a signal of this type is created. We will briefly outline the pros and cons of both strategies.

The dynamic memory management strategy has the advantage that the same buffer can be reused for different signals over time. This reduces the total size allocated for signal buffers in the system compared to the other strategy. However, the maximum size of signal memory that can be allocated at a time must be known in advance in order to choose a sufficiently large heap size. In most systems the exchange of signals happens more or less randomly, triggered by external events.

It is, therefore, difficult to analyze what is the maximum amount of memory required for active signal buffers. Even worse, the heap may become fragmented when signals of different sizes are allocated and deallocated in a non-deterministic order. In this case, even if the sum of available memory chunks in the heap would be large enough to accommodate a newly created signal, there might be no single contiguous chunk that is large enough. Designing the run-time system in such a way that a defragmentation of the heap could be performed during normal operation would make the system very complex.

In the case of statically allocated pools of signals with the same size, fragmentation cannot occur. Furthermore, it is comparatively simple to analyze the worst-case chain of signal allocations triggered by an external event or timer expiration (cf. activity thread model). When the maximum number of external signals allocated at a time is limited, the required worst-case buffer space for all signals that can possibly be created in the further course of actions can be determined. The number of timers in the system is fixed by the SDL model anyway.

For the aforementioned reasons and to guarantee that the SDL system implementation does not run out of memory for signal buffers at run-time, we propose a memory management scheme based on statically allocated signal buffers.

In some cases, the worst-case number of signals to be pre-allocated in pools may still exceed the size of available memory resources. Severely limiting the number of external signals that may be input into the SDL system could lead to low performance, because there is little buffer capacity for a burst of input signals, thus reducing the throughput.

As an alternative, we propose to allocate a single extra memory buffer large enough to accommodate one SDL signal buffer with the size of the largest signal type in the model. When a signal allocation fails because of insufficient memory resources, the SDL run-time system must provide a memory buffer where the application can safely write the signal parameters to, since in the generated code there is no provision to handle such a case. This signal, of course, is used only as a temporary memory buffer and is never actually consumed by another process. However, write access to parameter fields will not do any harm to the application, since there is a valid memory region for the signal buffer. The dummy signal is identified by its unique signal ID. The input queues of SDL processes ignore any signal with this identifier.

This dummy signal can only be used for such signals where a loss does not

cause incorrect behavior. Whether this is the case or not must be analyzed by the designer as it is based on semantic knowledge of the SDL model. For all SDL signals where a memory allocation failure would lead to a system failure, sufficient memory resources must be provided in the pre-allocated pools. Alternatively, the number of input signals from the environment could be limited to avoid an overload of the system.

Since each SDL system differs in the number of signal types and pool sizes, there can be no generic system implementation. We provide a `SignalBufferManager` class that can be easily adapted to new implementations—only the number of pools and their signal types and sizes must be given. This could be supported by a separate tool in a future development of our design methodology.

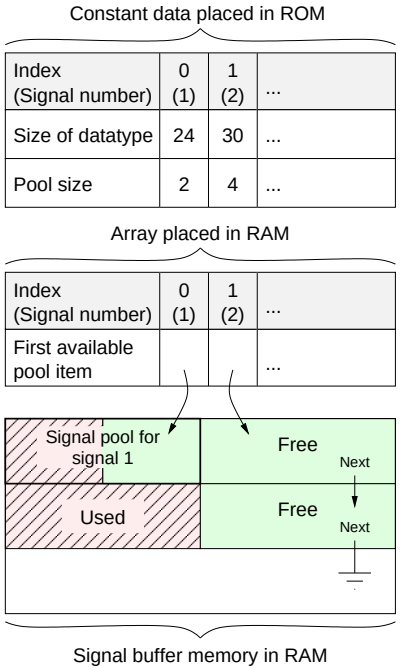
In Fig. 4.7 the organization of the RAM when using pre-allocated pools for each signal type is shown. The individual pools are organized as linked lists of available signal buffers. Pointers to the first available pool elements are kept in an array, which has as many items as there are signal types. To facilitate queue management in the process input and save queues, the data type used for SDL signals contains a `next`-pointer that is also used for referencing the next free signal buffer.

The pools are initialized at system start-up time. For this purpose, the number and size of the individual signal types must be known. This information is kept in a constant array, which can be placed in ROM.

Both, allocation and deallocation operations, are performed in constant time. Allocation simply returns the current first pool element and assigns the reference to the new first element to its `next`-pointer. When a signal buffer is deallocated, it becomes the new first element in the linked list and its `next`-pointer is set to the previous first element. The code generator assigns each signal type a continuous number starting from one. The signal number is a part of the data type representing an SDL signal. The right signal pool is accessed by using the signal number as the index for the array of `first`-pointers.

It is, however, also possible for an application developer to provide a `SignalBufferManager` implementation that uses a shared heap for all signals. We do not generally recommend this method, even though it saves on RAM resources required by the application.

Timer management SDL processes may contain timers to control the behavior




```
class SignalBufferManager {
  enum {
    ComputeRequest_PoolSize = 2,
    ComputeConfirm_PoolSize = 2,
    NUM_SIGNALS = 2,
    BUFFER_MEMORY_SIZE =
      ComputeRequest_PoolSize *
        sizeof(ComputeRequestType) +
      ComputeConfirm_PoolSize *
        sizeof(ComputeConfirmType)  }

  char buffer_memory[BUFFER_MEMORY_SIZE];
  static const PoolInfoType pool_info[NUM_SIGNALS];
  xSignalHeader first_item[NUM_SIGNALS];
  ...
  typedef union AnySignalType {
    ComputeRequestType dummy1;
    ComputeConfirmType dummy2; };
  AnySignalType dummy_signal;
  ...
};

const PoolInfoType
SignalBufferManager::pool_info[] = {
  PoolInfoType(ComputeRequest_PoolSize,
    sizeof(ComputeRequestType),
  PoolInfoType(ComputeConfirm_PoolSize,
    sizeof(ComputeConfirmType)     };
```

Figure 4.7: The realization of statically allocated signal buffer pools by using two arrays with control information and a memory area for the actual buffers is shown schematically on the left. On the right-hand side, a source code example of a `SignalBufferManager` class for the SDL system in Fig. 5.2 adopting this implementation scheme is presented. `xSignalHeader` is the base class for all signal buffer types.

of the state machines. The number of timers in a given model is fixed, however the timers can be set and reset at run-time, and, of course, generate a signal when they expire. One of the responsibilities of the run-time environment is to provide the timer management.

The CAdvanced code generator inserts a number of macros at places where a timer is defined, set, reset, checked for activity, and its expiration handled by the process. The tight integration library has to define these macros and to set up a timer handling infrastructure linked to the operating system services or hardware timer resources such that the SDL semantics with respect to timer handling is preserved.

Since expired timers are treated like other SDL signals, we represent timers in the same way as signals. Each timer signal has a unique identifier, which is

generated by CAdvanced.

In our design concept, all currently active timers, i.e. those that have been set and not yet expired, are managed by the so-called *timer process*. In this entity the timer signals are kept in a double-linked list, ordered by their expiration times. The timer process detects when the first timer in the list expires, removes the signal from the list, and sends it to the input queue of the process instance it belongs to.

We represent time as a 32-bit integer number. The time unit can be freely chosen by the application designer and depends strongly on the granularity of the system timer or hardware timer component. In any case, during the run time of an application an overflow of the continuously incrementing time may occur. Therefore, special care is taken in the timer process to handle the wrapping of the 32-bit integer number. Timer signals with a wrapped expiration time are ordered at the end of the non-wrapped timers, and a pointer to the first wrapped timer is maintained.

There are two possible approaches how the timer process can detect the expiration of a timer: either by receiving a periodic tick event or by an event that is triggered exactly at the time of expiration. The latter approach is, of course, more efficient, but may not be feasible on every hardware platform.

For instance on the MSP430 microcontroller, there are hardware timer components that can be programmed to generate an interrupt at a certain time. Such a hardware timer can be used to design the timer process in an efficient way.

In order to keep the tight integration layer flexible and make use of any hardware timers available on the target platform, we decided to realize processor-dependent functionality in a separate module. Then, only this class, which provides functions to query the current time and to schedule an event for a later point in time, must be adapted for a different hardware platform. This concept is shown in Fig. 4.8 in an UML-like notation. The timer process is a schedulable task. It is activated when the first timer in the list expires. This is realized by a platform-dependent timer service. The timer process will then remove all expired timer signals from its list and send them to the corresponding processes. An implementation which uses an interrupt-driven hardware timer expiration is presented in Chapter 5.

There are four functions that can be performed on timers: a check whether the timer is active, starting and stopping the timer, and its consumption by the SDL

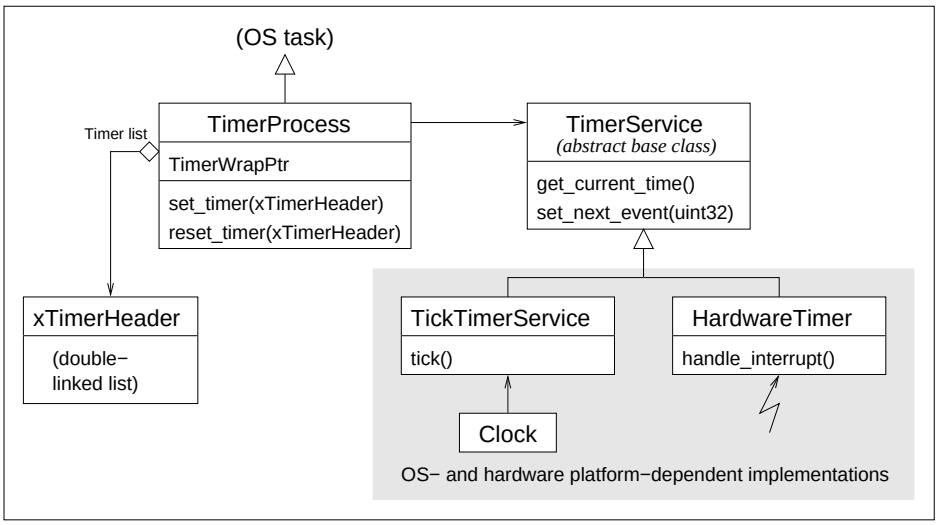


Figure 4.8: General software architecture for the timer management showing two variants for the detection of timer expiration.

process. In order to support these functions most efficiently, we added a field to the timer signal data structure that reflects the current state of timer signal. The state can be either idle (not started), running (not yet expired), currently in the input queue of the process (expired, but not yet consumed), or located in the save queue.

According to the SDL semantics, a timer is considered active until it is consumed by the process, i.e. the timer is not idle. This check can be performed in constant time.

Resetting a timer means removing it from either the list of running timers in the timer process, the input or save queue of the process. All these objects are double-linked lists, so that the delete operation can be realized in constant time. Access to the timer process list and the input queue must be exclusive to prevent concurrent write access to these data structures from other processes.

Before a timer is started, it is first reset. Then, the timer signal and its expiration time is passed to the timer process where it is inserted at the right place in the ordered list. This operation can be implemented in linear time by browsing the list from one end. Typically only few timers are running at the same time, so this should not be a performance bottleneck, and optimizations at the expense of larger code size are not necessary.

Finally, when a timer signal is consumed or discarded by the process, its state is set to idle again.

Interfacing the environment Without the possibility to interact with its environment, any SDL system would be useless. One obvious way of communication between SDL processes and other application parts outside the SDL system is via asynchronous signals—just like SDL processes communicate with each other. Since in the model there is no possibility to differentiate between a signal output to another SDL process and to the environment, exactly the same mechanism for signal exchange as presented above is applied. This means that direct addressing of signals to the receiver is used. There could be different environment processes all represented by their own addresses.

Such an environment process could also act as an interface to a hardware module. Received SDL signals could be transformed into a hardware access and trigger the desired functionality. Vice versa, an interrupt generated by the hardware might cause an SDL signal to be sent to an appropriate receiver process.

Another possibility to interface the environment from the SDL model is by calling imported functions or using abstract data types that have their operations implemented in C or C++. Telelogic, for instance, offers a tool that can read C++ header files and allows to create SDL abstract data types from C++ classes. These types can be used in the SDL model just like native data types. Their operations, however, are defined outside SDL.

The external functions are linked together with the generated C code. This makes it possible to access hardware peripherals of the microcontroller or dedicated hardware without the cumbersome exchange of asynchronous signals.

Reading a status byte from a peripheral would require the exchange of two SDL signals: a request to the environment and a response back into the model. Besides the run-time overhead for allocating and deallocating signal buffers, the parameters of the buffers must be set, and, most importantly, scheduling overhead is added. This makes it clear why we turned away from automatic hardware synthesis tools that map complete SDL processes to either software or hardware and realize all communication between SDL processes by exchanging signals. In the alternative approach, the mentioned function could be imported into the SDL model and directly called.

The two principle methods for interacting with the environment of an SDL

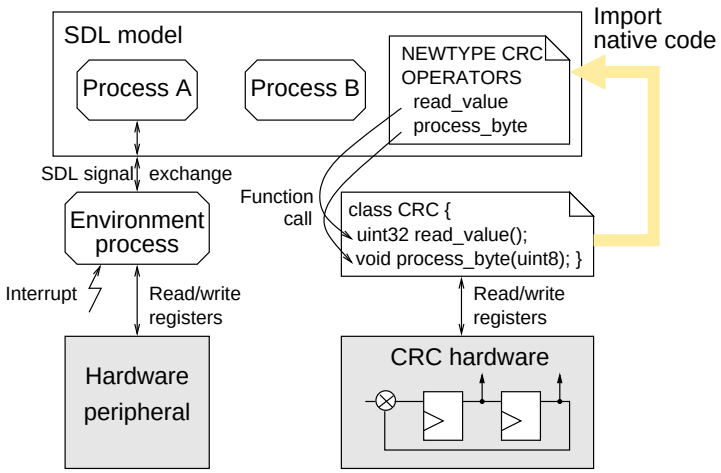


Figure 4.9: Principles for interacting with the environment, including hardware, from the SDL model.

model are illustrated in Fig. 4.9. On the left, the exchange of SDL signals is depicted, while on the right-hand side the use of an abstract data type linked to an external library is shown. The figure also illustrates how dedicated hardware can be accessed from those parts of the SDL model that are realized as software.

When integrating such external components with the SDL model, a co-verification must be performed to ensure the correctness of the whole system. For this purpose, a formal model or SDL specification of the external component could be designed and integrated with the existing SDL model.

4.3 Cosimulation with an instruction set simulator

Rationale and objectives When developing a communication protocol implementation following our design flow outlined in the first section of this chapter, that is by creating an SDL model of the protocol, applying automatic code generation and an integration with the target operating system, the result may be an implementation that does not meet the specified timing requirements or would necessitate a higher clock frequency and, hence, increase the processor's power consumption. In such cases, it is worthwhile to consider hardware/software partitioning of the protocol. This means that for the functionality that can be identified as a performance bottleneck a dedicated hardware component is designed. This

protocol accelerator serves as an application-specific coprocessor.

In Chapter 3 we presented SDL-based hardware/software codesign approaches with automatic generation of hardware descriptions from SDL processes (cf. Fig. 3.6). This methodology depends on annotations of the SDL model by which the designer maps SDL processes to software (CPU) or hardware (ASIC). The decision for this mapping is made intuitively, based on estimations of the computational complexity and timing requirements [Mut02]. We propose a simulation-based approach to obtain accurate estimations of the performance of protocol implementations and to identify performance bottlenecks.

The automatic generation of VHDL code from the SDL model is outside the scope of this thesis. The tools developed for this purpose (cf. Chapter 3) are targeted for rapid prototyping systems and allow only the mapping of complete SDL processes to hardware. In our approach, however, any part of the original software model can be realized in hardware. Hardware design and the definition of the interface to the software are manual activities. At the current state of the art for behavioral hardware compilers, this approach promises to yield more efficient implementations.

In the case of communication protocol implementations it is particularly important to analyze their timing behavior. We have discussed *static timing analysis* in Chapter 3 and found that worst-case estimations often lead to inefficient implementations. They are, however, required for the design of *hard real-time systems*. In this thesis, we focus on *soft real-time applications* and target ultra-low-power and resource-limited devices. Therefore, the implementations must be very efficient in terms of energy and resource usage.

As an alternative to static timing analysis, *system simulation* is a suitable and often applied technique to gain insights about the performance of an implementation.

The objective of a simulation-based approach is to obtain sufficiently accurate estimations of the performance of the target system. It should be possible to study the effects of variations in the processor parameters (clock frequency, caches, etc.), explore different hardware/software partitionings, while at the same time facilitating reuse of protocol test benches. A simulation run must be completed within a couple of minutes to be able to explore many design alternatives. All this can be achieved with the help of an instruction set simulator (ISS) and by running a functional SDL simulation in parallel to an ISS, as we discuss further below and

demonstrate in Chapter 6.

An ISS is a tool that emulates the real execution of a program on a target processor. Some instruction set simulators also allow the simulation of a coprocessor or user-defined hardware modules connected to the processor. Before going into the details of our cosimulation framework for hardware/software partitioning we will briefly discuss conceivable *alternative solutions* for simulation-based performance analysis.

- One simple approach would be to simulate the performance of the protocol software on the *development platform* rather than the target platform. For this purpose, the SDL simulation is performed, for instance, on a Windows PC, and conclusions are drawn from profiling information obtained during this simulation run. Some effort has to be spent to relate the execution time to the expected execution time on the target processor. To model the effects of hardware accelerators, simple stub functions could be used. In general, this proposed method can only provide first indications to possible performance bottlenecks, but is too inaccurate to produce reliable and useful results.

- The most accurate estimations of system performance could be obtained by means of an *register-transfer-level (RTL) simulation* of the microcontroller running the protocol software and any hardware accelerators. This RTL simulation could also be coupled with a high-level SDL network simulation. Unfortunately, due to the long simulation runs in the order of hours and days, this approach does not lend itself to the exploration of many design alternatives. Moreover, RTL or behavioral models for the protocol accelerators have to be designed before the simulation can be started. The RTL simulation of the target system, however, is often conducted at the end of the hardware design process to validate the design.

- To avoid the extensive simulation runs, it is a viable option to implement the target system on a *prototyping board*, for instance by making use of programmable hardware and an off-the-shelf target microcontroller. Hardware descriptions of the protocol accelerators have to be designed for this purpose. This could be a time-consuming task and severely limits the flexibility in exploring different design alternatives.

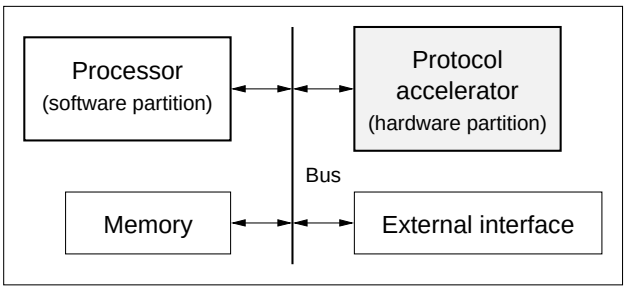


Figure 4.10: Principle building blocks of our target architecture for the hardware/software codesign process.

Our cosimulation approach supporting hardware/software partitioning is specifically aimed at embedded systems architectures with a single microcontroller and the possibility to integrate protocol accelerators. The microcontroller runs the software part of the communication protocol that is to be implemented. Of course, the embedded system may consist of other processors or DSPs, as well. However, we do not support the partitioning of protocol software onto multiple processors. Memory and external interfaces are further essential building blocks of the target architecture, as depicted in Fig. 4.10.

When studying the performance of a given protocol implementation, external events must be generated and presented as inputs to the system under test. Possible events are, for instance, a service primitive from a higher layer or a received protocol data unit from the lower layer. The time needed to react on the events is measured and compared with the specification.

Typically, when modeling a communication system in SDL, test benches are developed in order to validate that the system's functionality has been correctly represented. For this purpose, a number of individual communication systems—the protocol entities—are connected by a model of a communication network that allows to exchange protocol data units between the entities. In such a *functional SDL simulation*, the real execution time is irrelevant. Typically, a global simulation time for all entities is assumed. This global time does not advance when transitions are executed, but only by the expiration of the next scheduled timer.

The functional network simulation is ideally suited to generate input events for the target system. Then, it is necessary to study at what point in time the target system produced a response to the received event, for instance in the form

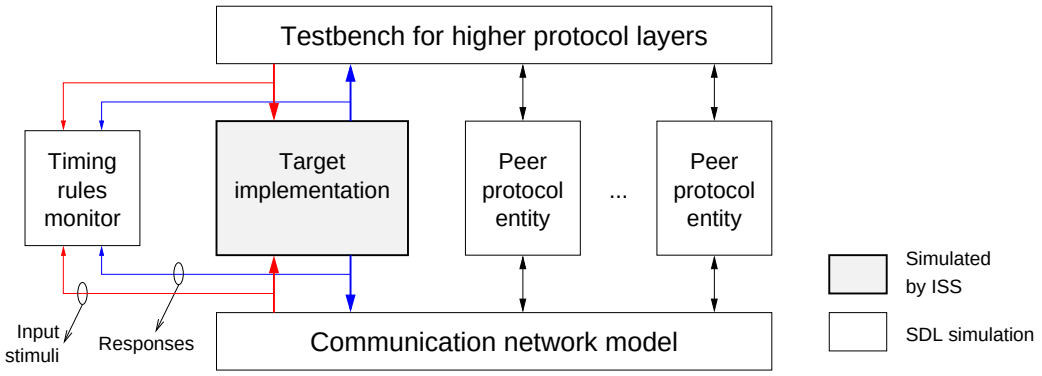


Figure 4.11: The target implementation simulated by the instruction set simulator receives inputs from and sends responses back to the SDL network simulation. An optional timing rules checker is used to flag violations of the protocol's timing specification by the target implementation.

of a transmitted packet or an indication to the higher layer. The protocol engineer has to check on the basis of the protocol specification if the observed behavior is correct. By using the output of the protocol executed by the ISS as an input to the functional network simulation, any deviation from the expected or tolerable timing behavior is made visible. For this purpose, the traces for the simulations with and without the system-under-test have to be compared.

As an extension to this approach, we give the designer the possibility to specify a set of rules of acceptable timing behavior for the protocol running in the ISS. These rules are described in SDL in a separate entity. All inputs to the target system and all of its output signals are copied and passed to this *timing rules monitor* as shown in Fig. 4.11. With the help of timers it is possible to detect if any responses were not generated in the acceptable time interval. Figure 4.12 exemplifies how one might specify the expected time interval for the transmission of an acknowledgment frame after the reception of another frame.

Naturally, the exact method of coupling the SDL simulator with an ISS depends on the provided interfaces by the ISS. In the remainder of this section, however, we first identify *general problems* related to an efficient cosimulation of SDL models and propose conceptual solutions that address these problems. In Chapter 6, we demonstrate how these concepts have been applied to a cosimulation framework connecting TSIM, the instruction set simulator for the LEON2 processor, with the

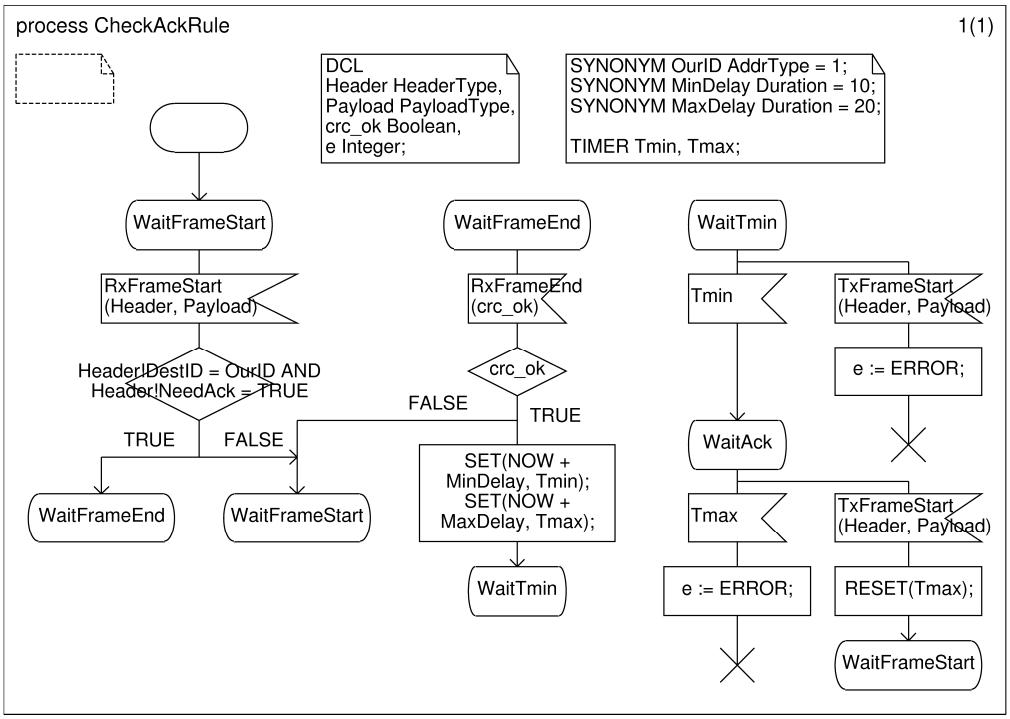


Figure 4.12: SDL process that illustrates a timing rule check. An acknowledgment frame is expected to be sent within an interval of 10 to 20 microseconds after the end of a successfully received frame.

SDL simulator from Telelogic, as an example.

Synchronization between SDL simulations As presented above, our goal is to provide an interactive cosimulation of two SDL models, namely the implementation model executed by an ISS and a functional network model acting as the test bench simulated by an SDL simulator. Since both simulators run independently of each other, there is no common simulation time and no single event queue.

The external implementation model in the ISS must be integrated in the communication network simulation as if it would be part of the functional SDL simulation. Since in SDL all processes run concurrently and synchronize only by exchanging signals, this means that the ISS and SDL simulator may proceed with their simulations independently until there is an SDL signal sent from one to the other. At the point of this signal exchange, both (local) simulation times must by

synchronized. A situation where one of the simulators receives a signal that was sent at a time ahead of the receiver's simulation time would violate the semantics of the model. This becomes evident by considering that the receiver process could have sent a signal to the sender process *before* it generated the other signal and might have controlled the behavior of the sender process in such a way that this signal would not be generated, at all. A rollback of simulation steps is not possible for the SDL simulator and, usually, also not for the instruction set simulator.

A straightforward and naive cosimulation approach would be to progress both simulators, one after the other, in infinitesimal time steps, for instance one clock cycle. Though the semantics of the model would be preserved with this technique, the performance would be very low due to the immense scheduling overhead.

Fortunately, a more efficient method can be applied since the SDL simulator uses an *abstract* time for the functional simulation as described above. The simulation time is advanced only by the expiration of timers. The execution of transitions does not take time. There are no external SDL signals into the functional simulation model except those sent from the instruction set simulator, as shown in Fig. 4.11. Under these preconditions, a highly efficient coupling of the two simulators can be realized that requires a synchronization of the simulators only when the ISS send a signal to the SDL simulation and when a timer in the functional SDL model expires.

The principle of operation can be summarized as follows. Initially, both simulations start at simulation time 0. The SDL simulator executes all transitions that are triggered at this time, i.e. the initial transitions of all processes and any transitions triggered by signals sent from these transitions. Time has not been advanced by this step. Then, the SDL system waits for a timer to expire or for an external signal from the environment.

Before advancing the simulation time to the timer expiration, the ISS is allowed to progress at most as many instructions as correspond to the time interval until the next timer expires. The ISS, however, must stop immediately when it outputs a signal to the functional SDL network simulation. The reason for this is that the signal might cause another signal to be sent back to the ISS. Once the ISS has stopped the simulation, the simulation of the SDL simulator is advanced to the time reached by the ISS, i.e. both simulation times are synchronized. If there was an SDL signal from the ISS, it is sent into the SDL model.

At that point, the SDL simulator will process any transitions that can be

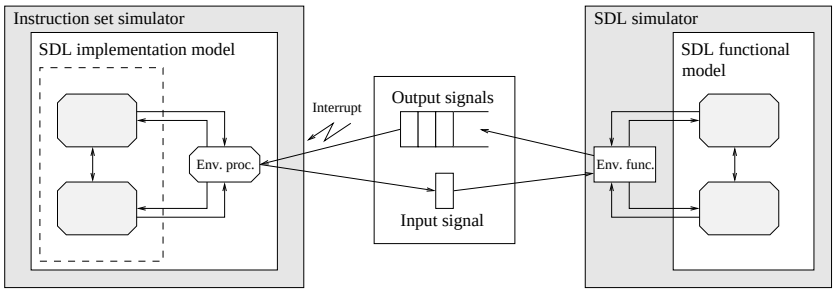


Figure 4.13: Principle of connecting an SDL simulator with an instruction set simulator (ISS). A FIFO signal queue is required to store all SDL signals sent from the functional model before the simulation time is advanced. In the other direction, from ISS to SDL simulation, at most one signal is sent at a time, because the ISS stops immediately when a signal was sent, and it is consumed by the SDL simulator.

triggered now. Of course, there could be one or more signals that have to be sent to the implementation model simulated by the ISS. These signals need to be queued in FIFO order for the ISS to retrieve (cf. Fig. 4.13). Then again, the instruction set simulation is advanced until the time of next timer expiration. If it had previously stopped because of sending a signal, the simulation will resume at that point in the SDL implementation model.

If there is one or more signals sent by the SDL simulator and queued for the ISS to process, this must be indicated to the implementation model. The easiest possibility would be by requesting an interrupt to the target processor when a signal is written into the queue. When this interrupt is unmasked, the target process will handle the interrupt and consume all signals from the queue for later processing. The interrupt handler has to allocate signal buffer memory for the new signals, send them to the appropriate receiver processes, and, thus, cause the processes to be added to the scheduler's ready queue. Depending on the SDL process priorities, these processes will then execute the transitions specified in the SDL model.

It is important to note that the order of the SDL signals is maintained as originally transmitted, because they are stored and retrieved in FIFO order. Hence, the SDL semantics are not violated.

The simulations are continued until a condition for their termination is reached.

Figure 4.14 summarizes the steps of the interleaving algorithm.

Figure 6.7 on page 156 illustrates the synchronization between SDL and instruction set simulations. It also shows examples for the exchange of signals between the two simulations.

Exchange of SDL signals As Fig. 4.13 shows, there must be memory buffers for queueing any output signals from the SDL simulator before the instruction set simulation is continued and consumes these signals from the queue. Similarly, memory space for an input signal has to be provided. A small software module must be developed that manages the queues and generates an interrupt for the ISS when a signal is placed in the output queue.

In the opposite direction, when it has completely received an SDL signal from the implementation model, the ISS must be stopped. The software module implementation is specific for each ISS, since there are no standardized interfaces to control an ISS. The queue management functionality, however, can be reused for other instruction set simulators.

The signal buffers must be mapped into the address space of the target processor, so that the implementation model can access the signal parameters and write its output signals to the allocated memory space. Therefore, the ISS must support the development of user-defined memory-mapped modules.

The protocol engineer has to extend the implementation model by an environment process which is responsible for copying signal parameters to the memory region of the input signal for the SDL simulator. Additionally, this environment process must handle the interrupt generated by writing a signal to the output queue and copy all signal parameters from the queue memory.

One problem that must be taken care of is the potentially different representation of signal parameters on the target processor and on the development computer system. A simple memory copy of the signal buffer is not sufficient, because the compilers for the target and for the host systems might use different memory layouts, i.e. byte ordering and alignment, for the C structures representing SDL signal parameters. Hence, SDL signals must be translated from the host system to the target system, and vice versa, such that the parameters are interpreted in the same way. One option would be to use a tool that can generate transformation functions for each signal from the signal data type definition automatically. Typically, the external interfaces of an SDL model consist of few signal types, probably in the

range of 10 to 20. For this small number of signals, the transformation functions can be written with little effort by the designer. A transformation tool could be designed as an extension of our work, in the future.

Concept for cosimulations with Telelogic's SDL simulator So far we have discussed general questions concerning the cosimulation of two SDL systems. In the following, we present how our concepts can be realized with the SDL tool from Telelogic, independent of the instruction set simulator.

For this purpose, the SDL model is first transformed into C code with the help of the CAdvanced code generator. Telelogic provides an SDL run-time system that can be used to create a stand-alone application from the SDL system on the host computer. At compile time, it must be configured to use an abstract simulation time, and not to the real execution time. A graphical simulator user interface can be, but does not have to be, connected to visualize the behavior of the SDL system and control its execution.

Additionally, the designer must provide four *environment functions*, that are called by the run-time system at initialization and termination time as well as for the exchange of SDL signals with the environment, when targeting a cosimulation. The four environment functions with their parameters and short descriptions are listed in Table 4.1.

Because the run-time system is responsible for the scheduling and execution of SDL processes, timer handling, as well as for interfacing with the environment, we use the term *SDL simulator* to describe this software. It does not require a graphical user interface and can be controlled with textual commands.

From the perspective of the SDL simulator, the instruction set simulation is part of its environment. This means that all signals addressed to an SDL process inside the implementation model are routed through the environment function `xOutEnv`. Vice versa, signals coming from the model inside the instruction set simulation are sent to the SDL simulation via the `xInEnv` function. Remember, the program executed by the ISS was created from the original SDL model of the protocol.

The synchronization of the SDL simulator with the ISS can be achieved by *interleaving the execution of the functional SDL simulation with the instruction set simulation* according to the general concept introduced before.

At the beginning of the cosimulation, the instruction set simulator must be

Table 4.1: Environment functions that are called by the SDL simulator and must be supplied by the designer.

Function	Purpose
`void xInitEnv()`	Initializes external code. Called once at simulation start.
`void xCloseEnv()`	Terminates the environment. Called when the simulation ends.
`void xOutEnv( xSignalNode *S)`	Called each time a signal is sent from the SDL system to the environment. `S` is a reference to this signal. After performing any appropriate actions, the signal must be released to free the memory it uses.
`void xInEnv( SDL_Time next_event)`	Enables the reception of signals from the environment by the SDL system. The parameter `next_event` will contain the time for the next event scheduled in the SDL system. To implement the sending of a signal into the SDL system, two functions are available: `xGetSignal`, which is used to allocate memory for the signal, and `SDL_Output`, which sends the signal to a specified receiver.

started and configured to simulate the target executable of the implementation model. This can be done from the `xInitEnv` function, which is called at system start-up time. The implementation model is just loaded, but not started, yet.

The `xOutEnv` function is called when the abstract SDL model sends a signal to the external implementation model. This function is responsible for placing the signal into the output signal queue, so that it can be consumed later by the implementation model (cf. Fig. 4.13).

When all transitions that could run at the current simulation time were executed, the `xInEnv` environment function is called with a parameter indicating the next scheduled timer event. If this is greater than the current simulation time[1], the ISS can be resumed for the number of clock cycles that correspond to the time interval until the next timer expires.

Control is returned to the `xInEnv` function when the ISS has performed this number of cycles or the implementation model sent a signal back to the abstract SDL model, as outlined previously. In any case, the abstract simulation time is advanced to the current ISS simulation time. If there is an input signal, it is

[1] In SDL, timers can also be set to expire immediately.

(0)	Initialization: start ISS and load application, both simulation times are set to 0.
(1)	Simulate SDL system until system is waiting for timer expiration or external event. Any signal to the environment (`xOutEnv` called) is passed to the ISS.
(2)	`xInEnv` is called with timestamp of next timer event as parameter.
(3)	Resume instruction set simulation for as many clock cycles that its simulation time reaches the time of the next timer event.
(4)	The ISS stops when a signal to the SDL system is emitted or the assigned number of clock cycles have been simulated.
(5)	Advance SDL simulation time to current ISS time. If a signal was sent by the ISS, send it into the SDL system. `xInEnv` returns with the effect of going back to step (1).

Figure 4.14: Principle operation of the interleaving algorithm to synchronize SDL and instruction set simulations.

sent to the appropriate receiver process, depending on the signal type. With this, `xInEnv` returns. The run-time system schedules the next SDL process and checks if any timers have expired. Then again, `xOutEnv` and `xInEnv` are called, until the simulation finishes.

Finally, `xCloseEnv` is called and will terminate the instruction set simulator. The interleaving algorithm and the role of the environment functions are summarized in Fig. 4.14.

Chapter 5

Efficient Integration of SDL Models into Reflex

In order to validate the concepts for an efficient SDL run-time environment presented in Sect. 4.2, the author of this thesis has developed a prototypical implementation of a tight integration layer for the real-time operating Reflex. This operating system was chosen because it is strictly object-oriented and programmed in C++. This makes the presentation of the software architecture for the run-time environment very clear and illustrates how our general concepts have been realized.

Furthermore, because it has very small memory footprint in the order of few kbytes, is comparable to TinyOS in its programming abstractions for event-driven applications, and has been ported to a number of microcontrollers, it stands as an excellent example of an operating system specifically designed for deeply embedded systems. Reflex features an earliest-deadline-first scheduler, which is particularly useful for real-time applications. We will revisit the design concepts of Reflex later in this chapter.

The tight integration library provides an execution environment for SDL models. It consists of a set of macro definitions for the macros contained in the generated C code[1] and a lightweight software library. The run-time environment, the generated code, and the operating system sources are compiled and linked for the target platform.

The implementation details of our tight integration library targeted for Reflex will be presented in this chapter. First, we will focus on the output of the

[1]The code generator CAdvanced is used to translate SDL models into C code.

CAdvanced code generator to provide a deeper understanding of the preconditions and starting point for our work. The header and source files automatically generated from the SDL specification contain the interfaces for an integration with an operating system. We will then present how our design concepts for an efficient integration of the transformed SDL model have been adapted to Reflex, before stating our implementation results in the last section of this chapter.

5.1 Output of the CAdvanced code generator

The output of the C code generator from Telelogic is interspersed with precompiler directives, i.e. macros and `#ifdef` statements. The macros are used to make the generated code adaptable to different integration models and target operating systems. An adaptation, consequently, consists of a set of macro definitions. Depending on the extent of supported SDL language concepts, this set can become rather extensive, in the order of more than 100 macro definitions. By means of compiler switches that affect the `#ifdef` statements the integrator may optimize the size of the compiled object code. Unfortunately, the heavy use of the precompiler directives makes the generated code very difficult to comprehend.

5.1.1 Transformation of system structure

SDL systems consist of a hierarchically structured set of blocks and processes (cf. Fig. 5.1). The processes are at the lowest level in the hierarchy. Signals routes and channels connect processes, blocks, and the environment. The SDL system must also contain a definition of the signals used in the model and their parameters.

The code generator creates a set of header and source files that reflect the structure of the SDL system. Most importantly, these files include:

- Static objects that collectively represent the hierarchical structure of the SDL system. This so-called *symbol table* is organized as a tree. The symbol table is useful, for instance, to determine the receiver process for an output signal if this receiver is not explicitly stated. For an efficient implementation that uses explicit addressing of signals the system structure is irrelevant since the entire behavioral description is enclosed in the processes. Therefore, it is possible to get rid of most of these static objects by means of precompiler directives.

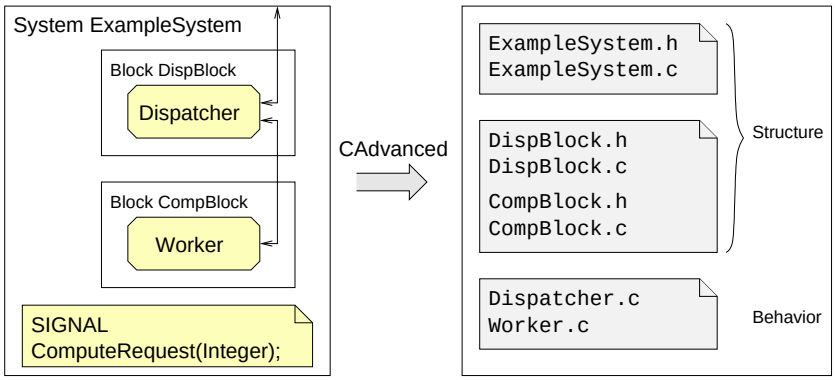


Figure 5.1: Hierarchical structure of SDL systems and corresponding output files from the CAdvanced code generator.

- *Signal type definitions.* For each signal type introduced in the SDL model, a data structure that encapsulates the signal parameters as well as control information needed by the run-time system to manage the exchange of signals is generated. An example for a signal type definition with an integer parameter is shown below. The macro `SIGNAL_VARS` comprises all run-time system-specific elements and must be defined by the integration library.

```
typedef struct {
    SIGNAL_VARS
    SDL_Integer  Param1;
} yPDef_z6_ComputeRequest;
```

- *Process type definitions.* Information related to processes is split into three categories in the generated code: common data for all instances of a process type which is encapsulated in a process node in the symbol table, instance-specific data, and dynamic behavior. The latter is dealt with in more detail in the following Sect. 5.1.2. As an example of instance-specific data, the lines below show the generated code for a process `Worker` with a single variable declaration (`number`) of type `Integer` and a timer `DummyTimer`. The macro `PROCESS_VARS` can be defined in different ways by the integration library. It must, however, define certain fields such as the `RestartAddress`, which identifies the transition of the state machine to be executed next.

```
typedef struct {
    PROCESS_VARS
```

```
    SDL_Integer z111_number;
    DEF_TIMER_VAR(yTim_DummyTimer)
} yVDef_z11_Worker;
```

- *Initialization routines.* The SDL system is initialized in a hierarchical manner. The system initialization function `yInit()` calls corresponding functions for all blocks, from where finally the process initialization functions are called. In order to create and set up a new instance of an SDL process, the data structures representing the process instance must be created and initialized. A startup signal is created and sent to every process. Since the initialization routines depend on the chosen integration approach, the function bodies in the generated code contain numerous macro calls. An appropriate definition of these macros has been conducted for our tight integration library.

5.1.2 Transformation of process state machines

The CAdvanced code generator applies the server model (cf. Sect. 2.1.4) for the implementation of process state machines. The state machine behavior is encapsulated in the so-called *process activity definition (PAD)* function. Whenever the run-time system brings a process to execution, the PAD function for this process type is called. As a parameter for this function, the instance-specific data structure for the activated process instance is passed. This way, the PAD function has access to all local variables of the process, its current state, and the input signal.

The body of the PAD function is a large `switch`-statement. Depending on the `RestartAddress`—a number that is determined by table lookup before calling the PAD function—the corresponding transition is executed. In the following, we will present the macros inserted by the code generator for the most important SDL concepts that may be present in a transition. For this purpose, in Fig. 5.3, we list and comment the code generator output for the (meaningless) SDL process in Fig. 5.2.

The source code shows the `switch`-statement of the PAD function. Because of the heavy use of macros and cryptic identifiers this code is hardly human-readable. We refrain from giving an explanation for each line and provide an overview of the most important macros in Table 5.1 instead. We refer to the Telelogic TAU User Manual [Tel06] for a comprehensive list of all relevant macros in the generated code. It gives also an overview on the data structures that represent the SDL model.

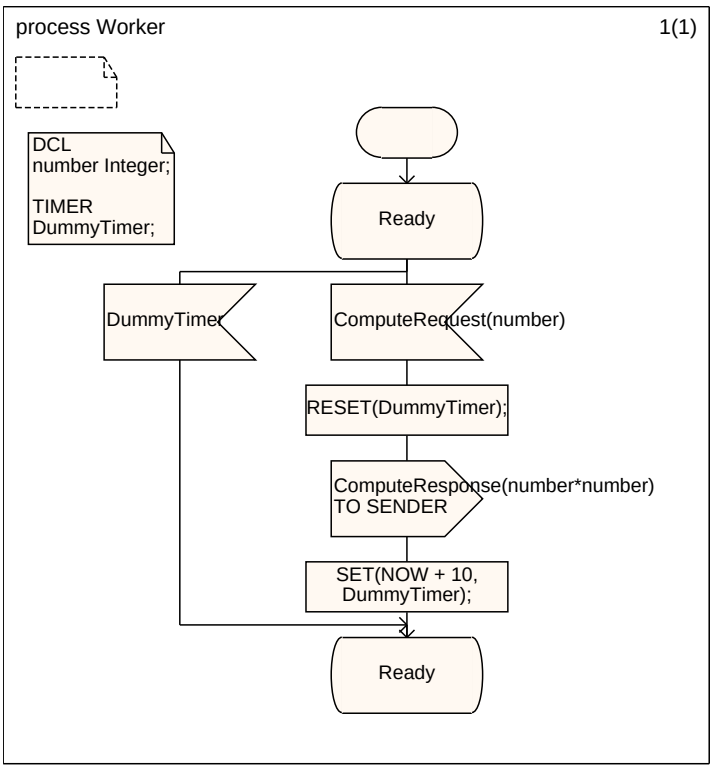


Figure 5.2: SDL process state machine with signal input and output as well as timer functions.

5.2 Tight integration model for Reflex

We elaborate on our software architecture for a tight integration model specifically targeted to the Reflex operating system in this section. It is based on the concepts that have been elaborated specifically for severely resource-constrained embedded systems, as discussed in the previous chapter.

It is essential to introduce the main concepts behind the Reflex OS, first, in order to understand the rationale for our design. A first version of our tight integration library was presented at the European Wireless Sensor Networks (EWSN) workshop in 2006 [WDEK06].

```
YPAD_FUNCTION(yPAD_z11_Worker)
{
  ...
  switch (yVarP->RestartAddress) {
  case 0: /* START */
    BEGIN_START_TRANSITION(yPDef_z11_Worker)
    INIT_TIMER_VAR(yVarP->yTim_DummyTimer)

  case 3: /* NEXTSTATE Ready */
    SDL_NEXTSTATE(Ready, z111_Ready, "Ready")

  case 1: /* INPUT DummyTimer */
    INPUT_TIMER_VAR(yVarP->yTim_DummyTimer)
  L_grst1:;

  case 4: /* NEXTSTATE Ready */
    SDL_NEXTSTATE(Ready, z111_Ready, "Ready")

  case 2: /* INPUT ComputeRequest */
    yAssF_SDL_Integer(yVarP->z113_number, ((yPDef_z6_ComputeRequest *)ySVarP)->
      Param1, XASS_AR_ASS_FR);

  case 5: /* RESET DummyTimer */
    SDL_RESET(DummyTimer, ySigN_z112_DummyTimer, yVarP->yTim_DummyTimer,
      "DummyTimer")

  case 6: /* OUTPUT ComputeResponse */
    ALLOC_SIGNAL_PAR(ComputeResponse, ySigN_z7_ComputeResponse,
      SDL_SENDER, yPDef_z7_ComputeResponse)
    SIGNAL_ALLOC_ERROR
    yAssF_SDL_Integer(((yPDef_z7_ComputeResponse *)OUTSIGNAL_DATA_PTR)->Param1,
      xMult_SDL_Integer(yVarP->z113_number, yVarP->z113_number), XASS_MR_ASS_NF);
    SDL_2OUTPUT(xDefaultPrioSignal, (xIdNode *)0, ComputeResponse,
      ySigN_z7_ComputeResponse, SDL_SENDER, sizeof(yPDef_z7_ComputeResponse),
      "ComputeResponse")

  case 7: /* SET DummyTimer */
    SDL_SET_DUR(xPlus_SDL_Time(SDL_NOW, SDL_DURATION_LIT(10.0, 10, 0)),
      SDL_DURATION_LIT(10.0, 10, 0), DummyTimer, ySigN_z112_DummyTimer,
      yVarP->yTim_DummyTimer, "DummyTimer")
    goto L_grst1; /* JOIN grst1 */
  }
}
```

Figure 5.3: Extracts of the C code generated by CAdvanced for the SDL process shown in Fig. 5.2.

5.2.1 The operating system Reflex

Reflex [Nol09] is an event-driven operating system designed at BTU Cottbus. It specifically targets embedded systems and has been ported to a number of commonly used 8-bit and 16-bit microcontrollers, as well as the 32-bit LEON2 microprocessor.

Table 5.1: Explanation of the most common macros found in the generated PAD functions.

Macro name	**Usage**
`SDL_NEXTSTATE`	The processes reaches a new state and ends the current transition (see `case 3` and `case 4` in the previous code example).
`ALLOC_SIGNAL_PAR`	Allocation of a signal buffer with parameters (see `case 6`).
`ALLOC_SIGNAL`	Allocation of a signal buffer without parameters.
`OUTSIGNAL_DATA_PTR`	Pointer to the allocated output signal (see `case 6`).
`SDL_2OUTPUT`	Output of a signal to a process with explicit addressing (see `case 6`).
`SDL_SET_DUR`	Start of a timer with a given duration (see `case 7`). The expiration of timers is treated like the reception of a signal.
`SDL_RESET`	Timer is stopped (see `case 5`).

The kind of systems Reflex was designed for do not require the rich set of services provided by general-purpose operating systems. Rather, it is tailored for the development of embedded real-time control applications.

For this purpose, Reflex offers different types of schedulers, for instance a FIFO scheduler and an earliest-deadline-first scheduler [WN07]. The latter facilitates preemption, that is the operating system interrupts the currently running task as soon as a task with a shorter deadline becomes ready for execution.

All schedulers implemented by Reflex have in common that only a single stack is required for all running tasks. This is major advantage for systems with limited memory resources.

The programming model of Reflex applications is based on the event-flow principle. Applications are composed of a number of functional components—the so-called *activities*—with input and output references. Communication between the activities is conceptually asynchronous. An activity assigns data or raises an event on its output which is connected to an input object of another activity. This input object buffers the received data item or event and marks the associated activity

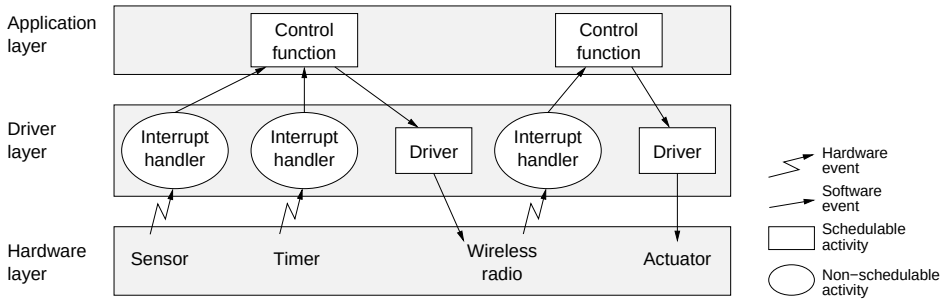


Figure 5.4: Simplified representation of a wireless sensor node application illustrating the event-flow model of Reflex.

as being ready for execution. The activity is scheduled according to the selected scheduling scheme and subsequently processes the event or data. An event flow, in any case, originates from an interrupt handler routine. The structure of an application composed of application-layer activities, interrupt handlers, device drivers, and the event flow between these components is shown in Fig. 5.4.

Reflex provides a library of object classes for *trigger variables* [WN06]. These are commonly used input objects, such as FIFO queues or single-value data buffers, that simplify the construction of event-driven applications. Activities, in Reflex, are represented by object instances of classes that are derived from the `Activity` base class. This class has a member function `run()` which is called by the scheduler to execute the corresponding activity.

A class diagram showing inheritance and typical interaction relationships between activities and trigger variables is shown in Fig. 5.5. Two activities, `Console` and `Serial` communicate with each other by means of a FIFO queue storing buffers to be transmitted by the serial driver. A `tx_done` event is raised by the `Serial` object each time a buffer was transmitted. Writing to the FIFO object triggers the `Serial` activity, the `tx_done` event activates the `Console` object.

A classical dynamic memory management with a heap is not part of Reflex. Alternatively, pre-allocated buffers managed in pools have to be used. Consequently, the operating system requires only very limited program and data memory. Typically, a Reflex executable, i.e. the application linked with the operating system, consumes a few kbytes of ROM and RAM.

With the exception of some low-level routines, for instance the boot sequence, Reflex is completely programmed in C++ following a clear object-oriented design.

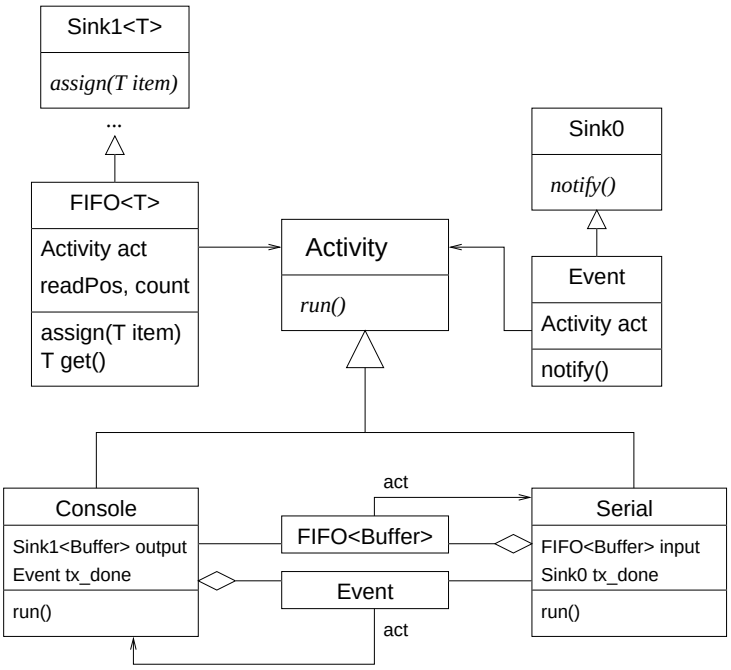


Figure 5.5: Class diagram showing the inheritance and interaction relationships between activity classes and trigger variables in Reflex.

This makes it easily portable to new platforms and processors.

5.2.2 Mapping of SDL processes

In our tight integration approach, each SDL process instance is represented by an instance of the `SDLProcess` class, which is derived from the `Activity` class. This `SDLProcess` instance acts as a wrapper for the process state machine contained in the PAD function. This way, the SDL process can be scheduled by the operating system.

As outlined above in Sect. 5.1, during system initialization static variables are declared for each SDL process. We have defined the macro `SDL_STATIC_CREATE`, which is inserted by the code generator, in such a way that not only a static variable for the instance-specific data structure (local variables etc.), but also a wrapper object for this process is created. This wrapper object receives a reference to the data structure of the process it represents (cf. Fig. 5.6).

The `SDLProcess` wrapper class also manages the input signal queue of the

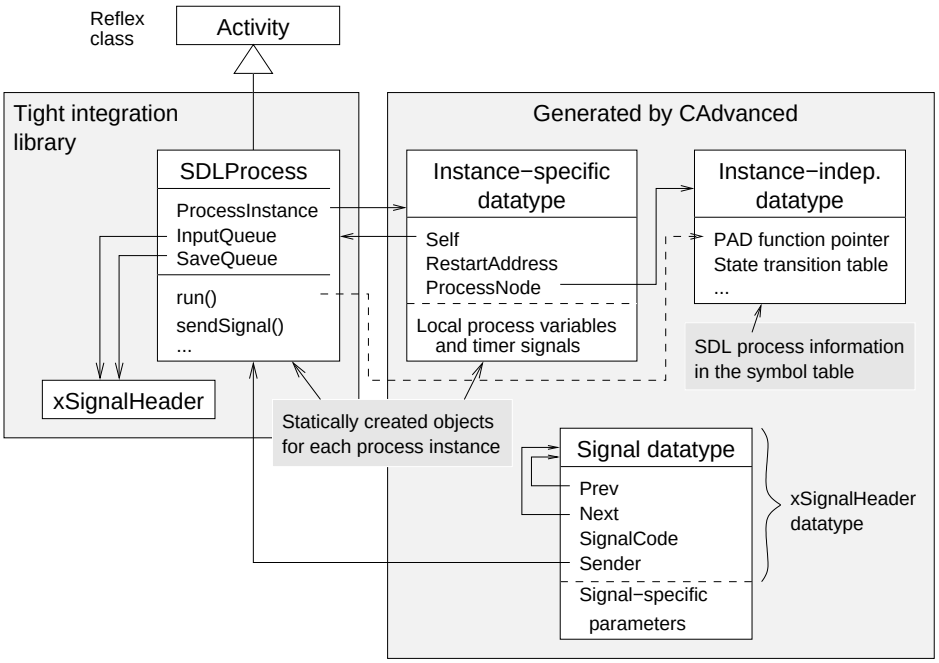


Figure 5.6: Diagram depicting the relationship between the process wrapper class `SDLProcess` and the objects created by the code generator CAdvanced.

process and its SAVE queue[2]. Other SDL processes, the timer process, or the environment send signals directly to the wrapper object by calling its `sendSignal()` function. By writing a signal into the input queue the wrapper object is activated, i.e. the scheduler will call its `run()` function when it schedules this activity. Since the signal queue may be accessed concurrently by multiple senders and the consumer, the integrity of its data structures must be protected by disabling interrupts during the access.

When the scheduler calls the `run()` function, the first signal from the input queue is consumed. At first, it must be determined, what action should be performed with the input signal—it must be either discarded, saved or it triggers a transition. In the latter case, the PAD function is called. The actions associated with an input signal for all the states of the process are contained in tables generated by the code generator, as well as the number of the transition that is triggered by the signal. The pseudocode implementation of the `run()` function is shown in

[2]The SAVE symbol is used in SDL models to indicate that a signal of the specified type should remain in the input queue.

```
function SDLProcess::run()
    consume first signal from input queue, this becomes
      the current signal to be processed
    loop while there is a signal to process
        find the input action for the current signal
          (and set the next transition number in case the action is INPUT)
        if DISCARD:
            deallocate memory for the current signal
            if the signal was from SAVE queue
                current signal = next signal in SAVE queue
            end if
        if SAVE:
            if the signal was in SAVE queue
                current signal = next signal in SAVE queue
            otherwise
                add signal to the end of the SAVE queue
            end if
        if INPUT:
            if current signal is from SAVE queue
                remove signal from SAVE queue
            end if

            set input signal of process instance variable to current signal
            call PAD function of associated process instance
            deallocate memory for the current signal

            current signal = first signal in SAVE queue
        end if
    end loop
end function run()
```

Figure 5.7: Pseudocode representation of the `run()` function of the `SDLProcess` class.

Fig. 5.7.

It should be noted that the scheduler manages a counter in each `Activity` instance that indicates how often the activity was triggered and calls the `run()` function as many times. This means that each time a signal was written into the input queue, the process is triggered and, eventually, `run()` will be called. Hence, there is no need to consume more than one input signal from the queue each time.

After calling the PAD function, the process state machine may have reached a new state. This means that the SAVE queue, if there are any signals, must be traversed to see if any of the saved signals will now trigger a transition or have to be discarded. The processing of signals from the SAVE queue must happen in FIFO order, just like signals from the input queue.

Finally, we address a practical problem concerning the explicit communication with the help of the `OUTPUT TO` statement. As we have outlined before, only direct communication is supported in our run-time environment to achieve high efficiency. This means that the process identifier, in our case: the wrapper object, has to be specified in the signal output, for instance:

```
// SDL PR
// Send signal "ComputeResponse.confirm" with parameter "100" to SDL
//   process identifier "Process1PId"
OUTPUT ComputeResponse.confirm(100) TO Process1PId;
```

The identifier `Process1PId` denotes the receiver process and must be declared in the SDL model. For our tight integration approach, this has to be done in the following form:

```
SYNONYM Process1PId PId = EXTERNAL;

/*#CODE
  #HEADING
    extern SDL_PId Process1PId;
    #define Process1PId Process1PId
  #BODY
    SDL_PId Process1PId;
*/
```

`PId` is an SDL data type representing a process identifier. The `EXTERNAL` keyword indicates that the constant `Process1PId` is defined externally. The lower part shows C code that will be placed by the CAdvanced in the generated header file for the SDL system (lines after `#HEADING`), and in a source file (lines after `#BODY`). The type `SDL_PId` is defined as a pointer to the `SDLProcessBase` class[3], which is a base class of `SDLProcess`, as will be covered later in Sect. 5.2.5.

During static process creation, the created wrapper object for the SDL process is assigned to the variable `Process1PId`. This is possible, because the name of the process is passed as a parameter to the `SDL_STATIC_CREATE` macro[4].

The CAdvanced code generator will use the macro `SDL_2OUTPUT` in the C code for the SDL output statement as follows:

```
SDL_2OUTPUT(xDefaultPrioSignal, (xIdNode *)0,
  ComputeResponseconfirm,  ySigN_z4_ComputeResponseconfirm,
  Process1PId, sizeof(yPDef_z4_ComputeResponseconfirm),
  "ComputeResponse.confirm")
```

The macro `SDL_2OUTPUT` is defined by us in such a way that the function `sendSignal()` will be called on the receiver process parameter, here: `Process1PId`, and a pointer to the signal (`ySigN_z4_ComputeResponseconfirm`) is passed as a parameter.

[3] In C++, this is expressed by: `typedef SDLProcessBase* SDL_PId;`

[4] The names of process identifiers are constructed by concatenating the given SDL process name, for instance `Process1`, and the suffix `PId`.

5.2.3 Memory management for signal buffers

As outlined in the previous chapter, signal buffer memory can be allocated on a heap or pre-allocated in pools. We do not recommend the first variant as it can lead to a situation where the system crashes because there is not enough free memory available on the heap. With pre-allocated signal pools for each signal type, the required maximum pool size can be statically determined.

In the case that pools are used for signal buffers, the exact number of pre-allocated elements and the signal types differ between system implementations. Therefore, it is only possible to provide a template for a signal buffer manager that must be adapted to the SDL system. The basic principles for the required signal pool data structures have been shown in Fig. 4.7 on page 104. Therefore, we restrict ourselves to a presentation of the C++ implementation of signal buffer manager functions, such as initialization of pools, allocation and deallocation of buffers.

The source code of a typical signal buffer manager implementation is shown in Fig. 5.8. The constant array with information about the sizes of all signal pools is defined in the first lines. In the constructor of the `SignalBufferManager` class, the dummy signal buffer, which is returned when a signal allocation fails due to unavailable memory, is initialized. The type of the dummy signal is a union of all signal types defined in the system such that the compiler automatically allocates the maximum required memory. The signal code 0 is not used by any signal and simply identifies that this signal shall never be sent to any process. The other member functions, `initPools`, `allocSignal`, and `freeSignal` are independent of the SDL model and the signal types defined by this model.

The signal type definitions are generated by CAdvanced from the SDL specification. The code generator also assigns consecutive signal numbers to each signal type, starting with 1. The automatic generation of pool declarations could be supported by a separate tool in a future development of our design methodology. Then, the designer would only have to assign the number of items in each signal pool.

Timer signals also require buffer space. As will be outlined in the next section 5.2.4, we allocate them statically as part of the process information for the SDL process where the timer is defined. Consequently, timer signals need not to be allocated dynamically.

```
// Constant array holding the buffer sizes and number of signal items for each type
const SignalBufferManager::PoolInfoType SignalBufferManager::pool_info[] = {
  SignalBufferManager::PoolInfoType(SigA_PoolSize, sizeof(SignalType_SigA)),
  SignalBufferManager::PoolInfoType(SigB_PoolSize, sizeof(SignalType_SigB)),
  ...
};

SignalBufferManager::SignalBufferManager() {  // constructor
    // SignalCode 0 identifies a signal that must not be output
    this->dummy_signal.SigA.SignalCode = 0;
}

void SignalBufferManager::initPools() {
    // "buffer memory" is contiguous chunk, will be structured into signal lists
    char *p = this->buffer_memory;
    for (unsigned int sig_id = 0; sig_id < NUM_SIGNALS; sig_id++)  {
        this->first_item[sig_id] = 0;

        // Initialize signal elements, construct linked list
        for (unsigned int item = 0;
          item < SignalBufferManager::pool_info[sig_id].num_items; item++) {
            ((xSignalHeader) p)->Suc = this->first_item[sig_id];
            ((xSignalHeader) p)->IsTimerSignal = 0;
            ((xSignalHeader) p)->SignalCode = sig_id + 1;
            this->first_item[sig_id] = ((xSignalHeader) p);
            p += SignalBufferManager::pool_info[sig_id].item_size;
        }
    }
}

xSignalHeader SignalBufferManager::allocSignal(int code) {
    xSignalHeader first;
    {
        InterruptLock lock(); // Protect concurrent access to "first"
        first = this->first_item[code - 1];
        if (first != 0)  {
            this->first_item[code - 1] = first->Suc;
            first->Suc = 0;
            first->Pre = 0;
        }
    }
    if (!first)  first = (xSignalHeader) &this->dummy_signal.SigA;
    return first;
}

void SignalBufferManager::freeSignal(int code, xSignalHeader signal) {
    if ((signal->SignalCode == STARTUPSIGNAL) ||
        (signal->IsTimerSignal))  return; // not in a signal pool

    InterruptLock lock();  // Protect concurrent access to "first"
    signal->Suc = this->first_item[code - 1];
    this->first_item[code - 1] = signal;
}
```

Figure 5.8: Implementation of the signal buffer manager functions in C++.

It should be noted that these functions are independent of the specific SDL system, and can be used also for other operating system integrations, as long as the same approach with pre-allocated signal pools is used.

5.2.4 Timer handling

We have implemented the concept for timer management described in Chapter 4 by creating a C++ class `SDLTimerProcess`, which is a schedulable Reflex activity. It is activated when the first timer in the timer queue expires. Its `run()` method called by the Reflex scheduler removes all expired timer signals from the queue and sends them to the corresponding processes.

Every Reflex application features a system clock that generates tick events in application-specific intervals. With such a clock module, it is straightforward to realize a concrete implementation and subclass of the abstract base class `TimerService`. The `SDLTimerProcess` instance requires such a `TimerService` instance (cf. Fig. 4.8 on page 106). As an example for a `TimerService` subclass and only possible solution on platforms with no available hardware timer module, we have implemented a `TickTimerService` class that makes use of the Reflex system clock.

Additionally, we developed a `TimerService` subclass called `HardwareTimerService` that takes advantage of the hardware timer module of the MSP430 microcontroller. This hardware timer uses a 16-bit counter and allows to generate interrupts at arbitrary points in time that can be defined by writing to a 16-bit capture-and-compare register. This makes it possible to avoid clock ticks in regular time intervals and generate expiration events by hardware interrupts. If the timer expiration interval is too long and cannot be expressed with 16 bits, the `HardwareTimerService` handles timer overflow interrupts and then checks again if the expiration time can be programmed into the 16-bit register.

This interrupt-driven implementation allows to set timers with a high precision, for instance 1 μs, and does not overload the system with clock tick processing at this rate (1 MHz in this example). The current time can be determined by reading the 16-bit counter register of the hardware timer and maintaining a counter for the number of overflows that have occurred. Remember that we use a 32-bit integer number to represent the system time. The approach is adaptable to other platforms with hardware timer modules, since they often provide similar features.

The classes implemented to provide the SDL timer management and their relationships are depicted in Fig. 5.9 in a UML-like notation. Only one of the two `TimerService` variants is needed by an application.

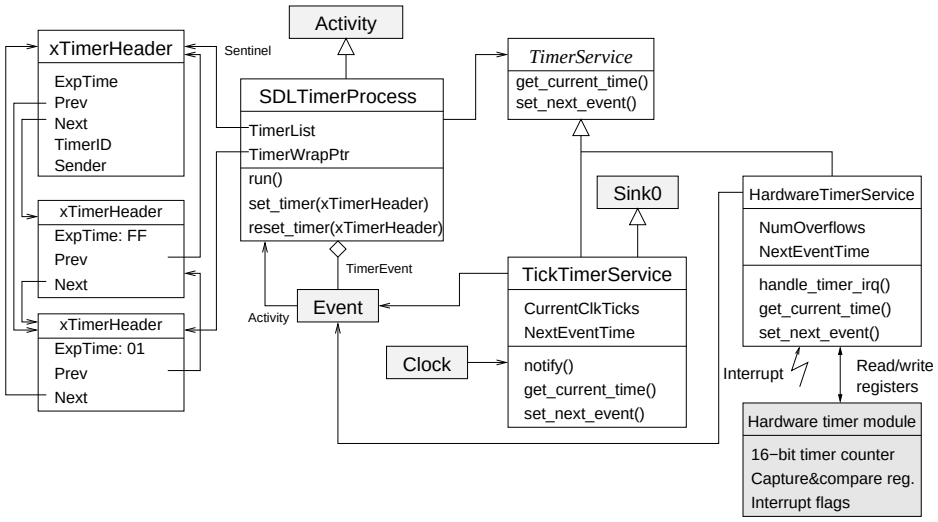


Figure 5.9: Software architecture around the `SDLTimerProcess` class.

For each timer defined in an SDL process, a timer signal instance is part of the instance-specific data structure representing the process. This is realized by the `DEF_TIMER_VAR` macro inserted by the code generator. Each timer signal has a unique identifier—the signal code—which is assigned by CAdvanced. Furthermore, the signal contains a pointer to the `SDLProcess` instance it belongs to, pointers to form double-linked lists (cf. 5.9), and a field that indicates the time when the timer expires.

5.2.5 Interfacing the environment

We have discussed two principle methods for interfacing the environment from an SDL system: by calling functions defined in an external library or by SDL signal exchange. Since the first method is already supported by a tool, we will focus only on the second approach, in the following.

In the SDL model, there is no difference between a signal output to the environment or to another, internal SDL process. It is possible to declare process identifiers for external processes, too. So, one can use explicit addressing in the signal output, i.e. the `OUTPUT TO` form, also for environment processes. Exactly the same mechanisms for signal exchange as presented above in Sect. 5.2.2 are applied. This means that the receiving object must provide a function `sendSignal()` to which the output signal is passed.

We have abstracted the common functionality of environment processes and the `SDLProcess` wrapper class for SDL processes in a base class called `SDLProcessBase`. This class provides the input queue and the signal handling functionality. It is subclassed from `Activity`. When a signal is written to the input queue by means of calling the `sendSignal()` function, this activity is scheduled. The `run()` function is a pure virtual method, i.e. concrete subclasses must provide an own implementation and define how to handle the input signals in the queue[5]. Furthermore, the `SDLProcess` class has additional fields required for acting as the wrapper for an SDL process that are not needed for an environment object.

The addresses of environment activities must be declared as external process identifiers in the SDL model such that signals can be sent to them. Vice versa, signals can be sent to any SDL process, that is its wrapper object, from outside the SDL system.

5.2.6 Putting it all together: the `SDLSystem` class

We have designed an `SDLSystem` class that provides access to the timer process and signal buffer management functions used by all SDL processes.

The `SDLTimerProcess` class implementing the timer management is universally applicable for all SDL system applications. It relies on an entity that allows to schedule a timer event and to get the current time. This class, derived from `TimerService`, has to be developed once for a specific target platform.

As discussed in Sect. 5.2.3 the memory management approach chosen by an application is not fixed. We recommend to utilize statically allocated pools, however also dynamic memory management based on a heap is possible. Therefore, the `SDLSystem` class has pure virtual functions for the allocation and deallocation of signal memory. Any concrete class must implement these functions, for instance relying on a signal buffer manager as shown in Fig. 4.7 on page 104.

The SDL process instances must have access to an instance of the system class derived from `SDLSystem`. We provide this access by defining a global variable of the system class in the application. For an application that is built from multiple SDL systems, different names for the system objects would have to be used. Within the system class, any number of environment processes could be defined. This is shown in the class diagram in Fig. 5.10 that also highlights which parts of an

[5]The `SDLProcess` class calls the PAD function of the associated SDL process from its implementation of the `run()` function.

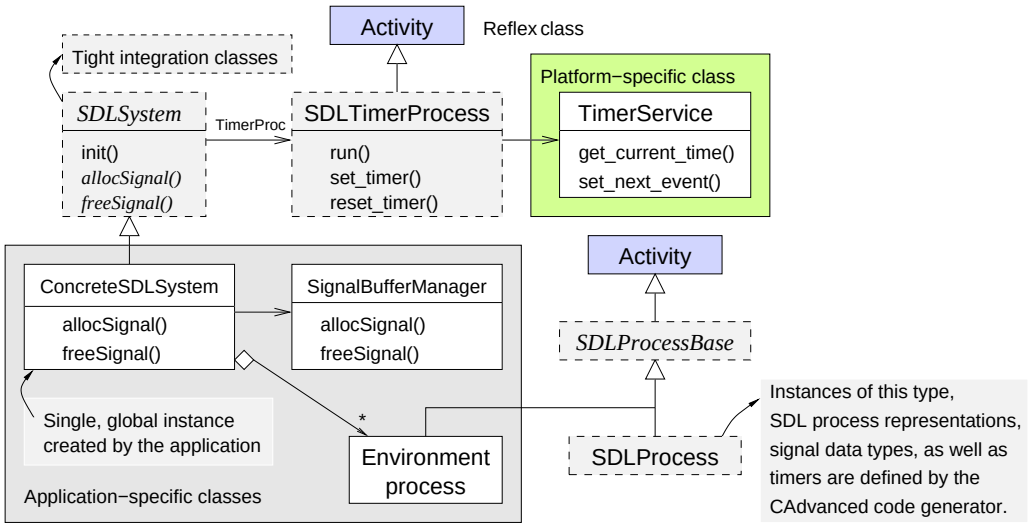


Figure 5.10: Classes required to build an application from an SDL model. Application- and platform-specific classes as well as those belonging to the Reflex operating system and to our tight integration library are distinguished.

application developed with our tight integration approach are created by the code generator and what has to be added manually.

For applications that use dynamic memory management all the sources can be taken from the tight integration library and the code generator output. There would be no application-specific classes except any environment processes. Otherwise, the signal buffer pools have to be provided based on the requirements of the application.

Finally, the initialization procedure of the SDL system shall be presented. The generated code contains a function `yInit()` which calls generated initialization routines of the SDL system structure in a hierarchical manner. The routines related to SDL processes initialize the process type instances in the symbol table and create static objects for each process instance, including the wrapper objects of type `SDLProcess`. When such an object is created, a so-called start-up signal is sent to the wrapper's own input queue, so that the activity is scheduled and the start transition can be triggered by this signal.

It is important that the signal buffer management is set up and all process wrapper objects are created before any of the start transitions is executed, because already within this transition signals might be sent to other processes. Therefore,

the application initializes the global `SDLSystem` object and calls the `yInit()` function before enabling the operating system scheduler.

5.3 Implementation results

Our tight integration model presented in the previous section has been fully implemented and adapted[6] to Reflex ports for the LEON2 processor and the MSP430 microcontroller.

LEON2 is a 32-bit general-purpose RISC processor conforming to the SPARC V8 instruction set [Gai05]. The tight integration library has been used for the software implementation of the IEEE 802.15.3 MAC protocol on this processor.

The MSP430 16-bit processor family [Tex06] from Texas Instruments is commonly used in ultra-low-power embedded systems and in particular on wireless sensor node platforms. Therefore, in this section, results for the tight integration model for this microcontroller will be presented. They have been obtained by targeting the TMote Sky wireless sensor node platform [Cor06] and conducting real measurements. The microcontroller variant on this platform is a MSP430F1611 with 48 kbytes program memory (Flash) and 10 kbytes RAM.

Memory consumption and execution speed are the most important performance figures to evaluate our approach. We designed simple SDL models to measure the required memory resources and processing speed of the generated code on the target platform. One set of SDL models consists of a number of processes that are arranged as a chain. The first process receives a signal (`Ping.request`) from the environment and, subsequently, sends a new signal of the same type to the process next in the chain as shown in Fig. 5.11. The receiver acts in the same way until the last process in the chain returns a `Ping.confirm` signal that is passed along the chain in reverse order. The signals are sent immediately without any processing delay after receiving the input signal that triggered the action. Finally, the `Ping.confirm` signal is sent to the environment.

Another SDL model designed for performance measurements is very similar to the previous model, but introduces signals that are stored in the save queue to reflect the behavior of typical SDL models. Our model consists of four processes that exchange `Ping.request` and `Ping.confirm` signals in the same manner as

[6] The interrupt-driven timer management is processor-specific and was adapted in order to take advantage of the available hardware timers.

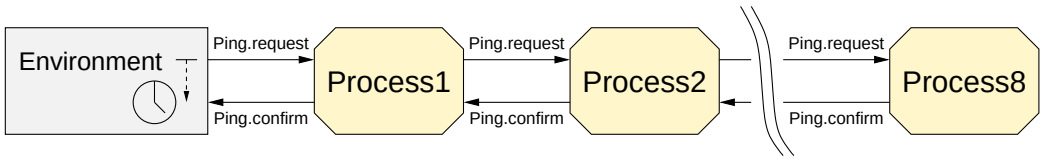


Figure 5.11: Simple SDL models composed of 2, 4, or 8 processes used for performance measurements.

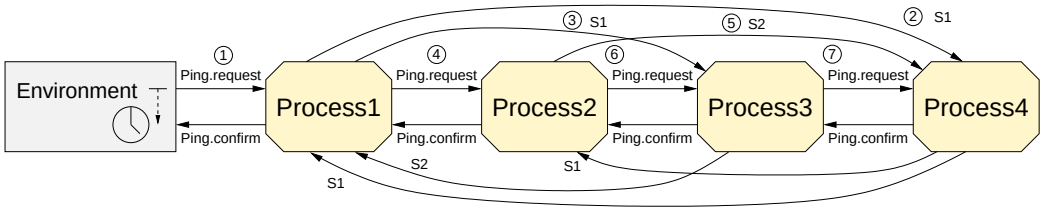


Figure 5.12: Performance measurements for an SDL model composed of 4 processes with signals `S1` and `S2` placed in the save queue.

described above. However, additional signals `S1` and `S2` are created and sent to processes further ahead in the chain where they are saved until the `Ping.request` or `Ping.confirm` signals are received. This is shown in Fig. 5.12.

To compare our results with a light integration approach, these models have been targeted to the TMote Sky platform using the SDL run-time environment from Telelogic. For a fair comparison, both approaches apply the same optimizations, for instance explicit addressing of signals, removal of code for unused data types, identical compiler optimization level, etc. Of course, the light integration model requires the standard C library for the dynamic memory management functions `malloc()` and `free()`, and the run-time environment supports all concepts of the SDL language. In both approaches a timer that generates ticks every 10 milliseconds is used, even though the SDL models have no timers declared. FIFO scheduling has been selected for Reflex in all cases.

Tables 5.2 and 5.3 show the memory consumption of four different applications: a chain of 2, 4, and 8 processes (Ping2, Ping4, and Ping8) and the model depicted in Fig. 5.12 (PingSave4) with the light and tight integration approaches, respectively.

The required memory space for the operating system Reflex, Telelogic's SDL run-time system or our tight integration library, the system environment func-

Table 5.2: Required memory space and processing speed of four SDL systems implemented with the light integration approach.

	Light integration approach			
	Ping2	Ping4	Ping8	PingSave4
Required memory space				
text	12128	12516	13284	12814
bss	358	578	1018	844
data	202	222	262	222
total	**12688**	**13316**	**14564**	**13880**
Execution time (5000 signals)				
ticks	548,364	1,212,656	2,450,632	2,291,606
seconds	4.28	9.47	19.1	17.9

Table 5.3: Required memory space and processing speed of four SDL systems implemented with the tight integration approach.

	Tight integration approach			
	Ping2	Ping4	Ping8	PingSave4
Required memory space				
text	7730	8174	9062	8576
bss	220	416	808	654
data	460	624	952	676
total	**8410**	**9214**	**10822**	**9906**
Execution time (5000 signals)				
ticks	371,371	851,502	1,729833	1,756,036
seconds	2.90	6.65	13.5	13.7

tions, library functions, and the code generated from the SDL model for the Ping2 application is listed in Table 5.4. Only the memory required for the SDL model varies between the different applications.

The read-only `text` segment contains all the executable code and read-only data, i.e. constants. The `data` and `bss` segments are used to store uninitialized data and data that will be initialized to zero, respectively.

The increase in the `text` segment size is caused by the PAD functions, i.e. state machine implementations, for the processes added to the model. The increased size of the `data` segments between the Ping2 models and the models with more

Table 5.4: Sizes of the text, data, and bss segments (in bytes) of the executable for the Ping2 application with the light and tight integration approaches, respectively.

Ping2 app.	**Light integration**			**Tight integration**		
	text	data	bss	text	data	bss
SDL model	824	238	152	888	190	442
Environment	286	0	12	320	0	6
Reflex	3370	8	4	3470	8	4
Run-time system / integration library	5562	110	34	2944	22	8
libgcc and libc	2086	2	0	128	0	0
Total (Ping2.elf)	**12128**	**358**	**202**	**7730**	**220**	**460**

processes can be explained by the additional arrays needed to store the state transition tables and the process information in the symbol table. This data cannot be removed from the generated code because it is essential for the proper operation of the process state machines. Finally, the `bss` segment grows because of the additional objects for the process-specific information and the process wrappers in the tight integration approach. The latter is the reason why the `bss` segment for the tight integration is larger. It should be noted that additional data memory is required for the stack and for the heap, in the case of the light integration approach. The memory required for the signal pools does not vary between the Ping2, Ping4, and Ping8 applications, because in all cases only two buffers for every signal type (`Ping.request` and `Ping.confirm`) are needed.

Due to the optimized run-time system, the re-use of the operating scheduler for the SDL process scheduling, and the absence of the libc library, the tight integration model saves more than 4 kbytes of program memory compared with the light integration approach. The results also show that the overall memory requirements for the applications are acceptable for a typical 16-bit microcontroller. However, the savings in the required memory space is not the only and primary advantage of our tight integration model, but rather that it enables preemptive scheduling, does not require dynamic memory allocation, and brings a performance increase as presented further below.

The toolchain MSPGCC for the MSP430, i.e. C/C++ compilers msp430-gcc and msp430-g++, and the linker msp430-ld, where used to create the executables. The compiler version 3.2.3 was used and the optimization level *O2* selected. The

Table 5.5: Performance results for the SDL model Ping2 obtained with different compiler optimization levels (*O2* and *Os*) for the mspgcc.

Ping2 app.	**Light integration approach**		**Tight integration approach**	
	Opt. level *O2*	Opt. level *Os*	Opt. level *O2*	Opt. level *Os*
Mem. breakdown				
text	12128 bytes	11830 bytes	7730 bytes	7636 bytes
data	358 bytes	358 bytes	220 bytes	220 bytes
bss	202 bytes	202 bytes	460 bytes	460 bytes
Sum	12688 bytes	12390 bytes	8410 bytes	8316 bytes
Execution time (1000 signals)	112227 ticks	114140 ticks	75926 ticks	74796 ticks

Table 5.6: Processing time with varying number of signals sent into the SDL model.

Ping2 app.	**Light integration approach**	**Tight integration approach**
Execution time		
1000 signals	112,227 ticks (0.88 s)	75,926 ticks (0.59 s)
5000 signals	548,364 ticks (4.28 s)	371,371 ticks (2.90 s)
10000 signals	1,096,376 ticks (8.56 s)	742,574 ticks (5.80 s)

optimization level *Os* gave better performance results for the tight integration approach and worse results for the light integration as presented in Table 5.5.

Furthermore, the time to process a signal sent by the environment into the SDL model has been measured. For this purpose, the environment generates a `Ping.request` signal, sends it to the first process in the chain, and waits for the reply (`Ping.confirm` signal). Upon reception of the `Ping.confirm` from the SDL system, the next `Ping.request` is sent immediately. The number of repetitions of this procedure has been varied. Table 5.6 reports the time it takes to process 1000, 5000, and 10000 signals by the Ping2 application.

The obtained figures are deterministic values, they have been measured with the help of the microcontroller's hardware timer module. The timer module's clock source is based on the processor's 4 MHz clock. The tick counter difference between the sending of the first signal and the reception of the last signal is communicated via a serial interface to a PC.

Table 5.6 clearly shows that the execution time grows linearly with the number of created signals, independent of the integration approach. The tight integration is

more than 30 percent faster than the light integration version. In Tables 5.2 and 5.3 the processing times for all four previously introduced systems with 5000 created `Ping.request` signals are given for the tight and light integration models. The applications using the tight integration approach outperform the light integration by 25–30 percent.

By extensive simulations and execution of the IEEE 802.15.3 MAC protocol implementation generated from an SDL model on real hardware for many hours without failures we could demonstrate that the tight integration library works correctly. A formal verification of our design and the Reflex operating system would be required to prove correctness and qualify applications developed using our SDL-based approach for safety-critical tasks.

Chapter 6

SDL Cosimulation with the TSIM Instruction Set Simulator

The feasibility of our concepts for a cosimulation of an abstract SDL simulation with an instruction set simulation has been demonstrated by a prototypical software implementation of all required components. This implementation is specific to the chosen ISS and also to the simulated SDL model, since functions for translating signal parameters from the host computer representation to the target system had to be developed.

We have selected the TSIM instruction set simulator for the LEON2 processor as an example. This was motivated by the fact that our IEEE 802.15.3 MAC protocol implementation was targeted for this processor and the cosimulation implementation could directly be used to support hardware/software partitioning of the protocol. For ultra-low-power microcontrollers, e.g. the Texas Instruments MSP430 or Atmel AVR, there are also instruction set simulators available. The basic principles for a cosimulation with, for instance, the MSPsim [EDF+07] or ATEMU [PBM+04] simulators are identical, even though some more effort has to be spent to implement the software framework for controlling the simulators.

In the remainder of this chapter, we will first give a brief introduction to TSIM and show how it can be controlled from the environment functions of the SDL simulator. Then, we present our software framework which implements all the cosimulation concepts that have been elaborated in Sect. 4.3. Hardware/software

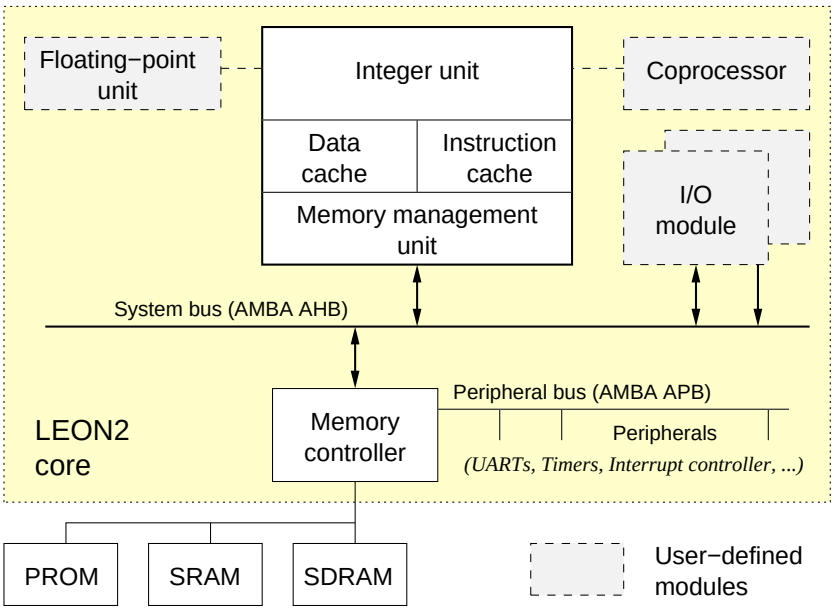


Figure 6.1: Architecture of the LEON2 processor. All the depicted components are simulated by TSIM.

partitioning results that have been created by applying the cosimulation to the IEEE 802.15.3 MAC protocol implementation are covered in Chapter 7.

6.1 The instruction set simulator TSIM

The TSIM simulator [Gai04] was developed by Gaisler Research AB as an instruction set simulator for the LEON2 processor, i.e. TSIM emulates the SPARC instruction set. Moreover, the complete LEON2 processor system, including data and instruction caches, memories, and peripheral devices, such as the interrupt controller, timers, and UARTs, are simulated by TSIM. It is possible to provide user-defined modules for a coprocessor, floating-point unit, and memory-mapped I/O devices. This way, system designers are able to analyze the performance and the behavior of LEON2-based designs consisting of hardware and software parts. The components that are part of the simulator are depicted in Fig. 6.1.

TSIM provides a number of features for system profiling and testing. So it is possible to display the number of cycles a program spent in different functions. For debugging purposes, breakpoints can be set, processor registers and memory

locations can be read or written, and code coverage can be recorded.

TSIM is a *cycle-true* simulator, which means that the simulation time is incremented according to the exact processor instruction timing and also takes into account memory latencies. As an optimization to achieve faster simulations, TSIM advances its simulation time to the next scheduled event in the event queue when it encounters the power-down instruction. This instruction is typically used in programs to save power when it is waiting for a timer interrupt or other external event and there is no current task for the processor.

The instruction set simulator is available as a stand-alone program and also as a library. The library version enables the integration into a larger simulation framework. Both modes of operation, stand-alone and library, are equivalent in terms of supported commands and features. When starting the simulator, the processor configuration can be fixed. This includes the selected clock frequency, cache configuration (sizes of data and instruction cache), or options for the arithmetic-logic unit (ALU).

The *TSIM library* provides a set of functions to control the simulation. The most important library functions are listed in Table 6.1 together with a short description of their purpose. In addition to these functions, the library exports two objects, `simif` and `ioif`, that allow, for instance, to read the current simulation time or to generate an interrupt. For a more comprehensive overview and detailed description of the interface to the TSIM library we refer to the manual [Gai04]. Under the Windows operating system, the TSIM library is provided as a dynamic link library (DLL), `tsimleon.dll`.

As already mentioned above, TSIM allows to extend the functionality of the LEON2 processor system by introducing *user-defined modules*. With the help of these modules it is possible to study the effects of designing a part of the system functionality in hardware. An example could be an extension of the instruction set which is realized by an application-specific coprocessor.

Alternatively, *hardware accelerators* could be introduced as *memory-mapped I/O modules*. In this case, an access by the processor to a memory location within the region of such an I/O module is interpreted by this module as an access to a control register or internal memory.

Depending on the addressed register, a user-defined operation is triggered. This could involve, for instance, direct memory access (DMA) to other memory locations or an interrupt request. The simulator's event queue can be used to schedule events

Table 6.1: Overview of the most important functions exported by the TSIM library.

Function	Purpose
`int tsim_init(char *option)`	Initialize TSIM with the desired configuration options (clock frequency, etc.)
`void tsim_exit(int val)`	Perform cleanup of the simulator
`void tsim_get_regs(int *regs)` `void tsim_set_regs(int *regs)`	Reading and writing processor registers
`void tsim_inc_time(uint64 t)`	Advance the simulator time without executing any instructions.
`int tsim_cmd(char *cmd)`	Execute a TSIM command, such as loading an application ("load app.elf"), starting a program ("go [address] [count/time]"), continue the execution for a certain amount of clock cycles or time ("cont [count/time]"), enable or disable profiling, etc. The command syntax is identical to the stand-alone mode of TSIM. The return value indicates the simulation status.

for a later time if the initiated operation does not finish immediately.

Address decoding is handled in a simple way by TSIM. Any access that does not belong to emulated memory or control registers is forwarded to I/O devices.

Any user-defined I/O module must provide a set of functions that are called by the simulator core. The most important ones of these are summarized in Table 6.2. The I/O module is compiled as a library and linked to TSIM. A structure with pointers to the module-specific functions in Table 6.2 must be exported by the I/O module DLL.

In summary, TSIM is an excellent tool to investigate design alternatives of hardware/software systems based on the LEON2 processor. The simulation is much faster than RTL simulations, and there is no need for the time-consuming design of hardware components in VHDL. In the following section, we will tackle the problem of integrating TSIM with an SDL simulation. Our goal is to perform hardware/software partitioning of the communication protocol implementation with the help of TSIM and to reuse the existing SDL-based test benches and communication network models to generate input stimuli for the hardware/software system under design.

Table 6.2: Relevant functions that have to be provided by user-defined I/O devices.

Function	**Purpose**
`void io_init()`	Called at simulator startup. Used to initialize the I/O device.
`void io_exit()`	Called when simulator exits.
`int io_read(unsigned int addr, int *data, int *ws)`	Read access to given address. The data value at that address as well as the number of wait states is passed to the simulator. The return value indicates the status of the access (success or memory error).
`int io_write(unsigned int addr, int *data, int *ws, int size)`	Write access to given address. The parameter `size` indicates how to interpret the `data` parameter (either as byte, half-word, word, or double-word value). The number of wait states and return value is identical to the read access.

6.2 Integrating TSIM with Telelogic's SDL simulator

The cosimulation framework consisting of TSIM and the SDL simulation is shown in Fig. 6.2. The TSIM library as well as an I/O module library are linked to the SDL simulator—a stand-alone application built from the SDL run-time environment for functional simulations and the SDL model of a communication network.

The four environment functions `xInitEnv`, `xCloseEnv`, `xInEnv`, and `xOutEnv` interact with the TSIM and I/O module libraries in such a way that the interleaved execution of the simulators and the signal exchange are realized. The principle operation was presented in Sect. 4.3 (cf. Fig. 4.14 on page 119) and shall not be repeated here.

For the simulated hardware/software system to be able to process signals sent from the SDL system, the signal parameters must be copied into the address space of the LEON2 processor. This can be achieved with the help of an I/O module. The I/O module DLL simply has to export functions which allow to access the memory space within the LEON2 system that is under control of this I/O module.

Some effort has to be spent to transform the signal parameters from one execution platform to the other. TSIM emulates the LEON2 processor, which conforms to the SPARC instruction set and, thus, uses big-endian byte ordering. This means that the most significant byte of a 32-bit data word has the lowest address. Intel

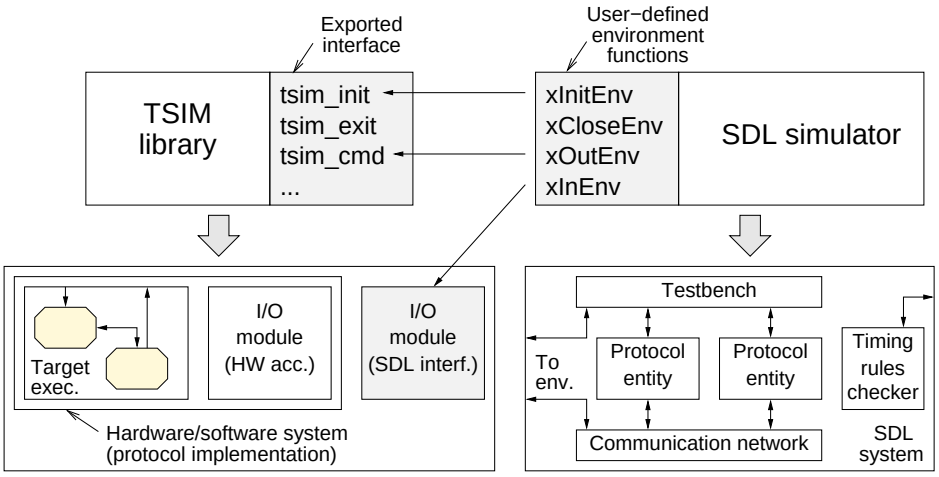


Figure 6.2: Applied scheme for coupling the SDL simulator with TSIM. The functions realizing the interface are shaded in gray.

x86 PCs, however, are little-endian machines. We have developed the cosimulation framework on x86 PCs, therefore it was necessary to convert SDL signals exchanged between the two simulators in order to preserve the meaning of the data, rather than simply copying the memory buffer occupied by the signal parameters.

In our approach, the conversion is performed within the environment functions `xInEnv` and `xOutEnv`. This way, the time required for the conversion does not affect the performance of the implementation model simulated by TSIM. We will go into the details of signal type conversion in the following section 6.3 on implementation aspects.

6.3 Implementation of the cosimulation framework

The exchange of SDL signals between the models simulated by TSIM and the SDL simulator is facilitated by an I/O module designed for this purpose. The implementation is specific for the TSIM simulator and must be adapted when targeting a different ISS. However, the signal queue handling can be easily reused as it is written in standard C. This I/O module must provide the functionality to read and write SDL signals from both, the implementation model and the SDL simulator. For the latter, library functions are exported and can be called from other applications.

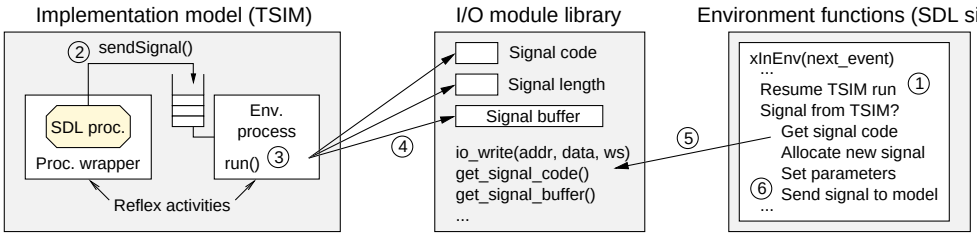


Figure 6.3: Chain of actions related to the output of an SDL signal from the implementation model simulated by TSIM to the external SDL system.

When an SDL signal shall be sent from the TSIM application to its environment, the signal parameters are copied to a certain memory buffer within the address range of the I/O module. Additionally, there is another address which is used to store the length of this signal data buffer, and an address for the signal code.

When the signal code is written into this designated address it has the effect that the I/O module immediately stops the simulation. Hence, the signal code must be written after all the other parameters have been copied. The signal identifier, the memory buffer for its parameters, as well as the length of this buffer can be read from the SDL simulator's environment function `xInEnv` by calling an exported function of the I/O module library. Subsequently, the parameters have to be correctly decoded from big-endian byte ordering to little-endian for the simulator platform.

Figure 6.3 shows the sequence of function calls related to the output of an SDL signal from the implementation model simulated by TSIM. At first, the TSIM simulation is resumed by calling the corresponding TSIM library function from the environment function `xInEnv`.

Any signal from the implementation model to the external SDL simulation is sent via the environment of the implementation model (step *(2)* in Fig. 6.3). The operating system scheduler eventually calls the `run` function of the environment process *(3)*. The signal is consumed from the input queue and written into the special memory region of the I/O module *(4)*. When the signal code has been written, the simulation is stopped by the I/O module library and the execution resumes in the `xInEnv` function from where TSIM was called. Here, the current simulation time of TSIM at the time when it stopped is retrieved (not shown

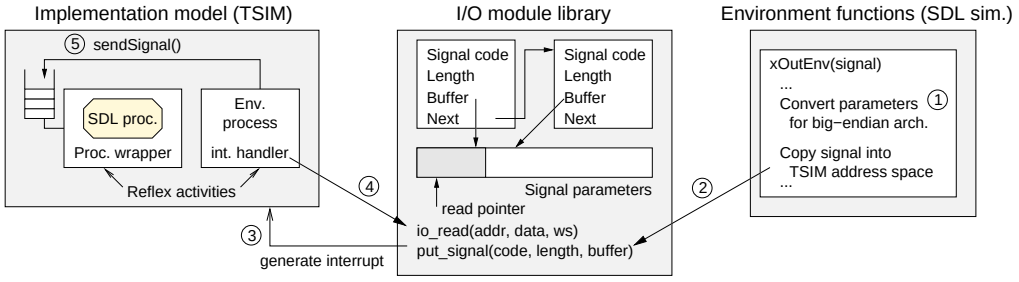


Figure 6.4: Interactions between the SDL simulator, I/O module, and TSIM related to the exchange of SDL signals from the Telelogic simulator to the implementation model.

in the figure) and the SDL simulation time is advanced accordingly. It is checked whether the simulation stopped because of a signal output *(5)*. If this is the case, a corresponding signal for the SDL simulation is allocated, its parameters are copied from the I/O module buffer, and, finally, the signal is output to the appropriate receiver process *(6)*. If a timing rules checker is utilized, the output signal is also sent to that process.

The I/O module provides also the mechanism to send SDL signals to the implementation model. Figure 6.4 illustrates the principle of operation and the entities involved in the signal exchange. Any signal that is sent from the SDL system running in the Telelogic simulator to the implementation model is passed to the environment function `xOutEnv`. As already outlined in the previous section, the signal parameters are converted from the PC platform with little-endian byte ordering to big-endian for the LEON2 processor (step *(1)*). The memory buffer containing the converted signal parameters, its length, as well as the signal code is passed to the I/O module by calling the exported library function `put_signal` *(2)*. The implementation model shall process these signals as soon as the TSIM simulation is resumed. Therefore, the `put_signal` function requests an interrupt of the LEON2 processor *(3)*. The environment process contains an interrupt service routine for exactly this interrupt.

More than one signal may be passed to the `xOutEnv` function before the simulation of TSIM is resumed by the `xInEnv` function. Therefore, all these signals must be kept in the I/O module until they are retrieved by the implementation model. The order in which the signals are received by the I/O model is preserved

by storing them in a linked list as indicated in Fig. 6.4. In addition to copying the signals to the I/O module, another copy could optionally be sent to a timing rules monitor process.

When there are no more executable transitions in the SDL simulator, `xInEnv` is called and, in this course, TSIM is resumed. Control is passed to the interrupt handler in the environment process. This function performs read operations on the memory region belonging to the I/O module in order to obtain all the relevant information about the stored input signals *(4)*. New SDL signals are allocated and initialized with the retrieved parameters. Finally, depending on the signal code, the signals are put into the input queues of the intended SDL processes *(5)*, that is of their process wrapper objects to be exact.

The order in which the signals are sent to the processes is the same order in which they have been output by the abstract model, because the signal queue in the I/O module operates as a FIFO buffer without exception. Hence, the SDL semantics are preserved in this respect.

Figure 6.5 summarizes the implementation of the four environment functions in a pseudocode representation. It has been presented in one of our conference papers [Die08]. In Fig. 6.6, a part of the I/O module library source code is listed. The queue for the signals from the SDL simulation is realized as a linked list with sentinel nodes `first_elem` and `last_elem` marking the beginning and the end of the list. The `next` pointer of the (dummy) end element indicates the last (real) signal in the queue. Two memory buffers, `sigin` and `sigout` are used to store the signal parameters. The function `get_signal_done` is called from `xInEnv` after running TSIM, in order to reinitialize the signal buffers.

The interleaving of the simulation runs is illustrated with a simple example in Fig. 6.7. This sequence chart shows the interactions of the SDL simulator, TSIM, and the I/O module. The time axis is drawn vertically with time increasing in downward direction.

At the beginning, `xInitEnv` is called by the SDL simulator. The TSIM simulation is initialized and the target application is loaded from this function. After that, the SDL simulation runs for the first time and executes all transitions at time 0. In this example, a timer is set at time 5 seconds. Hence, `xInEnv` is called with this time as the next timer event. At this point, TSIM is allowed to run for the first time, in fact for a duration of 5 seconds.

When the TSIM simulation returns, it is first checked what is the current

```
xInitEnv():
    Initialize TSIM
    Load application into TSIM
-------------------------------------------------------------
xCloseEnv():
    Exit TSIM
-------------------------------------------------------------
xInEnv(Time_For_Next_Event):
    Continue TSIM until Time_For_Next_Event
    If there is a signal from TSIM application
        Copy signal from I/O module into new SDL signal SignalIn
        Send SignalIn into SDL model
        Optionally: send a copy of SignalIn to timing rules monitor
        SDL system time = Get current TSIM time
    Else
        SDL system time = Time_For_Next_Event
    End if
-------------------------------------------------------------
xOutEnv(SignalOut):
    Optionally: send a copy of SignalOut to timing rules monitor
    Convert SignalOut parameters into big-endian representation
    Put a copy of SignalOut into signal queue in I/O module
```

Figure 6.5: Pseudocode implementation of the environment functions.

TSIM simulation time and whether a signal was sent from the implementation model. In our example, no signal was output and TSIM ran for the complete 5 seconds. This means that the SDL simulation resumes at time 5 and the timer is triggered. During this simulation, one SDL signal is sent to TSIM, namely `SigA`. This will cause an interrupt to the LEON2 processor such that the execution of the implementation model will continue with the retrieval of `SigA`.

Before TSIM is allowed to resume, all SDL transitions at time 5 are processed and a timer set to 10 seconds. In the example, the implementation models sends a signal (`SigB`) at time 6 seconds to the SDL simulation. TSIM stops immediately after this action. As described above, the current TSIM simulation time and the signal is retrieved by the environment function. Consequently, the simulation resumes at time 6 seconds.

A screenshot from a real cosimulation run of the same example as in Fig. 6.7 is displayed in Fig. 6.8. The implementation model generates `SigB` not exactly at time 6 seconds because it uses timer ticks with an interval of 10 milliseconds.

Cosimulation performance We have conducted measurements of the real simulation time for a wireless communication network modeled in SDL. Our model of the IEEE 802.15.3 MAC protocol, that will be introduced in the next chap-

```
// SIGOUT: signal into SDL simulation
#define SIGOUT_CONTROL_REG   0x20000010
#define SIGOUT_DATA_REG      0x20000014

// SIGIN: signal from SDL simulation
#define SIGIN_CONTROL_REG 0x20000020
#define SIGIN_DATA_REG    0x20000024
...
typedef struct sig_info_t
{
  unsigned short signal_code;
  unsigned short signal_length;
  unsigned int *buffer;
  struct sig_info_t *next;
} sig_info_t;

static sig_info_t first_elem, last_elem;

static void io_init(sim_interface sif,
                    io_interface iif) {
  sigout_control = 0;
  sigout = (char *) malloc(SIGOUT_SIZE);
  sigin = (char *) malloc(SIGIN_SIZE);

  first_elem.next = &last_elem;
      // pointer to last element in queue
  last_elem.next = &first_elem;

  sigout_write_ptr = (int*) sigout;

  sigin_write_ptr = sigin;
  sigin_read_ptr = (int*) sigin;
}

static int io_read(unsigned int address,
                   int *data, int *ws) {
  sig_info_t *p;
  ...
  if (address == SIGIN_CONTROL_REG)  {
    p = first_elem.next;
    if (p != &last_elem) {
      *data = (p->signal_length << 16) +
               p->signal_code;
      sigin_read_ptr = p->buffer;
      first_elem.next = p->next;
      if (p == last_elem.next)
        last_elem.next = &first_elem;
      free(p);              }
    else  *data = 0;
    return 0;                         }

  if (address == SIGIN_DATA_REG)     {
    *data = *sigin_read_ptr;
    sigin_read_ptr++;
    return 0;                         }

  return 1;
}

static int io_write(unsigned int address,
             int *data, int *ws, int size) {
  if (address == SIGOUT_DATA_REG)    {
    *sigout_write_ptr = *data;
    sigout_write_ptr++;
    return 0;                         }

  if (address == SIGOUT_CONTROL_REG) {
    sigout_control = *data;
    simif.sim_stop();
    return 0;                         }

  return 1;
}

static void put_signal(unsigned short code,
  unsigned short len, char* sig_buf) {
  sig_info_t *sig_info = (sig_info_t *)
     malloc(sizeof(sig_info_t));
  sig_info->signal_code = code;
  sig_info->signal_length = len;
  sig_info->buffer = (int*) sigin_write_ptr;
  sig_info->next = &last_elem;

  last_elem.next->next = sig_info;
  last_elem.next = sig_info;

  memcpy(sigin_write_ptr, sig_buf, len);
  sigin_write_ptr += len;

  ioif.set_irq(13, 0);
}

static unsigned short get_signal_code()    {
  return(sigout_control & 0xFFFF);         }

static unsigned short get_signal_length()  {
  return(sigout_control >> 16);            }

static void get_signal(char* sig_buf,
                       unsigned short len) {
  memcpy(sig_buf, sigout, len);            }

static void get_signal_done()              {
  sigout_control = 0;
  sigout_write_ptr = (int*) sigout;

  sigin_write_ptr = sigin;
  sigin_read_ptr = (int*) sigin;           }
```

Figure 6.6: Source code extracts for the I/O module library.

ter, served as the basis for the simulations. We instantiated four IEEE 802.15.3-compliant device entities, each consisting of a simplified physical layer and our MAC protocol model, and connected them with an airlink model. An SDL test bench was used to start one of the devices as a network coordinator, while the

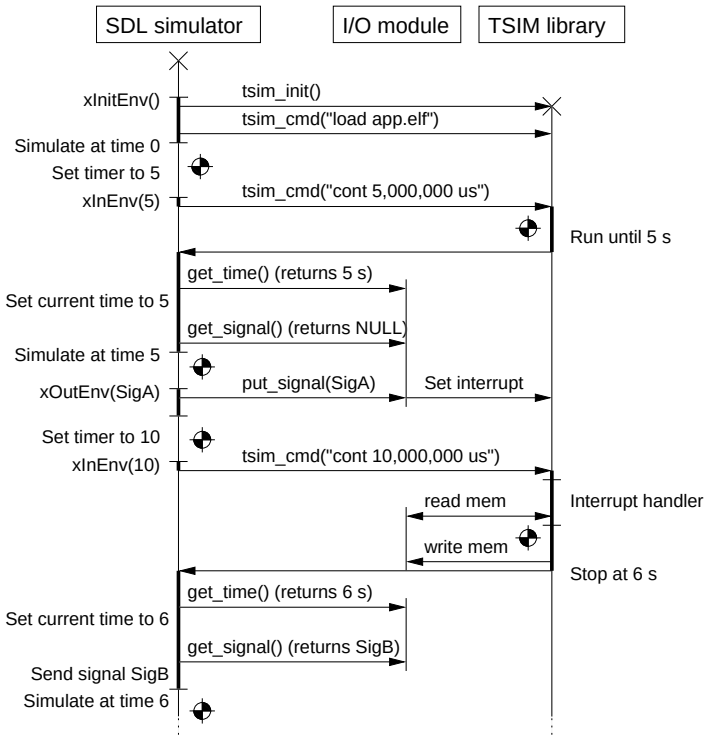


Figure 6.7: Interactions between the environment functions of the SDL simulator, the TSIM library, and the I/O module. The example shows the exchange of signals in both directions.

other three devices associated themselves with the coordinator. We conducted an abstract simulation, that is without using real-time. The real simulation time for a network simulation of 5, 10, and 20 seconds was measured.

In order to study the performance of the cosimulation framework, we extended the experimental set up by a fifth device simulated by the instruction set simulator TSIM. An implementation model of the same MAC protocol model was created by generating C code from the SDL model and compiling it for the LEON2 processor. The physical layer of the network model was extended in such a way that it allowed to receive and send frames from and to the TSIM model.

After performing the network creation and association procedures, one of the devices was ordered to periodically send an 8-byte asynchronous data packet. The interval between two data requests was set to 100 ms. The other devices were idle, which means that they only received and processed the beacon (broadcast every

```
No connection with the Postmaster.
Running stand-alone.
xInitEnv called
Initialising TSIM...

 TSIM/LEON SPARC simulator, version 1.3.3
 (professional version)

 Copyright (C) 2001, Gaisler Research
 ...
Loading application...
section: .rom_vectors, addr: 0x40000000,
  size 4104 bytes
section: .text, addr: 0x40001008,
  size 14696 bytes
section: .data, addr: 0x40004970,
  size 300 bytes
read 143 symbols
Starting application...
resuming at 0x40000000
 Current time 0 (0.000e+000 us)

Welcome to the SDL SIMULATOR.

Command : Proceed-Until 12

*** TRANSITION START
*      PId    : SenderProcess:1
*      Now    : 0.0000
*   SET on timer T at 5.0000
*** NEXTSTATE  Idle

xInEnv(5,0) called.
Executing TSIM command: cont 5000000 us
Advancing until time: 5.000000125 s

*** TIMER signal was sent
*   Timer    : T
*   Receiver : SenderProcess:1
*** Now      : 5.0000

*** TRANSITION START
*      PId    : SenderProcess:1
*      State  : Idle
*      Input  : T
*      Now    : 5.0000
*   OUTPUT of SigA to env:1
*      Parameter(s) : 0
xOutEnv:  SigA has been received by env
*   SET on timer T at 10.0000
*** NEXTSTATE  Idle

xInEnv(10,125) called.
Executing TSIM command: cont 5000000 us
io_write at 0x20000014, data=0x00000000
io_write at 0x20000010, data=0x00040002
Received signal: 2 (SigB)
*   OUTPUT of SigB to SenderProcess:1
*      Parameter(s) : 0
Advancing until time: 6.016702201 s

*** TRANSITION START
*      PId    : SenderProcess:1
*      State  : Idle
*      Input  : SigB
*      Sender : env:1
*      Now    : 6.0167
*      Parameter(s) : 0
*** NEXTSTATE  Idle

xInEnv(10,125) called.
Executing TSIM command: cont 3983298 us
Advancing until time: 10.000000325 s

*** TIMER signal was sent
*   Timer    : T
*   Receiver : SenderProcess:1
*** Now      : 10.0000
...
Command :
```

Figure 6.8: Screenshot from the SDL simulation of the example in Fig. 6.7.

50 ms) and data frames transmitted in the network. In the configuration with the connected instruction set simulator, the implementation model within TSIM was ordered to send these 8-byte data packets in the same intervals as before, while all other stations, this time, remained idle. This way, the overall network traffic could be kept on the same level as in the first set up.

We counted roughly 3000 transitions and 400 timer expirations per device within one second of simulation time, on average. In Table 6.3, the measurement results of the real simulation time for both scenarios are summarized. They have been obtained on a Windows PC with a Pentium 4 processor clocked at 2.53 GHz and 512 MB RAM. During the simulation, there was no polling of user input and no output of any trace information. As expected, the consumed real time grows

Table 6.3: Comparison of the real simulation times for wireless network simulations with and without an external instruction set simulator.

Simulation time	Pure SDL simulation of 4 devices	Cosimulation with additional implemenation model in TSIM	Number of signals exchanged between SDL and TSIM
5 s	6.0 s	25.3 s	TSIM → SDL: 116 SDL → TSIM: 992
10 s	11.0 s	49.5 s	325 / 2328
20 s	19.9 s	98.8 s	779 / 5147

linearly with the simulation time. At the beginning of the simulation, the device association was simulated and required more processing time than the simulation of a network with regular asynchronous data transmissions.

When looking at the difference between two simulation runs with identical simulation times, it can be found that the overhead caused by the instruction set simulation adds less than 4 s per second of abstract simulation time. The simulation of the implementation model by TSIM is between 10 and 20 times slower than the SDL simulation of a single protocol entity. This can be concluded by relating the additional execution time caused by TSIM to the pure SDL simulation processing time for four devices.

This clearly shows that instruction set simulators can provide performance estimations and profiling information for a target implementation within relatively short real processing times. They are orders of magnitude faster than RTL simulations. The effects of a hardware/software partitioning decision can be studied within a couple of minutes with this method.

Furthermore, the results confirm that our general concepts for an SDL cosimulation with an ISS are highly efficient and have been correctly and efficiently implemented. Synchronization between the two simulators occurs only at those times when a signal is sent from the ISS or when a timer expires in the functional SDL model. This avoids excessive scheduling overhead. The number of signals exchanged between TSIM and the functional SDL simulation is given in the last column of Table 6.3.

Chapter 7

Design Results

The protocol design methodology presented in the previous chapters has been validated by applying it to an embedded system implementation of the IEEE 802.15.3 MAC protocol. This work was conducted in our research group at the IHP as part of the Body Area System for Ubiquitous Multimedia Applications (BASUMA) project, a publicly-funded research project with the focus on developing a body-area wireless communication system supporting multimedia applications and, at the same time, requiring only very low power consumption [BAS06].

The IEEE 802.15.3 standard has been selected within the project for a number of reasons. Firstly, the supported data rates from 11 to 55 Mbit/s enable the exchange of MPEG-1 video streams, which was one of the requirements for the body area system. Secondly, TDMA-based channel access allows devices to conserve energy by turning on their radio transceivers only at scheduled time slots for transmission or reception. The channel time allocations are also useful to convey real-time or multimedia traffic due to their guaranteed bandwidth and periodic occurrence. Thirdly, effective power management schemes provided by the standard further enable groups of devices, that are part of so-called power-save sets, to synchronize their wake-up schedules in order to exchange data among them. The network coordinator role can be handed over smoothly to another device, thus providing a means to share the energy consumption equally among all devices in the network.

The basic operation of an IEEE 802.15.3 wireless network and its TDMA superframe structure has been explained already in Sect. 2.1.2 and shown in Fig. 2.4 on page 27. Due to the complexity of the protocol specification we refer to the

standards document [IEE03a] for a comprehensive presentation. The functionality most relevant for hardware/software partitioning will be covered where necessary in the following sections of this chapter.

We have designed and manufactured a chip that contains a LEON2 processor [Gai05] and a dedicated protocol accelerator for the IEEE 802.15.3 MAC. The LEON2 processor runs the protocol software and interacts with the hardware accelerator. The software was automatically generated from an SDL model of the protocol. Hardware/software partitioning of the inital SDL model was necessary to reach the data rates and timing requirements of the standard at a moderate clock frequency and, hence, with low power consumption. We have applied the cosimulation framework introduced in Chapter 6 to identify the functionality of the protocol accelerator.

The 32-bit LEON2 processor has been selected because it allows to address more than 64 kbytes of memory and, most importantly, the design is available as a soft core, i.e. the VHDL sources for the processor are open. It is, therefore, easily possible to extend the processor design with dedicated hardware connected to the processor bus.

We illustrate our embedded systems protocol design methodology by presenting the results of each step in the design flow applied to the IEEE 802.15.3 MAC protocol implementation. It should be pointed out that this is not an academic exercise based on a simplistic protocol, but rather a protocol whose complexity exceeds what is typically found in embedded systems. We begin with a brief overview of our SDL model in Sect. 7.1, followed by a presentation of cosimulation results and hardware/software partitioning in Sect. 7.2, and, finally, in Sect. 7.3 describe the architecture and behavior of the protocol accelerator.

7.1 SDL model of the IEEE 802.15.3 MAC protocol

Our primary design objective was to develop a complete functional model of the protocol. The functionality should be structured by means of SDL blocks and processes in such a way that the model would be easy to understand, validate, and extend by application-specific features. The model should also lend itself to hardware/software partitioning and make it easy to generate separate implementations for piconet coordinator (PNC)-capable and simple devices.

The design of the IEEE 802.15.3 MAC protocol SDL model was the focus of

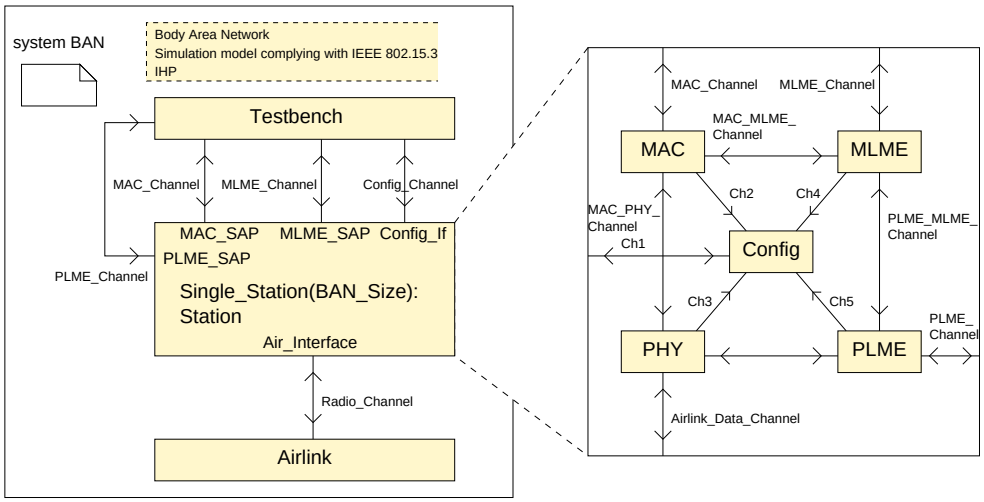


Figure 7.1: Structure of our SDL model for the IEEE 802.15.3 standard.

one of our conference papers [DBDK04].

7.1.1 Model architecture

Our top-level SDL system consists of a variable number of `Station` instances that are connected with each other through the `Airlink` block. This makes it possible to simulate the operation of a piconet consisting of several wireless devices. Stimuli to the individual stations as well as station configuration parameters are generated from within the `Testbench` block. This is shown in Fig. 7.1. The interfaces of the `Station` block to higher protocol layers are the MAC and MLME SAPs as defined in the standard. More details on the `Station` block structure are given below.

The `Testbench` contains SDL processes that send requests through the SAPs of the `Station` block. We use direct addressing to send requests to the respective stations. It is also possible to run a test program that is stored in a file instead of fixing the order of events in an SDL process.

In the `Airlink` block, a simple broadcast channel model is used. Frames that are transmitted by a station are received by all other stations at exactly the same time. It is possible for frames to collide or to be received in error due to random bit error insertion. This is indicated to the receiving stations by means of an extra signal parameter.

Referring to the 802.15.3 standard, the `Station` block contains five sub-blocks.

These are MAC, MLME, PHY, PLME, and Config as shown in Fig. 7.1. Only the MAC and MLME block functionality—the focus of the next section—will be used for the final MAC layer implementation.

The PHY and PLME sub-blocks contain abstract models for frame transmission, reception, and channel sensing as well as the PIB (personal information base) attributes of the PHY layer. The PHY layer functionality, such as synchronization, equalization, or coding, are outside the scope of this model.

The Config block is only required for SDL-specific reasons in order to handle more than one station in the piconet.

7.1.2 Behavioral description

The MAC layer functionality is contained completely in the MAC and MLME blocks, which corresponds to a clear separation of the so-called data path and control path. All data flow processing, such as

- check sum (CRC) calculation,
- encryption and decryption of the frame payload,
- interfacing with the PHY layer, and
- frame buffering

is modeled in the MAC block. The complexity of the MAC block functionality is rather low and the modeling of the mentioned algorithms straightforward, therefore we will focus more on the control path in the following. In the original model, the data path processing algorithms have not been optimized for efficiency, since at this point in the design flow we were aiming at a functional model of the protocol. The algorithms have been optimized later for hardware/software partitioning and implementation.

The MLME block is responsible for controlling the operation of the MAC block, maintaining protocol operation, and handling requests received via the MLME SAP. The design of the MLME block was driven by an object-oriented approach, which means that we first tried to identify basic modular units that are responsible for a single task and provide an interface, but no implementation details, to their clients. These units are known as classes in the object-oriented domain and are represented in our case by SDL processes. The exchange of signals between

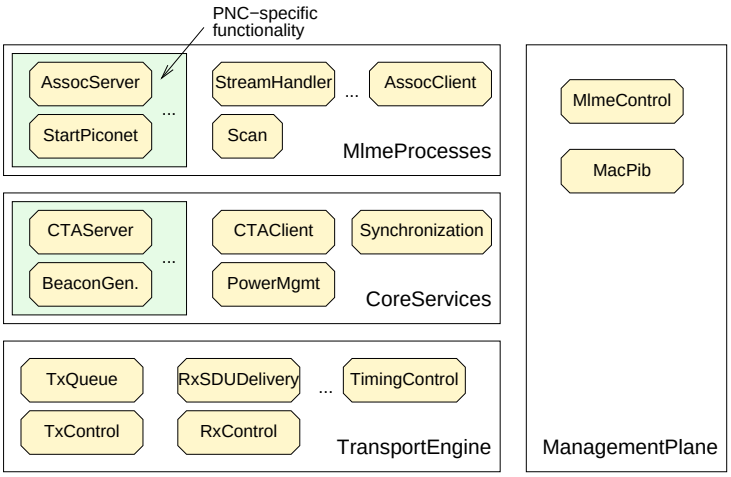


Figure 7.2: Functional layering of the processes in the `MLME` block

processes corresponds to the invocation of methods on an object. The concepts of concurrent process execution and asynchronous communication, which are native to SDL, have the great advantage of not anticipating any implementation choices regarding the hardware or software mapping and the kind of communication between processes.

The modularization approach has got the advantage that many designers in a team can work on the SDL model in parallel. Additionally, it leads to a decoupling of the individual modules (processes), so that the model can be modified or extended easily, without breaking the system. In short, applying the object-oriented design methodology for protocol design introduces the same benefits as seen for the development of large software systems, i.e. reduced complexity, clarity, etc. By taking the model for the `MLME` block as an example, our approach will be illustrated in some detail below.

Service layers in the `MLME` block The SDL processes in the `MLME` block can be grouped into three conceptual service layers and one management plane, as shown in Fig. 7.2. Additionally, within each layer, we identified those processes that are only needed for a PNC-capable device. This layering approach and the separation of PNC-specific functionality further enhances the clarity of the model and gives initial indications for a potential hardware/software partitioning.

The lowest service layer in the `MLME` block is called *TransportEngine*. It pro-

vides to the upper service layers the ability to receive and transmit service data units (SDU) such as commands, beacons or data. This means that upper layer processes do not have to deal with the channel access procedure, fragmentation and reassembly, retransmission, exact timing of transmission and reception, and so on. The timers in this layer require an accuracy of 1 μs.

On top of the `TransportEngine`, the *CoreServices* are placed. The processes in this layer are responsible for maintaining the piconet operation. The `CTAServer` process, for instance, manages channel time allocations in the superframe. A scheduling algorithm determines which stations are granted channel access based on previous channel time reservations. The `Synchronization` process observes the reception of beacons and takes action if the beacon was lost in several consecutive superframes.

The highest service layer contains the so-called *MlmeProcesses*. These are, for example, the `StartPiconet`, `Scan`, `AssocServer`, or `AssocClient` processes. They handle the service primitives received via the MLME SAP. Their behavior can be described on a high level by making use of lower-layer services such as frame transmission and reception. Note, that the `CoreServices` and `MlmeProcesses` do not require timers with an accuracy of 1 μs, but millisecond timers are sufficient.

Finally, the *ManagementPlane* contains the `MlmeCtrl` process, which distributes requests received via the MLME SAP and controls the overall station behavior, and the `MacPib` process that manages the personal information base (PIB) attributes.

Our layered approach leads to a decoupling of the functional modules of our model. Additionally, it facilitates the introduction of non-standard protocol extensions or new frame types. If desired, any additional functionality that relies on the basic frame exchange mechanisms can be placed above the `TransportEngine` layer with no or little impact on the overall model. Likewise, the basic channel access scheme or interframe spaces can be adapted by modifying the model at a single well-defined place.

The presented structure of the MAC protocol functionality gives good indications about which functions should be implemented in hardware and which in software. This is due to the fact that all bulk data processing is located in the `MAC` block and all time-critical control operations can be found in the `TransportEngine` layer.

The `TxQueue` process As an example of a module in the `TransportEngine` layer, we will present the SDL process called `TxQueue`. It is the responsibility of this process to queue service data units, i.e. beacons, commands or data units, for later transmission on behalf of other processes. This is initiated by sending a `TxAddBeacon.request`, `TxAddCmd.request` or `TxAddData.request`. When the SDU has been transmitted successfully, has timed out or has reached the retransmission limit, this is indicated via the `TxSDUStatus.indication` signal to the respective client process and the SDU is removed from the queue.

The `TxQueue` process will fragment SDUs into several frames, if necessary. Only if all fragments have been transmitted successfully and in the correct order, the SDU is removed from the queue. Another SDL process, called `TxControl`, queries the `TxQueue` process for the next frame to be transmitted depending on the current time slot. This can be either a beacon frame, a frame that is to be sent during the contention access period (CAP) or during a channel time allocation (CTA) for that device. The `TxQueue` process then determines which frame is to be transmitted next, based on frame priorities, remaining queuing time etc. It also performs aging of the queues in regular intervals, so that the SDUs are removed from the queue when their maximum queuing lifetime has expired. Note, however, that the `TxQueue` process is not responsible for maintaining the superframe timer and observing the channel access procedure. This is modeled in the `TimingControl` and `TxControl` processes, respectively. The `TxControl` process also informs the `TxQueue` process about a successful or failed frame transmission via the `TransmissionStatus.indication` signal.

Additionally, the `TxQueue` process initiates a request for more channel time for frame transmissions by sending `NewMCTAFrame.indication` and `AsyncChannelTimeRsv.indication` signals to the `CTAClient` process when a new SDU is queued and the current channel time reservation for the station is insufficient.

In Fig. 7.3, the relationships between the `TxQueue` process and other processes together with the signals that are exchanged between them are shown. A discussion of all the processes in the SDL model and their inter-relationships is outside the scope of this thesis.

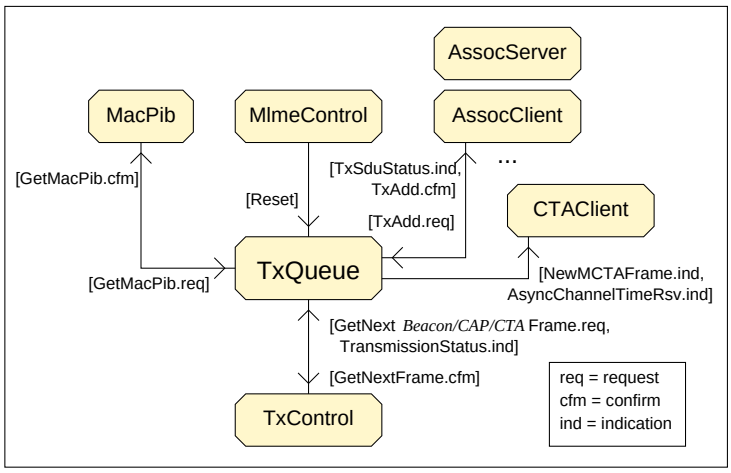


Figure 7.3: Process interaction diagram for the TxQueue process.

7.1.3 Results

Altogether, our functional model of the IEEE 802.15.3 MAC protocol consists of 24 SDL processes, with on average 10 pages per process. There are more than 160 declared SDL signal types for the communication with the physical layer and higher layer (as defined by the standard) as well as for the internal communication among the processes.

We have extensively simulated the protocol model in order to validate its functionality. For this purpose we instantiated four IEEE 802.15.3 stations, including a PNC, in simulation and tested the network behavior in response to requests received from a higher layer and with transmission errors in the physical layer. A formal verification by means of an external tool has not been conducted due to lack of time.

Exemplary message sequence charts involving two stations are given in Figures 7.4 and 7.5. They show the start of a new piconet by the first device after a scan procedure did not find any wireless network. The second device also performs a network scan and detects the newly created piconet by receiving beacons from the first device, now the PNC. Afterwards, the device synchronizes and associates itself with the network (Fig. 7.5).

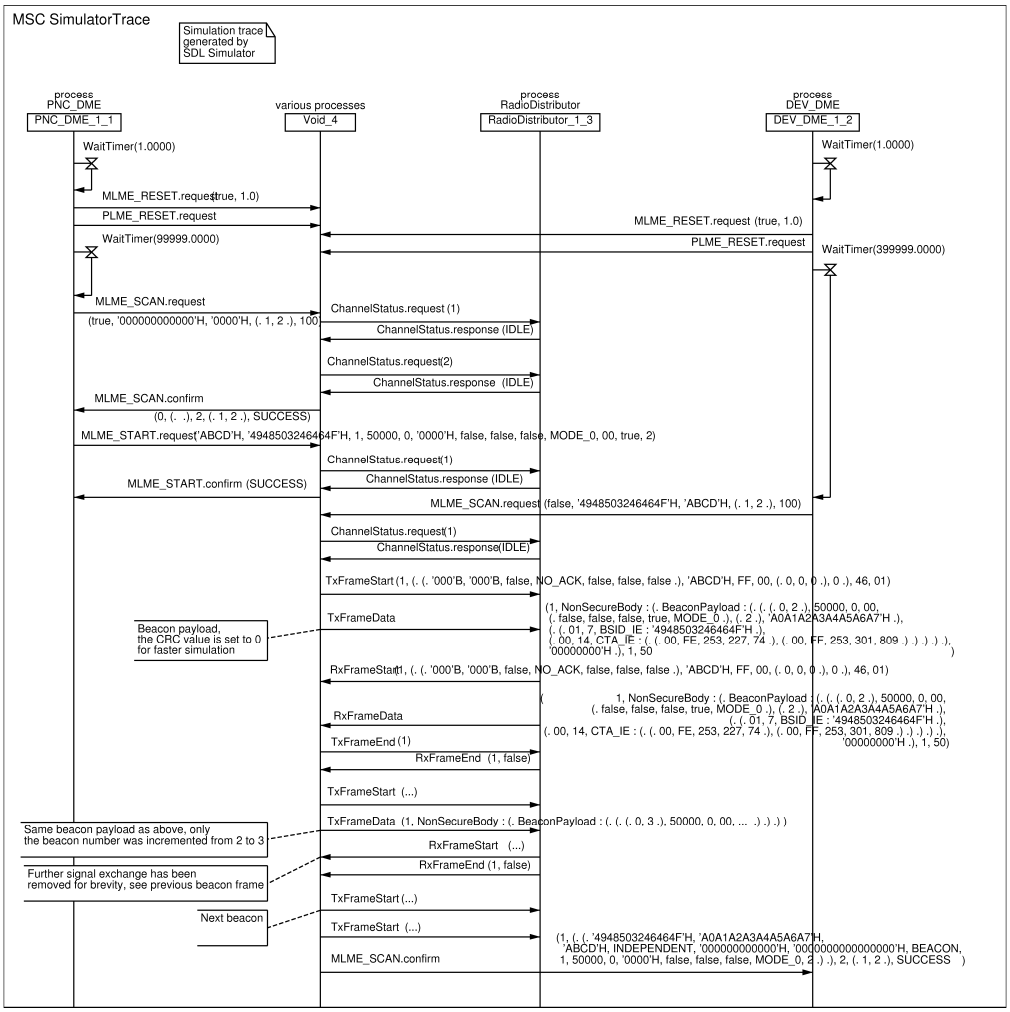


Figure 7.4: Message sequence chart showing the signal exchange from the higher layer (device management entity, DME) which initiates the formation of a new piconet.

7.2 Partitioning into hardware and software

In the following, we describe how our approach to hardware/software partitioning of communication protocol designs based on SDL/TSIM cosimulations was applied to the IEEE 802.15.3 MAC protocol. Rather than presenting a complete overview of all simulation runs and partitioning decisions, which would go beyond the scope of this thesis, we focus on a simple example that can be followed without knowing

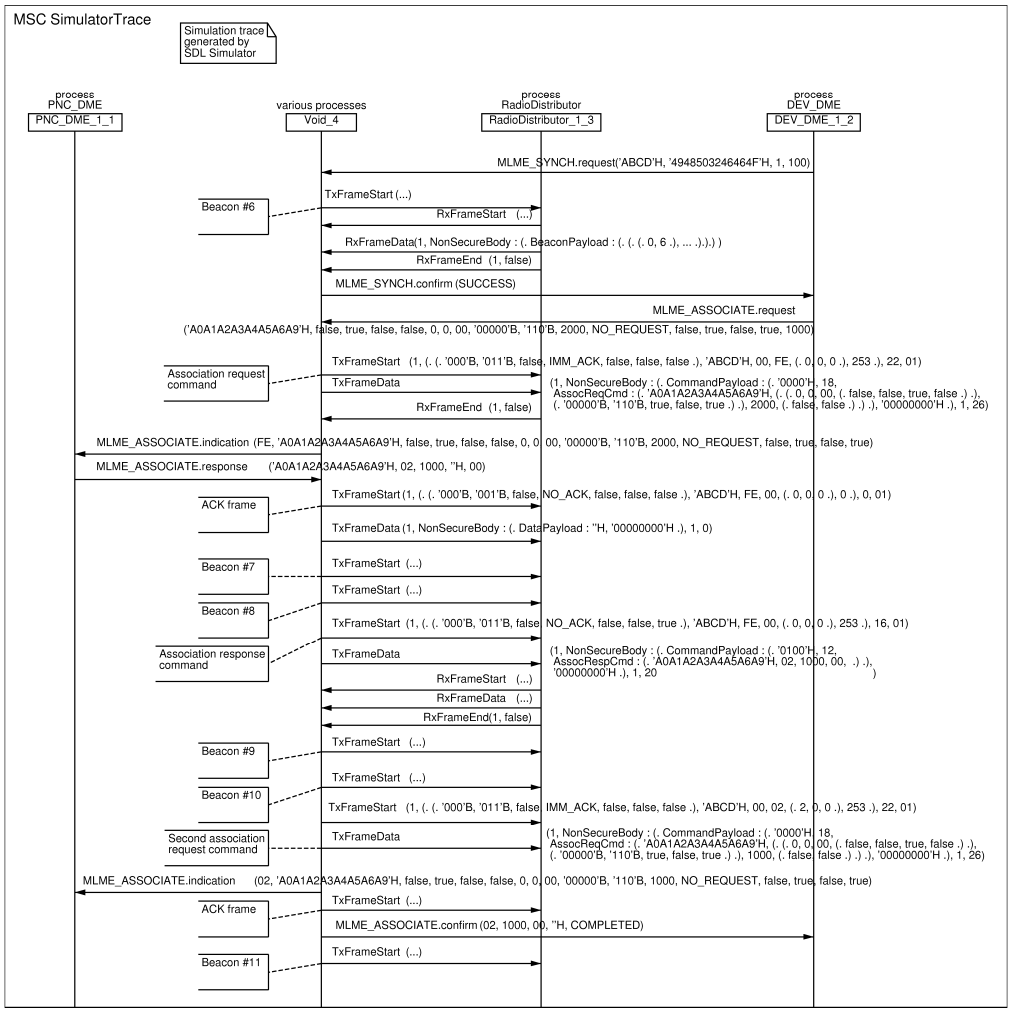


Figure 7.5: Continued message sequence chart from Fig. 7.4 showing synchronization with the PNC and association of a device. The frame exchange over the wireless medium is presented much shortened.

all the details of the protocol.

Hardware/software partitioning of the MAC protocol started from an all-software model. This initial implementation model was automatically generated from the SDL model introduced in the previous section. A number of optimizations have been applied to this model, for instance all signal outputs use explicit addressing of the receiver process, and inefficient SDL data types, such as strings

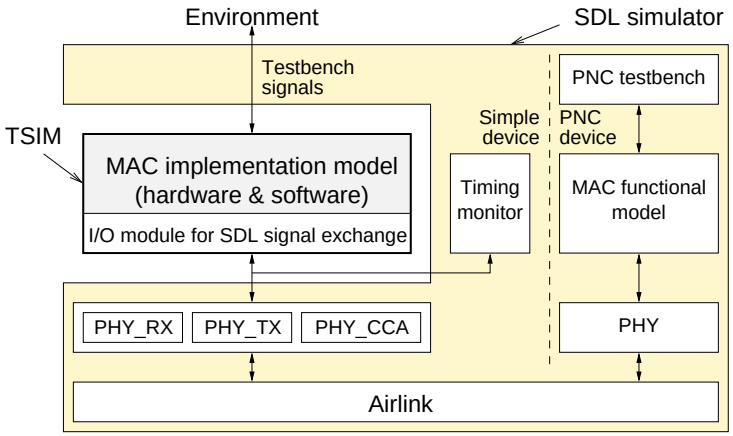


Figure 7.6: Experimental setup for the simulation of an IEEE 802.15.3 network consisting of a piconet coordinator (PNC) device and a non-PNC device. The latter is simulated by TSIM in order to obtain performance results for the MAC protocol implementation model.

or sets, were replaced by C++ implementations. Our tight integration library, which was presented in Chapter 5, has been used to target the SDL model to the Reflex operating system for the LEON2 processor.

In order to analyze the performance of the MAC implementation model and to study the effects of hardware accelerators, we simulated a piconet coordinator (PNC) device by means of the SDL simulator and connected it to a non-PNC device simulated by TSIM, in a first experimental setup. The two MAC protocol entities were interconnected by an abstract physical layer model for frame transmission and reception as well as clear channel assessment. The behavior of the physical layer was modeled according to the IEEE 802.15.3 physical layer standard, i.e. with data rates between 11 and 55 Mbit/s.

A test bench on top of the MAC layer served as the initiator of protocol operations, such as starting the piconet or exchanging data, and responded to service primitives received from the MAC layer, for instance an association indication. Our overall simulation environment for hardware/software partitioning is shown in Fig. 7.6.

The scenario that we consider in this section involves the following protocol functionality. The PNC device is ordered to start a new piconet. Subsequently, the other device simulated by TSIM associates itself with the new piconet. Upon

successful association, the PNC sends data to the newly associated device by using the asynchronous data service of the MAC protocol. For simplicity, all frame exchange may take place in the contention access period, which was chosen to have a duration of 20 ms in a superframe of 50 ms duration.

The expected frame exchange including the timing requirements are presented in Fig. 7.7. A violation of the expected timing behavior of the implementation model can be noticed either by a deviation of the expected protocol behavior of the PNC, for instance by repetitive retransmission attempts and, eventually, giving up the data transfer, or by means of a dedicated timing rules monitoring process. The latter approach helps significantly in the identification of missed deadlines.

The simulation of the scenario described above showed that the implementation model of the IEEE 802.15.3 MAC protocol managed to synchronize as well as associate successfully with the piconet coordinator. However, the exchange of data failed because the immediate acknowledgment (`ImmAck`) frame sent by the device was received by the PNC too late. After the end of transmission of the data frame a timer was started by the PNC. If no frame is received within the RIFS (retransmission interframe space), which equals to 27 microseconds, the timer will expire and trigger a retransmission of the data frame.

We investigated the reasons for the implementation model to miss the deadline for transmitting the `ImmAck` frame. The physical layer model for the device operates as follows. Precisely at the time when the frame preamble, physical layer and MAC header have been received, i.e. 22.5 μs after the start of frame transmission[1], an SDL signal indicating the reception of a new frame is sent to the MAC layer. This signal, `PHY_RX_START.indication` is defined by the standard and carries the MAC header information. This will request an interrupt to the LEON2 processor as outlined in Chapter 6. At the same time, the `PHY_DATA.indication` primitive containing the frame payload is also output to the MAC protocol implementation.

The assumption that the MAC header information is available from the physical layer immediately after it has been received by the antenna is unrealistic. We neglect the delays in the physical layer and make best-case assumptions in order to obtain the minimum requirements for the MAC protocol implementation. As is common in physical layer implementations, we assume the presence of a buffer

[1]The 2.4 GHz band physical layer of the IEEE 802.15.3 standard uses a preamble of 192 symbols and requires 56 symbols to transmit the PHY and MAC headers at a symbol rate of 11 Msymbols/s.

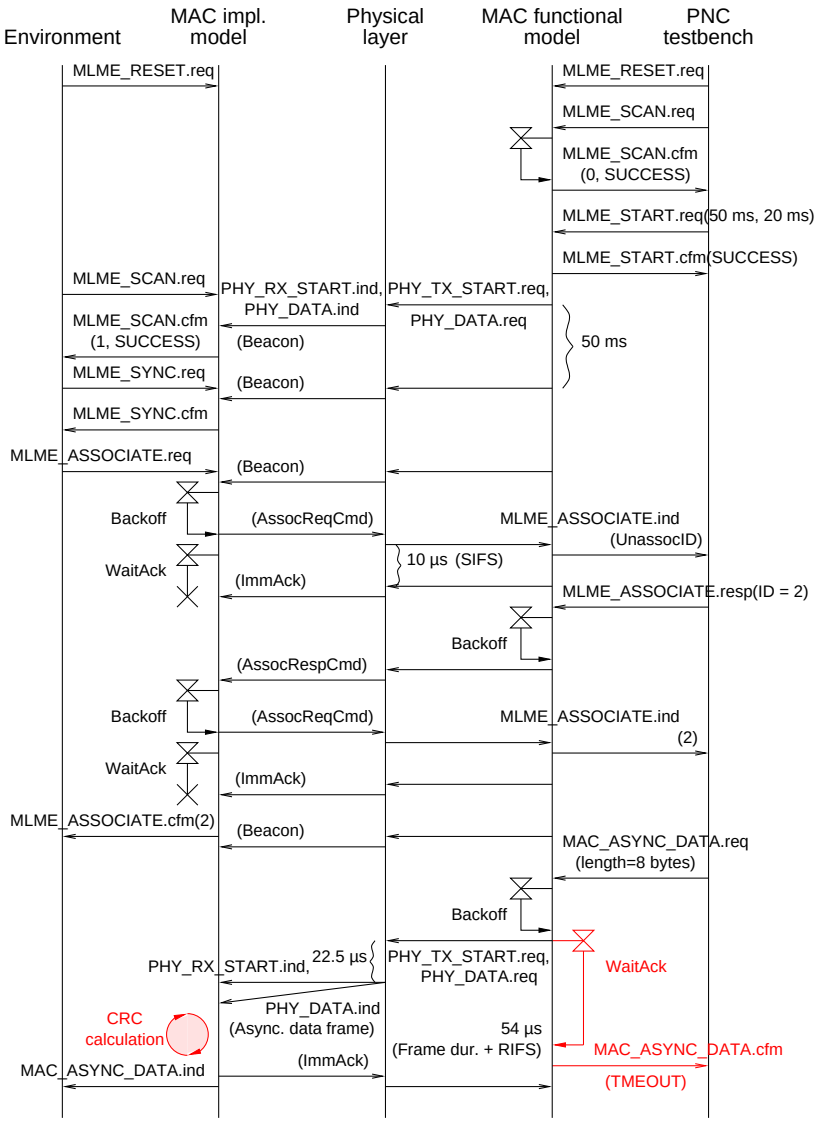


Figure 7.7: Message sequence chart showing the service primitives exchanged via the MAC and PHY layer interfaces for the start of a new piconet, association procedure, as well as asynchronous data exchange by the PNC. The data frame exchange fails because of the immediate acknowledgment frame being transmitted too late by the implementation model.

that can store received bytes and that can be read out by the MAC layer.

Before the `ImmAck` frame can be sent as a response to the received frame, the header information has to be evaluated, and the frame body must be checked for

any transmission errors[2]. For this purpose, a 32-bit CRC value over all payload bytes is calculated and compared with the received frame check sequence. The CRC check has been optimized for speed by using a table lookup algorithm. If the destination address in the MAC header corresponds to the device ID, an immediate acknowledgment is requested by the sender, and the CRC check is successful, the `ImmAck` frame has to be transmitted after the SIFS (short interframe space), i.e. 10 μs after the end of the received frame.

The protocol functionality related to the reception of a frame and generation of an `ImmAck` frame has been implemented in two different ways. In the straightforward approach directly resulting from the initial SDL model, several SDL processes are involved in the processing, namely `RxFrame`, `RxControl`, `TxControl`, and `TxFrame` (cf. Fig. 7.2 on page 163). In addition to interrupt latency, processing is delayed by scheduling overhead and the exchange of SDL signals. Interrupt latency consists of any delay caused by the application running with disabled interrupts in a critical section and by the execution of instructions necessary to save the processor state before actually jumping into interrupt handler code.

In the optimized approach for acknowledgment generation, all required functionality is performed by the interrupt service routine, thus avoiding any scheduling overhead. This optimization could only be achieved by manually re-implementing the algorithms in C++. The corresponding transitions and signals have been removed from the SDL model. The purpose of this approach was solely to measure if an all-software implementation of the acknowledgment generation would be feasible and what clock frequency required.

With the help of the instruction set simulator, we have measured the time required by the implementation model to handle the interrupt triggered by the `PHY_RX_START.indication` signal and to generate the `ImmAck` frame with both approaches and different frame lengths. We used the I/O module to print the exact clock cycle count of the TSIM simulator when a specific memory location was accessed. Table 7.1 provides the simulation results for frame lengths of 8, 80, and 800 bytes. The mandatory physical layer data rate of 22 Mbit/s has been assumed for frame transmissions.

The results show that, in all cases, not only the deadline for the timely trans-

[2]The physical layer and MAC header information is protected by its own CRC header check sum. According to the standard, only correctly received headers are passed by the physical layer to the MAC layer.

Table 7.1: Results from the simulation of data transmissions by the PNC with different frame lengths. (All times are relative to the start of the data frame transmission.)

Event		**CRC calculation and ACK generation by**	
		Original SDL model	Interrupt service routine
`PHY_RX_START.ind` sent by SDL simulation		23 μs	
`PHY_RX_START.ind` received by interrupt handler		64 μs	
Deadline for ACK received by PNC			
	8 bytes	54 μs	
Frame length	80 bytes	80 μs	
	800 bytes	342 μs	
Output of `PHY_TX_START.req` for `ImmAck`			
	8 bytes	1005 μs	114 μs
Frame length	80 bytes	1204 μs	151 μs
	800 bytes	2115 μs	498 μs

mission of the `ImmAck` frame has been missed, but also the deadline for the start of reception of this frame by the PNC was exceeded, which has the effect that a retransmission would be triggered. We observed that interrupts were disabled due to execution in a critical section at the time when the `PHY_RX_START.indication` signal was sent to the implementation model so that its processing was additionally delayed. In effect, the interrupt handler for processing the signal was executed only 64 μs after the start of the frame transmission.

We have measured that the interrupt latency amounts to 25 microseconds when the interrupt can be immediately served. Even this lower bound on interrupt latency leads to a late acknowledgment transmission for short received frames since the interframe space before the `ImmAck` frame is 10 μs and the frame body may well be shorter than 15 μs.

Furthermore, the results show that the CRC processing speed of the frame body within the interrupt service routine does not reach the physical layer data rate of 22 Mbit/s, but is in the order of 16 Mbit/s. This can be concluded by

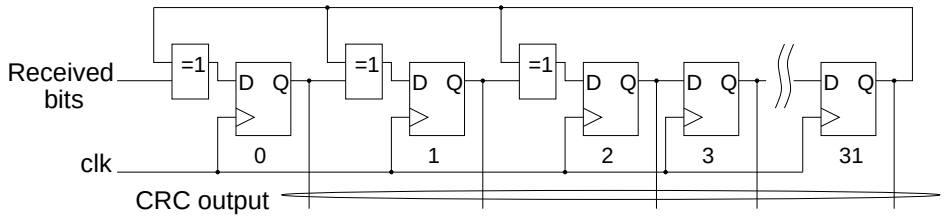


Figure 7.8: Hardware design of the 32-bit CRC algorithm consisting of 32 D flip-flops and a number of XOR gates according to the generator polynomial specified by the standard [IEE03a].

determining the time difference for a large frame (800 bytes) and a short frame (8 bytes). Except for the additional data processing, all other processing overhead remains the same.

The measurements have been obtained with the LEON2 processor clocked at 40 MHz. Though it is possible to use a higher clock frequency in order to meet the real-time requirements, this would also mean an increase of the system's power consumption. At a fixed supply voltage, the power consumption of a digital system is proportional to its clock frequency. The targeted application field of wireless body area networking with battery-powered devices excludes this design alternative. Besides, the processor would be busy calculating CRC values during frame reception and transmission, so that other protocol functionality or application processing would be delayed. This is particularly severe as the interrupts are disabled during the CRC processing.

Alternatively, the CRC calculation and acknowledgment generation could be performed in hardware. In fact, the CRC-32 algorithm can be realized with a single 32-bit shift register as shown in Fig. 7.8. It is easily possible to modify the design slightly in order to process 8 bits with every clock cycle in paralel, thus bringing the required clock frequency for the maximum data rate of 55 Mbit/s down to less than 7 MHz. The generation of the `ImmAck` frame in response to a previously received frame can also be designed with little effort in hardware. For this purpose, registers for the device ID (8 bits) and the piconet ID (16 bits), as well as for the source ID of the received frame have to be introduced. On reception of the MAC header, only few bit comparisons must be performed.

When comparing the presented design alternatives, mapping the CRC check and acknowledgment generation to hardware is preferable due to the reduced power

consumption compared with an all-software implementation. This result was expected and, in fact, has been realized in many other communication controllers before. We presented the partitioning problem in detail to illustrate the application of our cosimulation framework.

In the same manner as outlined above we identified the following MAC protocol functionality to be designed in hardware:

- *Frame reception and transmission procedure.* This involves the calculation of the CRC checksum over frame body data, encryption and decryption, acknowledgment generation as well as direct access to frame storage in memory in order to free the processor from copying data between the memory and protocol accelerator. The security algorithms have not been included in the first hardware design.

- *Superframe timing.* According to the channel time allocations broadcast in the beacon frame and based on timers with an accuracy of 1 microsecond, the channel access is controlled by the hardware accelerator. This means that exactly at the scheduled time the transmission of the right frame according to the current position in the superframe is triggered. This could be, for example, a beacon frame, or a data frame to be sent to another device in an allocated time slot. During the contention access period, the backoff procedure is performed in hardware. When an immediate acknowledgment is requested, a hardware timer is set and a retransmission started if the acknowledgment was not received, provided that there is enough time for it in the current time slot.

- *Beacon parsing.* Since beacon frames carry the information on how the time until the next beacon is allocated to the devices, it is necessary to parse the channel time allocations in the beacon immediately after it has been received or transmitted. Otherwise, a portion of the superframe would be missed due to delayed processing in software. As stated above, this information is required for superframe timing.

- *Parts of the transmission queue.* According to the current position in the superframe, the right frame for transmission must be selected. Since time slots can be rather short, in the area of few hundred microseconds, and devices may stop receiving when a transmission has not started shortly after

the beginning of a time slot, the frame data must be accessible immediately at the scheduled time. This can be achieved by placing parts of the transmission queue in hardware. We refrain from designing the complete queue in hardware in order to save resources for storing the frame information. From an implementation point of view, it is much easier and more flexible to manage the queues by software and keep the queue elements in data memory. It is sufficient to provide the hardware queue with the elements first in the queue and update this information after completed transmissions or for frame prioritization.

The results show that the line between the hardware and software partitions cuts through SDL processes, in some cases. The transmission queue, for instance, is not entirely designed in hardware as this would put an inflexible upper limit on the number of frames that can be handled by the implementation and waste hardware resources. Access to the first elements in the queue, however, is time-critical, and a part of the transmission queue is therefore designed in hardware. Another good example is the beacon analysis. Only the channel time allocation (CTA) information element is relevant for the time-critical medium access and, for this reason, part of the protocol accelerator, while all other information elements contained in the beacon frame are processed in software.

Some parts of the `Transport Engine` block remain in software, such as the defragmentation of received frames and the forwarding of complete SDUs to the appropriate receiver processes. These are control-dominated functions, whereas the processing-intensive and time-critical functionality of the channel access mechanism is handled by the protocol accelerator.

All SDL processes above the `Transport Engine` have no tight timing constraints. Consequently, they are mapped to software and handled by the LEON2 processor. Interrupts are used to signal protocol-related events from the hardware to the software. Conversely, the software interacts with the hardware block by writing to and reading from a number of control registers.

7.3 Protocol accelerator design

In this section we first introduce the system-on-chip target platform for the IEEE 802.15.3-compliant wireless communication system. Then, we present the architecture of the protocol accelerator and how its components work together to

provide the MAC functionality identified as hardware partition in the previous section.

Within the scope of this thesis we can only describe a small part of the RTL design of the protocol accelerator. We focus on the design of the transmission queue component as it exemplifies the tradeoff between hardware and software implementation, the interface between the software and hardware parts, and, last but not least, the hardware transmission queue consisting of several independent queues for different frame types is a novel contribution as far as we know. The protocol accelerator design was granted a patent [Die07].

The protocol accelerator is not limited to the IEEE 802.15.3 physical layer specification. We show how our design can be used for different data rates and non-standard physical layer timing parameters. Finally, the section is concluded with results from an FPGA implementation of the complete system. The LEON2 processor system including our protocol accelerator was also manufactured in 0.25 μm technology at the IHP as an ASIC. This chip is currently being tested on a specifically designed board.

7.3.1 Target hardware platform

Our wireless platform based on the IEEE 802.15.3 standard is composed of the LEON2 processor and communication subsystems integrated on a single chip [DEK07]. The processor runs protocol as well as application software. As shown in Fig. 7.9, the communication subsystem is composed of the RF front-end, digital baseband processor, and the MAC protocol accelerator as an important component of our wireless platform.

The protocol accelerator is connected to the system bus, the AMBA AHB bus (see Fig. 7.11). Via an AHB master interface it is possible for the accelerator to directly access the system memory, for instance to store and retrieve frame data without involving the LEON2 processor.

For data transfer to/from the baseband processor as well as status indications from the physical layer, the MAC-PHY interface was designed. It is of master/slave type with the MAC protocol accelerator acting as master. The accelerator sends commands to control the start of transmission or reception and to exchange frame data. The baseband processor contains a 32-byte data buffer for storing frame data in both, RX and TX, directions in order to decouple the two protocol layers with respect to timing (see Fig. 7.10). There are signal lines from the physical

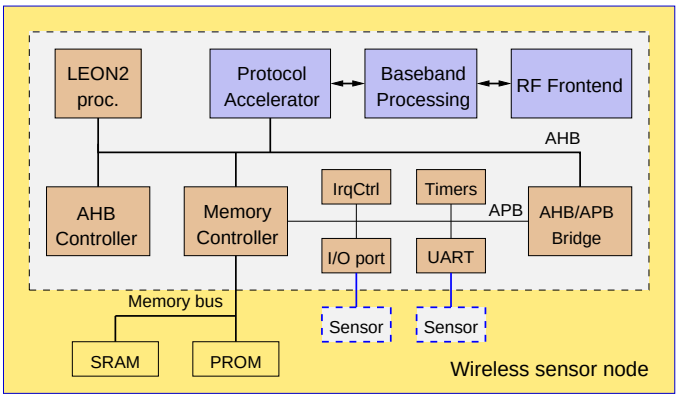


Figure 7.9: Single-chip wireless communication platform consisting of the LEON2 processor, protocol accelerator, and modem. It can be easily used for the design of wireless sensor nodes.

layer to the MAC hardware accelerator that indicate the status of these buffers and can be used for flow control.

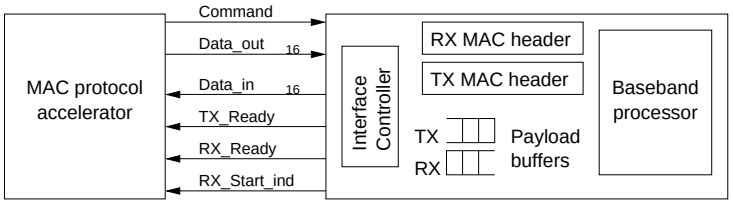


Figure 7.10: Interface between MAC protocol accelerator and the physical layer. *TX_Ready* is applied when the TX buffer is not full, *RX_Start_ind* when the reception of new frame has started, and *RX_Ready* when the RX buffer contains data.

7.3.2 Architecture

Figure 7.11 shows the main components of the protocol accelerator. The tasks performed by each of the main components are listed below. They reflect the protocol functions that have been identified in Sect. 7.2 to be designed in hardware.

- In receive direction, to retrieve frame data from the physical layer byte by byte, perform packet filtering and CRC check, and to store the data at a given memory location by means of direct memory access (components *Rx controller*, *CRC*, and *DMA*).

- In transmit direction, to retrieve frame data from a memory location, calculate and append the check sum, and write the data to the physical layer (components *Tx controller*, *CRC*, and *DMA*).

- To signal a successful reception or transmission of a frame to the processor by an interrupt (component *Interrupts*).

- To analyze received and transmitted beacon frames and extract information on channel time allocations (component *Beacon parser*).

- To manage a queue of frames that are to be transmitted, and to select an appropriate frame for transmission (component *Transmission queue*).

- At the start of a time slot or following a frame transmission, to query a new frame from the queue and, in the case that the frame must be acknowledged by the receiver, wait for the acknowledgment frame (components *Scheduler* and *Timers*).

- To perform the backoff procedure in the contention access period (components *Scheduler* and *Timers*).

- To send an acknowledgment at the right time upon reception of a frame that needs to be acknowledged (components *Scheduler*, *Timers*, and *Tx controller*).

An additional component (*CalcDuration*), that is not shown in Fig. 7.11 for simplicity, calculates the actual duration of a frame transmission based on its payload length and data rate. This component is used to determine if a frame transmission fits into an available time slot and when a transmission initiated by the protocol accelerator will be completed by the physical layer.

7.3.3 Transmission queue

The *Scheduler*, *Transmission queue*, and *DMA* components in the protocol accelerator facilitate a frame transmission operation that is not directly controlled by the processor. This allows to reduce the clock frequency of the processor and, hence, leads to possible energy savings. The *Transmission queue* component, that shall be discussed in this section, has a key role in this operation.

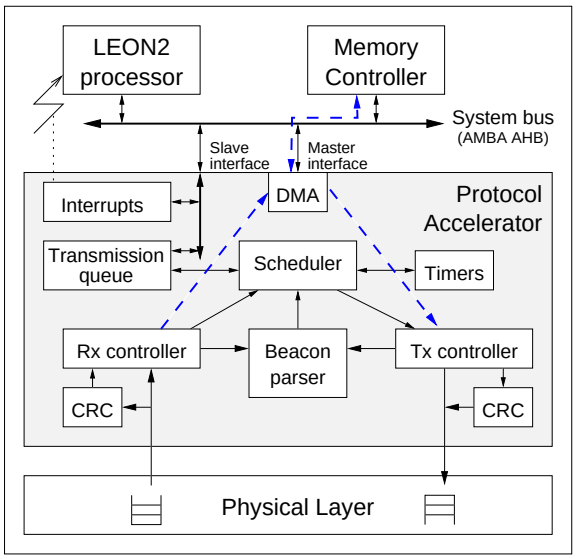


Figure 7.11: Hardware architecture of the protocol accelerator (direct memory access data path highlighted by dashed blue lines) [DEK07].

The transmission queue designed for the IEEE 802.15.3 MAC protocol accelerator contains and manages a table with information on frames to be transmitted—not the frame data itself, which is stored in application memory. The protocol software running on the LEON2 processor fills the table according to the generated frames and their transmission order. An interrupt is signaled to the processor as soon as a frame from the queue has been transmitted successfully or its maximum retransmission limit has been reached. This indicates to the software that the entry in the table is free and can be reused by another frame. An update, however, does not have to happen immediately as there are still enough frames in the hardware transmission queue.

The current design contains 8 table entries, but there might be many more, e.g. 32 or 64. In order to find the right frame upon request from the *Scheduler* quickly, there are ordered lists for different frame types, for instance for beacon frames or frames that can be transmitted in the contention access period. The table index of the first list item is kept in a register. From this item the complete list can be traversed by following the table index of the next list item, which is stored in the table. Furthermore, there is a reference to the previous list item to support delete operations efficiently. In essence, this forms a double-linked list. These lists are also used to preserve the right order of data frames where the MAC protocol

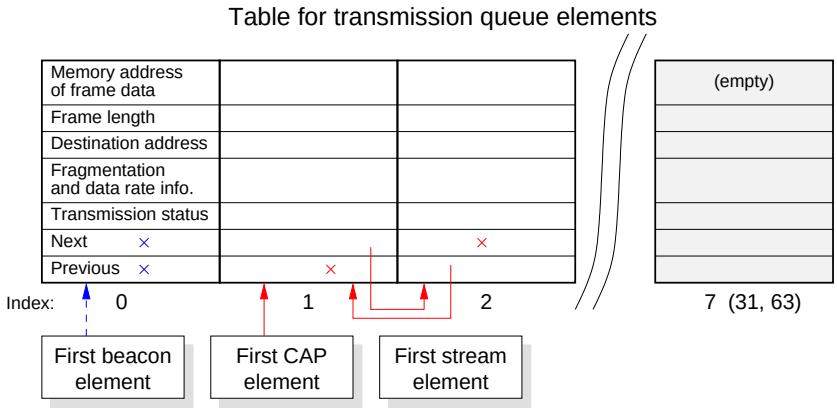


Figure 7.12: Structure of the transmission queue table and its elements.

provides an ordered delivery service to the higher layers. Figure 7.12 presents the design concept of the transmission queue graphically.

While the software is updating the transmission queue, the hardware may not access its contents in order to avoid inconsistencies. Similarly, when the hardware is browsing the lists for a suitable frame to transmit, the software may not write to the queue. Therefore, a lock must be acquired by the software before an operation on the queue can be performed and released when it is done. When the queue is locked, the hardware must defer access to it.

7.3.4 Support for flexible timing

The protocol accelerator has been designed such that it is not limited to a fixed data rate and time intervals between consecutive frame transmissions (interframe spaces). Instead, a number of software-programmable registers have been introduced that completely determine the timing behavior of the protocol. These registers are read by the *Scheduler* to calculate the point in time when the next frame can be transmitted. There are registers for SIFS, MIFS, and backoff slot duration, each given in microseconds.

Additionally, there is one programmable register which contains a rate factor and is used by the above mentioned *CalcDuration* component to calculate the duration of frame transmissions.

7.3.5 Software interface

The protocol software interacts with the hardware accelerator through a set of *registers* that are accessible from the processor via *memory-mapped I/O*. A special memory region is reserved for the accelerator. Additionally, *interrupts* are used to signal events from the protocol accelerator to the processor.

There are a number of configuration registers that store MAC protocol information like the piconet or device identifier. It is possible to enable or disable the scheduler with another register. When the scheduler is enabled, the protocol accelerator will analyze beacon frames and seize transmission opportunities, otherwise it is just in scanning mode and delivers received frames.

The protocol accelerator features four maskable interrupts: superframe start, MAC header received, MAC frame received, and transmission queue interrupt. The latter is triggered when a queued frame has been sent or was discarded. Two registers indicate the first free queue index and the first index that contains a transmitted or discarded frame. This way, the software does not have to browse the complete queue to get this information.

To control the receive operation, there are registers for the address where the next payload can be stored and for the size of this buffer. When the accelerator has written data to this buffer, no other payload can be received unless a new payload buffer address is provided by the software. An acknowledgment frame will be generated only if the header and payload of the received frame have been stored.

A complete list of all memory addresses in use by the accelerator and a brief description of their purpose is summarized in Table 7.2.

7.3.6 Results

In a first step, we have implemented the LEON2 system and protocol accelerator on a Xilinx Virtex-II FPGA. The functionality designed in hardware has been removed from the protocol software by updating the SDL model. An interface to the protocol accelerator has been added. The executable for the LEON2 processor requires about 140 kbytes of program memory and 60 kbytes of data memory (bss and data segments).

We have successfully tested the complete MAC protocol implementation, i.e. the protocol software running on the LEON2 processor and the hardware accel-

Table 7.2: Protocol accelerator registers and their purpose.

Offset	Name	Short description
00 hex	RESET	Protocol accelerator reset
10 hex	RX_HEADER	Last received MAC header
1C hex	TX_CONFIG	Filter frames based on correct destination ID, network ID
20 hex	PHY_PARAMS	Configure PHY layer timings, interframe spaces
24 hex	TIMING_CONTROL	Enable scheduler, scanning mode
2C hex	CALC_DUR_PARAMS	Configure non-standard preamble length and data rate
34 hex	INTERRUPTS	Pending interrupts register
38 hex	INTERRUPT_MASK	Interrupt mask register
3C hex	MLME_PIB	Set device and network ID, and whether device is currently acting as PNC
4C hex	ACK_DUR	Set duration of ACK frame, retransmission timeout (RIFS), and backoff slot duration (in microseconds)
60 hex	RX_RESET	Reset receive operation
64 hex	RX_BUFFER	Set memory address where to write next received frame
68 hex	RX_BUFFER_SIZE	Set maximum size for received frame
6C hex	RX_START_PARAMS	Read PHY header information (frame length and data rate, header check sequence)
70 hex	RX_FRAME_PARAMS	Result of receive operation (CRC correct, security check)
C0 hex	TXQ_PENDING	Bitfield indicating which of the TX queue elements were either successfully transmitted or discarded
C4 hex	TXQ_FIRST_FREE	Returns a TX queue index which is not currently filled with an element
C8 hex	TXQ_FIRST_DONE	Returns a TX queue index with completed transmission
CC hex	TXQ_OPERATION	Add or remove a TX queue element, lock TX queue

erator, by connecting two FPGA boards with wires. This emulates a network of two devices. The wires couple the boards below the MAC layer, data symbols are transferred serially at a rate of 20 Mbit/s. Table 7.3 shows the usage of FPGA resources of the same LEON2-based system with and without the protocol accelerator.

After the successful test on the FPGA, the complete LEON2-based MAC processor including Flash memory and peripherals has been designed and taped out as an ASIC in 0.25 μm CMOS technology [SDP+07]. The chip occupies an area of 31.9 mm^2 and consumes 15 mW/MHz. A die photo of the chip is shown in Fig. 7.13. Based on our synthesis results, the silicon area of the protocol accelerator is about 1.8 mm^2.

Table 7.3: FPGA resources used by the MAC protocol system.

Resources	LEON2 system		Difference
	Original	With prot. acc.	
4 input LUTs	11,582	24,034	12,452
Occupied slices	6,828	14,365	7,537
Block RAMs	20	22	2
Equivalent gate count	1,427,060	1,681,651	254,591

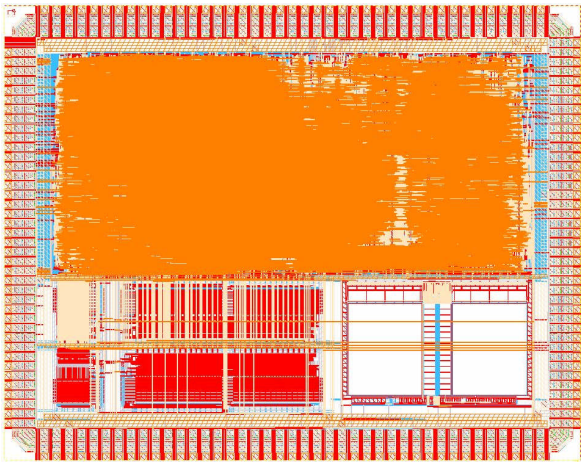


Figure 7.13: MAC processor chip layout.

This chip is currently under test on an evaluation board. First tests have been successful. By connecting this board with the baseband processor running on an FPGA and an RF board we will be able to demonstrate a complete IEEE 802.15.3-compliant wireless system. Its components will then be integrated on a single chip enabling applications with demand for high data rates, efficient power management, and guaranteed quality-of-service, such as multimedia and real-time applications.

Chapter 8

Critical Assessment and Future Work

This thesis focused on the development of an SDL-based protocol design and implementation methodology for embedded systems. It has been successfully applied to the design of an IEEE 802.15.3-compliant wireless communication system.

Driven by Moore's law, i.e. the exponentially increasing count of transistors that can be inexpensively placed on an integrated circuit, we expect that the trend for miniaturization and growing complexity of embedded systems will continue. This creates the potential of developing cheap, battery-powered, autonomous devices with enough computing resources for smart applications and the ability to communicate wirelessly. As key requirements for a massive deployment of such devices we consider their reliable operation and ability to use the available scarce energy efficiently.

However, as stated in the HiPEAC Roadmap on Embedded Systems [V+06], "at present a huge gap exists between specifications and implementation of embedded systems". The utilization of methods that can use the system specification for automatic or semi-automatic synthesis of the implementation was regarded as a future trend in EDA tool development [V+06].

From our perspective, ideally, a high abstraction level functional specification complemented by specifications of the non-functional requirements should be the starting point for the design flow (see Fig. 8.1). The functional specification would be composed of elements that use different models of computation, such as finite state machines or synchronous data flow. Non-functional requirements comprise

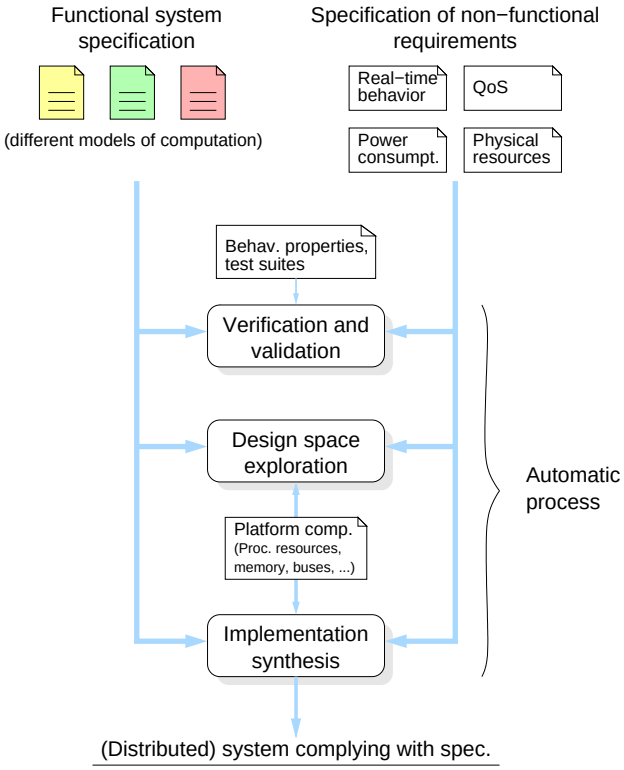


Figure 8.1: Elements of an ideal design flow that supports hardware/software system synthesis by an automatic transformation from high-level specifications.

the real-time characteristics, expected quality-of-service, but also constraints on power consumption and physical resource usage.

Tools supporting the formal verification of the specification against a set of formally defined properties should be employed. An automatic transformation of the high-level model into a, possibly distributed, system implementation that satisfies the non-functional requirements would yield products that are correct by design.

Though having been the focus of research in systems design in the past decades, we are still far away from the ideal solution sketched above. Tools that address some of the challenges have been proposed, for instance the automatic hardware generation from SDL specifications or system specification using heterogeneous models of computation. Especially the integration of non-functional requirements into the design process still lacks support. Another hurdle for the wide adoption of automatic synthesis tools is the efficiency gap compared with hand-optimized

hardware/software systems.

Our work addresses system design from high-level specifications and contributes to an increased efficiency of automatically generated software from SDL models. Furthermore, we support hardware/software partitioning with our cosimulation framework. We are aware that SDL is not a language that is equally well suitable for all kinds of applications. However, its benefits in the area of protocol design are well-known. One of the major drawbacks of the language is its inability to specify real-time requirements, as discussed in Sect. 2.1. For hard real-time systems development, extensions of SDL that have been proposed in the literature, or other languages must be used.

We have demonstrated that our proposed methodology is well-suited for the design and implementation of a complex wireless MAC protocol. We developed an SDL model of the IEEE 802.15.3 MAC protocol from scratch and used the C code generator from Telelogic to transform this model into a first, all-software implementation, which was then optimized and the basis for hardware/software partitioning. We have shown that the tight integration approach for the Reflex operating system outperforms the standard light integration approach in terms of memory consumption (by 25–30 percent) and speed (by 30 percent).

With the help of an available instruction set simulator and by performing cosimulations with the abstract SDL model, we partitioned the protocol in such a way that the clock frequency of the general-purpose processor could be lowered while still satisfying the real-time properties of the protocol. A protocol accelerator, which realizes the hardware functionality, was patented and designed as an ASIC. The CRC calculation is the bottleneck of the design, a clock frequency of 7 MHz is required to achieve the 55 Mbit/s throughput of the maximum allowed data rate.

Therefore, we are convinced that the proposed design flow and our tools advance the state-of-the-art in protocol engineering. Since we have specifically focused on resource-limited device in our work, the SDL-based design flow can be employed in the area of wireless sensor networks that has attracted much research interest in recent years. Even without hardware/software partitioning, but solely by using the efficient automatic transformation of SDL models and their integration into a sensor node operating system, our approach can facilitate the development of formally verified applications and protocols.

In order to assess the suitability of the SDL-based design flow for typical wire-

less sensor networks protocol implementations, we developed an SDL model of the S-MAC protocol [YHE02], which was briefly described in Sect. 2.1.2, and created a pure software implementation by applying the tight integration approach. We targeted the application for the TMote Sky platform [Cor06], since the operating system Reflex and device drivers for the hardware timers, radio transceiver, and other peripherals were already available.

We used the S-MAC source code available in the TinyOS operating system[1] as the basis for our SDL model. The entire functionality of the MAC layer was included in the model. The only adaptations to the protocol we had to make were related to the different timing of the radio transceiver and the timer handling mechanism in the MAC protocol, as we will explain briefly in the following.

The original TinyOS S-MAC module was designed for the Mica and Mica2 sensor nodes featuring a different radio transceiver, the RFM TR3000 with 20 kbit/s data rate, than found on the TMote Sky board, where a CC2420 with 250 kbit/s is used. Unfortunately, no open S-MAC implementation for TMote Sky exists. The TinyOS implementation of S-MAC uses a timer that creates a tick every millisecond. With every tick, a number of timer variables is decremented and a corresponding action triggered when a variable reaches zero. Due to the higher data rate on our board, the required timer granularity must be shorter than 1 ms. This would lead to an excessive timer tick processing. Therefore, we decided to use the TMote Sky hardware timer and perform the timer handling interrupt-driven. This is more efficient, since the microcontroller may switch into sleep mode while waiting for the next timer interrupt, while this was not possible in the original design. However, this modification added some complexity to the model.

We tested our SDL-based S-MAC implementation by developing a simple demonstration application. Its software architecture, which is typical for applications generated following our design flow, is shown in Fig. 8.2. On the lowest software layer, the device drivers are situated. They must be provided by the operating system and cannot be specified in SDL. As an example, the driver for the CC2420 transceiver provides an interface to access configuration registers, receive and transmit buffers, as well as trigger commands such as starting a frame transmission. It uses an SPI interface driver to communicate with the hardware module. However, the interrupt service routine to handle the reception of a frame

[1] Source: TinyOS CVS repository on `http://sourceforge.net`. S-MAC is located at `tinyos-1.x/contrib/s-mac` (last change: September 2005).

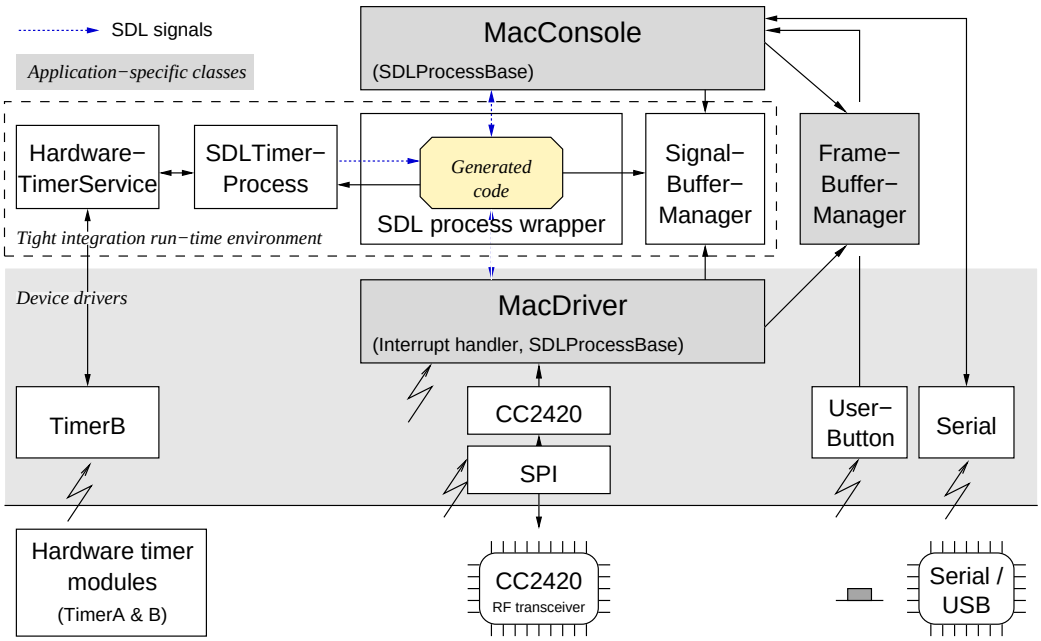


Figure 8.2: Software architecture of the S-MAC demonstration application on Reflex. It incorporates automatically generated code from the SDL model of the protocol.

is not included in the CC2420 driver, as this functionality is application-specific.

On top of the driver layer are the automatically generated SDL processes, any environment processes, and other application-specific classes. In the case of the S-MAC application, there is one environment process, `MacDriver`, which acts as the physical layer interface for the MAC protocol model. It communicates with the S-MAC SDL process via SDL signals. Another environment process, `MacConsole`, makes use of the S-MAC services and initiates communication with other devices upon user requests received via a serial interface or by pressing the user button on the board.

The figure also shows classes that are an integral part of the SDL run-time environment, namely the `SDLTimerProcess` and `SignalBufferManager`. A `HardwareTimerService` is responsible for providing the current system time and notifying the `SDLTimerProcess` when the next timer event has expired.

An application-specific `FrameBufferManager` class is used to allocate memory for received frames or user data that shall be transmitted. Like the signal buffer manager, it uses pre-allocated pools of memory buffers. It could also have

Table 8.1: Sizes of the text, data, and bss segments (in bytes) of the executable for the S-MAC demonstration application created from an SDL model of the protocol.

S-MAC demo application	**Tight integration**		
	text	data	bss
SDL model	14186	267	2246[a]
Run-time system	4196	24	6
Environment, application classes	6636	8	0
Reflex, device drivers	5882	12	4
libgcc	210	0	0
Total (smac.elf)	**31110**	**267**	**2246**

[a]This number includes local process variables, timer signals, as well as signal pools for the 18 signal types.

been modeled in SDL, but would have hampered comparability with the TinyOS application.

Other OS classes, such as the scheduler or clock, are not shown in the figure for simplicity.

The memory breakdown for our demonstration application is given in Table 8.1. Similarly to the presentation of the tight integration results in Sect. 5.3, we differentiate between the automatically generated SDL model, the SDL run-time system, application-specific classes including the environment processes, operating system and device driver classes, and the gcc library.

Only the share of the SDL model and the run-time environment has to be considered for the comparison with the TinyOS S-MAC implementation, as all the other parts would be required by a similar TinyOS demonstration application, as well. As stated in [Pol05] the size of the S-MAC implementation for TinyOS requires slightly more than 6 kbytes of program memory and about 500 bytes of RAM. This result was obtained for the 8-bit ATmega128L microcontroller from Atmel. The MSP430 processor on the TMote Sky board, however, is a 16-bit microcontroller, which has the effect that the code size will be larger, particularly when manipulating 8-bit data as is often the case in the S-MAC protocol implementation.

Our results show that the automatically generated code and the necessary SDL run-time environment require more memory space (in total 18 kbytes program memory) than a native C implementation. The size of the application,

which is slightly larger than 30 kbytes, is well within the region of available memory resources of today's ultra-low-power microcontrollers. The SDL-based design approach, however, offers a fast validation by means of simulations and a formal verification of the protocol.

The memory required for the run-time environment is slightly increased compared to the results of Sect. 5.3. This comes from the fact that the S-MAC SDL model uses more data types, for example *structs*, than the simple Ping application. The code generator recognizes which data types are used in the model and includes certain functions to support these data types by defining precompiler switches. For instance, when *structs* are used in the model, a function that allows to check if two instances are equal by comparing all *struct* fields, is included. This is a drawback of the code generator as it does not recognize if such an operation is actually needed by the model.

Future Work The proposed design flow could be extended in a number of ways to support the designer and to add more benefits.

Currently, the designer still has to write some C++ source code when using the tight integration model and our cosimulation framework. This applies mainly to the handling of SDL signals. We recommend to use pre-allocated signal pools so that there is no need for dynamic memory allocation. The signal pools could be automatically generated from a list of signal names and a user-defined number of signals of each type to be pre-allocated. Furthermore, related to the cosimulation framework and the exchange of SDL signals and their parameters between the instruction set and SDL simulators, the source code responsible for copying the signals to and from the I/O module could be generated automatically based on a list of the signal types to be exchanged.

The tight integration model could be easily extended to support process or signal priorities. The SDL tool from Telelogic allows to assign such priorities in the SDL model. Our integration library would have to be adapted in such a way that the priority-based scheduler of the Reflex operating system is chosen and that the priority information in the generated code is passed as a parameter to the process wrapper instances, which are Reflex activities.

Today, protocols are typically not designed by reusing a set of previously designed and verified protocol elements. The design process could be fastened and design quality improved by creating the possibility to compose complex protocols

from atomic building blocks. Approaches in this direction have been briefly looked at in this thesis, however they are still not common practice. A future research topic could be the design of such miniprotocols in SDL and of mechanisms that enable their composability. Similarly, pre-designed hardware components for each miniprotocol could be added to a protocol design framework that facilitates modular protocol engineering.

The automatic generation of VHDL designs from SDL descriptions has been the focus of a number of researchers, and an elaborate hardware/software codesign tool for rapid prototyping of hard-real time applications has been presented by Muth in [Mut02]. This hardware compiler could be combined with our work. Ideally, optimizations that would allow to map smaller chunks than complete SDL processes into hardware should be added.

Chapter 9

Conclusions

Embedded systems, such as wireless sensor nodes or microcontrollers embedded in another device, are characterized by limited processing, memory, and energy resources, the need for reliable operation for months or years without maintenance, and the ability to communicate with other electronic devices. Applications and communication protocols developed for such platforms, therefore, must use the available energy as efficiently as possible and must not contain design errors that lead to system failures or other unexpected behavior.

In this thesis we presented a design methodology for embedded systems with this kind of requirements. The high abstraction level, formal language SDL was chosen to model system behavior because of its popularity in protocol design. SDL models can be simulated, formally verified, and transformed to C code by an automatic transformation. We explicitly do not support hard real-time systems design.

The first contribution of this thesis is a tight integration library for Telelogic's CAdvanced code generator targeting the Reflex operating system. Reflex is a real-time operating system for deeply embedded systems and has been ported to a number of 8-, 16- and 32-bit microcontrollers. We compared our approach with the existing light integration approach and achieved significant improvements. Our tight integration library is lightweight and, therefore, meets the requirements of the intended application area.

Our second contribution addresses hardware/software partitioning. We provide a cosimulation framework that allows coupling of an implementation model—consisting of software and hardware models—simulated by the TSIM instruction

set simulator with an abstract SDL simulation. This helps to identify timing bottlenecks of the implementation. The functional SDL model, which was the starting point for the generation of the implementation, is reused as a test bench for the implementation model. Identified timing bottlenecks can be tackled by developing more efficient software algorithms, using a higher clock frequency or, in some cases, by designing an optimized hardware solution. An automatic generation of VHDL from a high-level specification was outside the scope of this thesis.

Finally, we applied our integrated SDL-based design flow to the design and implementation of a complex communication protocol, the IEEE 802.15.3 wireless MAC protocol. The LEON2 processor was chosen as the target processor for the software generated from our SDL model of this protocol. With the help of the cosimulation framework we performed a hardware/software partitioning of the protocol and identified the required protocol accelerator functionality. A hardware design of this accelerator was presented in this thesis. Together with the LEON2 processor it was integrated on a single chip and manufactured at the IHP. This successful result proves the validity and effectiveness of our approach.

List of Acronyms

ARQ Automatic Repeat reQuest. ARQ mechanisms are used to initiate retransmission of lost or destroyed PDUs.

ASIC Application-Specific Integrated Circuit.

ASIP Application Specific Instruction set Processor.

BASUMA Body Area System for Ubiquitous Multimedia Applications, BMBF project (2004–2006).

CDMA Code Division Multiple Access.

CPU Central Processing Unit.

CRC Cyclic Redundancy Check. A special class of algorithms used for error detection where a binary message is treated as polynomial and is divided by a so called generator polynomial. The remainder of the division serves as error check sequence and is attached to the original message. Generator polynomials can be selected such that they detect error bursts up to a certain length.

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance.

DECT Digital Enhanced Cordless Telecommunications.

DLL Dynamic Link Library. A software module that can be dynamically linked to an application.

DMA Direct Memory Access.

DSP Digital Signal Processor.

EDA Electronic Design Automation.

FDMA Frequency Division Multiple Access.

FDT Formal Description Technique.

FIFO First In, First Out. This principle describes the operation of queues. With the FIFO scheme, the queue must return the elements in the same order as they have been written into the queue.

FPGA Field Programmable Gate Array.

FSM Finite State Machine.

GPRS General Packet Radio Service.

GSM Global System for Mobile communications.

IEEE Institute of Electrical and Electronics Engineers. It is one of the leading standardization bodies for wired and wireless communication systems through its IEEE Standards Association.

IP Internet Protocol. The packet-oriented network-layer protocol of the Internet protocol suite.

ISS Instruction Set Simulator.

MAC Medium Access Control. The MAC protocol sublayer is part of the OSI reference model [ISO94] (layer 2) and provides the protocol and control mechanisms that are required for a certain channel access method.

MPGA Mask Programmable Gate Array.

NFC Near Field Communication, a short-range wireless communication technology.

OS Operating System.

PAD Process Activity Definition.

PCB Printed Circuit Board.

PDU Protocol Data Unit. General term for the messages exchanged between peer protocol entities. PDUs of protocols in OSI layer 2 are sometimes called *frames*, in OSI layer 3 *packets*.

PNC Piconet Coordinator. Name of those device in IEEE 802.15.3 networks that transmit beacons and control the allocation of time slots in the superframe.

QoS Quality of Service.

RAM Random Access Memory.

RF Radio Frequency.

RISC Reduced Instruction Set Computer.

RTL Register Transfer Level.

SDL Specification and Description Language. A high-level specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems. It is defined by the ITU-T (formerly CCITT) Recommendation Z.100 [ITU02].

SDMA Space Division Multiple Access.

SDU Service Data Unit. In a layered protocol stack, the data passed from a higher layer protocol to the lower layer, which acts as service provider.

SoC System-on-a-Chip.

TCP Transmission Control Protocol. A connection-oriented transport protocol of the Internet protocol suite.

TDMA Time Division Multiple Access.

UML Unified Modeling Language.

UMTS Universal Mobile Telecommunications System.

VHDL VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.

WLAN Wireless Local Area Network.

WPAN Wireless Personal Area Network.

WSN Wireless Sensor Network.

Bibliography

[AAMO05] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems*, 10(1):136–156, 2005.

[ÅCH05] Karl-Erik Årzén, Anton Cervin, and Dan Henriksson. Implementation-Aware Embedded Control Systems. In William S. Levine and Dimitrios Hristu-Varsakelis, editors, *Handbook of Networked and Embedded Control Systems*. Birkhäuser, 2005.

[AHV03] Annikka Aalto, Nisse Husberg, and Kimmo Varpaaniemi. Automatic Formal Model Generation and Analysis of SDL. In Rick Reed and Jeanne Reed, editors, *SDL Forum*, volume 2708 of *Lecture Notes in Computer Science*, pages 285–299. Springer, 2003.

[ART08] ARTIST Network of Excellence on Embedded Systems Design. http://www.artist-embedded.org/, 2008.

[ASI+98] Arthur Abnous, Katsunori Seno, Yuji Ichikawa, Marlene Wan, and Jan M. Rabaey. Evaluation of a Low-Power Reconfigurable DSP Architecture. In *IPPS/SPDP Workshops*, pages 55–60, 1998.

[AUT08] AUTOSAR Homepage. http://www.autosar.org, 2008.

[AZW+02] Arthur Abnous, Hui Zhang, Marlene Wan, George Varghese, Vandana Prabhu, and Jan Rabaey. *The Application of Programmable DSPs in Mobile Communications*, chapter The Pleiades Architecture. John Wiley & Sons, Ltd, 2002.

[BAS06] IHP GmbH. BASUMA - Body Area System for Ubiquitous Multimedia Applications. http://www.basuma.de, 2006.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, and Attila Jurecska. *Hardware-Software Co-design of Embedded Systems: the POLIS Approach*. Kluwer Academic Publishers, 1997.

[BDHS00] Dragan Bošnački, Dennis Dams, Leszek Holenderski, and Natalia Sidorova. Model Checking SDL with Spin. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2000.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. IF: An intermediate representation for SDL and its applications. In *SDL Forum*, pages 423–440, 1999.

[Bou07] Yves Bourgeois. Design and Verification of Concurrent Real-Time Systems using SDL and MSC. Master's thesis, Delft University of Technology, July 2007.

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45–52, April 2003.

[CCH+99] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew J. McNelly, and Lee Todd. *Surviving the SOC revolution: a guide to platform-based design*. Kluwer Academic Publishers, 1999.

[Cel05] Survey of System Design Trends. Technical report, Celoxica Inc., 2005.

[Cin07] Cinderella ApS. Cinderella SDL. http://www.cinderella.dk, 2007.

[Cla85] David D. Clark. The structuring of systems using upcalls. *ACM SIGOPS Operating Systems Review*, 19(5):171–180, 1985.

[Cor06] Moteiv Corporation. TMote Sky datasheet. http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf, November 2006.

[DB89] Piotr Dembinski and Stanislaw Budkowski. Specification language Estelle. In Michel Diaz, Jean-Pierre Ansart, Jean-Pierre Courtiat, Pierre Azema, and Vijaya Chari, editors, *The formal description technique Estelle*, pages 35–75. North-Holland, 1989.

[DBDK04] Daniel Dietterle, Irina Babanskaja, Kai F. Dombrowski, and Rolf Kraemer. High-Level Behavioral SDL Model for the IEEE 802.15.3 MAC Protocol. In Peter Langendörfer, Mingyan Liu, Ibrahim Matta, and Vassilios Tsaoussidis, editors, *Proc. Wired/Wireless Internet Communications*, volume 2957 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2004.

[DEK07] Daniel Dietterle, Jean-Pierre Ebert, and Rolf Kraemer. A hardware accelerated implementation of the IEEE 802.15.3 MAC protocol. In Luis Orozco-Barbosa, Teresa Olivares, Rafael Casado, and Aurelio Bermúdez, editors, *IFIP WG 6.8 First International Conference on Wireless Sensor and Actor Networks (WSAN'07)*, volume 248 of *IFIP International Federation for Information Processing*, pages 215–226. Springer, September 2007.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *IEEE Conference on Local Computer Networks*, pages 455–462. IEEE Computer Society, 2004.

[Die07] Daniel Dietterle. Hardware-Protokollbeschleuniger für eine Verbindungssicherungs-Protokollebene eines Senderempfängers. Deutsche Patentanmeldung, Januar 2007.

[Die08] Daniel Dietterle. Embedded System Protocol Design Flow Based on SDL: From Specification to Hardware / Software Implementation. In *Proceedings 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008)*. ACM, March 2008.

[DMTS00] Matthias Dörfel, Andreas Mitschele-Thiel, and Frank Slomka. CORSAIR: HW/SW-Codesign von Kommunikationssystemen mit SDL.

PIK - Praxis der Informationsverarbeitung und Kommunikation, 23(1):3–13, 2000.

[Dol00] Michael Dolinsky. High-Level Design of Embedded HardwareSoftware Systems. *Advances in Engineering Software*, 31(3):197–201, March 2000.

[DSH99] Matthias Dörfel, Frank Slomka, and Richard Hofmann. A Scalable Hardware Library for the Rapid Prototyping of SDL Specifications. In *IEEE International Workshop on Rapid System Prototyping*, pages 120–125, 1999.

[dV02] Bianca de Vree. Formal control used on requirements analysis of MARS. Applying formal methods in UML context to the specification of requirements of real-time systems. Master's thesis, Katholieke Universiteit Nijmegen, 2002.

[dW04] Nico de Wet. *Model Driven Communication Protocol Engineering and Simulation-Based Performance Analysis Using UML 2.0.* PhD thesis, University of Cape Town, December 2004.

[DZM01] Christos Drosos, Michel Zayadine, and Dimitris Metafas. Embedded real-time communication protocol development using SDL for ARM microprocessor. *Dedicated Systems Magazine*, January 2001.

[EDF+07] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, and Thiemo Voigt. MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007.

[FDT95] Joachim Fischer, Evgeni Dimitrov, and Udo Taubert. Analysis and formal Verification of SDL'92 Specifications using Extended Petri Nets. Technical report, Humboldt Universität zu Berlin, 1995.

[FHvLP00] Joachim Fischer, Eckhardt Holz, Martin von Löwis, and Andreas Prinz. SDL-2000: A Language with a Formal Semantics. In *The Third Workshop on Rigorous Object-Oriented Methods (ROOM 2000)*, January 2000.

[FL98] Stefan Fischer and Stefan Leue. Formal Methods for Broadband and Multimedia Systems. *Computer Networks & ISDN Systems*, 9-10(30):865–899, 1998.

[FN01] Laura Marie Feeney and Martin Nilsson. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1548–1557, 2001.

[Gai04] Gaisler Research AB. TSIM Simulator User's Manual. Version 1.3. http://www.gaisler.com, March 2004.

[Gai05] Gaisler Research AB. LEON2 Processor User's Manual. http://www.gaisler.com, July 2005.

[GGK07] Reinhard Gotzhein, Rüdiger Grammes, and Thomas Kuhn. Specifying Input Port Bounds in SDL. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007: Design for Dependable Systems, 13th International SDL Forum, Proceedings*, volume 4745 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 2007.

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Oscar Nierstrasz, editor, *ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993.

[GLM02] Hubert Garavel, Frèdèric Lang, and Radu Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.

[Hän02] Marko Hännikäinen. *Design of Quality of Service Support for Wireless Local Area Networks*. PhD thesis, Tampere University of Technology, 2002.

[HBB04] Malek Haroud, Ljubica Blažević, and Armin Biere. HW accelerated ultra wide band MAC protocol using SDL and SystemC. In *Proc. IEEE Radio and Wireless Conference (RAWCON'04)*, pages 525–528, 2004.

[HHL+01] Jörg Hintelmann, Richard Hofmann, Frank Lemmen, Andreas Mitschele-Thiel, and Bruno Müller-Clostermann. Applying techniques and tools for the performance engineering of SDL systems. *Computer Networks*, 35(6):647–665, 2001.

[HHM98] Holger Hermanns, Ulrich Herzog, and Vassilis Mertsiotakis. Stochastic Process Algebras – Between LOTOS and Markov Chains. *Computer Networks*, 30(9-10):901–924, 1998.

[HKHS00] Marko Hännikäinen, Jarno Knuutila, Timo Hämäläinen, and Jukka Saarinen. Using SDL for Implementing a Wireless Medium Access Control Protocol. In *IEEE International Symposium on Multimedia Software Engineering (MSE2000)*, pages 229–236, 2000.

[HML03] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with LISA*. Springer-Verlag, 2003.

[HMTKL96] Ralf Henke, Andreas Mitschele-Thiel, Hartmut König, and Peter Langendörfer. Automated Derivation of Efficient Implementations from SDL Specifications. Technical report, FIN/IRB, Otto-von-Guericke-Universität Magdeburg, 1996.

[Hoa78] Charles A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hog89] Dieter Hogrefe. *Estelle, LOTOS und SDL: Standard Spezifikationssprachen für verteilte Systeme*. Springer-Verlag, Berlin, 1989.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[HP85] David Harel and Amir Pnueli. *Logics and models of concurrent systems*, chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, Inc., 1985.

[HS96] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer-Verlag, 1996.

[HS07] Thomas A. Henzinger and Joseph Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, 40(10):32–40, October 2007.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[IAJ94] Tarek Ben Ismail, Mohamed Abid, and Ahmed Amine Jerraya. COSMOS: a codesign approach for communicating systems. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, pages 17–24. IEEE Computer Society, 1994.

[IEE99] *IEEE Std 802.11-1999 (ISO/IEC 8802-11: 1999), IEEE Standards for Information Technology–Telecommunications and Information Exchange between Systems–Local and Metropolitan Area Network–Specific Requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* IEEE Standards Association, 1999.

[IEE03a] *IEEE Std 802.15.3-2003, IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 15.3: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for High Rate Wireless Personal Area Networks (WPAN).* IEEE Standards Association, 2003.

[IEE03b] *IEEE Std 802.15.4-2003, IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks– Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer*

(PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs). IEEE Standards Association, 2003.

[IEE05] *IEEE Std 802.15.1-2005, IEEE Standard for Information Technology–Telecommunications and Information Exchange Between Systems–Local and Metropolitan Area Networks– Specific Requirements Part 15.1: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANS)*. IEEE Standards Association, 2005.

[Ins07] Texas Instruments. True System-on-Chip with Low Power RF Transceiver and 8051 MCU (Rev. F). Datasheet available from http://focus.ti.com/docs/prod/folders/print/cc1110f32.html, November 2007.

[ISO] Information Processing Systems–Open System Interconnection, ISO International Standard 9074: Estelle - A Formal Description Technique Based on an Extended State Transition Model.

[ISO91] ISO, OSI Conformance Testing Methodology and Framework, IS 9646, 1991.

[ISO94] ISO/IEC 7498-1:1994(E), Information Technology–Open Systems Interconnections–Basic Reference Model: The Basic Model, 1994.

[ISO97] ISO, Formal Methods for Conformance Testing, IS 13245, 1997.

[ITR05] International Technology Roadmap for Semiconductors. 2005 Edition. Technical report, ITRS, 2005.

[ITU02] International Telecommunication Union. ITU-T Recommendation Z.100: Specification and Description Language (SDL), 2002.

[ITU04] International Telecommunication Union. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 2004.

[JG01] Guoping Jia and Susanne Graf. Verification experiments on the MASCARA protocol. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 123–142, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[Kal95] Asawaree Kalavade. *System-Level Codesign of Mixed Hardware-Software Systems*. PhD thesis, Dept. of EECS, University of California, Berkeley, 1995.

[KB03] Hermann Kopetz and Günter Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[KK96] Hartmut König and Heiko Krumm. Implementierung von Kommunikationsprotokollen. *Informatik Spektrum*, 19(6):316–325, 1996.

[KMN+00] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19(12), December 2000.

[Kön03] Hartmut König. *Protocol Engineering. Prinzip, Beschreibung und Entwicklung von Kommunikationsprotokollen*. B. G. Teubner Verlag, 2003. in German.

[Kur97] Robert P. Kurshan. Formal Verification in a Commercial Setting. In *Design Automation Conference*, pages 258–262, 1997.

[KV04] Klaus Kronlöf and Nikolaos S. Voros. *System Level Design Model with Reuse of System IP*, chapter System Design Practices in Industry Today, pages 5–11. Springer, 2004.

[LC02] Bo Lincoln and Anton Cervin. Jitterbug: A Tool for Analysis of Real-Time Control Performance. In *Proceedings of the 41st IEEE Conference on Decision and Control*, December 2002.

[Leu95] Stefan Leue. Specifying Real-Time Requirements for SDL Specifications–A Temporal Logic-Based Approach. In *Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 19–34, 1995.

[LIS07] LISA–Language for Instruction Set Architectures. http://servus.ert.rwth-aachen.de/lisa/, 2007.

[LK99] Peter Langendörfer and Hartmut König. COCOS - A configurable SDL compiler for generating efficient protocol implementations. In Rachida Dssouli, Gregor von Bochmann, and Yair Lahav, editors, *SDL Forum*, pages 259–274. Elsevier, 1999.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[LMP+05] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. *Ambient Intelligence*, chapter TinyOS: An Operating System for Wireless Sensor Networks. Springer-Verlag, 2005.

[LO96] Stefan Leue and Philippe A. Oechslin. On parallelizing and optimizing the implementation of communication protocols. *IEEE/ACM Trans. Netw.*, 4(1):55–70, 1996.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, December 1997.

[LVL03] Marisa López-Vallejo and Juan Carlos López. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):269–297, 2003.

[Mal99] Mazen Malek. PerfSDL: Interface to protocol performance analysis by means of simulation. In *SDL Forum*, pages 441–456, 1999.

[MAR07] The official OMG MARTE Web site. Modeling and Analysis of Real-time and Embedded systems. http://www.omgmarte.org/, August 2007.

[Mat07] The Mathworks. Simulink® - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/, 2007.

[MdCP00] Jean-Pierre Moreau, Philippe di Crescenzo, and Lloyd Pople. Hardware software system codesign based on SDL/C specifcations. *Microelectron. Eng.*, 54(1-2):181–191, 2000.

[MED03] The MEDEA+ Design Automation Roadmap. Technical report, MEDEA, 2003.

[Met07] Metropolis: Design Environment for Heterogeneous Systems. http://www.gigascale.org/metropolis/, 2007.

[MF00] Annette Muth and Georg Färber. SDL as a System Level Specification Language for Application-Specific Hardware in a Rapid Prototyping Environment. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS'00)*, pages 157–162. IEEE Computer Society, 2000.

[MHA+02] C.A.M. Marcon, F. Hessel, A.M. Amory, L.H.L. Ries, F.G. Moraes, and N.L.V. Calazans. Prototyping of embedded digital systems from SDL language: a case study. In *Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop (HLDVT'02)*, pages 133–138. IEEE Computer Society, 2002.

[MIJ03] Marius Minea, Cornel Izbaşa, and Calin Jebelean. Experience with Formal Verification of SDL Protocols. *Computing*, 2(3), 2003.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mut02] Annette Muth. *SDL-based Design of Application Specific Hardware for Hard Real-Time Systems*. PhD thesis, Chair of Real-Time Computer Systems, Technical University Munich, 2002.

[Nol09] Jörg Nolte. Reflex - Realtime Event FLow EXecutive. http://www-bs.informatik.tu-cottbus.de/index.php?id=38&L=2, 2009.

[ÖBE+97] Achim Österling, Thomas Benner, Rolf Ernst, Dirk Herrmann, Thomas Scholz, and Wei Ye. *Hardware/Software Co-Design: Principles and Practice*, chapter The COSYMA System. Kluwer, 1997.

[OHC07] Martin Ohlin, Dan Henriksson, and Anton Cervin. *TrueTime 1.5—Reference Manual*, January 2007.

[PBM+04] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J.S. Baras. ATEMU: a fine-grained sensor network simulator. In *First Annual IEEE Com-*

munications Society Conference on Sensor and Ad Hoc Communications and Networks (IEEE SECON 2004), pages 145–152, 2004.

[Pol05] Joseph Robert Polastre. *A unifying link abstraction for wireless sensor networks*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2005. Adviser-David E. Culler.

[Pop06] Miroslav Popovic. *Communication Protocol Engineering*. CRC Press, 2006.

[PS91] Robert L. Probert and Kassem Saleh. Synthesis of Communication Protocols: Survey and Assessment. *IEEE Trans. on Computers*, 40(4):468–476, 1991.

[Ram07] Ulrich Ramacher. Software-Defined Radio Prospects for Multistandard Mobile Phones. *IEEE Computer*, 40(10):62–69, October 2007.

[RB98] Franz Regensburger and Aenne Barnard. Formal Verification of SDL Systems at the Siemens Mobile Phone Department. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 439–455, London, UK, 1998. Springer-Verlag.

[RS82] Anders Rockström and Roberto Saracco. SDL–CCITT Specification and Description Language. *IEEE Transactions on Communications*, COM-30(6):1310–1318, June 1982.

[SDP+07] Zoran Stamenković, Daniel Dietterle, Goran Panić, Wojtek Bocer, Gunter Schoof, and Jean-Pierre Ebert. MAC Processor for BASUMA Wireless Body Area Network. In J. G. Delgado-Frias, editor, *Proceedings of the 5th IASTED International Conference on Circuits, Signals and Systems*, pages 47–52, 2007.

[SHH02] Tuomo Saari, Marko Hännikäinen, and Timo Hämäläinen. Hardware Acceleration of Wireless LAN MAC Functions. In *IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS2002)*, pages 398–401, 2002.

[SS01] Natalia Sidorova and Martin Steffen. Verifying Large SDL-Specifications Using Model Checking. In *Proceedings of the 10th International SDL Forum Copenhagen*, volume 2078 of *Lecture Notes In Computer Science*, pages 403–420, London, UK, 2001. Springer-Verlag.

[SVDN07] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded System Design for Automotive Applications. *IEEE Computer*, 40(10):42–51, October 2007.

[Svo89] Liba Svobodova. Implementing OSI Systems. *IEEE Journal on Selected Areas in Communication*, 7(7):1115–1130, 1989.

[Sys05] IEEE Computer Society. IEEE Standard SystemC®Language Reference Manual. IEEE Std 1666™-2005, 2005.

[Sys07] OMG Systems Modeling Language, September 2007.

[Tel06] Telelogic AB. Telelogic TAU 4.6 User's Manual, 2006.

[Tes] Proceedings of the IFIP International Conference on Testing Communicating Systems (TestCom), formerly IWTCS and IWPTS.

[Tex06] Texas Instruments. MSP430x1xx Family User's Guide. http://focus.ti.com/lit/ug/slau049f/slau049f.pdf, 2006.

[TTT07] TTTech website. http://www.tttech.com, 2007.

[V+06] Stamatis Vassiliadis *et al.* The HiPEAC Roadmap on Embedded Systems. Technical report, European Network of Excellence on High-Performance Embedded Architecture and Compilation, 2006.

[VdJ07] Eric Verhulst and Gjalt G. de Jong. OpenComRTOS: An Ultra-Small Network Centric Embedded RTOS Designed Using Formal Modeling. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007: Design for Dependable Systems, 13th International SDL Forum, Proceedings*, volume 4745 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007.

[vDL03] Tijs van Dam and Koen Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *1st ACM Conf. on*

Embedded Networked Sensor Systems (SenSys 2003), pages 171–180, 2003.

[WDEK06] Gerald Wagenknecht, Daniel Dietterle, Jean-Pierre Ebert, and Rolf Kraemer. Transforming Protocol Specifications for Wireless Sensor Networks into Efficient Embedded System Implementations. In Kay Römer, Holger Karl, and Friedemann Mattern, editors, *Proceedings of the Third European Workshop on Wireless Sensor Networks*, volume 3868 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2006.

[WFGG04] Christian Webel, Ingmar Fliege, Alexander Geraldy, and Reinhard Gotzhein. Developing Reliable Systems with SDL Design Patterns and Design Components. In *ISSRE04 Workshop on Integrated-reliability with Telecommunications and UML Languages (WITUL)*, 2004.

[WKSN08] Karsten Walther, Reinhardt Karnapke, André Sieber, and Jörg Nolte. Using Preemption in Event Driven Systems with a Single Stack. In *The Second International Conference on Sensor Technologies and Applications*, Cap Esterel, France, 2008.

[WN06] Karsten Walther and Jörg Nolte. Event-Flow and Synchronization in Single Threaded Systems. In *First GI/ITG Workshop on Non-Functional Properties of Embedded Systems (NFPES)*, 2006.

[WN07] Karsten Walther and Jörg Nolte. A Flexible Scheduling Framework for Deeply Embedded Systems. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 784–791. IEEE Computer Society, 2007.

[YHE02] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *21st Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1567–1576, 2002.

[ZDSV+06] Haibo Zeng, Abhijit Davare, Alberto Sangiovanni-Vincentelli, Sampada Sonalkar, Sri Kanajan, and Claudio Pinello. Design Space

Exploration of Automotive Platforms in Metropolis. In *Society of Automotive Engineers Congress*, April 2006.

[ZHT93] Belhassen Zouari, Serge Haddad, and Mohamed Taghelit. A Protocol Specification Language with a High-Level Petri Net Semantics. In *Decentralized and Distributed Systems*, pages 225–241, 1993.

[ZMDJ98] N.E. Zergainoh, G.F. Marchioro, J.M. Daveau, and A.A. Jerraya. Using SDL for Hardware/Software Co-design of an ATM Network Interface Card. In *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*, June 1998.

Curriculum Vitae

Daniel Dietterle received his Dipl.-Inf. degree in Computer Science from the Brandenburg University of Technology at Cottbus, Germany, in 2002. In the same year he joined the IHP GmbH in Frankfurt (Oder) as a research assistant. His research work was mainly focused on the design and implementation of low-power wireless communication systems. The developed protocol design methodology and the respective results are presented in this thesis.