Generating Mixed Hardware/Software Systems from SDL Specifications

Frank Slomka, Matthias Dörfel, Ralf Münzenberger
University of Erlangen-Nuremberg
Department of Computer Networks and Communication Systems
Martensstr. 3, 91058 Erlangen, Germany
{slomka, doerfel, muenzenberger}@informatik.uni-erlangen.de

Abstract

A new approach for the translation of SDL specifications to a mixed hardware/software system is presented. Based on the computational model of communicating extended finite state machines (EFSM) the control flow is separated from data flow of the SDL process. Hence for the first time it is possible to generate a mixed hardware/software implementation of an SDL process. This technique also reduces the complexity for high-level and register-transfer synthesis tools for the hardware parts of the system. The advantage of this methodology is shown by a design example of a wireless communication chip.

1 Introduction

Since several years the Specification and Description Language SDL is widely used for the specification of large distributed communication systems. SDL is popular in the communication industry because it provides language constructs needed for protocol development and systems engineering. The computational model of SDL is based on asynchronous communication between extended finite state machines which allows the specification of distributed concurrent embedded systems. The increasing communication aspect of modern automotive systems makes SDL also feasible for the design of car control systems.

SDL allows the designer to specify electronic systems independent from the implementation. Communication systems like routers, switches and the complete mobile telephone network are typical hardware/software systems. Therefore SDL becomes popular as a language for hardware/software co-design. To support the co-design process some tools have been developed to generate automatically VHDL descriptions from SDL specifications in addition to commercial software code generators. Integrating these tools together is only possible when the partitioning is performed on the grain level of processes.

In common cases the designer has this in mind during the specification phase. As a result the designer intuitively creates small SDL processes with less functionality because he takes into consideration that the specification will be partitioned. Such a disadvantageous approach leads to a large communication overhead after synthesis.

After partitioning the SDL processes are synthesized separately. The current approach for hardware generation is to translate each EFSM to one VHDL process and to connect this process to a run-time support system (RTSS) which implements the communication. The resulting hardware description contains a complete finite state machine with data operations which is difficult to translate by high-level synthesis to the register transfer level. Mixing a large control flow dominated description with only a few data operations leads to inefficient results of the high-level synthesis. On the other hand in dataflow dominated applications the control-structure of SDL leads to a large amount of overhead.

As a solution for the aforementioned problems this paper presents an integrated approach for the translation of SDL specifications to mixed hardware/software systems. First the explicitly formulated control-flow of the state machine is separated from the data operations of the transitions. The separate control flow can be translated to a software or hardware description. The hardware description can be given at register transfer level because the scheduling of the operations is already defined. In the next step every SDL transition is translated on its own. Using separate VHDL processes for the transitions with complex data operations simplifies high-level synthesis. Resource sharing between the different transitions is supported by multi-process high-level synthesis.

Using this approach it is possible to perform a fine grain hardware/software partitioning where the control-flow is performed by software and high-performance data-operations are performed by hardware.

2 Related Work

Translating designs described in SDL [10] to implementation languages was first made for software. The commercially available case tool TAU [16] allows the generation of C and Chill code. The main application area for TAU is the emulation of the SDL system on a host computer. New code generators of the TAU framework also allow the implementation of embedded software systems. But when performing an integration on commercial real-time operating systems a lot of SDL constructs are not supported by these code generators.

The TAU framework only supports the server model approach when generating an implementation. During the last

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG) under grant He 1408/4-3.

years a new application model has become more popular: the activity thread model [11] which eliminates signal queues from time critical paths of the SDL specification. Tools which translate hardware implementations from SDL are generally using the server model. With the server model approach the EFSM is directly translated to hardware while the communication model is selected from a library [9].

In [3] the control flow is also separated from the data flow while for the transitions an application independent data processor is used which was manually designed before. For many cases the application independency of the data processor leads to an unnecessary amount of gates.

The SDL2VHDL tool described in [2] allows the generation of VHDL code for different abstraction levels. For SDL processes with a high amount of computational data operations behavioral VHDL is generated. Automatically this behavioral description can be connected to a hardware RTSS [6] which is described at RT-level or which can already be presynthesized. On the other hand the hardware of control flow dominated processes can be synthesized directly from RTL VHDL descriptions which can also be connected to the RTSS. In [13] and [14] an extension of the SDL2VHDL code generator is described. This extension allows the automatic implementation of the activity thread model in hardware. The generation of software parts is not supported by these tools. To support hardware/software co-design they are used with commercial code generators as described in [8] and [14].

The only integrated framework to generate hardware and software from a SDL specification is COSMOS. It is based on an intermediate system description format called SOLAR. The framework allows the designer to split and to merge SDL processes before starting the code generation process but does not allow to split control- from data-flow or the use of different synthesis techniques after partitioning. The hardware translator of COSMOS was described in [4] while a case study is presented in [5]. In this case study a partitioning of the SDL process is not considered which leads to the disadvantages of the discussed approach.

3 Generating Mixed HW and SW Implementations from SDL

3.1 Specification and Description Language

The language SDL was originally designed for the specification of protocol automata in communication systems. These automata are extended by data operations which leads to the model of an extended finite state machine. The computational model of SDL is based on asynchronous communication between the EFSM. Additionally to the specification of communicating automata SDL allows to describe the module structure of a system. Each system can be partitioned into blocks, subblocks, processes, and services. Each process or service contains an independent EFSM and each process has an own FIFO input queue. The communication structure of the system is described by channels between blocks and signal routes between processes or services. The communication is performed by sending a signal from one process to another.

As mentioned before in SDL behavior can be described with processes and with services. While all processes may run in parallel, services are part of a process and are executed sequentially. Each transition of a EFSM is triggered by the receipt of an asynchronous signal, the change of a process local variable (continuous signal) or both of them (enabling condition). Using services and continuous signals it is possible to specify synchronous communication inside processes. These features are important for the design of real-time systems with SDL.

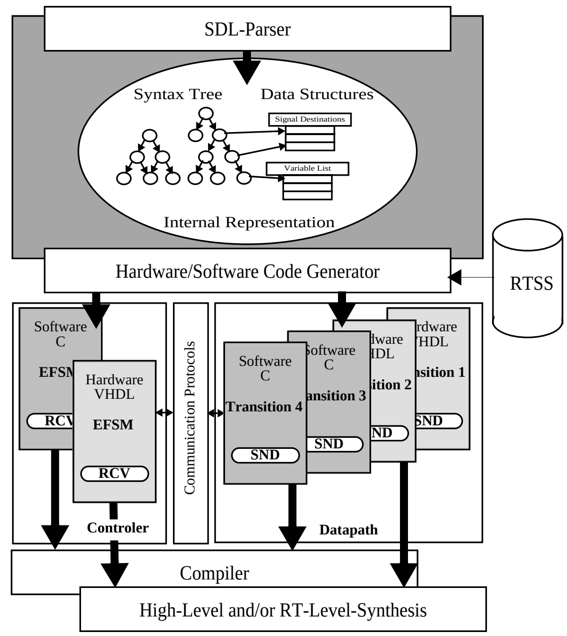

Figure 1: The SDL co-design code generator

3.2 Architecture of the Tool

In fig. 1 the architecture of the integrated co-design code generator for the implementation synthesis of SDL specifications is shown. The SDL specification is translated by a Java parser to an internal representation. The syntax tree is based on Java classes generated by the tree generator JavaTree. For the semantic analysis of SDL some additional Java classes are implemented to store e.g. variable lists, signal identifiers, signal destination lists and process identifiers.

From that internal representation the software and the hardware parts are generated. The code generator uses a template which defines the structure of the final C or VHDL description. We have defined different templates for C and VHDL to support different implementation strategies and to be flexible to change the implementation strategies in the future. This concept also allows to support the different code generation strategies described in [2]. The hardware/software code generator only has to emit code for the application specific behavior of the SDL system. The application independent functionality like inter process communication or process scheduling is encapsulated in the RTSS which is connected to the application by procedure or function calls.

3.3 Generating Hardware

Generating Hardware from SDL specifications means to translate the structural and behavioral SDL specification to a hardware description language like VHDL or Verilog. Earlier proposals as discussed in section 2 generate one VHDL process for each SDL process. In the following we will call this the monolithic approach. Normally the behavioral part of the SDL specification is translated by system level synthesis to a controller and a datapath which can be synthesized to a gate

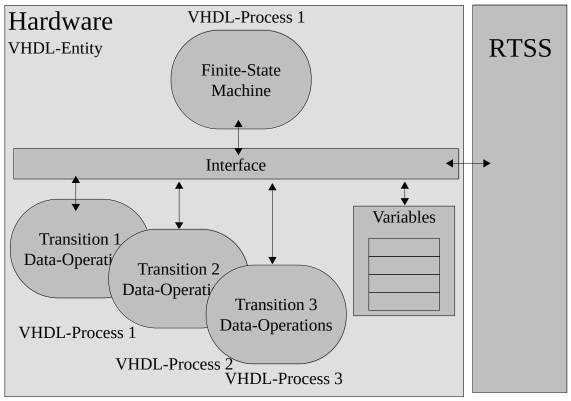

Figure 2: Hardware architecture of a EFSM

level circuit. The RTSS system comes from a library and contains VHDL code at register transfer level.

The new approach of generating hardware from SDL follows the philosophy to separate control flow from data flow. Therefore each SDL process is translated to at least two VHDL processes. The SDL state machine is translated to a separate VHDL state machine and for each SDL transition a separate VHDL process is generated. The finite state machine of the process will directly be synthesized by RTL synthesis to a controller engine while the transitions containing data operations will be synthesized by high-level synthesis or by RTL synthesis depending on the number of data operations. The communication between all VHDL processes is performed by a small communication interface which also contains shared memory to store the variables of the SDL process. In fig. 2 the architecture of the new approach is shown. The complete SDL process is connected to a RTSS module as described in [6].

3.4 Generating Software

Generating software from SDL is a well known area. The SDL extended finite state machine is translated to a case- or if-structure of a high-level programming language and the computational model is supported by an operating system with processes and asynchronous message passing. Another approach to translate SDL to software is table driven: Each transition of the state machine is encapsulated in a function. The identifier of the next signal and the state of the SDL process are indices in a two dimensional array which contains references to the functions implementing the transitions.

In dependency from the application area of the final system the documentation of TAU [16] differentiates between two implementation strategies: One where all SDL processes are located in a single operating system process and the communication and scheduling of processes is managed by a RTSS also belonging to this process, and another strategy where one operating system process is used for each SDL process. In the second case the complete RTSS is implemented by system calls to the operating system. While TAU does not support important language constructs like continuous signal or enabling conditions when mapping SDL to a real-time operating system, the SDL2VHDL code generator was extended to generate software implementations especially for embedded real-time systems. Such constructs are very useful to perform synchronous communication, which reduces the overall communication overhead.

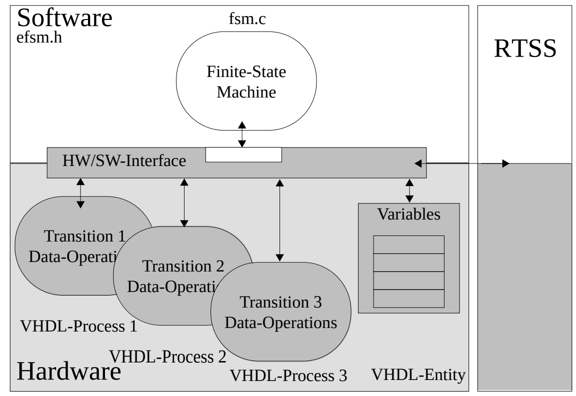

Figure 3: HW/SW Partitioning: Control Flow in SW

3.5 Mixing Hardware and Software

Using the new concept it is possible to mix hardware and software on a fine level of granularity. A software implementation for the state machine of the SDL process can be implemented while the transitions will be implemented on an application specific coprocessor which was synthesized by high-level synthesis. Using a new approach for multi-process high-level-synthesis as described in [1] it is possible to share resources of the hardware over all transitions. In fig. 3 and fig. 4 two different mixed implementations are shown. In fig. 3 only the state machine is implemented in software while in fig. 4 also a transition is implemented in software. All communication between the different implementation entities is supported by a generic hardware/software interface described in the next section of the paper.

4 Run-Time Support System

4.1 Hardware Interfaces and Hardware Library

The RTSS for SDL processes implemented in hardware is built as described in [6]. Scalable modules containing message queues, timer modules, and communication infrastructure are instantiated by the compiler. For data passing the SDL compiler emits procedure calls which are resolved by a behavioral compiler. As an alternative in case of not using high-level synthesis, the compiler is also able to emit code fragments given

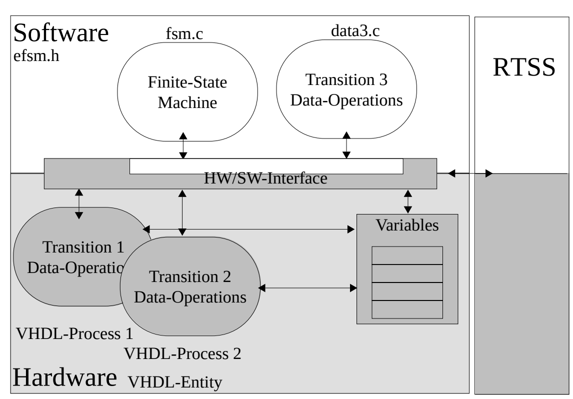

Figure 4: HW/SW Partitioning: Mixed Implementation

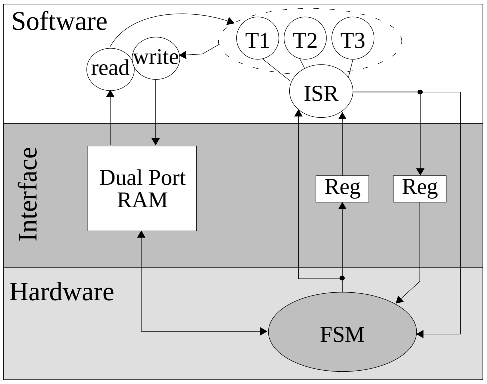

Figure 5: SW/HW Interface

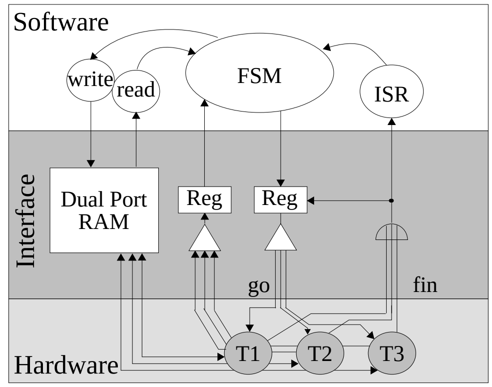

Figure 6: HW/SW-Interface

as templates, e.g. the body of the aforementioned procedures themselves.

4.2 Software Interfaces and Real-Time Operating-System

To implement the RTSS the code generator focuses on the real-time operating system RTEMS [15] which is available for a broad range of processors and target boards. The limitation to a single kernel allows a strong optimization of the generated code and the exploration of new implementation strategies for relevant SDL constructs.

To implement an SDL specification as a real-time system every SDL process is started as an own operating system process. Further the communication mechanisms of the real-time operating system like message queues or asynchronous signals are used for the implementation of the communication model of SDL. Hence if a waiting process receives a signal in his queue the operating system puts the process into the ready state and schedules it according to its priority and the state of other processes.

The handling of synchronous communication by using the construct continuous signal in conjunction with variables shared between SDL processes is a weak point of state-of-the-art code generators. In a real-time operating system the state of a process can only be changed in an active way triggered by a system call or by an interrupt. So for the aforementioned construct every write access to the shared variable has to be encapsulated by the code generator with a wrapper function. Inside the wrapper function the kernel is called which itself calls a continuous signal handler which was installed by the waiting process. If the condition is fulfilled the waiting process is activated.

4.3 Hardware/Software Interfaces

When implementing a SDL process in hardware and software the RTSS has to be accessible from both parts. The communication and synchronization between the hardware and the software entities of the process is implemented by a generic hardware/software interface. Two alternative implementations are discussed: the control flow is implemented in hardware or the control flow is implemented in software. For sharing the variables of the process the interface contains a dual port memory.

For the implementation of the state machine in hardware a generic structure like the one shown in fig. 5 is used. After reading a signal from the hardware RTSS the state machine places a transition code in a communication register. The write access to the register triggers an interrupt on the processor. The processor executes an interrupt service routine (ISR) which reads the communication register and activates the according transition. The end of the transition is signalled back to the controlling state machine by another communication register and the ISR finishes. The content of the second register is used to calculate the new state of the state machine. Using a processor with hardware supported context switching like the MSPARC described in [12] the execution delay after activation of a transition can be minimized.

In fig. 6 the state machine is implemented in software and the transitions are implemented as application specific hardware. The software state machine is activated by a non-empty queue by means of the scheduler of the operating system. According to the current state and the received signal a transition code is written into a register implemented in the interface. After decoding the content the appropriate transition is started. At the end of the transition the transition code register is cleared by the hardware, the code for the next state is placed into a second register and an interrupt is generated. The software state machine is activated again by the ISR.

4.4 Connecting External Components

To integrate ready designed intellectual property (IP) cores for different application areas, e.g. analog line coders or physical data modems, a hardware component was designed which allows asynchronous signal communication with the IP cores. To instantiate IP cores it is possible to assign them to a gate of the SDL specification. The compiler recognizes the instantiation of the IP core and generates an interface component which is parametrized with attributes of the IP core given in the library. The sending of a signal to the gate is converted to the appropriate access of the interface component.

In case an IP core has an addressable memory-like interface every signal has its own address in the memory space. For every signal sent to the gate the signal header and parameters are written to this memory address. This concept is implementation independent: It can be used by a software SDL system to access a component connected to the processor bus or by a hardware SDL system based on an application-specific communication bus. For asynchronous communication based on shared variables an IP core can also export register contents. The communication interface is performed as shown in fig. 5.

5 Results

The two different approaches for the code generation of SDL processes have been compared for a small typical example from the application area of real-time communication systems: The monolithic approach leads to an implementation of the datapath which needs 2306 gates on an ALTERA FLEX10K100 device and a controller with 54 states.

Using the new approach the design was split in processes for the data transitions and a separate RTL description for the state machine. The synthesized datapath results in 2153 gates plus 28 additional gates for the controller of the data path while the state machine has 30 states.

6 Co-Design of a DECT MAC-Layer Chip

To demonstrate the usability of the approach for the specification, design, implementation, and validation of a real co-design problem we consider a burst mode controller chip for the wireless communication system DECT. The DECT example was used because the system behavior is well known and we already have experience developing DECT systems. Additionally DECT is very similar to GSM or UMTS so the case study covers most problems which will appear during the development of future mobile communication systems. The case study only considers the main functions of a DECT fixed part to demonstrate how the approach works. Functions like connection management, measurement of radio signal strength, and handover algorithms are not considered yet. The main focus of the case study is to show how hardware and software for real-time systems can be developed keeping a single description for the whole system and how the implementation can be generated automatically.

6.1 DECT system architecture

The DECT protocol specification describes layer one, two and three of the OSI reference model for wireless speech and data services and supports high traffic loads. It is designed to provide large cordless private branch exchange (PBX) installations or wireless LANs and is also available for domestic consumers. The radio fixed parts (base stations) of the system are connected to a PBX via a standard telephone line interface (analog or digital). Portable parts, mobile phones or laptops with a DECT interface are communicating with the telephone system via the air interface. DECT supports seamless handover to change the radio fixed part being connected to a portable part, or to switch the radio channel, if the quality of a connection gets worse. This may occur when the user leaves the area of the current radio fixed part or the radio channel is interfered by other radio signals. To support high traffic load, the air interface is realized with time division multiplex access (TDMA) where each 10 ms frame is subdivided into 24 time slots. Using 12 time slots for the uplink and 12 time slots for the downlink, DECT supports 120 logical channels on 10 radio frequencies.

6.2 SDL Design Specification

The design of the DECT fixed part burst mode controller starts with an abstract specification using SDL which describes the DECT MAC layer [7] shown in fig. 7. The design is split in six different SDL processes. The process Lower Layer Management Entity (LLME) manages the complete MAC

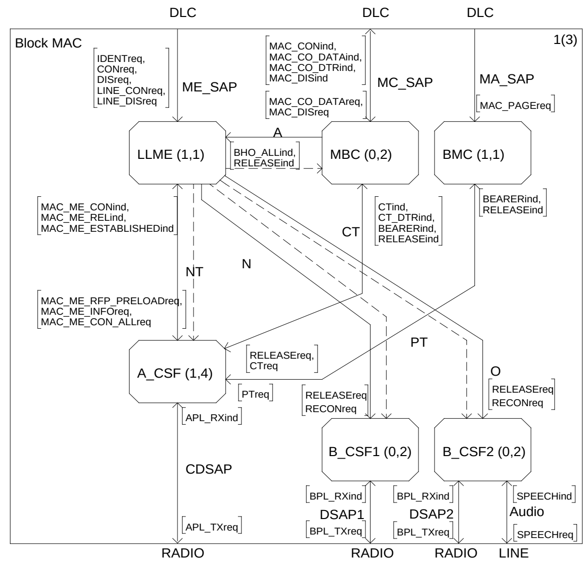

Figure 7: DECT MAC: SDL specification

layer. Handover and broadcast management are performed by the process Multi Bearer Controller (MBC) and the process Broadcast Message Controller (BMC). The control of the radio links is performed by the cell site function processes (CSF). The process A_CSF performs the setup of the so called bearer - the radio links - while the other two CSF processes perform speech data transmission.

6.3 Implementation

The SDL specification was validated by a functional simulation using the commercial SDL tool TAU [16]. Afterwards the specification was implemented on the rapid prototyping platform Phoenix [17]. Phoenix contains four Altera FLEX10K100 FPGAs which are used for the implementation of the DECT burst mode controller and is linked to a Motorola MC68060 microprocessor. The VHDL- and C-Code were generated using the concepts described.

A small manually developed IP core connects a Tenovis DECT transceiver with the hardware RTSS. This VHDL module contains a CRC generator, a digital PLL and a small radio control logic. The hardware RTSS was fully developed in VHDL and the software RTSS is based on the real time operating system RTEMS. The RTSS implements the complete computational model of SDL in the application. For a manual design space exploration each SDL substructure was implemented in hardware as well as in software. A possible partitioning of the system is shown in fig. 8. All time critical modules of the DECT specification have been implemented on the FPGAs in hardware. The protocol automata for the bearer (MTCtrl) and the connection management (MBC), the LLME and the BMC processes are implemented in software on the Motorola microprocessor. The MAC application does not contain large data operations, so the hardware/software partitioning was performed for the CSF processes. Only the A_CSF was partitioned along the border between its synchronous communicating services AMUX and MTCtrl. The other CSF processes were completely implemented in hardware. The architecture shown in the figure supports the setup of up to four bearer. Larger base stations are easily implemented with this

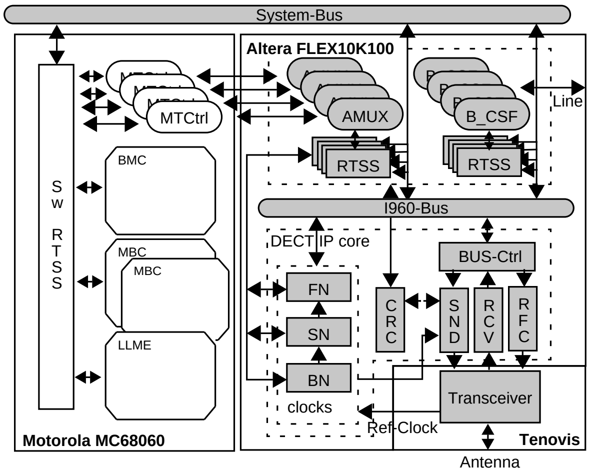

Figure 8: Architecture of the implementation

approach because only more instances of the SDL processes A_CSF, B_CSF1 and B_CSF2 have to be implemented.

7 Conclusion

The paper presents a new approach for the code generation of SDL specifications. The solution allows a fine grain hardware/software partitioning of SDL processes by separating the control flow from the data flow. As a result of this approach high-level synthesis yields better results than the monolithic approach presented in earlier work. Different interface techniques for the communication of the hardware/software parts and the communication with the environment are discussed. Finally the specification and implementation architecture of a typical application was discussed.

8 References

[1] O. Bringmann, W. Rosenstiel. *Resource Sharing in Hierarchical Synthesis*. Proceedings of International Conference on Computer Aided Design (ICCAD), San Jose 1997

[2] O. Bringmann, A. Muth, F. Slomka, W. Rosenstiel, G. Färber, R. Hofmann. *Mixed Abstraction Level Hardware Synthesis from SDL for Rapid Prototyping.* 10th IEEE International Workshop on Rapid System Prototyping (RSP'1999), Clearwater, USA, 1999.

[3] G. Carle, J. Schiller: S*emi-automated Design of High-Performance Communication Subsystems.* 31st IEEE Hawaii International Conference on System Sciences, HICSS 98, Kona, 1998.

[4] J.M. Daveau, G.F. Marchioro, et. al.: *VHDL generation from SDL specifications.* XIII IFIP Conference on Computer Hardware Description Languages (CHDL '97), Toledo, Spain, 1997.

[5] J.M. Daveau, G. Marchioro, A.J. Jerraya. *Hardware/Software Co-Design of an ATM Network Interface Card: a Case Study.* 6th International Workshop on Hardware/Software Codesign (Codes/CASHE '98), Seattle, USA, 1998.

[6] M. Dörfel, F. Slomka, R. Hofmann. *A Scalable Hardware Library for the Rapid Prototyping of SDL Specifications.* 10th IEEE International Workshop on Rapid System Prototyping (RSP'1999), Clearwater, USA, 1999.

[7] ETSI. DIN ETS 300-175-3, *Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 3: Medium Access Control (MAC) layer*. ETSI, 1996.

[8] F. Fischer, A. Muth, G. Färber: *Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping environment.* 6th International Workshop on Hardware/Software Co-Design (Codes/CASHE '98), Seattle, USA, 1998.

[9] W. Glunz, T. Kruse, T. Rössel, D. Monjau: *Integrating SDL and VHDL for System-Level Hardware Design.* XI IFIP Conference on Computer Hardware Description Languages (CHDL '93), Ottawa, Canada, 1993.

[10] ITU-T. *Recommendation Z.100. Specification and Description Language*. ITU, 1993.

[11] R. Henke, H. König, A. Mitschele-Thiel. *Derivation of Efficient Implementations from SDL Specifications Employing Data Referencing, Integrated Packet Framing and Activity Threads*. In A. Cavalli, A. Sarma (eds.), SDL'97 Time for Testing. Elsevier, 1997

[12] A. Mikschl, W. Damm. *Msparc: A multithreaded sparc*. Lecture Notes on Computer Science, 1996

[13] A. Muth, T. Kolloch, T. Maier-Komor, and G. Färber. *An evaluation of code generation strategies targeting hardware for the rapid prototyping of SDL specifications*. In Proceedings of the 11th IEEE International Workshop on Rapid Systems Prototyping (RSP'2000), Paris, France, 2000.

[14] A. Muth and G. 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'2000), Madrid, Spain, 2000.

[15] OAR Corporation. *RTEMS*. http://www.OARcorp.com/RTEMS/rtems.html

[16] SDT. *SDT Reference Manual*. Telelogic, Malmö, 1996.

[17] F. Slomka, M. Dörfel, R. Münzenberger, R. Hofmann. *Hardware/Software Codesign and Rapid-Prototyping of Embedded Systems.* IEEE Design & Test of Computers, Special issue: Design Tools for Embedded Systems, Vol. 17, No. 2, April-June 2000.