An Evaluation of Code Generation Strategies Targeting Hardware for the Rapid Prototyping of SDL Specifications *

Annette Muth Thomas Kolloch Thomas Maier-Komor Georg Färber
Institute for Real–Time Computer Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
{**Annette.Muth,Thomas.Kolloch,Georg.Färber**}@rcs.ei.tum.de

Abstract

The specification of an embedded system at system level together with co–joint hardware/software synthesis is a goal of many rapid prototyping projects. SDL has been proposed as a formal and abstract specification language well suited for this purpose. In the automated generation of hardware however, SDL's asynchronous communication model (directly implemented in the so called server model) can lead to a large overhead in area and response time. The activity thread implementation model on the other hand is more similar to hardware description language concepts, respectively an execution in hardware, due to its synchronous communication and execution scheme. This paper compares VHDL code generation from SDL using these two models regarding implementation architectures, resource usage, throughput and response time. The integration in an existing rapid prototyping design process is presented as well as results gained form several application examples.

1 Introduction

Embedded hard real–time systems show growing functional complexity as well as an increasing demand for short response times and high computing performance. Rapid prototyping is a means to reduce development times and costs of such systems by confirming the *functional* and *timely* requirements of the application at a very early stage of development. The rapid prototyping environment REAR[1] is focussed on event–driven reactive systems with hard real–time requirements. It combines an automated design flow starting from a formal specification with the real–time analysis necessary to prove that the timing requirements have been modeled correctly, and that the prototype will be able to meet all deadlines ([11]). The REAR rapid prototyping target architecture ([4]) is a tightly coupled heterogeneous multiprocessor system consisting of microprocessor based processing units for different classes of real–time tasks, and a FPGA based configurable I/O processor (CIOP) acting as a flexible link to the embedding process and as execution unit for tasks with deadlines too short to be met in software.

Starting point of the rapid prototyping process is a specification in the "Specification and Description Language" **SDL**. SDL, originally from the telecommunications domain, is standardized by the ITU [10], and is increasingly being used in embedded systems design as a formal, abstract description technique at system level. Structure in SDL is expressed with hierarchical blocks and processes, and signal bundling with channels. The lowest level of refinement is a network of parallel SDL processes, each with its private infinite message queue, communicating via asynchronous messages (SDL signals). An extended finite state machine (EFSM), which can contain local variables, specifies the behaviour of each SDL process. Signals are processed in order and trigger a state transition, which in turn can contain arbitrary code inside a task block, output–statements and flow control statements like decisions. With the save–statement, the processing of a signal can be postponed, while a signal marked with priority input is processed at once, even if it is not the first signal in the queue. Timers send signals to the requesting process, using the process' message queue.

The idea of rapid prototyping implies an as far as possible automated design process transforming the specification, in our case the SDL model, into software and configurable hardware on the prototyping target architecture. Two **implementation models**, which preserve the semantics of SDL are the server model and the activity thread model. In the *server model*, each SDL process is implemented as a single RTOS thread in SW, respectively as

*The work presented in this paper is supported by the *Deutsche Forschungsgemeinschaft* as part of a research programme on "Rapid Prototyping for Embedded Hard Real–Time Systems" under Grant Fa 109/11-2.

[1]Rapid Prototyping Environment for Advanced Real–Time Systems

a separate VHDL entity in the HW implementation, each with its own message queue. In contrast to this, the *activity thread model* maps each activity thread, i.e. each chain of activations in the SDL model caused by a stimulating event to one RTOS thread respectively HW entity.

Code size and area usage, while being of critical importance in production code for embedded systems, cannot be completely neglected in rapid prototyping as well. Throughput and response time are indispensable attributes for the functionality of a real–time system's prototype. In hard real–time systems, however, analyzability and guaranteeable worst case performance takes precedence over average or best case throughput and response time.

This paper evaluates code generation using the two implementation models mentioned above, concentrating on the generation of configurable hardware (VDHL) from SDL specifications. The next section surveys related work, after which code generation according to the server model and the activity thread model is examined in detail in Section 3 and 4. Section 5 explains the integration of these code generation strategies in our automated rapid prototyping design process, after which the experiences gained in application examples are presented in Section 6. In the last Section we summarize our results and indicate future work.

2 Related Work

The terms server model and activity thread model stem from the telecommunications area, where they are used to describe different strategies to implement multi layer communication systems ([12]). In the server model, each protocol entity from one layer is implemented as a single software process, communicating with other layer entities via messages. In the activity thread model, one software task processes an incoming or outgoing request trough several layers. [8] proposes the employment of efficient methods known from the manual implementation of communication systems in an automated software design process based on SDL. In their realization of the activity thread model, each SDL process is implemented as a reentrant procedure. Two execution models, the basic and extended activity thread model, are presented, which ensure a semantically correct order of calls to these procedures in the execution of the activity threads.

Commercial code generators for SDL, like SDT's CAdvanced ([13]), only support the generation of software after the server implementation model. The automated implementation of hardware from SDL is still mainly the object of ongoing research.

Several approaches generate VHDL using the server model. The main focus here is the mapping of the abstract communication between the processes to existing interfaces and protocols. The SDL–to–VHDL translator presented in [6] uses a textual implementation description to select functions from a library of channel and protocol descriptions. In [2], the VHDL generation is embedded in the codesign environment COSMOS. An SDL description is translated to an intermediate format. During an interactive refinement process, the abstract channels of this model are replaced by protocols, communication units and interfaces from a library. A commercial spin–off of the TIMA laboratory[2] offers a tool for architectural exploration, supporting SDL as input and VHDL as target language.

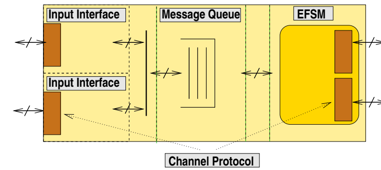

Figure 1. Server Model Architecture

[9] is based on a concept aiming to support SDL's dynamic process creation feature also in hardware. Here, one entity is created for each SDL process class, storing and loading the context of each process instance after a simple schedule. Due to the dynamic instantiation, a signal's receiver cannot be determined statically. This necessitates a central supervisor unit in the communication subsystem.

3 Server Model Implementation

The server model maps each SDL process to one hardware entity with its own message queue. Figure 1 shows a typical hardware architecture implementing one SDL process according to the server model. One input interface for each communication channel receives SDL messages, implementing the channel's protocol. The message is put at the end of the queue, which can be realized as a simple FIFO only if save and priority input are excluded from the supported SDL language subset. The extended finite state machine is implemented in an infinite loop, where in turn a message is taken from the queue, the appropriate transition executed, and, if applicable, a new SDL message is sent via an output channel. To realize the asynchronous communication of SDL (non-blocking send, receive via message queue), the input interface, the message queue and the EFSM must run parallel.

The server model is a direct, semantically correct implementation of SDL. In contrast to software, the implementation in hardware allows a parallel execution of the SDL processes. Code generation is fairly straightforward, due to the semantic similarities between SDL and the server model (asynchronous computation and communication). While the EFSM part has to be generated for each new SDL model, the message queue, timers, and the entire inter-process com-

[2] www.arexsys.com

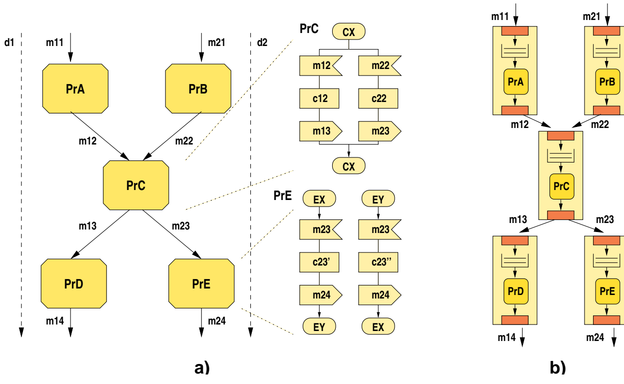

Figure 2. a) SDL specification, b) server model implementation

munication can be implemented as a library of reusable components, the "HW run–time system" ([1], [3]).

Figure 2 shows an exemplary SDL specification and the corresponding server model implementation. The specification consists of a network of five SDL processes, communicating via messages m_{ij}. A transition triggered by message m_{ij} consists of a task c_{ij} and the sending of a new message $m_{i(j+1)}$. The response to the external events m_{11} and m_{21}, i.e. the output of messages m_{14} and m_{24} has to occur within the given deadlines d_1 and d_2.

Area Efficiency For a single SDL process, a large contribution to the occupied area comes from the message queue. While the hardware effort for a simple FIFO is high enough (see Figure 5 in Section 6), it becomes prohibitive for a queue correctly implementing save and priority input, i.e. permitting message insertion and removal at random positions. The queue size has to be dimensioned very carefully.

Next to the message queues a considerable hardware effort is incurred by the sending and receiving of SDL messages. Each input channel implicates a protocol implementation and data conversion functions, plus handshake logic and a multiplexed input to the queue. Each SDL output–statement also implies an output interface, implementing data conversion and the channel protocol, plus multiplexer logic if several output–statements write on one channel.

Typically, a SDL specification consists not of one, but of several processes, interconnected by signals. For such a network of processes, the hardware effort increases not only linearly with the EFSMs and message queues, but disproportionate to the number of processes with each signal interconnection between two processes. Each additional connection generates a larger input interface, and, depending on the number of output–statements, a very high overhead for the output interfaces.

Response Time and Throughput The response time t_r to a certain event consists of the computation time t_c needed to process the event and a possible waiting time t_w that elapses while a message stands in the queue. While t_c is directly determined by the hardware architecture, an upper bound of t_w has to be computed by a real–time analysis. t_c of a single SDL process consists of the time needed to receive and queue the message, the time to remove the message from the queue and process it, plus the time to output a new message. The parallel and pipelined execution of a network of SDL processes allows, depending on the application, a very good average and worst case throughput. The computation time necessary to respond to an external event, however, can be very high, since the t_cs of all involved SDL processes have to be summed. Typical values for t_c realized in application examples can be found in Section 6.

4 Activity Thread Model Implementation

In the activity thread implementation model the chain of activations triggered by a stimulating event, i.e. an event from the environment or a timer output is analyzed. All actions and state changes contained in the transitions along this "activity thread" are executed sequentially, thereby avoiding the message send and receive overhead between the processes. If the minimal time distance between two events triggering an activity thread is smaller than the worst case execution time of the thread, a message queue at the input of the activity thread has to ensure that no events are lost.

To ensure a correct implementation of the SDL semantics, the following issues have to be treated with special attention:

Atomicity of the state transitions: The state transitions of one process must exclude each other mutually. If transitions of one process are part of different activity threads, it must be ensured that they are not executed at the same time. In an parallel execution model they must be protected by a lock mechanism.

Branches in an activity thread: An activity thread branches when several output–statements occur in one transition, or when an output–statement stands before a task–statement. A semantically correct ordering of the execution can be fixed at compile time.

Continuous signals: Transitions guarded by continuous signals have no stimulating event. For them, an own activity thread has to be created.

Architecture Alternatives In hardware, each activity thread could be executed in parallel. Depending on the type of application and on the temporal specification of embedding and embedded system, however, other implementation alternatives can be more efficient in area and response

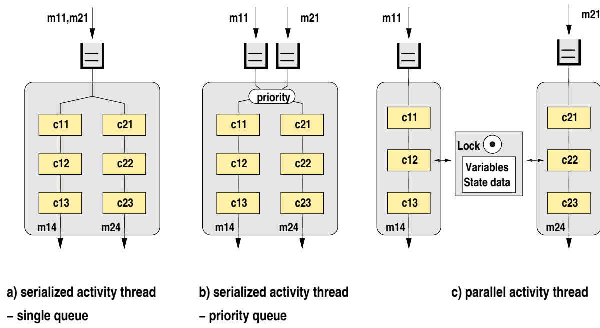

Figure 3. Activity thread model alternatives

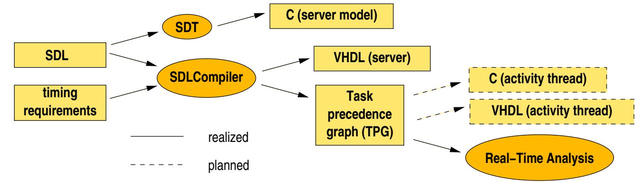

Figure 4. REAR design process

time. Figure 3 shows three architecture alternatives, using the SDL example from Figure 2.

The least degree of parallelism is achieved in the *serialized activity thread – single queue* architecture. Here, all activity threads are implemented in one VHDL process. Only one event at a time is taken from the input message queue, and the corresponding activity thread is processed. To ensure a response in real–time, the processing time t_c of an event plus the t_c's of all events that can block the queue before it, must be lower than the event's deadline.

In order to be able to guarantee short deadlines, the *serialized activity thread – priority queue* architecture has more than one input message queue. At compile time, external events and timer outputs are assigned to different classes according to their deadlines respectively priorities. For each "priority class", a separate input message queue is implemented. Again, one event at a time is taken from the queues, serving queues with high priorities first, and processed in its activity thread.

The *parallel activity thread* architecture implements each activity thread in its own parallel VHDL process with its own input message queue. Like pointed out above, this requires a shared access to the local variables and state data protected by a lock mechanism, if a SDL process has been split into more than one activity thread.

The three implementation architectures can be easily combined. For each activity thread a decision can be made to either join it with other threads, or to implement it separately. A combination of server and activity thread model is also easily possible, since the interface of each activity thread is the asynchronous sending and receiving of messages, identical to each process in the server model.

Area Efficiency For input interface, message queue, timers and output interface, the same "HW runtime–system"–components as in the server model can be used, and the same area usage can be assumed. The number of these components needed in an activity thread implementation is equal only in a model consisting of a single SDL process. In all other cases it is lower, since no messages are sent inside an activity thread. An increased hardware effort can occur when one SDL signal is output in different transitions, i.e. the transition triggered by this signal appears in several activity threads. The *serialized activity thread – single queue* architecture requires one message queue, and input and output interfaces only for messages at the border of the SDL model. The *serialized activity thread – priority queue* architecture requires more input queues, and a more sophisticated queue access. Due to the sequential specification in one VHDL process, the first two design alternatives allow resource sharing between SDL processes. In the *parallel activity thread* architecture, each additional activity thread requires a message queue. Further overhead is caused with each divided SDL process by the shared process data access protected by the lock mechanism.

Response Time and Throughput The calculation time of an event in the *serialized activity thread – single queue* architecture can be very low due to the small overhead. This is especially true, when the computational complexity of the SDL process is is relatively low. In that case, the time required to execute an entire activity thread is low compared to communication or shared data access times. A real–time analysis has to determine the feasibility of this architecture by calculating the maximum number of events blocking the queue, leading to additional t_w. The *serialized activity thread – single queue* model is advantageous if the hardware computation time is small compared to the timely distance of external events. The priority–queue leads to a slightly higher queue access time, but the worst case waiting time is shorter. In the case of disjunct activity threads, the *parallel activity thread* architecture has the same computation time as the other two models. When the mutual exclusion mechanism is necessary, however, the calculation time increases because of the access time to the shared data. Additionally, blocking times of the lock mechanism by another thread have to be taken into account. The parallel activity thread architecture allows parallel execution, but no pipelining. Its throughput is therefore lower than the server model's.

5 Design Process

Figure 4 depicts the tool chain of the automated rapid prototyping design process realized in REAR. The specifi-

cation in SDL is annotated with a specification of the timing requirements using deadlines and a temporal description of the embedding system with event streams ([7]). Currently, the SDL model is partitioned manually by allocating SDL processes on the target architecture's HW and SW processing units according to the task classification model ([5]). For the software part, C code is generated using SDT's code generator CAdvanced, automatically including functions from the underlying real–time operating system RTEMS and from a scalable IPC library for the inter–unit communication. VHDL after the server model is generated using the SDLCompiler presented in [1]. The SDLCompiler also generates a task precedence graph (TPG) which (next to the worst–case execution times) is the basis of a real–time analysis delivering the proof, that the realized embedded system will be able to meet all deadlines. A detailed description of the framework can be found in [11].

The TPG, representing the activity threads contained in the SDL process network, is also used for the generation of C and VHDL code using the activity thread model, which is currently being integrated in the design environment. With the inclusion of a second implementation model, the design space increases significantly, and the partitioning step becomes less straightforward. Section 3 and 4 pointed out the trade-offs between response time, throughput and area that can result from the different implementation alternatives. An extended real–time analysis has to eliminate the design alternatives that do not conform with the timing requirements. An interactive partitioning process is planned, where the designer is being assisted with resource usage estimations and real–time analysis.

6 Experimental Results

Figure 5 shows the influence of the message queue on the area usage. It depicts the CLB[3] usage on the CIOP's Xilinx XC4025E FPGA of a simple "**ping–pong**" application example, depending on message size and queue length (FIFO message queue). The example consists of a single SDL process with one input and one output channel, each implementing a handshake protocol. VHDL code was generated using the SDL-to–VHDL tool presented in [6]. t_c of this example amounted to 3 cycles for input interface and message queue, 2 cycles for the execution of the EFSM, and 2 cycles for the output interface.

A simple **event counter** specification, consisting of three SDL processes, was used for a first comparison between the implementation models. The area usage of a 8 bit wide server model implementation was 444 CLBs, while a activity thread implementation required only 117 CLBs.

As a non–trivial real–world example with stringent real–time requirements, a **CAN controller and monitor** application was implemented on the REAR target architecture. CAN is a serial field bus with bit rates of up to 1 Mbits^{-1}. Figure 6 depicts the SDL process structure of the CAN physical layer, which implements the access to the physical medium (sending and receiving single message bits) and the according low–level timing, bit stuffing and synchronization functionality of the protocol. To achieve a precise timing, the duration of one message bit has to be divided by a configurable number of internal controller ticks. The SDL process `Timing` is triggered by the emission of these ticks (signal `ctrl_clock`) and, depending on its state, notifies other processes when a new bit frame starts or the sampling point inside the bit has been reached (signals `can_clock` and `sample_now`). Figure 7 shows the corresponding activity thread. The branches of this activity thread, e.g. the sampling of the bus level followed by output of signal `rx` to the data link layer, have to be finished before the emission of the next tick. The deadline $d_{c,rx}$ of the activity thread `ctrl_clock` → `rx` is $d_{c,rx} = \frac{1}{8 \cdot f}$, for bus frequency f and a number of 8 internal ticks per message bit.

[3]configurable logic block

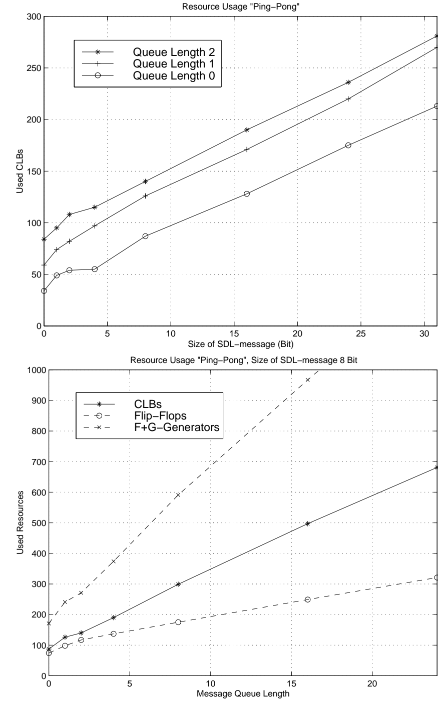

Figure 5. CLB–Usage Ping–Pong Example

A partly automated implementation of the CAN physical layer on the CIOP's FPGA after the server model, using the SDLCompiler, required 1022 CLBs. The process chain `ctrl_clock` → `rx` takes 16 cycles. This is due firstly to message sending overhead of the three processes Clock, Timing and Receiver, and secondly to a delay in the timing–process, where the relevant signal is the last of three

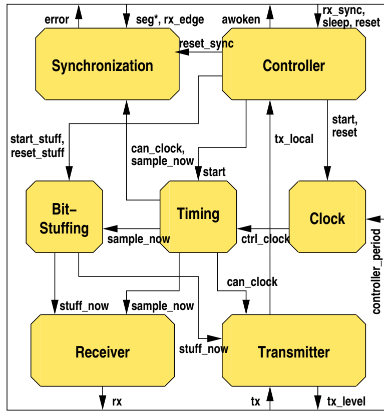

Figure 6. SDL model CAN physical layer

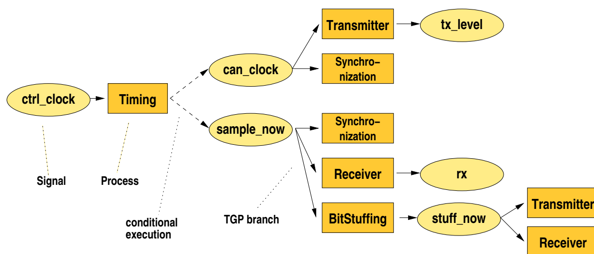

Figure 7. Timing activity thread (clipped)

sequential output–statements. With a cycle period of 80 ns this results in a maximal possible CAN bus frequency of 98 kbits^{-1}. In contrast to this, a manual implementation using the serialized activity thread model took 426 CLBs. The longest path in the design was 4 cycles, leading to a achievable bus frequency of 390 kbits^{-1}.

7 Conclusions and Future Work

This paper has compared two implementation models which enable automated generation of hardware from SDL. First experimental results indicate a large advantage of the activity thread model over the server model in applications where the amount of computation is small compared to the communication overhead. In the investigated examples a smaller area usage by a factor of 2.4 to 3.8 and shorter response time by a factor of 4 were achieved. General considerations on area, throughput and response times of different hardware architectures supported by the implementation models indicate, that there are trade–offs to be investigated depending on the type of application and its real–time constraints. Future work will include the integration of code generation after the activity thread model in our rapid prototyping design process, and the extension of the real–time analysis in order to support the evaluation of the different design alternatives.

References

[1] O. Bringmann, W. Rosenstiel, A. Muth, G. Färber, F. Slomka, and R. Hofmann. Mixed abstraction level hardware synthesis from SDL for rapid prototyping. In *Proc. of the 10th IEEE International Workshop on Rapid Systems Prototyping (RSP'99)*, Clearwater, USA, June 1999.

[2] J. Daveau, G. Marchioro, C. A. Valderrama, and A. A. Jerraya. VHDL generation from SDL specifications. In *Proceedings of the XIII Conference on Computer Hardware Description Languages, CHDL'97*, Toledo, Spain, Apr. 1997.

[3] M. Dörfel, F. Slomka, and R. Hofmann. A scalable hardware library for the rapid prototyping of SDL specifications. In *Proc. of the 10th IEEE International Workshop on Rapid Systems Prototyping (RSP'99)*, Clearwater, USA, June 1999.

[4] F. Fischer, T. Kolloch, A. Muth, and G. Färber. A configurable target architecture for rapid prototyping high performance control systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 3, Las Vegas, Nevada, USA, June 30 – July 3 1997.

[5] G. Färber, F. Fischer, T. Kolloch, and A. Muth. Improving processor utilization with a task classification model based application specific hard real–time architecture. In *Proceedings of the 1997 International Workshop on Real–Time Computing Systems and Applications (RTCSA'97)*, Academia Sinica, Taipei, Taiwan, ROC, Oct. 27–29 1997.

[6] W. Glunz, T. Kruse, T. Rössel, and D. Monjau. Integrating SDL and VHDL for system–level hardware design. In *Proc. XI IFIP Conference on Computer Hardware Description Languages (CHDL '93)*, Ottawa, Canada, 1993.

[7] K. Gresser. An event model for deadline verification of hard real–time systems. In *Proc. Fifth Euromicro Workshop on Real Time Systems*, pages 118–123, Oulu, Finland, June 1993. IEEE.

[8] R. Henke, H. König, and A. Mitschele-Thiel. Derivation of efficient implementations from SDL specifications employing data referencing, integrated packet framing and activity threads. In *Proceeding of the Eighth SDL Forum, SDL'97*, pages 397–414, Evry, France, Sept. 1997. Elsevier Science Publishers B.V.

[9] W. Horn, B. Svantesson, S. Kumar, A. Jantsch, and A. Hemani. Hardware synthesis of an ATM multiplexer from SDL to VHDL: A case study. In *Proceedings of the IEEE Workshop on VLSI'99 (WVLSI'99)*, Orlando, Florida, USA, Apr. 1999.

[10] ITU. *ITU–T Recommendation Z.100: CCITT Specification and Description Language (SDL)*. ITU–T, June 1994.

[11] S. Petters, A. Muth, T. Kolloch, T. Hopfner, F. Fischer, and G. Färber. The REAR framework for emulation and analysis of embedded hard real–time systems. In *Proc. of the 10th IEEE International Workshop on Rapid Systems Prototyping (RSP'99)*, Clearwater, USA, June 1999.

[12] L. Svoboda. Implementing OSI systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, 1989.

[13] Telelogic, Kungsgatan 6, S-20312 Malmö, Sweden. *SDT Reference Manual*. Version 3.1.