In: Proc. of the 13th International Symposium on System Synthesis (ISSS'2000), Madrid, Spain, September 20-22, 2000.

SDL as a System Level Specification Language for Application–Specific Hardware in a Rapid Prototyping Environment

Annette Muth Georg Färber
Institute for Real–Time Computer Systems Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
{Annette.Muth,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. The implementation of SDL's asynchronous communication model in application specific hardware, however, is unproportionally expensive in terms of area and response time. This paper discusses the efficiency of the server model — the implementation model used in all known codesign approaches based on SDL — and compares it with an alternative implementation model for SDL known from software, the activity thread model. In a combination of both implementation stategies, the communication and synchronization overhead for each application can be minimized, and an efficient implementation on distributed architectures realized. The integration in an existing rapid prototyping design process is presented as well as results gained from application examples.

1 Introduction

The challenge in system level design of embedded systems of today is to master their ever increasing complexity in the short development times and costs the market allows. Modern design methodologies, realized in HW/SW–codesign environments like [5], [3], [1] (see also [6], [14]), start with a model of the system in one or several specification languages. Using such a model, validation with simulation, formal verification, real–time analysis or rapid prototyping are possible, providing a means to find design flaws in early phases of the development. Embedded systems typically consist of microprocessors respectively processor cores executing software, and application specific hardware. A suitable architecture of processors, buses and custom hardware is determined and the specification is partitioned and mapped to the available execution units. The next steps towards implementation are communication refinement, automated codegeneration and synthesis of hardware and software. The first commercial tools based on these research results are now emerging. The Cadence Cierto VCC environment[1] is based on Berkeley's Polis approach. Arexsys, the commercial spin–off of the TIMA laboratory, markets a codesign environment based on Cosmos[2].

A multitude of specification languages and methods are used in embedded systems design, i.e. C, C++, VHDL, Esterel, Statecharts, Matlab and SDL. Several properties of the "Specification and Description Language" SDL (see also Section 2) make it well suited for a design methodology like outlined above. SDL allows implementation independent system modeling at a high level of abstraction, in graphical and textual form. Its formal semantics form a solid basis for validation and simulation. Due to SDL's asynchronous execution and communication scheme, dividing and joining models is easy, which allows for more flexibility in architectural synthesis, and efficient implementations on distributed systems. Other, non–technical criteria, are maturity and popularity of the language, and the well developed tool support. SDL's standardization ([10]) makes it suitable for official authorization procedures, in some of which it is already mandatory. SDL is a supported input format of both commercial codesign tools mentioned above.

The use of SDL in a HW/SW–codesign environment requires automated generation of hardware from SDL — in order to use a homogenous SDL specification for

[1]www.cadence.com/technology/hwsw/ciertovcc
[2]www.arexsys.com

hardware and software, or at least to enable a flexible allocation to hardware or software when a heterogenous specification containing SDL parts is used.

This paper discusses the efficiency of SDL implementations in hardware. It identifies the cases when the prevalent SDL implementation model — the server model — leads to unreasonable designs in terms of area and timing. It presents the activity thread implementation model, which can reduce the — in hardware unproportionally expensive — communication overhead caused by SDL's asynchronous communication model. Section 2 introduces the language SDL and gives an overview on SDL implementation strategies used in different approaches. Section 3 and 4 describe the server model and the activity thread model in greater detail. Section 6 presents the integration of both implementation schemes in a rapid prototyping environment for hard real–time systems, where first experimental results were gained (Section 7).

2 Implementation Strategies from SDL

Structure in SDL is expressed with hierarchical blocks and processes. 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 SDL task block, SDL 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. Timers send signals to the requesting process, using the process' message queue.

In the design process, the specification is transformed into software and configurable hardware for the target architecture. The rules for this transformation, the **implementation model**, must make sure that the implementation is semantically conform with the specification. 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 task in SW, respectively as 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 one stimulating event (an event from the environment or a timer output), to one RTOS task respectively HW entity.

The terms server model and activity thread model stem from the telecommunications area, where they are used to describe different stategies to implement multilayer communication systems ([12]). In the server model, each protocol unit from one layer is implemented as a single software task, communicating with other layers via messages. In the activity thread model, one software task processes an incoming or outgoing request through several layers. A recent approach, [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. Each activity thread is a sequence of calls to these procedures. Commercial codegenerators for SDL targeting software, like SDT's CAdvanced ([13]), only support the server implementation model.

Several approaches generate VHDL using the server model. The main focus here is mapping the abstract communication between SDL processes to existing interfaces and protocols. The SDL–to–VHDL translator presented in [7]uses a textual implementation description to select functions from a library of channel and protocol descriptions. In [4], 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. This approach is now integrated in a commercial tool (cf. Section 1). A further approach, aiming to support SDL's dynamic process creation feature also in hardware, is presented in [9].

3 Server Model

The server model maps each SDL process to one HW–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. The extended finite state machine is implemented in an infinite loop with two nested case-statements. In turn a message is taken from the queue, the transition belonging to message type and process state executed, and, if necessary, a new SDL message is sent via an output channel.

The server model is a straightforward, semantically correct implementation of SDL. While the EFSM part has to be generated for each new SDL model, the message queue, timers, and the entire inter-process

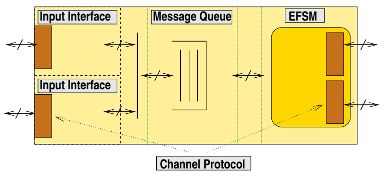

Figure 1. Server Model Architecture

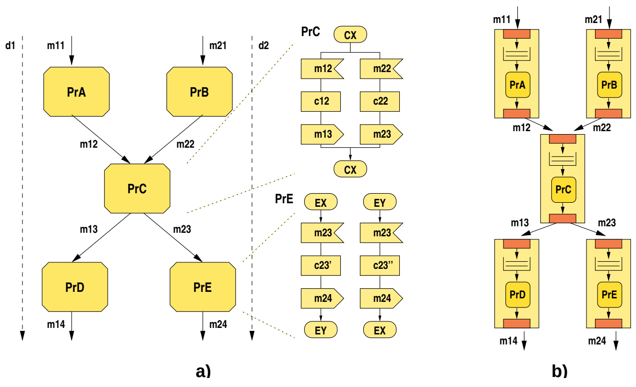

Figure 2. a) SDL specification, b) server model implementation

communication can be implemented from a library of reusable components.

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 $m_{i(j+1)}$.

Area Efficiency For a single SDL process, a large portion of the occupied area comes from the message queue. While the hardware effort for a simple FIFO is high enough, it becomes prohibitive for a queue correctly implementing save and priority input, i.e. permitting message insertion and removal at random positions. 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, plus handshake and multiplex logic for the queue's input. 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.

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 resides 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 enqueue the message, the time to remove the message from the queue and process it, plus the time to output a new message.

Summary Typically, an 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, very high overhead for the output interfaces. The parallel and potentially pipelined execution of a network of SDL processes allows, depending on the application, good average and worst case throughput. The computation time necessary to respond to an external event, however, can be very high, since the communication overhead of all involved SDL processes adds up.

The communication overhead incurred in area and time can be dramatic, since asynchronous communication is especially expensive in hardware, and can easily surmount the effort for the computation actually included in the SDL processes.

4 Activity Thread Model

The activity thread implementation model analyzes the chain of activations in an SDL system triggered by an external event or timer output: The event (in form of an SDL message) is received by an SDL process, triggers a transition, where in turn an SDL message may be sent to a second process. This chain of activations is called "activity thread". Activity threads contain state choices, branch at multiple SDL output statements in a transition, and terminate with the sending of a message to the environment or with the consumation of an SDL message in a process without triggering a new SDL output. All actions and state changes contained in the transitions along an activity thread are implemented sequentially, thereby avoiding the message send and receive overhead between the processes. Special attention has to be paid to a semantically correct implementation, especially concerning process data consistency and the correct ordering of messages.

Architecture Alternatives In hardware, each activity thread could be executed in parallel. Depending on the type of application, other implementation alternatives can be more efficient. Figure 3 shows two architecture alternatives, using the example from Figure 2.

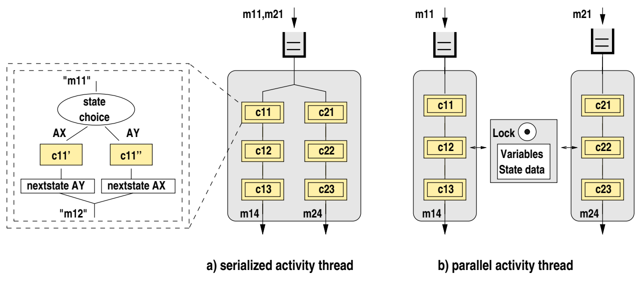

Figure 3. Activity thread model alternatives

In the *serialized activity thread* architecture, all activity threads are implemented in one VHDL process. One event at a time is taken from the input message queue, and the corresponding activity thread is executed.

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

Area Efficiency For input interface, message queue, timers and output interface, the same library components as in the server model can be used. The *serialized activity thread* architecture requires message queue, and input and output interfaces only for messages at the border of the SDL model.

In the *parallel activity thread* architecture, each activity thread requires its own message queue. Each SDL process divided in several activity threads needs a component implementing shared data access protected by a lock mechanism. This implies an additional synchronization overhead.

The activity thread model displays the effect that a single functionality inside one SDL process, when triggered by different external events, will be multiplied, because it appears in several activity threads. In the *serialized activity thread* architecture, a joining of activity threads is conceivable. In the *parallel activity thread* architecture, this functionality will be implemented multiply, leading to additional area overhead.

Response Time and Throughput The *serialized activity thread* architecture serializes the activity threads for all incoming events. The computation time t_c consists of a very small communication overhead plus the computation required for the entire activity thread. Due to the serialization, a potentially increased waiting time t_w has to be taken into account. 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 variables. Additionally, blocking times of the lock mechanism by another thread have to be considered. The parallel activity thread model allows parallel execution, but no pipelining. Its throughput is therefore lower than the server model's.

Serialized Activity Thread Summary For the extreme example of a single SDL process, the server implementation and the serialized activity thread implementation are identical. In the general case, $t_{c,atm}$ is smaller than $t_{c,server}$. t_w has to be determined in a real–time analysis. The realizable throughput of the *serialized activity thread* architecture is lower because of the unused parallelization potential. The used area of the activity thread implementation is smaller than of the server model, if the area added by multiply implemented functionality is smaller than the server model's communication overhead, which is true for a large class of applications.

Parallel Activity Thread Summary In the extreme example of a single SDL "server"–process, that performs one functionality for a number of "clients", the area overhead due to the multiple implementation of this functionality is maximal, as well as the synchronization overhead. For such a case, a server model implementation is most suitable. In a different extreme example of a model consisting of disjunct chains of SDL processes (i.e. multilayer protocol stacks), synchronization overhead and multiple implementations are low. Here the parallel activity thread implementation shows high throughput and response time, but a slightly higher area compared to the serialized activity thread implementation. Here, both activity thread alternatives are very efficient.

In the general case, the parallel activity thread implementation is smaller in area than the server model, if the server model's communication overhead is larger than synchronization and computation overhead of a activity thread model. This is for example the case in a control–dominated SDL model with fine process granularity. Analogously, $t_{c,at}$ is smaller than $t_{c,server}$, if the communication overhead is larger than the synchronization overhead.

5 Combining Implementation Models

Each SDL message connects two SDL processes in an asynchronous manner. Because of this quality it can serve as a connection point between parts of the SDL system implemented after different implementation models. In particular this means:

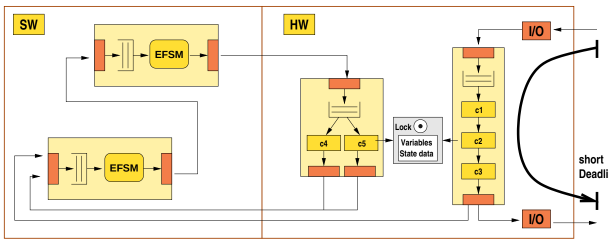

Figure 4. Partitioning Example

- Each SDL message can start an activity thread, coming from the environment, a timer, or an SDL process implemented after the server model.
- Each activity thread (or a branch of an activity thread) can be terminated by the sending of an SDL message to the environment, to a different activity thread, or to a server process.

The server model and the activity thread model represent two different views of the SDL specification. The SDL model is implemented in parallel VHDL processes, each with input interface and message queue. Mapping an entire SDL process to one VHLD process corresponds to a server model implementation of this SDL process. Accordingly, mapping one or several activity threads entirely or partly to one VHDL process is a serialized or parallel activity thread implementation.

Figure 4 shows a typical combination of the implementation models. Here, an entire activity thread with a short deadline is implemented in hardware. For a short worst–case response time blocking times of shared variable areas caused by an SDL process part implemented in software cannot be accepted. Here, asynchronous communication at the HW/SW boundary is more appropriate.

6 Application in Rapid Prototyping

The presented hardware implementation models for SDL are used in an automated design process of a rapid prototyping environment for hard real–time systems ([11]). The rapid prototyping target architecture is a heterogenous multiprocessor system, tightly coupled by a global PCI bus. It consists of processor–based execution units running software and FPGA–based execution units serving as a flexible link to the embedding system, and as a platform for tasks with deadlines too short to be met in software.

Currently, the rapid prototyping design process supports codegeneration for software after the server model using Telelogic's SDT CAdvanced code generator. The SDLCompiler presented in [2]generates

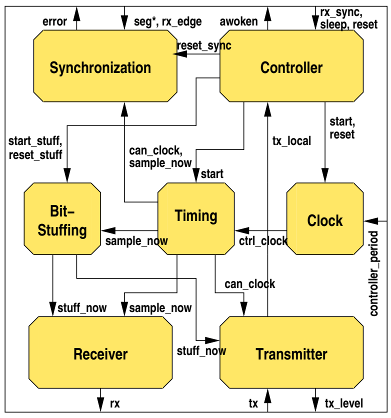

Figure 5. SDL model CAN physical layer

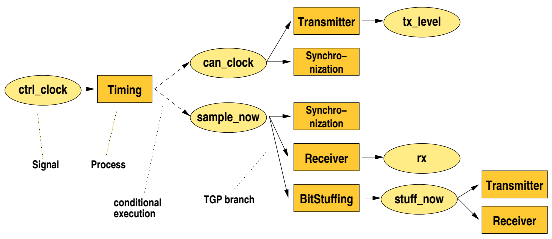

Figure 6. CAN Timing activity thread

VHDL code after the server model and after the serialized activity thread implementation model. SDL processes are mapped to HW and SW manually, using timing constraints and computational complexity as main criteria.

7 Experimental Results

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 on a Xilinx 4025E, while an 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 rapid prototyping target architecture. Figure 5 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 6 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 $\texttt{ctrl\_clock} \rightarrow \texttt{rx}$ is $d_{c,rx} = \frac{1}{8 \cdot f}$, for bus frequency f and a number of 8 internal ticks per message bit.

An 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 $\texttt{ctrl\_clock} \rightarrow \texttt{rx}$ 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 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 automated implementation using the serialized activity thread model took 564 CLBs. The longest path in the design was 4 cycles, leading to a achievable bus frequency of 390 kbits^{-1}.

8 Conclusions and Future Work

Due to SDL's asynchronous execution and communication semantic, a parallel implementation in hardware inevitably causes communication overhead (server model) or synchronization overhead (parallel activity thread model). A serial implementation (serialized activity thread model) on the other hand reduces syncronization and communication overhead to a high degree, but fails to make use of the possible parallelization in hardware. The different implementation models can be freely combined, which makes it possible to find the most efficient implementation specifically for each application, particularily on a distributed architecture.

The future vision for the rapid prototyping design process is an interactive partitioning and mapping step, where the designer can explore the trade–offs in response time, throughput and area between the implementation alternatives, assisted with partitioning suggestions and evaluations based on area and real–time estimations.

References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A, Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B.Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* Kluwer Academic Press, 1997.

[2] 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. IEEE International Workshop on Rapid Systems Prototyping (RSP'99)*, Clearwater, USA, June 16–18 1999.

[3] P. Chou, R. Ortega, K. Hines, K. Partridge, and G. Borriello. ipChinook: An integrated IP-based design framework for distributed embedded systems. In *Proceedings of the 36th Design Automation Conference — DAC'99*, pages 44–49, New Orleans, USA, 21–25 1999. ACM Press.

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

[5] R. Ernst, J. Henkel, T. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny. The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, 20(3), May 1996.

[6] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[7] 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.

[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 *Proc. of the Eighth SDL Forum, SDL'97*, Evry, France, Sept. 1997.

[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)*, pages 100–105, Orlando, Florida, USA, Apr. 8 – 9 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. *Design Automation for Embedded Systems*, 5(3), 2000.

[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.

[14] W. H. Wolf. Hardware–software co–design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.