Lehrstuhl für Realzeit-Computersysteme

SDL-based Design of Application Specific Hardware for Hard Real-Time Systems

Annette Muth

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Die Dissertation wurde am 17.01.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 02.05.2002 angenommen.

Meinen Eltern

Abstract

Specification languages and automated design methods are increasingly being used to master the growing complexity in the development of embedded electronic systems. The work presented here uses the "Specification and Description Language" SDL as basis of an automated design process targeting application specific hardware particularly for hard real-time systems.

The SDL specification is annotated with deadlines, event streams and event dependencies which capture the timing requirements and properties of the embedding system. The next step towards an electronic circuit is a VHDL description of the required behaviour. Different principles for the transformation of SDL into VHDL, the implementation models, are presented. The server model maps each SDL process to its own VHDL entity with its own message queue. The activity thread implementation in contrast executes all transitions, which are triggered in the SDL processes by one external signal or timer output, directly one after the other, abolishing the internal communication between the processes. In the presented design process, a SDL-Compiler generates VHDL from textual SDL. The statemachine part is linked with so called run-time components, which implement reusable functions like message queues, timers, and communication channels. Commercial synthesis tools create the electronic circuit from the VHDL design. The complete design flow was integrated with a HW/SW rapid prototyping environment.

Hard real-time systems require the beforehand proof that all deadlines will be met. A real-time analysis is presented which calculates the worst case response times to external events, considering the timing constraints, different implementation models and run-time components. An upper bound on the necessary length of the message queues is derived as well. The consideration of event dependencies during real-time analysis brings a relaxation of a possibly too pessimistic worst case scenario. The presented methods have been tested in the rapid prototyping environment on a FPGA-based target architecture with the help of several application examples. The results from these experiences allow an evaluation of both the resource requirements and real-time properties of the hardware automatically generated from SDL.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The development of embedded electronic systems presents an increasing challenge for industry and academia: Most new features in many domains, e.g. in automotive and telecommunications, depend on electronics. The increasing density and speed of electronic circuits correspond with an increasing complexity to be mastered during the design process. Sharp competition leads to shortening design cycles and pressure on the development costs, while engineering manpower is turning into a scarce resource. At the same time, embedded systems have to meet stringent requirements, among others functionality, reliability, size and power consumption. Very central among these rank soft and hard *real-time requirements*. These express the embedded system's ability to respond not only correctly but also within a given time.

Modern design methodologies, realized in HW/SW–codesign environments like [EHB+96], [COH+99], [BCG+97] (see also [GVNG94], [Wol94]), aim to meet these challenges by introducing systematic, formal design methods and increasing the degree of automation in the design process. They start with a model of the system under development in one or several specification languages. The use of a system model in a formal specification language has several advantages. Firstly, its defined semantics allow a concise description and avoid the ambiguities of a natural language. Secondly, validation with the help of simulation, formal verification or real–time analysis is possible. Thirdly, a formally defined language makes it possible to write compilers which translate it into a different language. This opens the path to an automated design process, which ensures that an implementation is reached in shorter time, is hopefully error free and consistent with the specification.

Embedded systems typically consist of microprocessors respectively processor cores executing software, and application specific hardware. During the automated design process, 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 code generation and synthesis of hardware and software.

Rapid prototyping has been proposed as a method to find errors and flaws in the embedded system's requirements at a very early stage of development. To make this possible, the specification is executed in the real environment in form of a working prototype. In order to rapidly obtain such a prototype, an automated design process like outlined above is used, targeting a rapid prototyping platform. In a rapid prototyping application, the execution of the specification in software means high flexibility, short design cycles and good debug facilities. *Application specific hardware* on the other hand is very often necessary as a link to the embedding process, i.e. for glue logic or a custom bus protocol, and as execution unit for processes with short deadlines. Here, too, it is possible to reach a high degree of flexibility by using reprogrammable FPGAs.

In many application areas, the focus of rapid prototyping lies only on the functional correctness of the embedded system. For *hard real–time systems* however, it is also necessary to include the real–time aspects already at the prototyping stage: In the design of a system that has to meet hard deadlines, the timing requirements of the embedding system have to be modeled correctly. This is a non-trivial, error-prone task. A real–time analysis in the prototyping stage and the real–time execution of the prototype show if the timing situation has been understood correctly. In contrast to simulation, the idea of a prototype is to be executed in the real world with real data. If the application to be designed has to meet hard timing requirements, they generally will have to be met by a prototype executing in the same environment, too. A schedulability analysis must give the proof that it is safe to execute the prototype.

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 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 ([ITU94]) makes it suitable for official authorization procedures, in some of which it is already mandatory. SDL is a supported input format of a considerable number of commercial and academic codesign environments, as outlined in chapter 2.

The automated design flow from SDL to software is state of the art and widely used. The ability to automatically generate application specific hardware from SDL enables the use of SDL in a HW/SW–codesign environment. This makes it possible to use SDL specifications for hardware and software.

1.2 Scope

The work presented here deals with the automated design of application specific hardware from SDL in a rapid prototyping HW/SW–codesign environment. It is targeted towards reactive hard real-time embedded controllers. Timing constraints are assumed to have the form of end-to-end deadlines, i.e. a given response time that may elapse until the reaction to an external event is finished. In hard real-time systems, in contrast to soft real-time, the violation of a deadline can lead to catastrophic consequences, up the loss of lives. In this class of systems the correct real-time behavior has to be *proven* during design. Analyzability and guaranteeable worst case response times therefore take precedence over average or best case throughput and latency.

In the rapid prototyping scenario, for cost and flexibility reasons, as much functionality as possible will be implemented in software. Application specific hardware is necessary for processes with deadlines that are too short to be met in software and as a connection to the embedding environment. The envisioned type of application therefore are fast I/O-tasks with relatively low computational complexity. In this sense, application specific hardware is regarded like a pre-processor for external events rather than a co-processor.

The automated design process is part of the rapid prototyping environment REAR presented in [PMK$^+$00]. Its rapid prototyping target architecture is a configurable and scalable heterogenous multiprocessor system. A board with a field programmable gate array (FPGA), the so called configurable I/O processor CIOP, is tightly coupled with the microprocessor based units. Functionality targeted to hardware can be specified in SDL. Additionally, standard or custom hardware modules, e.g. written in the hardware description language VHDL, can be included in the HW/SW-codesign environment. To automatically generate hardware from SDL, a compiler translates textual SDL to VHDL. This work investigates the efficiency of the generated hardware in terms of resource usage and timing properties. Different implementation models are developed together with the real-time analysis algorithms needed to determine the guaranteeable worst case behaviour.

1.3 Overview

There is a number of research groups and commercial efforts in the context of embedded systems design with SDL, some of which also include application specific hardware. An overview of them is given in chapter 2.

Chapter 3 introduces the specification language SDL. Next to the functional specification, real-time analysis requires a description of the timely properties of the embedding system, which are expressed with the help of event streams. These, together with the given deadlines, define the timing contstraints of the system to be designed. Finally, a method for including predefined, non-SDL components in the specification is suggested.

The transformation principle behind the automated generation of hardware from SDL is called the implementation model. Chapter 4 presents three implementation models for SDL, the server model, the serialized and the parallel activity thread model, as well as the combination possibilities of them. As will be seen, next to the hardware generated for each new specification, there is a number of reusable components like e.g. message queues implementing SDL's asynchronous communication, and timers to be considered. In analogy to software they are named "run-time components".

With functional specification, timing constraints and implementation model known, the real-time analysis from chapter 5 is possible. The primary task here is the calculation of the worst case response times to external events, considering the different implementation models and run-time components. As by-product, an upper bound on the necessary length of the message queues can be given. The consideration of event dependencies during real-time analysis can bring a relaxation of a possibly too pessimistic worst case scenario.

Chapter 6 details the automated design process from SDL to hardware. It describes the rapid prototyping environment in which the presented methods were integrated. The abstract SDL specification is not directly translated to a netlist of logic gates. Instead it is compiled into a high level model in the hardware description language VHDL, using the so called SDL-Compiler. After this step, reusable and additional external hardware components are integrated with the generated VDHL blocks. This includes the interface to the likewise automatically generated software. The complete VHDL model is processed further using commercial synthesis and place-and-route tools.

The rapid prototyping environment also formed the testbed for several application examples. They were used to gather on-hand experience in a real environment. The gained hardware synthesis and real-time analysis results are presented together with an evaluation in chapter 7. The work finishes with conlusions and indications of future work in chapter 8.

Chapter 2

Related Work

In the last 30 years, an extremely large number of specification languages has been developed. For a classification, [GVNG94] identifies four basic concepts: concurrency (control or data oriented concurrency), hierarchy (structural or behavioral hierarchy), communication (message passing or shared memory) and synchronization (synchronous or asynchronous). In different fields of application, languages with different characteristics have been found to be useful. A good overview is e.g. given in [JRM$^+$99].

The degree of abstraction required depends on the phase of the design process. During the requirements phase, a modeling at *system level* is appropriate. Here, the required functionality is defined. Architecture and implementation details are not fixed yet. There are few indications about timing, except for some global constraints. At the *algorithmical level* and *register transfer (RT) level*, the level of detail increases, until the implementation is completely defined.

Table 2.1 which was adapted from [MdCP01] lists typical time granularities and models of computations at the different levels of abstraction. It shows representative specification languages used in the hardware and software domain and their association to these design levels.

For the description of hardware at the algorithmical and RT-level, the languages VHDL and Verilog ([IEE00],[IEE95]) are de-facto standard with an excellent tool support. In a VHDL model at RT-level, be it a structural or a behavioural description, each operation is assigned to one clock cycle. RT-level synthesis tools, like e.g. Synopsys Design Compiler translate it to a equivalent network of registers and combinational logic. After logic optimization, resource allocation and mapping a netlist of components from a

	Time Granularity	**Computational Model**	**HW Languages**	**SW Languages**
System Level	Transactions	Task graphs, communicating processes	StateCharts, SDL, Java	SDL, Matlab, Java, Esterel
Algorithm and Functional Level	Computational Steps	Control Flow Graph, Data Flow Graph	VHDL, Verilog, C/C++ with extensions, e.g. SystemC	C/C++
Register Transfer Level	Clock Cycles	Finite State Machines, Boolean equations	VHDL, Verilog, RT-C	Assembler

Table 2.1: Specification Language Overview

target library is generated. From this netlist, vendor-specific place-and-route tools generate a layout of the digital circuit.

In contrast to this, at the algorithmical level, the timing is not or only very roughly fixed. Here as well, a description in VHDL or Verilog is usual. This kind of model can be handled by a high-level synthesis tool like e.g. [BR98]. It generates a processor, consisting of controller and data path, which implements the specified behaviour. To achieve this, the steps scheduling, resource allocation and binding have to be performed. The loosely defined timing means, that a given operation, e.g. an addition, can be scheduled in different clock cycles. This makes it possible to find a schedule, where components, e.g. the adder, can be reused and therefore resources saved.

In the last years, many institutions have attempted to introduce C-based languages for hardware design at algorithmical and RT-level (e.g. Stanford's HardwareC, Irvine's SpecC, SystemC by Synopsys, C-Level by EASICS, Cynlibs by CynApps). The idea is to use the wide-spread language C and C-based development environments by extending C with libraries and classes to express hardware specific concepts, in particular concurrency and timing. In most of these frameworks, the level of detail is exactly the same as in dedicated hardware languages. Therefore, even though the same language as for software is used, due to the implementation specific degree of detail, no unified view on the entire system can be taken. The acceptance in the hardware design community on the other hand is difficult. The willingness to adapt a new language is especially low if the advantages are not clearly seen. A further problem lies in the lacking standardization of the various proposals.

A number of HW/SW-Codesign approaches are based on the specification language SDL. A SDL specification consists of parallel processes, whose behaviour is specified with extended finite state machines. SDL processes communicate via asynchronous messages.

Pulkkinen ([PK92]) and **Turner** ([CT97]) use SDL to model electronic circuits at RT-level. SDL's event triggered semantics, the lacking features for an exact modeling of time and the missing broadcast mechanism make it poorly suited for a detailed description of electronic devices and circuits. This results in long winded, clumsy specifications. The advantage over the already presented HW modeling methods, e.g. VHDL, remains unclear, as well as how these approaches could be integrated in an automated hardware design flow.

All other approaches use SDL for a functional specification at system level. 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 ([Svo89]). In the server model, each protocol unit from one OSI 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, avoiding queueing and process management overhead.

Henke and **Mitschele-Thiel** ([HKMT97]) 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, whereby each SDL-signal-output statement is replaced by the procedure call corresponding to the signal's destination process. Here, special attention has to be paid towards a sementically correct implementation, especially in the cases "multiple output statements" and "action after output" (for more details see section 4.4). Two execution models, the basic and extended activity thread model, are presented. In the basic activity thread model, all activity threads are implemented in one operating system (OS) task; the processing of a new external signal is postponed until the processing of the previous signal has been finished. In the extended activity thread model, each activity thread is implemented in its own OS-thread. Here, timely interleaving of several activity threads is possible and depends on the applied scheduling strategy. Obviously, the procedures implementing SDL processes have to be protected with a semaphore.

Commercial code generators for SDL targeting software, like SDT's CAdvanced, CBasic and CMicro C-code generators ([Tel]), only support the server implementation model. In an implementation of a SDL specification on a single processor, the theoretically parallel SDL processes are sequentialized. The timely ordering depends on the scheduling algorithm. In the so called light integration (all SDL processes in one OS task), scheduling is implemented in the task body which is generated by the code generator. In a tight integration scheme, one SDL process equals one OS task; here, the scheduling is determined by the operating system. Naturally, these implementation decisions have a strong impact on the real-time behaviour. This problem area is investigated in [Kol01].

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.

A framework for the automated design of high-performance communication subsystems is presented by **Schiller** in [CS98]. The parallel, hardware-based target architecture consists of RISC-processors, specialized protocol function units, programmable and synthesizable protocol automata, and memory, connected via a crossbar switch. C-code for the processors, microcode for the programmable protocol automata and especially RT-level VHDL for the synthesizable protocol automata are generated from SDL. One synthesizable protocol automaton, which implements one SDL process, consists of an I/O-interface connecting it to the input queue and the crossbar switch, an ALU-interface to a local ALU, and an execution and control unit; i.e. control and data path are seperated. I/O-interface and ALU-interface are predefined components adapted to the target architecture. The execution and control unit is generated by a SDL-to-VHDL compiler. It implements only the bare finite state machine of the SDL process. The storage of state and variables and all data operations are located in the ALU. The direct generation of register-transfer level, i.e. cycle-fixed VHDL is possible, because the data width is fixed to 32 bit and the execution duration of operations of the ALU is known, once the compiler has mapped all parts of the transition to ALU-commands. This work is very specialized and efficient for communication protocols, but not easily transferable to other applications.

The SDL–to–VHDL translator described by **Glunz** in [Glu94] and [GKRM93] presents an architecture for implementing general SDL specification in hardware. The target architecture for each SDL process consists of

3 parallel VHDL entities: a receiver unit, a message queue and a processing unit. The receiver unit is responsible for the input of SDL messages. It must run independently from the other units to ensure SDL's non-blocking send-operation. The message queue stores the incoming SDL messages. The processing unit directly implements the extended finite state machine (EFSM) of the SDL process. This architecture is the basis for several other approaches ([DMVJ97],[BRM$^+$99]), and is also presented in greater detail in section 4.3. An important focus of the work is SDLs abstract communication. It presents a formalism to describe the actual hardware channels and their protocols which realize the communication. For modularity and reuse, the protocols can be layered, distinguishing between application independent and application specific. At the lowest level, the protocols are defined in RT-level VHDL. An implementation description finally maps the logical SDL signal channels to these physical channels, which are taken from a library. It further defines the necessary length and the coding and decoding procedures for the message queue. Even though the main target of this SDL-to-VHDL system seems to have been on simulation, Glunz emphasizes that the generated VHDL is synthesizable, which has been confirmed in [Rin98]. The direct generation of RT-level VHDL from SDL is possible, because the integrated protocol implementations are at RT-level and all data operations at SDL level are written in "native" VHDL and directly inserted in the generated finite state machine. The scheduling of these operation is hence the responsibility of the designer at SDL level, which represents somewhat a contradiction to the desired implementation independent SDL system specification. The modular and layered specification of receiver unit and also message queue is very flexible. Compared with "monolithic" components, however, it leads to a resource and execution time overhead, which adds to the communication overhead inherent to all server model implementations.

In [DMVJ97], the VHDL generation is embedded in the codesign environment **COSMOS**. An SDL description is translated to an intermediate format, which consists of processes communicating via remote procedure calls over abstract channels. Each SDL process is allocated one abstract channel, which represents the input message queue and its interface. The SDL processes' behaviour is translated to the corresponding finite state machine. During a partitioning step, state machines may be splitted and merged. During an interactive, human guided compilation, the abstract channels are mapped to communication units. For these, finally an implementation from a library is selected and the required interfaces and interconnections are generated. The generated VHDL model is targeted at simulation and high level synthesis, i.e. timing and the final RT-level hardware architecture are not fixed in the SDL-to-VHDL step. **Arexsys**[1], the commercial spin–off of the TIMA laboratory, markets a codesign environment based on COSMOS. The CASE-tool environment, which includes cosimulation, architectural exploration and design implementation is described in [MdCP01]). A case study, where these tools were used for the HW/SW codesign of an image processing unit at Aerospatiale Missile Division (specification in SDL, generation of C and VHDL) is presented in [ABG$^+$99]. Interestingly, in the COSMOS environment high level synthesis is no longer used for further processing of the generated VHDL. As also noted in [BRM$^+$99], high level synthesis often yields poor results when generating the controller and datapath for the VHDL model of the SDL processes' behaviour, which already is in form of a finite state machine. On the other hand, it is not possible to predict in the general case what kind of operations will be specified in the SDL transitions, and therefore how many cycles they will need in hardware. The solution, like described in [CSMJ00], is to use only the scheduling step of high level synthesis, that is to assign all operations in the generated VHDL to clock cycles. The structure of the FSM is left intact and resource allocation and binding are performed by RT-level synthesis tools.

The **Cadence Cierto VCC** environment ([Cad99a]) [2] is based on Berkeley's Polis approach ([BCG$^+$97]). Similarily to the Arexsys evironment presented above, it aims at a continuous design flow from a system level specification towards implementation. It supports a number of input languages, e.g. Esterel, C, VHDL and a subset of SDL. Next to architectural exploration, cosimulation and performance estimation, also a link to an implementation in C and VHDL is described in [Cad99b]. However, no details on the implementation of SDL processes in hardware have been found.

Slomka et al. present in [SDMH00] a rapid prototyping system for high performance communication systems based on SDL. The core of their co-design system is a Java based SDL-Compiler, which was utilized and developed further in cooperation with the rapid prototyping project presented in this work. For the data-intensive applications from the communication systems domain, a mixed hardware synthesis approach is applied ([BRM$^+$99]). Run-time components are defined and synthezised at RT-level while the behaviour of the SDL process, possibly with many data operations, is synthesized using a high-level synthesis system. To address the already mentioned limitations of high-level synthesis, a novel implementation scheme is proposed in [SDM01]. Similar to Schillers work ([CS98]), each SDL process is separated into control and data path. The finite state machine of the SDL

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

process is implemented at RT-level in one hardware or software entity, while the body of the transitions form a separate entity, which can be synthesized using high-level synthesis. In contrast to Schiller, no predefined ALU is used, but an own processing unit is synthesized for each transition. First results indicate a considerable communication overhead, which leads to a only minimally reduced resource usage.

The work of **Hemani et al.** ([SKH98], [HSK+99]) 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. What remains unclear is at which point the overhead caused by this infrastructure is smaller than the resources multiple instances provided from the start would cause. For a large number of active instances, on the other hand, the serialization due to the shared processing unit would cause a very slow response time. This in turn challenges the decision for implementation in hardware compared with software.

Concluding, it can be remarked that the large body of work concerning automated implementation of SDL in hardware utilizes the server implementation model. Much attention is paid to the realization of SDLs abstract communication on concrete components and protocols. A further issue is the refinement of timing down to the cycle-true specification required for the synthesis of a digital circuit. None of the approaches known address the problem of real-time behaviour and analysis.

Chapter 3

Specification with SDL

This work uses the Specification and Description Language SDL for the functional specification of the system under development. Section 3.1 presents the language SDL, concentrating on the language features relevant for implementation in general and on the subset which is supported in hardware.

SDL is however not intended for the specification of non-functional aspects. In the context of hard real-time systems, particularly the real-time constraints are of importance. This is dealt with in section 3.2, which introduces the language annotations chosen for expressing timing constraints, and the underlying event stream model for describing the temporal behaviour of the embedding system.

Next to the parts of the design functionally described in SDL, it is also possible to include predefined hardware components and IP in the design process, which is described in section 3.3.

3.1 Functional Specification with SDL

3.1.1 Background

SDL is a Specification and Description Language standardized as ITU (International Telecommunication Union) Recommendation Z.100 for the specification of telecommunication systems ([ITU94]). The first description of the language stems from 1980. Since then, a new language version is presented every four years, with a varying size of differences between subsequent versions. In SDL-88, the basic concepts and model of computation as well as syntax and semantic are fully defined. SDL-92 introduced object oriented design and is the language version currently supported by all commercial SDL-tools and is also the basis of this work. The latest major SDL version is

SDL-2000, which aims at a seamless link with object modelling, in particular UML, and an improved use of SDL for implementation.

SDL can be regarded as a profile of the well known UML (Unified Modeling Language) model from the Object Management Group. In some respects it exceeds the UML requirements and has well defined semantics where UML has vaguer "semantic variation points". The class models of SDL follow the usual UML object model notation. The language SDL has a graphical representation (SDL-GR) and an equivalent textual representation (SDL-PR).

Initially, SDL has been developed for the telecommunications domain. It is however well suited to describe event-driven, reactive systems in general, and has increasingly been used in the design of all kinds of embedded systems. Today, there exists an excellent tool support for SDL. It includes support for system modeling, syntax and semantic check, formal verification, simulation, links to testing and implementation in software. Largest vendor is Telelogic AB, Sweden, with its tool SDT. This work used SDT Version 3.1.

3.1.2 Language Elements of SDL-92

Like mentioned above, SDL-92 introduces object orientated features. Object orientation is useful for the specification and design process, since objects with similar characteristics can be combined in classes. In SDL-92, object classes are called "types". An actual implementation, however, always consists of instantiated objects, but not of classes. Instances in SDL can be either instantiated types or can be directly described. How these instances were derived is not important from the implementation point of view; particularly if dynamic object creation is not supported, which is assumed in this work. Therefore, without restriction of generality, SDL's object oriented features are not considered for the rest of this work. All language objects, e.g. processes, blocks, signals, are regarded as instances.

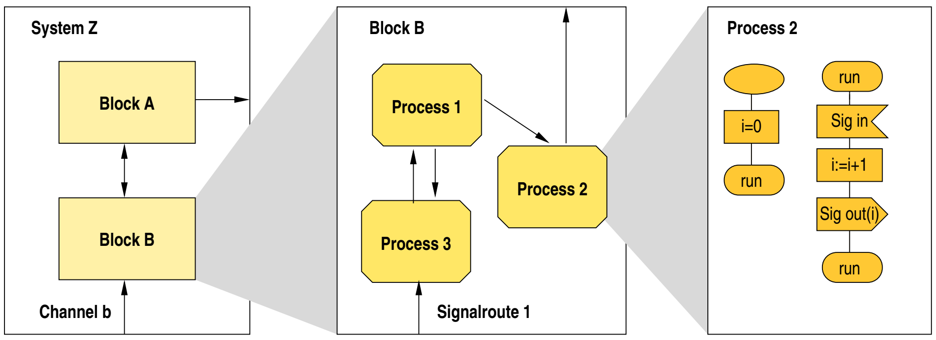


Figure 3.1: Example SDL System Structure

Structure in SDL is expressed with a hierarchy of *blocks* and *processes* (see Figure 3.1). A SDL *system* consists of one or several *blocks*, which can in turn be build of *blocks*. Communicating *blocks* and the *environment* are connected via *channels*. The lowest level of refinement are the parallel SDL *processes*, which can communicate via *signals*[1] over *signal routes*. Each *process* keeps incoming signals in its own private *message queue*.

The **Behaviour** of a SDL process is described in form of an extended finite state machine (EFSM), consisting of *states* and *transitions*. Figure 3.2 summarizes the most important elements of the EFSM with their graphical and textual representation.

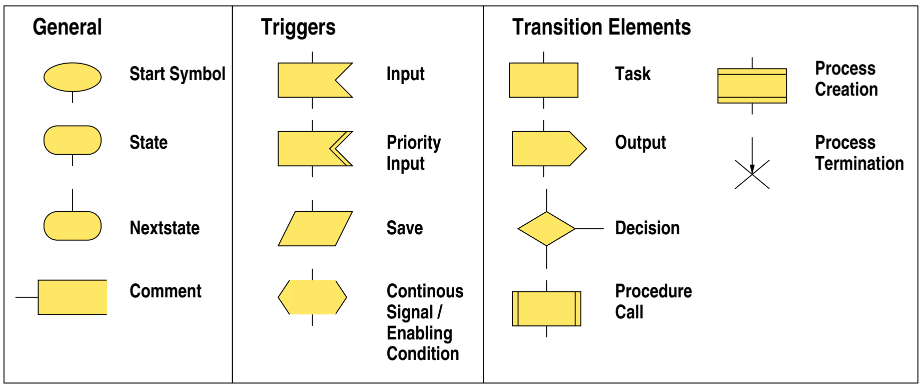


Figure 3.2: SDL-92 EFSM Elements

The *start symbol* denotes the *initial transition*, which is executed once at the start of the SDL system. It brings the SDL process into a defined state, but can also contain arbitrary transition elements.

SDL knows a number of different *triggers* which can be defined for a transition from a state: The signal *input* triggers the transition upon the arrival of the denoted signal, removing the signal from the message queue. A *continuous signal* triggers when the given boolean expression evaluates true. *Enabling conditions* combine the first two, triggering only when the signal is availiable and the condition is fulfilled; the signal remains in the queue until this is the case. *Priority inputs* are a special case of the *input* symbol. They define that the given signal is to be consumed first, even if it is not the first in the message queue. The *save*-statement indicates that the given signal has to be kept in the queue, even if it has no transition to trigger in the current state.

[1]In cases where a confusion with VHDL signals is possible, SDL signals will exchangeably be termed SDL messages.

The transition body can contain an arbitrary number of *tasks*, *outputs*, *decisions*, *procedure calls*, and the *nextstate*-symbol, to name the most important. A *task* is a container for data operations, which can be expressed formally using arithmetic expressions, but also informal text. The *output*-statement sends the given signal to a given process or signal route. Recurring parts of the state machine can be put into a *procedure*, which can be called from anywhere in the process. *Nextstate* defines the state that has to be assumed at the end of the transition. The *process creation* and *termination*-symbol specify the dynamic creation and termination of the given process at run time, which is however excluded from the implementation in hardware.

SDL's chief communication mechanism is asynchronous sending and receiving of SDL signals, which can optionally carry data. Generally, data is declared local to each process and cannot be accessed from outside. There are however three exceptions to this rule. *Remote procedures* belong to the context of the exporting process, but can be called from a different process, and therefore can allow access to the remote process's data. The *import*-construct allows reading local variables of a remote process, but only values, which have been released with the explicit *export*-statement. These two mechanisms form no principal exception from SDL's basic communication scheme. They can in fact be implemented using signal interchange and additional states. In contrast to this stands the *view/reveal*-mechanism, which allows a process to always read the actual value of the revealed variable of the remote process. The use of view/reveal is not recommended by Z.100.

Data types in SDL are abstract data types, i.e. the definition of literals, operators and their semantics is possible in the system specification. There are, however, a few predefined data types for frequently used types like *boolean*, *integer*, *real*, *character*, *charstring*, *time* and *duration*, as well as composite types like *array* and *struct*. With the help of a *syntype*-definition, a new name for an existing data type can be given and, useful with regard towards an implementation in hardware, its range of values can be limited.

SDL assumes a global **Time**, whose actual value is accessible with the *now*-statement. It makes, however, no assumptions on a timely synchronization of actions, nor on communication and execution times. Similarily, the scaling of the predefined *time* and *duration* data types is left as an implementation issue. A *timer* can be started from inside a process transition. After the given duration, it sends a signal to the requesting process, using the process' message queue.

3.1.3 Model of Computation

Blocks in SDL are useful as hierarchical structuring element, and e.g. define the visibility of viewed variables, but they have no influence on the behaviour of the SDL system, which is entirely defined by the processes. SDL processes are defined to be concurrent, i.e. no specific execution order is prescribed. Synchronization and communication take place asynchronously over the SDL signals, with a non-blocking send and blocking receive. Processes are therefore modelled as independent from each other as possible, meaning communication and synchronization must only be realized where it is explicitly demanded by the specification with the statements send, import, view and remote procedure call.

Processes are activated when a transition is triggered. Like outlined in section 3.1.2, the two basic trigger mechanisms are signal input and continuous signals, where a boolean condition is evaluated. The variables contained in such a boolean equation, however, can only be local variables or imported/viewed variables from other processes. To trigger a transition, the values of these variables would have to change, which in turn can only happen inside a triggered transition. This means, that an activation of any part of the SDL system can only be initiated by either a signal from the system's environment or an output from a timer. Signals from the environment, i.e. external events, and timer outputs are subsumed with the expression *triggering events* of the SDL system. [2]

The detailed model of computation of a single SDL process is as follows: First, the message queue is checked if it contains a signal which has a priority input in the current state, and if true the corresponding transition is executed. Secondly, the first signal in the queue is evaluated. If an input for this signal is specified in the current state, and if applicable the enabling condition evaluates true, the transition is triggered, and the signal is removed from the queue. The signal remains in the queue as long as the enabling condition is false. If there is no input statement for the signal, it is removed from the queue and discarded. In the case of a save-statement, it is kept in the queue. At this point, the next signal in the message queue is regarded. Only if there is no more signals in the queue to be processed, the continuous signals are evaluated.

This execution policy means that the processes' message queue cannot be a simple First-In-First-Out-Buffer (FIFO). In order to realize priority input, save and enabling conditions, it has to be possible to view all buffer elements, and to remove signals from random positions. If these three language

[2]An exception are the so called *spontanous transitions*, which are used to model random behaviour, like e.g. system failures.

elements are excluded, the message queue is reduced to a FIFO. Obviously, this has consequences on the modeling style. Without a save-statement, for instance, it is no longer possible to use the message queue as a buffer for data to be processed later, since the signals are removed from the queue as fast as possible, either to be consumed or discarded.

Returning once again to the classification criteria proposed by Gajski mentioned at the beginning of chapter 2, the specification language SDL is characterized by *control oriented concurrency.* In contrast to data-oriented languages, which describe data streams and their interaction, SDL processes specify concurrency and synchronization of the control structures. SDLs blocks imply a *structural hierarchy.* Since process behaviour cannot be further refined in SDL-92, no behavioural hierarchy is given. *Communication* is mainly realized via *message passing* with the exception of the view/reveal-mechanism which implements a shared memory concept. *Synchronization* is *asynchronous*, in an event-driven fashion using messages (SDL signals).

3.1.4 Discussion

Summarizing the previous sections, SDL is excellently suited for the specification of control-dominated, event-driven systems, whose behaviour can be described in the fashion of a finite state machine. Descriptions in SDL are advantageous for implementation on distributed systems, since a minimum on assumptions is made concerning synchronization and communication between processes. SDL favours an implementation independent specification at an abstract level. It dissociates the functionality from the communication and the interface to the outside, with advantages in reusability and maintainability. Further, already mentioned advantages are its standardizition, maturity, popularity and well developed tool support.

On the other hand, SDL is not intended for an explicit description of data flows, nor for continuous systems. It is possible to create a periodic behaviour with the help of one or several timers, which trigger arithmetic operations inside the finite state machines. Tools like Matlab or MatrixX, however, are certainly better equipped for the design of e.g. a closed loop control. Neither has SDL been envisioned for a detailed description of an implementation or for instance defining an architecture of processing units, busses and software components.

A major weak point in SDL is the concept of time. The language has no provision for a precise specification of timing constraints. Expressing time with the help of SDL timers is, according to [Ols94], only possible in systems where “tolerances on the time intervals are $>$ 100 times the average instruction time of the CPU”. The problems arising with the SDL timer concept particularly in software implementations on a single processor are adressed in [Kol01]. Further disadvantages include the lack of a broadcast mechanism and the lack of a behavioural hierarchy inside SDL processes, which can lead to cluttered processes.

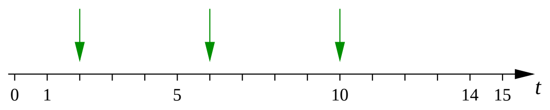
3.2 Timing Constraints

Reactive systems can be modelled adequately in SDL, where a SDL system is activated by triggering events, i.e. a timer output or an external event from the environment. In reactive real-time systems, this response to an event has to occur within a given time, the *deadline*. In the context of reactive systems, relative, end-to-end deadlines can be assumed. A real-time analysis has to prove that the system under development will meet the given deadlines. To be able to perform such a real-time analysis, the timely behaviour of the environment has to be modelled. It is necessary to have information, e.g. bounds on, the timely distribution of the triggering events, since the reaction time to one event can be influcenced by other events, depending on the time of their occurence.

In hard real-time systems, where a deadline miss has to be avoided under all circumstances, the *worst case* has to be covered. In this case, the well known methods which have been developed to describe queueing systems ([Kle75]) cannot be applied, since they are based on statistics and therefore only allow statements on e.g. mean values and distribution, but not on the worst case. On the other hand, it is not useful to enumerate all possible points in time of the occurence of the different events, and it is a severe restriction to stipulate that events may only occur at a few, discrete points in time (e.g. only periodically). The method employed instead is to give *bounds* on the timely distribution of the events. For real-time analysis, as will be seen in chapter 5, the minimum timely distance of events is relevant. A very powerful model to express this is the *event stream model* proposed by Gresser ([Gre93b]), which will be outlined below.

Event stream model The basic information expressed by the event stream model is how many events of a type can occur within a given time interval I, under the assumption that I can be located anywhere in time.

$$\textbf{Definition: } a_j \text{ is the smallest time interval in which } j \text{ events can occur} \tag{3.1}$$



$a_1 = 0, \quad a_2 = 4, \quad a_3 = 8$

Figure 3.3: Finite event sequence

Obviously, $a_1 = 0$ is always given, since one event can occur in one instant. The small example in Figure 3.3 depicts the worst case event sequence consisting of three events that would satisfy $a_1 = 0, a_2 = 4, a_3 = 8$. The time in this and in the following examples is given in absolute numbers without measurement unit; it can be multiplied with any time unit. Time values from the set of real numbers are assumed. To find out if any given event sequence complies with a_j, a window of size a_j has to be moved along the time axis. At no time, more than j events may show in the window.

In order to be able to also completely characterize a *infinite* sequence of events, a second parameter, the cycle z_j is introduced:

Definition: $a_j + k \cdot z_j$ is the smallest time interval in which
$(k + 1) \cdot j$ events can occur,

$$k \in \mathbb{N}_0 \quad (3.2)$$

Now, like depicted in Figure 3.4, an event sequence with period 4 can be described with $a_1 = 0, z_1 = 4$.

a_j and z_j together are termed an **event tuple**:

$$\begin{pmatrix} \mathit{Cycle}\ z_j \\ \mathit{Interval}\ a_j \end{pmatrix}$$

If no cycle can be given, $z_j = \infty$ is assumed. An **event stream** is a set of event tuples, whereby the restrictions given by all event tuples have to be equally fulfilled. Now it is possible to concisely describe complex event combinations, e.g. sporadic events, jitter and event groups.

$$ES: \quad \left\{ \begin{pmatrix} z_j \\ a_j \end{pmatrix} \right\} = \left\{ \begin{pmatrix} z_1 \\ a_1 \end{pmatrix}, \begin{pmatrix} z_2 \\ a_2 \end{pmatrix}, \ldots, \begin{pmatrix} z_n \\ a_n \end{pmatrix} \right\} \qquad (3.3)$$

The **event function** $E(I)$ expresses the constraints given by the event stream in a different fashion: For any given time interval I it gives the maximum number of events which can occur in it, taking into account all event tuples. The contribution $E_j(I)$ of one event tuple j to the event function is given as:

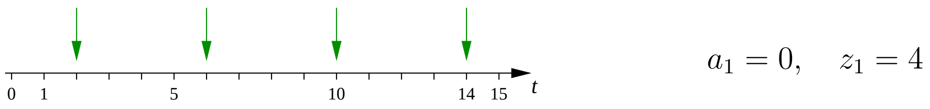


Figure 3.4: Periodic event sequence

$$E_j(I) = \begin{cases} 0 & ; I < a_j \\ \left\lfloor \dfrac{I - a_j}{z_j} + 1 \right\rfloor & ; I \geq a_j \wedge z_j < \infty \\ 1 & ; I \geq a_j \wedge z_j = \infty \end{cases} \tag{3.4}$$

The operator $\lfloor x \rfloor$ denotes the integer-function, which rounds off its argument to the nearest integer smaller or equal than x. Figure 3.5 shows graphically the contribution $E_j(I)$ of one event tuple i to the event function.

The event function $E(I)$ of the event stream is the sum of the contributions of all n event tuples:

$$E(I) = \sum_{j=1}^{n} E_j(I) = \sum_{j=1}^{n} \begin{cases} 0 & ; I < a_j \\ \left\lfloor \dfrac{I - a_j}{z_j} + 1 \right\rfloor & ; I \geq a_j \wedge z_j < \infty \\ 1 & ; I \geq a_j \wedge z_j = \infty \end{cases} \tag{3.5}$$

Figure 3.6 depicts an example event stream with one possible event sequence compliant with these restrictions on the left-hand side of the figure. Expressed in words, the event stream poses the following bounds: In a time interval of length 3 never more than 3 events may occur, whereby any time interval of length 1 may contain 2 events at maximum, and only one event may occur at the same time (interval length 0); this can repeat itself after 7 time units. The event function corresponding to the given event stream is drawn on the bottom of the figure.

However, not every combination of event tuples constitutes a valid event stream, in the sense that in contains no internal inconsistencies. Such a inconsitency is given when the minimum interval for a number of events can only be reached by violating the restrictions given by a different event tuple. This is expressed by the following equation, which has to be met for all I to obtain a valid event stream:

$$\forall I_2 > I_1 : E(I_2) - E(I_1) \leq E(I_2 - I_1) \tag{3.6}$$

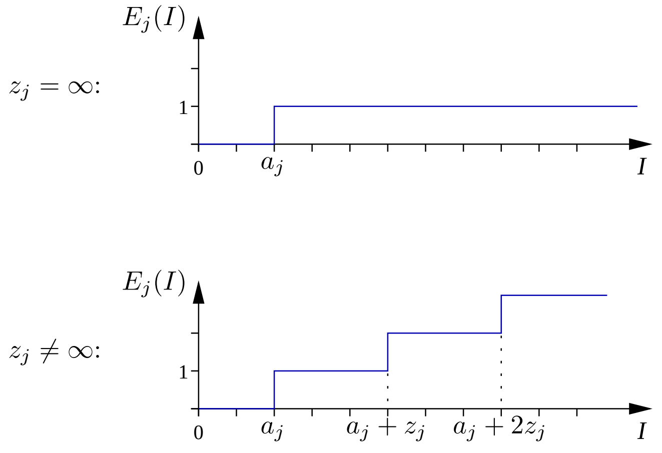


Figure 3.5: Contribution to the event function by one event tuple

Event dependencies While event streams describe the temporal characteristic of one event type, event dependencies deal with relations between events of different types. Like before, minimal time distance between events are of interest for a worst case analysis. Therefore, in addition to the event streams, for each group of dependent events a **event dependency matrix EDM** is given:

$$\mathbf{EDM} = \begin{pmatrix} ed_{11} & ed_{12} \\ ed_{21} & ed_{22} \end{pmatrix}$$

Each element e_{kl} of the EAM denotes the minimal temporal distance between an occurrence of k and a subsequent occurrence of l. Obviously, ed_{kk} is equal to the distance between two events of a type a_2 given by the event stream.

Summary The temporal characteristics of the embedding system are specified with deadlines, event streams and event dependencies, which give a lower bound on the timely distance of events. Event streams and deadlines are annotated to the triggering events (external signals and timer outputs), event dependency to the system properties in the SDL system. Keywords and syntax of these annotations are subsumed in Table 3.1. Deadlines, event streams, respectively the derived event functions, and event dependencies are input to the real-time analysis presented in chapter 5.

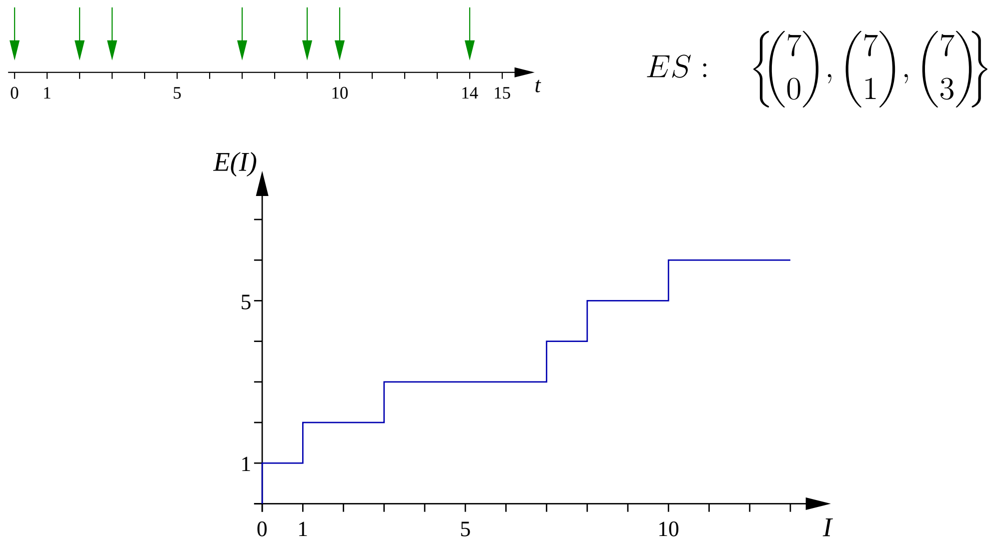


Figure 3.6: Event stream example

Real–Time Extensions	
`#RZA ES{` $(z_1\ \ a_1)\,[,(z_j\ \ a_j)]*$ `}`	event stream ES
`#RZA Deadline` d	deadline d
`#RZA ED` $k\ \ l\ \ ed$	event dependency ed_{kl}

Table 3.1: SDL Annotations

3.3 Predefined Components

From the point of view of the SDL system, predefined components are functional blocks which are specified in a different language than SDL. The scope of this work does not include general multi-language design like outlined in [JRM$^+$99], e.g. a specification given in equal parts in SDL and Matlab. The focus here, with a specific view towards implementation in hardware, is on standard components for which highly optimized designs already exist. An example are peripheral components at the interface to the embedding system like e.g. a PWM module, for which commercial IP versions are available.

It is possible to explicitly include such a predefined component in the SDL system specification. A SDL block is assigned to represent the predefined component as a black box. In an annotation to SDL, the assignment to the component is fixed. The SDL block can either be left empty; this is allowed in SDL to account for early specification phases. Or it can contain a description of the component's behaviour in SDL, if the component should be included in the simulation at SDL level. The predefined component is linked to the rest of the SDL system using the well known communication methods: When SDL signals are used, the included components actively communicate with the SDL processes via the message queues. With shared variables and remote procedure calls, on the other hand, the component can be accessed from within a SDL process, i.e. in the task bodies of the finite state machine transitions.

The inclusion of the components in the final SDL implementation, as well as the realization of the specified communication with the rest of the SDL system, have to be performed as part of the automated design process, which is described in chapter 6.

Chapter 4

Implementation of SDL in Hardware

This chapter describes how an implementation of a SDL specification can be realized in an application specific electronic circuit. For this, an important influence is the state-of-the-art hardware design method, which is shortly introduced in section 4.1. It is based on an abstract specification of the hardware in the description language VHDL, which is synthesized into a synchronous circuit design using commercial tools.

Section 4.2 shows the hardware implementation of a single SDL process, which is specified in VHDL. The basic architecture consists of a hardware implementation of the processes' finite state machine and additional so called run-time-components, which implement SDL's abstract communication, the message queue, and e.g. timers. This VHDL model is independent of the intended target architecture, be it a single FPGA tightly coupled with software execution units like in the already mentioned rapid prototyping system, a multi-FPGA system or an ASIC. How such a VHDL specification can be automatically generated from SDL, and the integration in a hardware/software-environment, is detailed in chapter 6.

The architecture presented in section 4.2 is basis of the different implementation models for an entire SDL system. The most straightforward of these, the server implementation model, is presented in section 4.3. It directly reflects the process structure of the SDL system, and is the model used in all other hardware implementation approaches for SDL.

In contrast to this stands the activity thread model known from the software domain, which functionally combines the activities in the SDL system which are triggered by one event. It is introduced, with two implementation alternatives, in section 4.4. Section 4.5 finally presents techniques for combining the two implementation models.

4.1 Hardware Design with VHDL

The language VHDL (VHSIC hardware description language) originated in the 1980ies from the VHSIC (Very high speed integrated circuit) programme of the U.S. department of defense. It has been standardized by the IEEE ([IEE00],[LWS94]), and is by now, next to Verilog, quasi-standard in the hardware design domain.

A VHDL model consists of two major parts: The *entity* defines a component's interface, i.e. its inputs and outputs. One or several *architectures* describe the component's functionality. VHDL allows different views on the described hardware. In a *behavioral description* a component is defined through the reactions of its output signals to changes in the input signals. *Sequential statements*, which may occur inside a so called VHDL *process*, include constructs like if-else-branches, loops, data operations or procedure calls, similar to software programming languages. Every statement, e.g. a signal assignment, outside of a process is a *concurrent statement*, i.e. it expresses the hardware's parallelity. A *structural description* defines a component by its composition of sub-components, which in turn are defined in VHDL. Structural and behavioral descriptions can be used side by side in one model.

The different levels of abstraction known in hardware design have already been briefly introduced in chapter 2. A VHDL description at *algorithmic level* typically uses functions, procedures, processes and control structure to express functionality, but has no relation to a concrete realization in hardware, and no timing is fixed. At *register-transfer level* the circuit's properties are defined by operations and the transfer of the processes' data between registers. The temporal sequence of operations is given, i.e. all operations are assigned to fixed clock cycles and reset-signals are integrated. The description can be structural, where registers, adders, multiplexers, etc. are connected, or behavioural. Behavioural descriptions at RT level typically are of finite state machine style, where statements of the type `wait until clock'event and clock = '1'` indicate the different clock cycles. A VHDL description at RT level already determines implicitly the structure of the electronic circuit. At the *logic level* the circuit is described by logic operations on digital signals and their temporal properties (delays). A behavioural view at this level consists of boolean equations, while a structural description is a netlist connecting basic elements, i.e. AND/OR-gates and Flip-Flops, from a target technology dependent library.

While it is possible to write a VHDL specification at all levels of abstraction, the state-of-the-art design flow starts at algorithmic or RT-level and uses commercial tools to create a circuit design. Figure 4.1 depicts a possible design flow targeting a FPGA architecture. High level synthesis takes a specification at algorithmic level and generates a RT-level architecture of a data-path and a controller. The main steps in this process are the assignment of components (resource allocation) and of clock cycles (scheduling) to the operations. Logic synthesis transforms a VHDL specification at RT level into a netlist at logic level. Depending on the tool, only a subset of the VHDL language can be synthesized. Usually vendor specific tools perform placing and routing of the netlist, generating the programmable bitfile, which contains the circuit design for the FPGA.

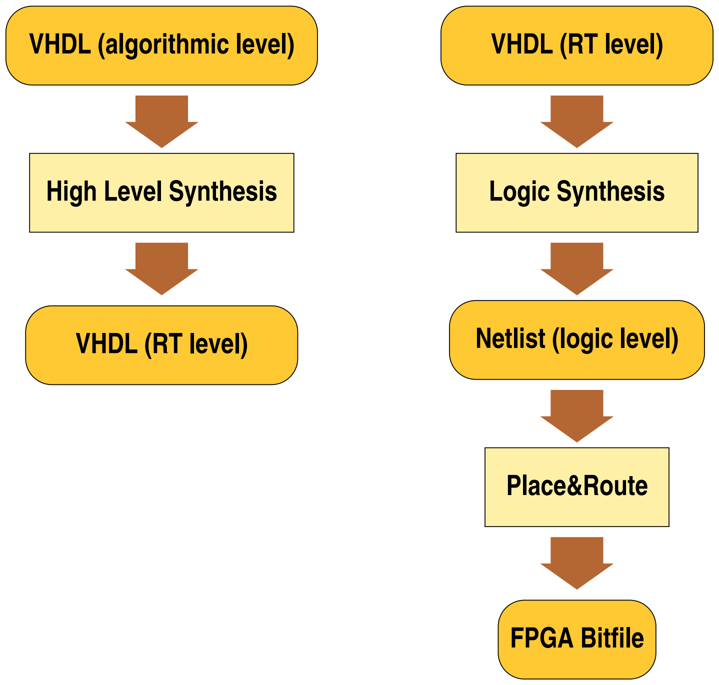


Figure 4.1: Hardware design flow targeting FPGA

4.2 Implementation of a Single SDL Process

4.2.1 Component view

A SDL specification consists of processes, which communicate over abstract communication primitives. For the implementation of a SDL process in hardware, the processes' behaviour must be realized, and a concrete implementation of the communication must be found. The latter, in contrast to the process behaviour, is not explicitly contained in the specification, but is defined implicitly by SDL's model of computation. The functionality, i.e. message queue, send- and receive-primitives, is required by all processes, but at the same time highly dependent on the actual conditions of the implementation, e.g. the available communication hardware. The concept therefore is to employ reusable components from a library, termed run-time components. In analogy to the software world, they provide an infrastructure for the SDL process and serve to isolate the application from the underlying target architecture. Next to the SDL communication, this concept is also applied to the implementation of SDL timers. The run-time components are not generated automatically, but hand-written and optimized. They are most efficiently specified in VHDL at RT-level. Like indicated in [BRM^{+}99], further run-time components are conceivable, e.g. for the storage of process variables in a separate memory. They are however not considered in this work. Further it should be noted that dynamic process creation is not supported in this work. The result of the above considerations is the basic architecture for implementing SDL in hardware, which is shown in Figure 4.2. Expressed in VHDL, it corresponds to a structural view of the implementation.

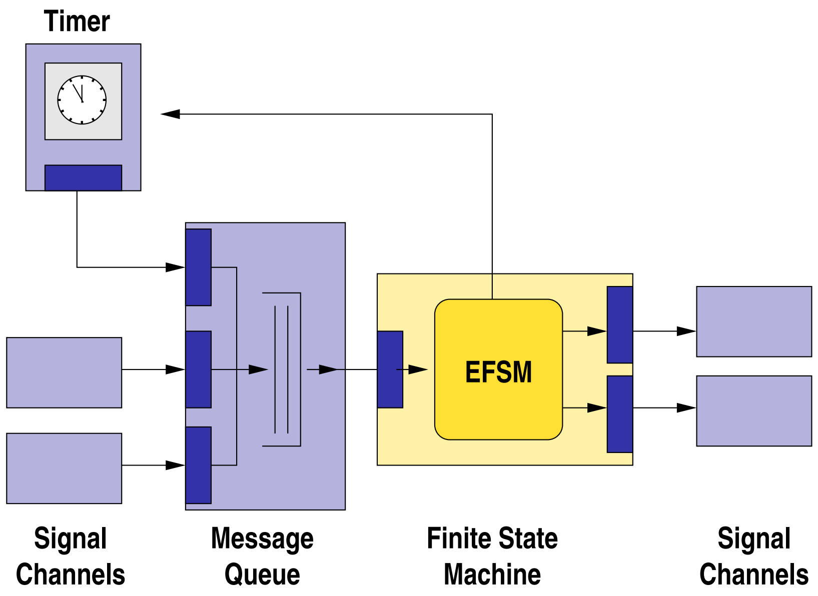


Figure 4.2: Basic hardware architecture implementing one SDL process

Section 4.2.2 describes the implementation of the process behaviour in the block EFSM (extended finite state machine) in detail. The refinement of SDL's communication, the selection of the run-time components and their inclusion in the hardware design, which is a focus of most projects targeting SDL to hardware (e.g. [Glu94], [DMVJ97]), is addressed in chapter 6. In the following, the functionality required of the run-time components is defined, and a brief characterization of the component library which has been used in the exemplary implementations on the rapid prototyping target architecture is given.

The **signal channel** is the actual medium over which SDL signals are sent, which consist of a signal ID and optionally additional data. Obviously, a great variety is possible, depending on the source and destination of the channel – hardware, software, the system environment –, the realization of the channel – point-to-point, bus – and the channel's protocol. For a signal output, a write on the channel occurs from inside the EFSM. Therefore, a send macro or procedure implementing the channels protocol is needed for the EFSM implementation. Analogously, the message queue requires a macro for receiving signals from the channel. At both sides, coding and decoding functions for signal ID and data have to be defined.

The processes' **message queue** realizes SDL's asynchronous communication insofar as a signal can be received even if the process is currently not ready to process it. This makes a non-blocking send possible, within the limits given by the signal channel's protocol. The message queue has input interfaces to one or several signal channels, implementing the channels' protocol, and an interface to the finite state machine, where the EFSM requests a signal when it is ready to process it. If the process contains no save-statement, priority input or enabling condition, the message queue can store the signals after a simple first-in-first-out principle. In this case, the requested SDL signal is always removed from the queue. If one of the three mentioned trigger conditions occurs in the process, the EFSM determines if the current SDL signal is consumed in the current state, or if it has to be kept in the message queue. In the latter case, the next signal in the queue is regarded.

In SDL specifications, a message queue of infinite length is assumed. For the implementation, however, the queue has to be dimensioned carefully. Since the signal storage causes a very high hardware effort, it should be as short as possible. A message queue which is too short, on the other hand, violates SDL's semantics since it causes an unexpected delay, in an extreme example it can cause a deadlock. A real-time analysis, like presented in chapter 5, can give bounds on the required size. In practice, a message queue length of 1 or 0 is often sufficient. If it can e.g. be shown, that the minimal distance of triggering events is always longer than the execution time of the SDL process, the special case of message queue length 0, i.e. the completely synchronous coupling between sender and receiver, can be applied.

The SDL **timer** is well suited to be implemented in an independent runtime component. It can be reset and started from inside the process, and the duration can be set. If started, it runs independently of the EFSM and notifies its termination by sending a SDL signal to the message queue. For the message queue, the interface to the timer is not different from any other signal channel.

The run-time components used in the examples (see also Appendix C) are implemented fairly straightforward. Connections are made point-to-point over simple wires, with a simple hand-shake protocol. The message queue's input interface guards all input channels, performing the hand-shake if it detects a send request. If several channels attempt to send simultanously, a fixed order is given. EFSM and message queue equally perform a simple hand-shake. The queue `queue_ln` stores the SDL signals in a FIFO from a commercial IP-library for queue sizes larger than 1. A message queue of size 1 (`queue_l1`) contains one register. In the queue of length 0 (`queue_l0`), signal channel is directly connected with the EFSM, and no signal storing is performed.

Further components form the necessary interfaces between SDL process and message queue and the embedded system's environment and the HW/SW-interface. One set of components (`rwdecode` and `doutmux`) provide the interface to the local bus connecting the FPGA with the software-based units, decoding the bus signals and accessing the bus over three-state buffers. In the case of a write access from software, the component `write_to_sdl_signal` intermediately stores the SDL signal and sends it to the destination process using the hand-shake protocol. For the direction hardware to software, the component `hw_sw_queue` receives the signals to send from all processes and stores them in a queue. If the queue is not empty, the software-based unit is notified over an interrupt and can read the signal from the queue. At the border to the embedding system, the component `edge_to_sdl_signal` observes the signal level of a defined FPGA output pin, and upon observing a previously defined rising or falling edge sends a predefined SDL signal to a process. In addition to that, direct read and write access to external pins is possible from within the SDL processes.

4.2.2 Finite State Machine Implementation in VHDL

One hardware entity implements the behaviour of the SDL process strictly after SDL's model of computation presented in section 3.1.3. The specification of this hardware entity is given in VHDL. Before the details of the implementation are presented, however, a discussion is necessary which level of abstraction is suitable.

The SDL processes' behaviour is given in form of a extended finite state machine. Like mentioned in section 4.1, this kind of description is customary in a RT-level specification. It has to be observed, though, that the states of the SDL EFSM do not directly correspond to states of the hardware circuit. The state of a synchronous electronic circuit denotes everything that has to be stored in a register between two clock cycles. A complete SDL transition, however, will usually require more than one clock cycle, since a hand-shake with the message queue has to be performed and possibly a signal has to be sent requiring the execution of the signal channel's protocol. Additionally, the transition can contain arbitrarily complex computation inside the SDL tasks. A direct transcription of the EFSM in VHDL will therefore yield a so called implicit state machine description, containing additional states in the transitions. Commercial synthesis tools are able to translate this type of specification into an explicit state machine for further processing.

SDL however does not define the exact timing information required at RT-level. In the case of the interfaces to message queue and signal channels, the exact timing is given by the RT-level specification of the components, like outlined in the previous section. This is not the case for the computation inside the transitions. If the process behaviour is specified at algorithmic level of abstraction, the high-level synthesis tool performs the scheduling of operations to clock cycles. In addition, the separation of controller and data path makes component reuse possible, which is particularly interesting if many complex data operations are contained in the SDL tasks. [BRM+99] indicates how an algorithmic specification and the cycle-fixed communication interfaces can be integrated during high-level synthesis. The drawback of high-level synthesis, on the other hand, is a considerable overhead incurred by separate controller and data path. It can particularly be found in applications with low computational complexity, where little can be gained by resource sharing.

Taking into account the assumption of section 1.2 that mainly simple control-dominated processes are targeted to hardware, the following proceeding is therefore proposed: SDL processes are implemented in RT-level VHDL, applying a very rough scheduling whereby each SDL task is assigned to a clock cycle. The underlying assumption that this is a valid schedule has proven to be true in the majority of the conducted application examples. For cases where one SDL task's operation does not fit in one clock cycle, the scheduling step of high level synthesis is used. The EFSM structure is conserved, and after scheduling RT synthesis is performed. A similar combination of RT and high level synthesis is presented in [CSMJ00].

Figure 4.3 depicts the basic VHDL frame implementing one SDL process, considering only the case of a simple FIFO message queue. The EFSM is specified in one VHDL process, i.e. in a behavioural description, which can be easily derived from the SDL model. During reset, signals and local variables are initialized. The second branch is evaluated only once after reset

```
main: PROCESS
   BEGIN
   IF reset = '1' THEN
      -- initialize data --
   ELSIF init = '1' THEN
      -- execute start transition --
   ELSE
      IF (signal in queue) THEN
         -- get signal from queue --
         CASE signal IS
            WHEN a =>
               CASE state IS
                  WHEN Z =>
                     -- execute transition 1 --
                  WHEN Y =>
                     -- execute transition 2 --
               END CASE;
            WHEN b =>
               -- execute transition 3 --
         END CASE;
      ELSE
         -- evaluate continuous signals
      END IF;
   END IF;
   WAIT UNTIL clock = '1' AND clock'event;
END PROCESS main;
```

Figure 4.3: VHDL frame for one SDL process

and contains the start transition. During normal operation, the message queue is regarded first. If it contains a signal, it is removed and evaluated. Depending on the current state, which is kept in a local variable, a transition is executed. The EFSM can contain conditional branches, data operations and the assignment of a new state, which are all directly translated to VHDL. For the sending of a SDL signal, the appropriate channel protocol is inserted. Only if the message queue is empty, continuous signals are evaluated and the respective transitions executed.

Local variables of the SDL process are translated to VHDL variables belonging to the process scope, using the VHDL data types analogous to the predefined SDL data types "boolean" and "integer". View/reveal and import/export are implemented by direct wires to and from the process, the latter with explicit assignment of the new visible value. Remote procedure calls can be resolved at SDL level by additional wait states and signal exchange. In that fashion, they can be likewise integrated in the presented VHDL frame.

Commercial CASE-tools often extend SDL by proprietary commands which are inserted inside SDL comments, e.g. for assigning a priority to SDL processes or the inclusion of code in the target implementation language. In the latter case, the specified native C-code is inserted in the implementation by the code generator. The same is possible for a hardware target as well, by inserting VHDL-code contained in the specification inside a task statement. This proceeding leads to implementation dependent descriptions and breaches the benefits of standardizing SDL, but is often useful for a last fine-tuning for efficiency.

4.3 Server Implementation Model

A SDL specification typically consists not of one, but of several processes interconnected by signals. In the server implementation model, each SDL process is implemented on its own after the basic architecture presented in section 4.2. The processes, each with its own EFSM and private message queue, are connected via signal channels. The term server model is chosen for this straightforward implementation scenario because here each SDL process acts like a server, which exclusively waits for requests in the form of triggering SDL signals. In SDL, processes are concurrent. In a software implementation, several processes usually have to share one processor. In a hardware implementation, in contrast to this, SDL processes are implemented in truly parallel fashion. Several processes may be active at the same time. In fact, each SDL process in hardware can be regarded like one small processor implementing the specified behaviour.

Figure 4.4 shows an exemplary SDL specification and figure 4.5 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 $m_{i(j+1)}$.

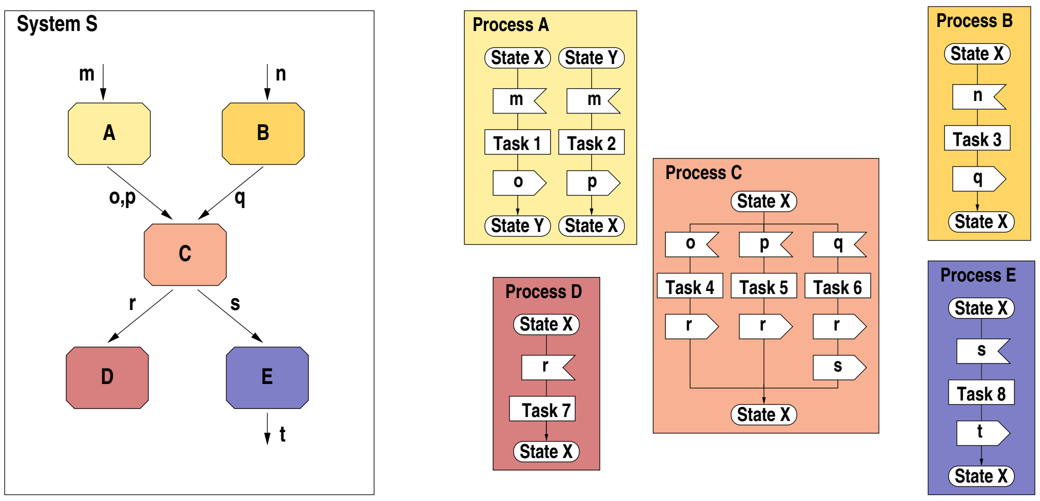


Figure 4.4: SDL specification example

4.4 Activity Thread Implementation Model

The activity thread implementation model takes an alternative view of the SDL specification. Seen from a black-box view, the implementation displays the behaviour defined in SDL, but is internally built after a different fashion. The activity thread model takes advantage of SDL's property outlined in section 3.1.3, that a SDL system is only active if triggered by an external event from the environment or a timer output. It analyzes the chain of activations in a SDL system caused by such a triggering event. The event is received by an SDL process, triggers a transition, where in turn an SDL signal may be sent to a second process, and so forth. This chain of activations in such a task precedence system 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 consummation of an SDL message in a process without triggering a new SDL output. Figure 4.6 shows the task precedence graph of the SDL example introduced in figure 4.4, consisting of two activity threads.

All actions and state changes contained in the transitions along an activity thread are implemented sequentially. Signal outputs are replaced by the activities triggered in the receiving SDL process , thereby avoiding the message send and receive overhead between the processes.

An activity thread is implemented in one VHDL process. It requires the basic architecture and run-time support presented in section 4.2. A message queue, whose length is determined by real-time analysis, ensures that no signals are lost. SDL signal output to the environment is realized over

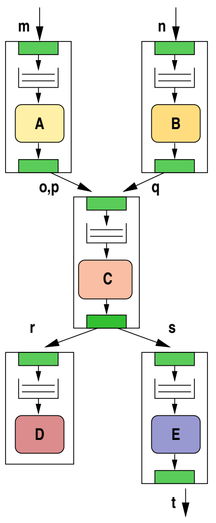


Figure 4.5: Server model implementation

send-macros to signal channels. The activity thread's behaviour is likewise implemented in a finite state machine, which is depicted in Figure 4.7. The VHDL realization of the content of the transitions is identical to the already presented EFSM implementation of a single SDL process. The only difference lies in the control flow. Several levels of conditional branches are introduced for the state choice of all SDL processes involved in the activity thread. Depending on the SDL model, a high degree of nesting may be reached. Conditional branches for the various input signals on the other hand are superfluous.

Since all internal communication is abolished, signal channels and message queues are saved. This has the additional benefit that, given a low complexity of transition, several transitions belonging to different SDL processes can be scheduled in one clock cycle. This reduction of execution time is not possible in the server model, because transitions are separated by the signal channels.

Special attention has to be paid to a semantically correct implementation concerning process data consistency. The state transitions of one process must exclude each other mutually, i.e. they may not be executed at the same time and one transition must be finished, before the next transition of the same SDL process is executed. To ensure this, it is necessary to reorder the execution of each transition in such a fashion that the signal output, which in the activity thread model is replaced by the transition of the destination process, comes after all other operations. This constitutes no change in the specified behaviour, but only delays the start of the next transition until the

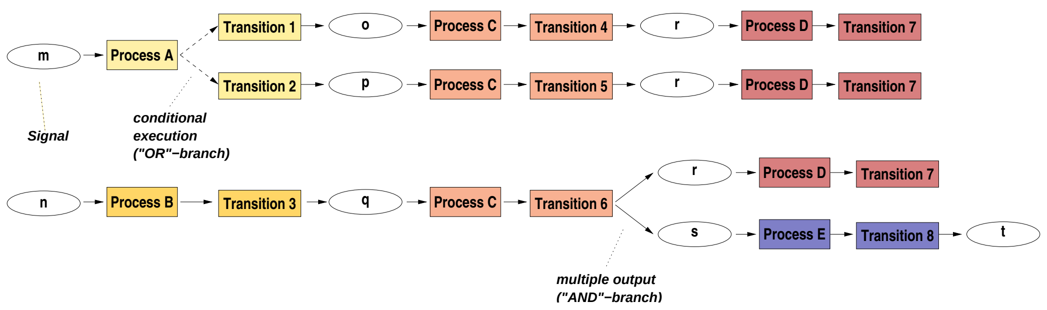


Figure 4.6: Task precedence graph of the SDL system from figure 4.4

current transition is finished. In the other case, a loop in the SDL system could make it possible that a second transition of one process is started before the first has been finished.

The activity thread branches when several SDL outputs are contained in one SDL transition. This can be resolved inside the VHDL process by executing the branches one after the other. To achieve a parallel execution, the branches have to be implemented in concurrent VHDL processes, which can be connected with the help of the methods presented in section 4.5.

In general, a SDL system has more than one triggering event and therefore consists of several activity threads. In hardware, each activity thread could be executed in parallel. This would correspond to a dedicated processor exclusively waiting for each external event. Depending on the type of application, and on the temporal specification of embedding and embedded system a second alternative can be more efficient in area and response time. Figure 4.8 shows the two architecture alternatives, using the example from figure 4.4.

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 VHDL finite state machine frame in Figure 4.7 therefore is complemented by conditional branches depending on the triggering signal. The activity threads are serialized inside one VHDL process. This has the effect that no problems due to mutual access on the data shared by the activity threads, i.e. state and local variables of the SDL processes, can occur.

The **parallel activity thread** architecture implements each activity thread in its own parallel VHDL process with its own input message queue. It is obviously possible that transitions belonging to one SDL process are part of different activity threads. In this case, different concurrent activity threads must be able to access this processes' state and local variables. At the same time, it must be provided that only one transition belonging to one

```
main: PROCESS
   BEGIN
   IF reset = '1' THEN
      -- initialize data --
   ELSIF init = '1' THEN
      -- execute start transitions --
   ELSE
      IF (signal in queue) THEN
         -- get signal from queue --
         CASE state_process_1 IS
            WHEN X =>
               -- execute task 1 --
               -- execute state change --
               -- signal output to process_2 is replaced:

               CASE state_process_2 IS
                  WHEN A =>
                     -- execute transition 2 --
                  WHEN B =>
                     -- execute transition 3 --
               END CASE;
            WHEN Y =>
               -- execute transition 4 --
         END CASE;
      END IF;
   END IF;
   WAIT UNTIL clock = '1' AND clock'event;
END PROCESS main;
```

Figure 4.7: VHDL frame for one activity thread

SDL process is executed at the same time. Otherwise, the consistency of the shared data is not guaranteed and faulty execution is possible. Therefore, an additional run-time component is required, which implements the storage of shared process state and local variables, protected by a lock mechanism. Like shown in chapter 5, short blocking times are crucial for real-time execution and analysis. The implementation of signal outputs after the rest of the transition, which is required for semantical correctness, has the additional advantage of minimal blocking times of the shared process data.

A shared data component stores the local state and variable data of one

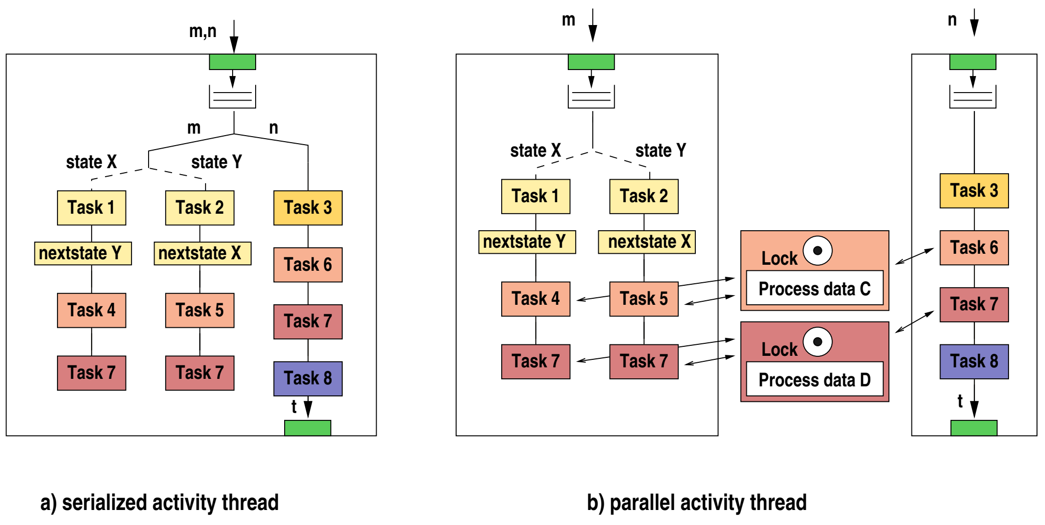


Figure 4.8: Activity thread model alternatives

SDL process. It has to provide read and write access to several activity threads as well as a mutual exclusion mechanism. For the case of several activity threads attempting to execute a transition of the process, different strategies are possible. A first-come-first-serve policy resembles most closely the original execution order of the SDL specification. A second option is a locking order after priorities, which are firmly assigned to the different activity threads. In the case of the shared data component, the implementation after FCFS creates higher cost than the priority scheme, because the locking request (in contrast to a SDL signal) needs not to be further transmitted. A FIFO storing the requests therefore creates a hardware overhead. In both cases, the preemption of transitions is not possible. For the experiments on the rapid prototyping system, a priority based component was used.

4.5 Combination of Implementation Models

Each SDL signal 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:

- Each SDL signal 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 signal to the environment, to a different activity thread, or to a server process.

The already mentioned scenario with several consecutive output statements in one transition can be implemented concurrently by a SDL signal connection to a second VHDL process implementing the part of the activity thread belonging to this signal.

A second interface between different implementation models can be given by the shared data component. It ensures the SDL processes' integrity, while its transitions can arbitrarily be implemented in different VHDL processes.

Chapter 5

Real-Time Analysis

Once the implementation model for the SDL system has been determined and the run-time components have been chosen from the library like described in chapter 4, the temporal behaviour of the implementation is fixed. The worst-case and best-case execution times of the EFSM and the run-time components can directly be read from their respective VHDL descriptions. As part of the specification outlined in chapter 3, the temporal properties of the environment have been defined with the help of event streams and event dependency matrices. With all this information assembled, a real-time analysis is possible, which gives guarantees on the worst-case reaction time to external events. As a by-product, the necessary length of message queues can be determined.

After the definition of the analysis model and the terms used throughout this chapter in section 5.1, section 5.2 presents the real-time analysis for the server implementation model. In a first step, the reaction time to a triggering signal is determined from the execution time and the worst case waiting time of the signal in the message queue. Since the latter is influenced by the occurence of other signals, taking event dependencies into account can greatly reduce too pessimistic assumptions. The inclusion of event dependencies in the real-time analysis is described in section 5.2.2. The event streams of triggering events from the environment or timers are assumed to be given as part of the specification. In a network of SDL processes, a SDL process can be triggered by a signal originating from a different SDL process. The event stream of such an "internal" SDL signal can be derived from the triggering event streams and the worst and best case execution times of the sending process, like presented in section 5.2.3.

As outlined in section 4.4, the structure of a serialized activity thread implementation is identical to a server process. In section 5.3, therefore, the real-time analysis methods of the server model are applied to serialized activity threads. In the parallel activity thread model, the server model's waiting times in the message queue correspond to blocking times at the shared process data components. Section 5.4 presents the real-time analysis for this implementation model, taking into consideration the execution scheme inside the separate activity threads. Section 5.5 deals with the analysis of an entire SDL system consisting of several processes, implemented after the server model or a combination of the different implementation models. Section 5.6 finally investigates the required depth of the message queue and the effects of a queue dimensioned too small on the real-time behaviour.

5.1 Real-Time Analysis Model

5.1.1 Definitions

As mentioned before, this work concentrates on event-driven reactive hard real-time systems, where the worst-case, i.e. maximum response time to a triggering event has to be determined.

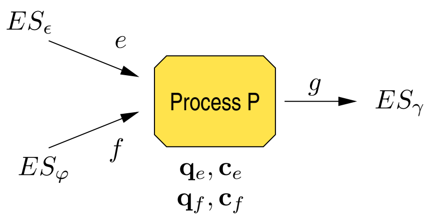


Figure 5.1: Top-level view of an implementation process P

An implementation entity P to be investigated, which can either be a SDL server process or an activity thread, has a set I_P of SDL input signals. Each SDL signal $i \in I_P$ can originate from one of several signal sources $\iota \in S_i$. S_i is the set of all signal sources which can output signal i. Each signal source ι is characterized by an event stream $ES_\iota : \left\{\binom{z_j}{a_j}_{j=1\ldots n}\right\}$. For each signal i, the queueing time $\mathbf{q}_i$ is required to receive and enqueue it, and the computation $\mathbf{c}_i$ is triggered by it in the receiving implementation process. Hereby, $\mathbf{c}_i$ denotes the worst case execution time and $\mathbf{c}_{min,i}$ the best case execution time necessary to process the event in the EFSM.

For event tuples describing periodic events, i.e. $z_j < \infty$, the worst case steady utilization of the implementation process is given by:

$$U = \sum_{i \in I_P} \sum_{j=1}^{n} \frac{\mathbf{c}_i}{z_j} \tag{5.1}$$

A correlation between two signal sources can be expressed with an event dependency matrix EDM, whose elements $ed_{\kappa\iota}$ denote the minimum temporal distance between an occurrence of a signal originating from κ and a subsequent occurrence of a signal originating from ι.

The worst case waiting time $\mathbf{w}$ of a signal in the message queue and the overall worst case reaction time $\mathbf{r}$ have to be determined by the real-time analysis.

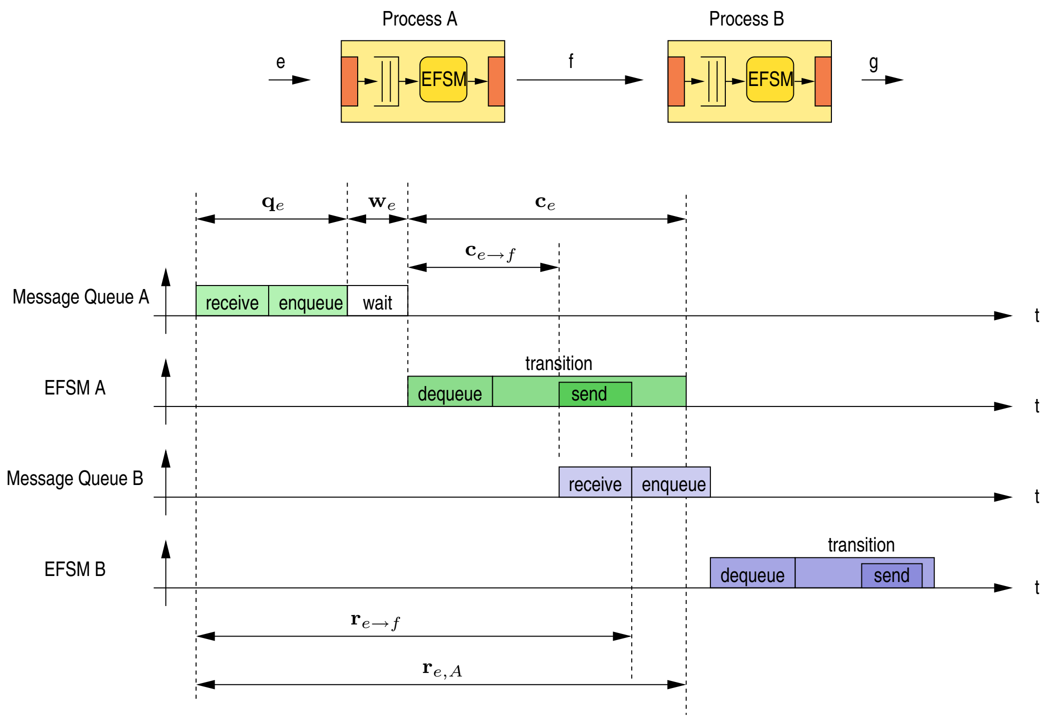


Figure 5.2: Timing relationship in a precedence system

The temporal relationship between these values is detailed in figure 5.2. A signal e is received using the channel's protocol and put into the message queue. The EFSM, running independently of the queue, removes the signal from the queue and executes the appropriate transition, during which one or several new SDL signals can be output. The queueing time $\mathbf{q}$ encompasses $\mathbf{q}_{\text{receive}}$ and $\mathbf{q}_{\text{enqueue}}$, where depending on the message queue's implementation

$\mathbf{q}_{\text{enqueue}}$ can already be contained in $\mathbf{q}_{\text{receive}}$. Computation $\mathbf{c}_e$ caused by signal e in the EFSM consists of the time $\mathbf{d}$ to remove the signal from the message queue plus the time to execute the transition, which is detailed below. Additionally, the time $\mathbf{c}_{e \to f}$ is defined, which denotes the time until a signal f, which is triggered by e, is output during the transition.

Accordingly, $\mathbf{r}_{e,A}$ denotes the time that elapses until the reaction of implementation process A to e is finished. $\mathbf{r}_{e,A}$ consists of queuing time $\mathbf{q}_e$, execution time $\mathbf{c}_e$ and additional waiting time $\mathbf{w}_e$ in the message queue. The reaction time $\mathbf{r}_{e \to f}$ in contrast denotes the time until the signal f has been output in reaction to e. Like detailed in figure 5.2, $\mathbf{r}_{e \to f}$ comprises $\mathbf{w}_e$, $\mathbf{q}_e$, the execution time $\mathbf{c}_{e \to f}$ plus the time $\mathbf{s}_f$ necessary to output f on the signal channel.

Figure 5.2 reveals that due to the hardware's parallelism, the activities in the EFSMs and message queues of a precedence system overlap in time. A task precedence graph is an alternative view of the SDL system, which clearly displays the precedence relations in the SDL system and helps determining which signals, processes and transitions are involved in the reaction to an external signal. An example of a task precedence graph was already shown in chapter 4 in figure 4.6.

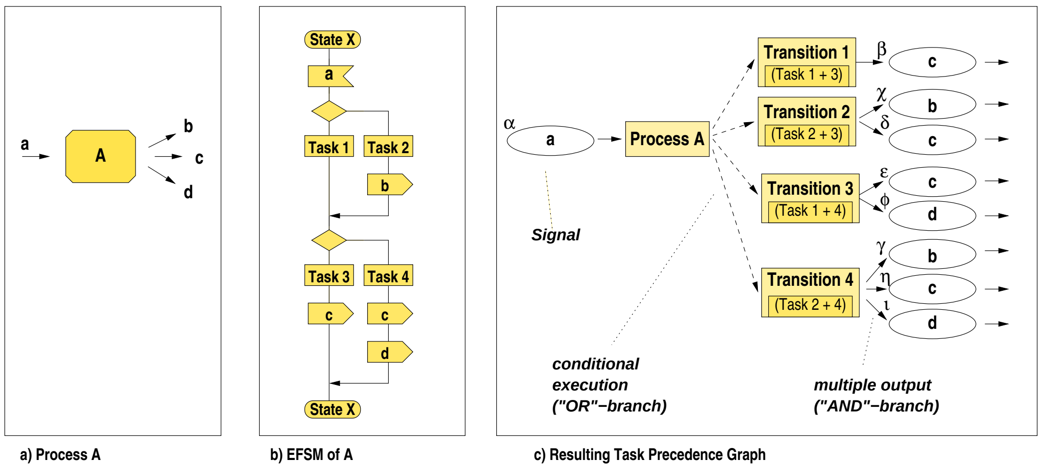


Figure 5.3: Derivation of the task precedence graph

As can be seen in figure 5.3b, a transition triggered by a signal in the EFSM can contain different computations and signal outputs due to conditional execution. The transitions contained in the task precedence graph (5.3c) therefore do not correspond directly to the original transitions of the EFSM. Instead, they enumerate the different behaviour alternatives which can be triggered by an input signal. In the following, t stands for a transition in the task precedence graph. T_e denotes the set of transitions t which can be triggered by signal e. $\mathbf{t}_t$ represents the time required to execute the transition t, not including the time to output any signals. O_t is the set of signals which are output in transition t.

A signal e can appear several times in the task precedence graph. This is the case when, like e.g. in figure 5.3, the signal output is inside conditional branches, occurs in several transitions, or when the signal is output of different SDL processes. In the following, it is however required that the destination process P of each signal e has to be unique. If this is not the case in the SDL specification, differentiating signal names have to be introduced. Each occurence of the signal in the task precedence graph is a unique source ϵ of this signal e. S_e denotes the set of all sources of signal e in the SDL system. The derivation of the event streams and event dependencies of the internal signal sources is adressed in section 5.2.3 and 5.2.4.

The part of the task precedence graph triggered by one external signal is termed task precedence system (TPS). A branch R_e describes one specific path through the task precedence system triggered by the external signal e. It is a sequence of transitions t, beginning with the first transition triggered by e. It constitutes one possible reaction to the external signal e. B_e is the set of all branches in the task precedence system triggered by signal e, and therefore fully defines this task precedence system. Z denotes the set of all task precedence systems of the entire SDL system, characterized by their triggering signals e. Two further sets of signals are needed for the activity thread implementation model: E_e is the set of all signals of the TPS e which are sent to the environment. $A_{P,e}$ is the set of all signals of TPS e which are received by process P.

Figure 5.4 summarizes the variables relevant for real-time analysis.

5.1.2 Level of Abstraction

The view on the system to be analyzed outlined in the previous section shows a high level of abstraction. The SDL system is simplified to external signals triggering a computation, described only by an upper and lower bound on its duration, during which new signals can be sent. As will be seen in the following, at this level of detail it is possible to perform a real-time analysis which is guaranteed to cover the worst case. The analysis algorithms require a limited effort and can be automated. This method however has two major drawbacks. Firstly, the general use of the maximum computation time without regard of the function described in the EFSM can lead to overly pessimistic worst case assumptions. An example is a frequent external signal that only every n occurences triggers the output of an internal signal with

I_P	set of input signals of process P
S_e	set of signal sources of signal e
T_e	set of transitions triggered by signal e
O_t	set of output signals of transition t
Z	set of task precedence systems (TPS) forming the SDL system
R	sequence of transitions characterizing one path of a TPS
B_e	set of branches R constituting the TPS triggered by signal e
E_e	set of all signals sent to the environment in TPS e
$A_{P,e}$	set of all signals received by process P in TPS e
ES_ϵ	event stream of signal source ϵ
$ed_{\epsilon\varphi}$	minimum distance between signals from sources ϵ and φ
$\mathbf{w}$	waiting time in message queue
$\mathbf{b}$	blocking time of shared data
$\mathbf{r}_{e,P}$	reaction time of process P to signal e
$\mathbf{r}_{e\to f}$	time required for output of f in reaction to e
$\mathbf{c}$,$\mathbf{c}_{max}$	worst case execution time
$\mathbf{c}_{min}$	best case execution time
$\mathbf{c}_e$	execution time required to process signal e
$\mathbf{c}_{e\to f}$	execution time for the output of f
$\mathbf{t}_t$	time required to execute transition t (without signal sending)
$\mathbf{s}_e$	time required to send signal e
$\mathbf{q}$	queueing time
$\mathbf{d}$	dequeueing time
$\mathbf{l}$	time required to lock shared data
$\mathbf{u}$	time required to unlock shared data

Figure 5.4: Variables used during real-time analysis

a very long computation. Secondly, the behaviour of a SDL process using priority input, save and enabling conditions statements can not be described using this abstract model. This is due to the fact that in this case signals are not always consumed instantaneously, but may be kept in the message queue. It then depends on the functionality of the EFSM and not the given event stream of the external signal, when the respective computation is triggered.

The two adressed problems would require including the functionality described in the SDL system in the real-time analysis in order to obtain more detailed information which transition is executed in which situation. At this level of detail, however, it is no longer possible to find a general solution yielding the worst case reaction time. By executing the SDL specification, different scenarios would have to be created, with an enormous effort and no guarantee that the worst case has been met.

Taking into account this tradeoff, the following procedure is proposed. The use of SDL is restricted to a subset excluding the priority input, save and enabling conditions statements, and real-time analysis is performed at the described high level of abstraction. If the result is too pessimistic, such that no feasible implementation can be found, in a second step further information can be added to the real-time model by the designer. With additional event streams and event dependencies, knowledge about the functionality can be expressed. In the example mentioned above, an event stream annotated to the internal signal could express that it occurs only infrequently. A second helpful notion is the concept of modes, which can express the mutual exclusion of entire groups of signals or computations depending on the operation mode of the system. These ideas are seized in the treatment of SDL systems in section 5.5.

5.2 Server Implementation Model

5.2.1 Reaction Time

A SDL process' reaction time to a signal, like outlined in section 5.1, consists of the queueing time $\mathbf{q}$, the computation time $\mathbf{c}$ and a variable waiting time $\mathbf{w}$. In a server model implementation, the computation time $\mathbf{c}_e$ consists of the dequeuing time, the time to execute the transition triggered by e plus the time required to output the signals:

$$\mathbf{c}_{e,max} = max\,(\,\mathbf{d}_e + \mathbf{t}_t + \sum_{i \in O_t} \mathbf{s}_i \qquad \forall t \in T_e) \tag{5.2}$$

Waiting time $\mathbf{w}$ arises when the processing of the signal is delayed because of other signals. Such a delay can occur at both entities which comprise the SDL process, the message queue and the EFSM. Figure 5.5 shows such a situation, where three signals are sent simultaneously to a SDL process.

Depending on the channel protocol and the implementation of the queue, a first delay will arise if the message queue can not be written simultaneously. The EFSM processes the signals sequentially in first-come-first-serve order, resulting in a possible second delay. It becomes clear from figure 5.5 that those two delays overlap due to the parallel execution of message queue and EFSM. For the real-time analysis it is therefore sufficient to calculate the waiting time only from the execution order in the EFSM. An upper bound of

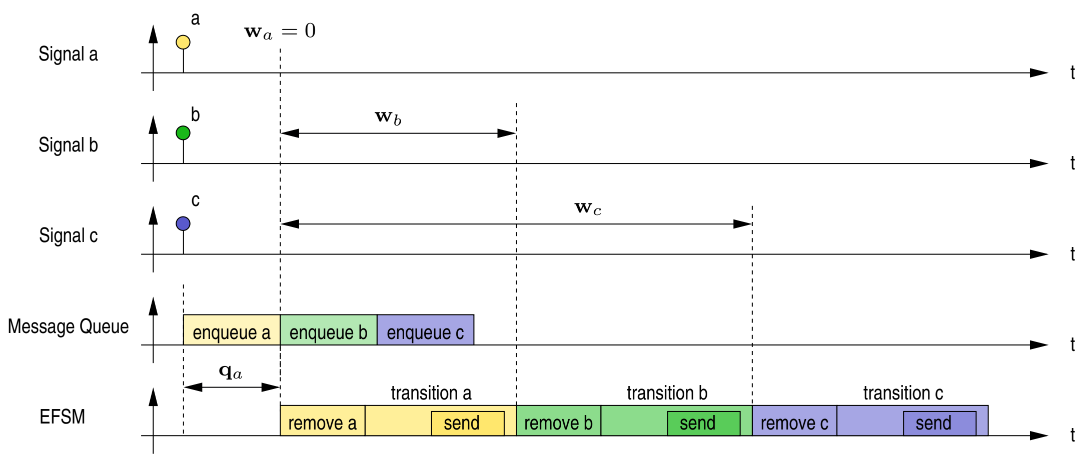


Figure 5.5: Detailed view of the waiting time

the response time of one SDL process to a signal e is given by equation (5.3), utilizing the maximum possible queueing times if the queueing times of the signals differ.

$$\mathbf{r}_e = max(\mathbf{q}_{i \in I_P}) + \mathbf{w}_e + \mathbf{c}_e \tag{5.3}$$

The problem to be solved now is identical to the determination of the reaction time of software tasks on a single processor with a first-come-first-serve-scheduler and no preemption, where the computation triggered by one signal corresponds to one task. In a first step, no event dependencies are considered.

Figure 5.6 serves to illustrate one basic effect of a pure first-come-first-serve strategy when the waiting time of one event e, in the example signal b, has to be found: Events arriving after e have no effect on the waiting time, since they are strictly proceeded later. Instead, the history of events arriving before e determines the waiting time of e. To qualify the worst case waiting time, a bound has to be given on the number of signals with their respective required amount of computing which can already wait in the message queue at the time of arrival of signal e.

t_0 denotes the time of arrival of signal e. Now, for each time interval I before t_0, the maximum number of all signals $i \in I_P$ that can arrive from signal source $\iota \in S_i$ within I is specified as $E_\iota(I)$, each arrival requesting c_i computation time. Therefore, the entire amount of computation time requested within interval I can be written as

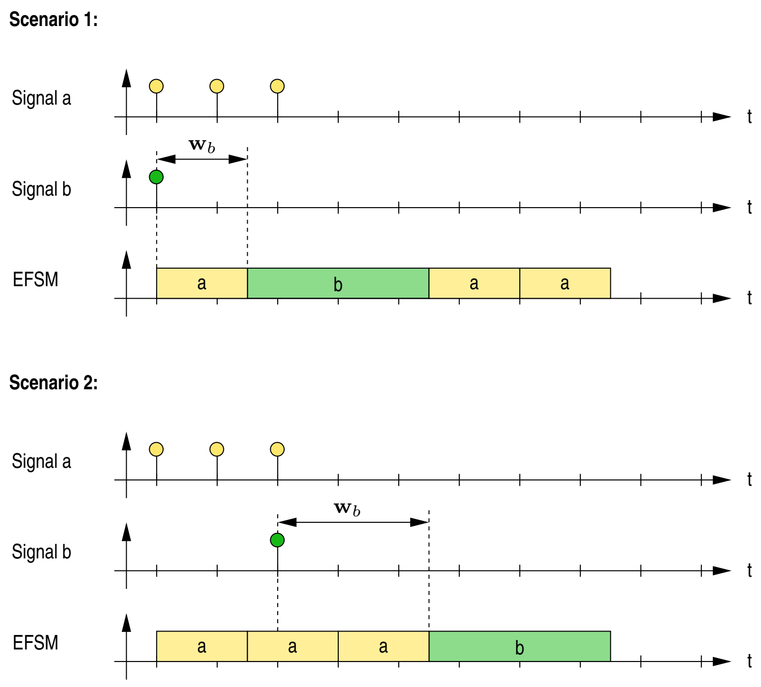


Figure 5.6: First-come-first-serve computation with different event sequences

$$C(I) = \sum_{i \in I_P} \sum_{\iota \in S_i} E_\iota(I) \cdot \mathbf{c}_i \quad \text{for independent signals}$$

Since $C(I)$ is calculated for all signals $i \in I_P$, it also includes all previous occurences and the current arrival of the observed signal e. Under the assumption that I is a busy period, i.e. that the EFSM is never idle respectively the message queue never empty within I, the amount of computation already performed within I is I. Taking also into account, that the computation for the current signal e is by definition not part of the waiting time, the waiting time for e considering interval I is:

$$\mathbf{w}_e(I) = C(I) - I - \mathbf{c}_e$$

An interval I is a busy period if its requested computation is larger than or equal to the interval, i.e. $C(I) - I \geq 0$. The term $C(I) - I$ is equivalent to the unfinished work $U(t)$ which is used in queueing theory (e.g. [Kle75]). The worst case waiting time now is given as the maximum $\mathbf{w}_e$ of all I which meet the busy period criterion:

$$\mathbf{w}_{e,max} = max\Big(C(I) - I - \mathbf{c}_e\Big) \tag{5.4}$$

Starting with $I = 0$, only intervals until the first break of the busy period, i.e. the first time $C(I) - I$ drops below zero, have to be investigated. Taking into account the required property of a valid event stream formulated in equation 3.6, it can be shown that the first maximum of $\mathbf{w}_e$ can never be exceeded later.

It should be noted that equation 5.4 makes no assumption on the temporal distribution of the signals, the required computation and the signal's deadlines, i.e. it is valid for all event streams. Obviously, if the steady maximum utilization is larger than 100%, which means that the requested computation $C(I)$ exceeds I for all possible I, the worst case waiting time increases without bound.

Figure 5.7 shows the derivation of $\mathbf{w}_{e,max}$ for the example already used in figure 5.6. Note that the worst case reached for signal B corresponds to scenario 2 in figure 5.6.

An important special case, which simplifies the derivation of $\mathbf{w}_{max}$, is given when the entire computation requested by all signals which can occur simultaneously is smaller than the given minimum distance of any two signals i from source ι, i.e. $C(0) \leq a_{2,min}$. For this situation it can be shown that the worst case reaction time is already found at $I = 0$, and is given as:

$$\mathbf{w}_{e,max} = C(0) - \mathbf{c}_e \quad \text{for} \quad C(0) \leq a_{2,\iota} \quad \forall\, \iota \in S_i, i \in I_P \tag{5.5}$$

The guarantee that the worst case response time is lower than the minimum distance of any two signals in turn is equivalent to the statement that the message queue will never contain the same two signals.

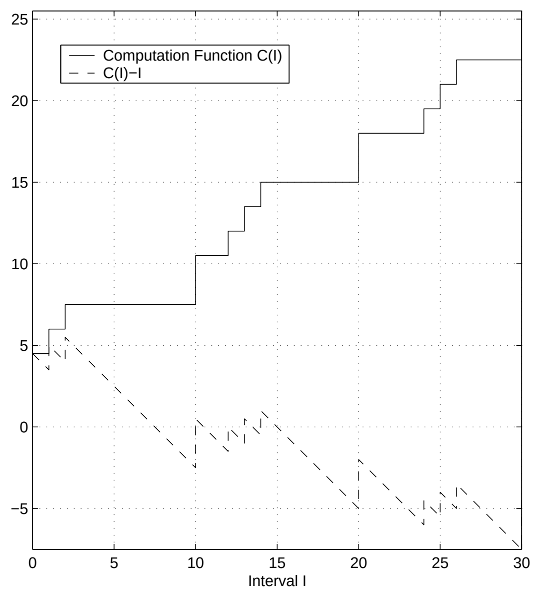
In purely periodic systems, the mean steady utilization must be below 100%:

$$U = \sum_{i \in I_P} \sum_{j=1}^{n} \frac{\mathbf{c}_i}{z_j} \leq 1 \tag{5.6}$$

If this is the case, it can be shown that the worst case waiting time is also already reached at $I = 0$. The necessary and sufficient condition ensuring that a deadline equal to the period can always be met is again $C(0) \leq a_{2,min}$.

5.2.2 Event Dependencies

Taking event dependencies into account, in the present case the specified pairwise minimum event distances $ed_{\alpha\beta}$ between two event sources α and β,



$ES_\alpha : \left\{ \binom{12}{0}, \binom{12}{1}, \binom{12}{2} \right\}$; $\mathbf{c}_a = 1.5$
$ES_\beta : \left\{ \binom{10}{0} \right\}$; $\mathbf{c}_b = 3$;

$\mathbf{w}_{max,a} = \mathbf{w}_a(I = 2) = 7,5 - 2 - 1.5 = 4$
$\mathbf{w}_{max,b} = \mathbf{w}_b(I = 2) = 7,5 - 2 - 3 = 2,5$ (see Scenario 2)

Figure 5.7: Determination of **w** for the example from figure 5.6

can greatly relax real-time analysis. It has no longer to be assumed that all events can occur at the same time. Since the FCFS processing order is not affected, like before, the maximum computation which can wait in the message queue at the time of arrival of an observed event e has to be determined. In contrast to the previous section, however, it can not be directly derived from the event functions. The sum of $E_\iota(I)$ assumes the worst case of all events occuring as soon as the event distances a_j allow. Depending on the given event dependencies, this might no longer be possible. It is then no longer evident in the general case, which event sequence leads to the worst case.

The task to determine the worst case waiting time of a signal e now can be formulated as follows: For each interval I preceding a point in time t_0, find the maximum possible required computation $C(I)$ which satisfies the specified event functions *and* event dependencies, given the occurence of signal e at t_0.

One possible solution to the described problem is the enumeration of all possible events sequences like described in [Gre93a], using a branch-and-bound search algorithm. Each node of the search tree built by the algorithm corresponds to an event sequence which precedes the occurrence of signal e at the time interval $I = 0$ as root node, as illustrated in figure 5.8.

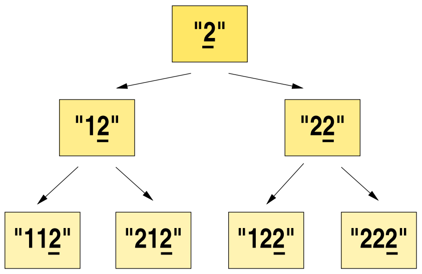
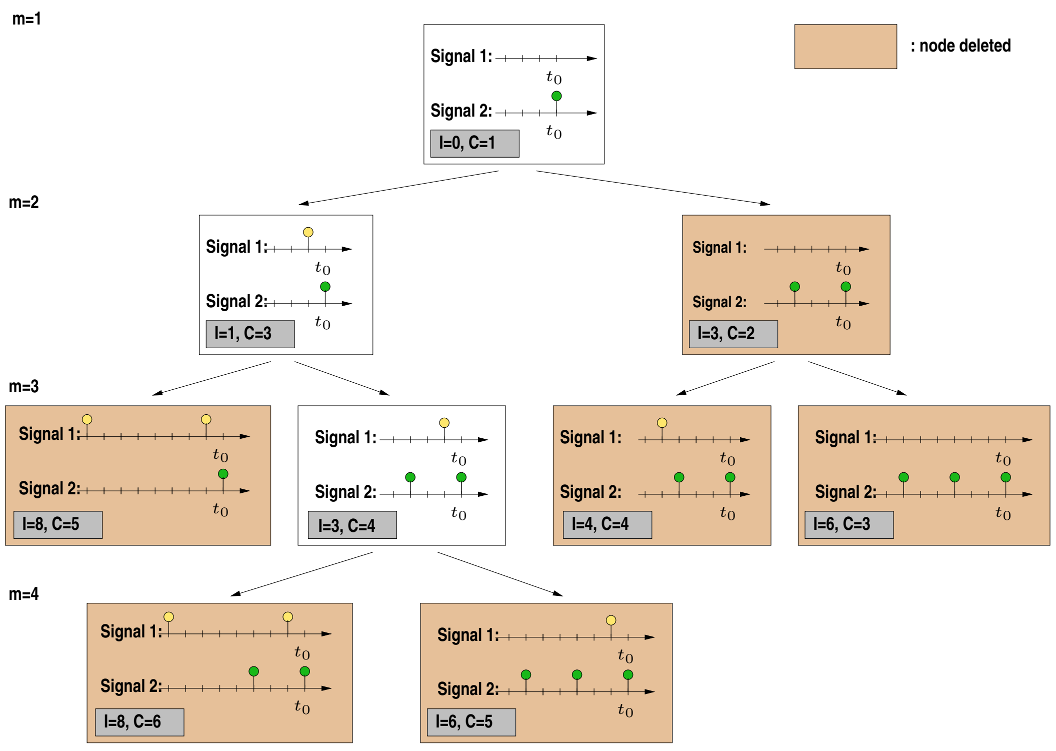


Figure 5.8: Possible event sequences preceding an occurence of signal 2 at t_0

Each node is characterized by the minimum time interval I in which this particular sequence can occur, and the overall computation C requested by this sequence in the interval. Starting from one node, a branch to a new node is introduced for each different signal i from source ι that can occur before. The time distance to this new signal, which determines the I of the new node, is given by the event streams and event dependencies. A node which will clearly not lead to the worst case, since there exists at least one other node with both a larger C and a smaller I, needs not to be explored further and can be deleted. The search tree has to be built breadth first to avoid the exploration of branches which could be deleted early.

Each node contributes a candidate for the worst case requested computation $C(I)$ for its interval I. $C(I)$ is the sum of the computation requested by the node's event sequence plus the maximum computation which can be caused by additional independent signals in this interval. From figure 5.9 it becomes clear that the time intervals I do not increase uniformly from node to node. It is therefore useful to record each new node in a list sorted after the time intervals I. All possible candidates for one particular I_x have been found if the nodes of all branches still active have reached intervals larger than I_x.

Analogously to section 5.2, it can be shown that only intervals I have to be investigated until the first I_0 where all nodes show $C(I) - I < 0$. Once the maximum requested computation $C(I)$ for each interval I has been found, the worst case waiting time can be determined using equation 5.4. It is clear that the search tree grows exponentially with n^m, where n is the number of dependent events and m the number of signals in the event sequence. This



$$ES_1 : \left\{ \binom{7}{0} \right\};\; c_1 = 2;\;\; ES_2 : \left\{ \binom{3}{0} \right\};\; c_2 = 1;\;\; ed_{12} = ed_{21} = 1;\;\; (n = 2)$$

Figure 5.9: Search tree example

becomes particularly dramatic if groups of $n > 2$ dependent events are given. In practice, however, only a very limited number of sequential events has to be considered: In the special case where each signal has to be processed before the next of its kind arrives, only nodes in the depth $m \leq n$ can contribute to the current signals waiting time. In general, all nodes with $m > n$ and $C(I) - I > 0$ signify that signals accumulate in the message queue. If the steady utilization is not larger than 100%, obviously this can occur only for limited time intervals I.

Figure 5.9 depicts an exemplary search tree for an occurence of signal 2 at time t_0. It can be seen that at each level only one node needs to be followed further, since all other nodes can be deleted after the condition $C(I) - I < 0$. The worst case waiting time for signal 2 in this example results in $\mathbf{w}_{2,max} = C(1) - 1 - c_2 = 1$.

5.2.3 Output Event Stream

The analysis of entire SDL systems also requires the event streams of internal SDL signals, which do not originate from the environment, but are output of SDL processes. Figure 5.10 depicts such a scenario where several signals are input to a SDL process P. The transition triggered by signal a outputs the internal signal d.

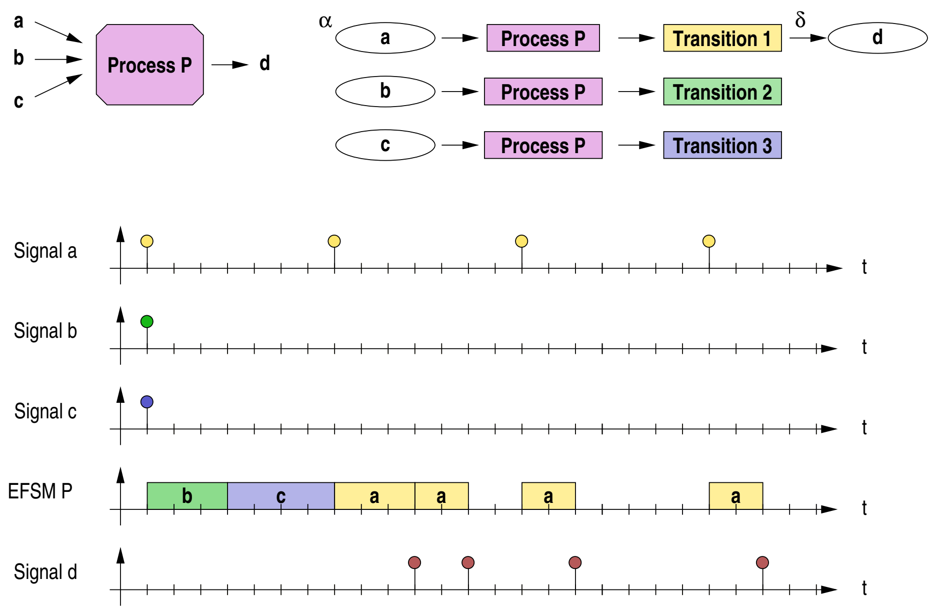


Figure 5.10: Example for internal signal d

Figure 5.10 illustrates the effect of a possible delay of the processing of a from signal source α due to other signals. The timing diagram shows that d can occur in shorter time distances than specified in the event stream ES_α, albeit obviously never shorter than the minimum execution time $\mathbf{c}_{a,min}$.

The problem to be solved in order to determine a new valid event stream ES_δ describing the new source δ of signal d is to find the shortest possible time intervals a_1, a_2, a_3, etc., in which one, two, three, etc., signals d can occur. It becomes evident from figure 5.10 that the minimum event distance between *two* signals d results when one occurence of a, which is maximally delayed, is followed by a signal a which is immediately processed. When the minimum interval for two signals is known, the time interval for *three* signals obviously is minimum when a third signal also occurs as early as possible.

The worst case scenario is therefore characterized as follows: One signal a has to wait the maximum time in the queue and requires the maximum execution time $\mathbf{c}_{a,max}$. All following events a are processed as early as possible in the shortest execution time $\mathbf{c}_{a,min}$. Regarding the other signals, this means that as many as possible occur before the observed signal a, and none after[1]. All parameters of this worst case are known already. $\mathbf{c}_{a,min}$ and $\mathbf{c}_{a,max}$ are given, the maximum waiting time $\mathbf{w}_{a,max}$ of signal a can be determined using the algorithms from section 5.2.1 and 5.2.2, and the minimum distances of following signals a is given by ES_α.

The basic rule for deriving the event stream ES_δ from ES_α can be directly read from the description of the worst case scenario. The minimum time interval for j signals d is equal to the minimum interval specified for j signals a, diminished by the worst case delay for a, consisting of the maximum waiting time and the difference between $\mathbf{c}_{a,max}$ and $\mathbf{c}_{a,min}$. Expressed in other words, the event stream ES_δ results from event stream ES_α subjected to a jitter J with

$$J = \mathbf{w}_{a,max} + \mathbf{c}_{a,max} - \mathbf{c}_{a,min} \tag{5.7}$$

Each interval of ES_α has to be shortened by J, while at the same time observing the two given conditions:

$$a_j \geq 0 \qquad \text{(negative time intervals are not possible)} \tag{5.8}$$

$$a_j - a_{j-1} \geq \mathbf{c}_{a,min} \qquad \text{(minimum event distance is } \mathbf{c}_{a,min}\text{)} \tag{5.9}$$

This is identical to a left shift of event function $E_\alpha(I)$ by J, which when necessary has to be modified such that the two conditions are met.

Figure 5.11 gives an algorithm which generates the worst case ES_δ from ES_α for any given J, generating an event stream that complies with the conditions given in equation 5.8 and 5.9. Since condition 5.9 is independent of J, the algorithm has to be applied also in cases where no waiting time due to other signals can occur. The example from figure 5.10 is seized again in figure 5.12, where both the graphical modification of the event function and the application of the algorithm are demonstrated.

The special case $C(0) \leq a_{2,\iota}\ \forall \iota \in S_i, i \in I_P$, which has been already mentioned, greatly simplifies the derivation of ES_δ. It guarantees that $J \leq a_2 - \mathbf{c}_{a,min}$. In this case, for each event tuple of ES_α the rules given in table 5.1 can be directly applied.

Remark on the utilization of $\mathbf{w}_{a,max}$: It is possible that the worst case $\mathbf{w}_{a,max}$ results from previous occurences of the observed signal a. If a displays a bursty behaviour, which means that only a limited number of signals

[1]Since their event streams give only a minimum and no maximum event distance, this is compatible with their time specification.

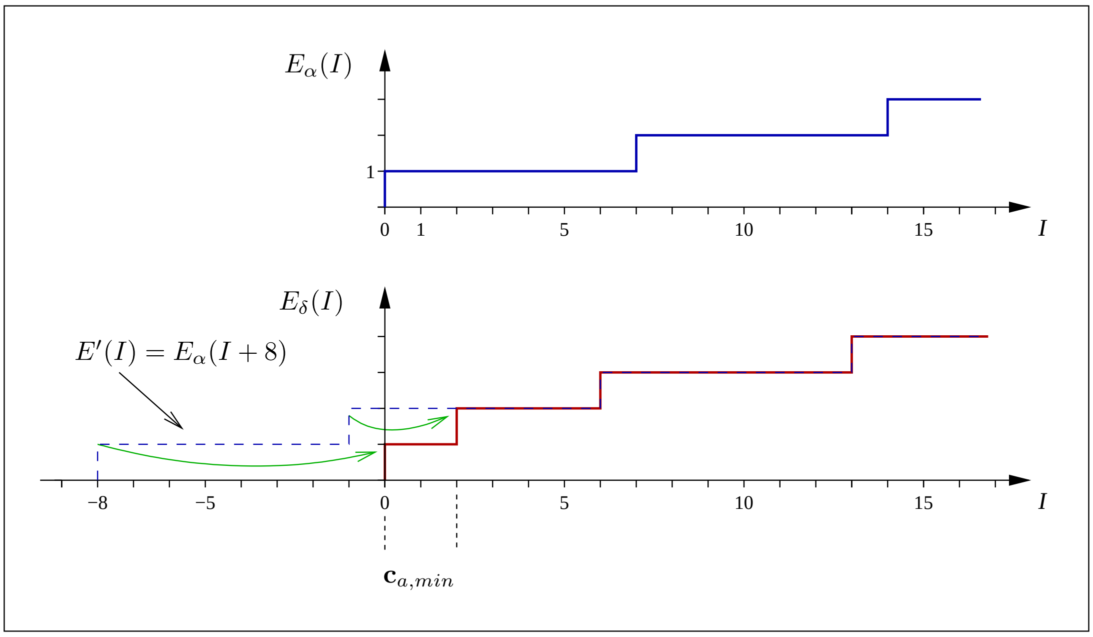
ES_α	ES_δ
$\left\{\binom{\infty}{0}\right\}$	$\left\{\binom{\infty}{0}\right\}$
$\left\{\binom{\infty}{a_j}\right\}$	$\left\{\binom{\infty}{a_j - J}\right\}$
$\left\{\binom{z_j}{0}\right\}$	$\left\{\binom{\infty}{0}, \binom{z_j}{z_j - J}\right\}$
$\left\{\binom{z_j}{a_j}\right\}$	$\left\{\binom{z_j}{a_j - J}\right\}$

Table 5.1: Derivation of ES_δ from ES_α for $J \leq a_2 - c_{a,min}$

a can occur one after the other, the proposed algorithm is slightly too pessimistic. It assumes the maximum number of consecutive signals a after t_0, which is inconsistent with the occurences of a before t_0 necessary to reach the maximum $\mathbf{w}_{a,max}$. If a very exact analysis is required, a modified ES_α which takes the previous occurences of a into account can be used in the presented algorithm. In general, however, the described slightly too negative assumption can be accepted.

```
**** Generation of event stream $ES_\delta$ ****

(1) generate intermediate event stream $ES'$
      with $z'_l := z_{j,a}$, $a'_l := a_{j,a} - J$
(2) investigate smallest $a'_l$ of $ES'$
IF ($a'_l < 0$)
      $k := 1$
      $a_{k,d} := 0$
      $z_{k,d} := \infty$
      IF ($z'_l \neq \infty$)
            $a'_l := a'_l + z'_l$
            $z'_l := z'_l$
      ELSE
            remove $z'_l, a'_l$ from $ES'$
      END IF
      LOOP until $ES'$ is empty
            (3) investigate smallest $a'_l$ of $ES'$
            IF ($a'_l - a_{k,d} < \mathbf{c}_{a,min}$)
                  $k := k + 1$
                  $a_{k,d} := a_{k-1,d} + \mathbf{c}_{a,min}$
                  $z_{k,d} := \infty$
                  IF ($z'_l \neq \infty$)
                        $a'_l := a'_l + z'_l$
                        $z'_l := z'_l$
                  ELSE
                        remove $z'_l$, $a'_l$ from $ES'$
                  END IF
            ELSE
                  $k := k + 1$
                  $a_{k,d} := a'_l$
                  $z_{k,d} := z'_l$
                  remove $z'_l$, $a'_l$ from $ES'$
            END IF
      END LOOP
ELSE
      $ES_\delta := ES'$
END IF
```

Figure 5.11: Derivation of ES_δ from ES_α for general J



$ES_\alpha : \left\{\binom{7}{0}\right\}$; $\mathbf{c}_{a,min} = 2$; $\mathbf{c}_{a,max} = 3$; $w_{max,a} = 7$; $\Rightarrow J = 8$

```
(1) intermediate event stream $ES' : \left\{\binom{7}{-8}\right\}$
(2) smallest $a'_l = -8 < 0$:
        $k := 1$; $a_{k,d} := 0$; $z_{k,d} := \infty$
        $\Rightarrow$ new $ES_\delta : \left\{\binom{\infty}{0}\right\}$
        $z'_l = 7 \neq \infty$:
        $a'_l := a'_l + z'_l = -1 \Rightarrow$ new $ES' : \left\{\binom{7}{-1}\right\}$
LOOP until $ES'$ is empty
(3) smallest $a'_l = -1$
        $a'_l - a_{k,d} = -1 - 0 < \mathbf{c}_{a,min}$:
                $k := 2$; $a_{k,d} := a_{k-1,d} + \mathbf{c}_{a,min} = 0 + 2 = 2$; $z_{k,d} := \infty$
                $\Rightarrow$ new $ES_\delta : \left\{\binom{\infty}{0}, \binom{\infty}{2}\right\}$
        $z'_l = 7 \neq \infty$:
                $a'_l := a'_l + z'_l = 6 \Rightarrow$ new $ES' : \left\{\binom{7}{6}\right\}$
(3) smallest $a'_l = 6$
        $a'_l - a_{k,d} = 6 - 2 > \mathbf{c}_{a,min}$:
                $k := 3$; $a_{k,d} := a'_l$; $z_{k,d} := z'_l$
                $\Rightarrow$ new $ES_\delta : \left\{\binom{\infty}{0}, \binom{\infty}{2}, \binom{7}{6}\right\} \Rightarrow$ new $ES' : \{\}$
END LOOP
```

$\Rightarrow ES_\delta : \left\{\binom{\infty}{0}, \binom{\infty}{2}, \binom{7}{6}\right\}$

Figure 5.12: Graphical and algorithmical derivation of $E_\delta(I)$ for the example from figure 5.10

5.2.4 Derived Event Dependencies

Event dependencies are given as part of the specification for external SDL signals. For specific scenarios they can additionally be derived by an analysis of the implementation.

Event dependencies between signals from one SDL process Between two internal SDL signals originating from the same SDL process, a minimum event distance is automatically given by the fact the EFSM can only execute one transition at a time. Figure 5.13 details this causal relation. In the depicted example, signal a triggers the output of signal c and signal b triggers d. The minimum distance of an event d to a predecessor c from a different transition is at least equal to the minimum execution time of the rest of the transition where c is sent plus the execution time of the transition triggered by b up to the sending of d:

$$ed_{\gamma\delta} = \mathbf{c}_{a,min} - \mathbf{c}_{a\to c,max} - \mathbf{s}_c + \mathbf{c}_{b\to d,min} + \mathbf{s}_d \tag{5.10}$$

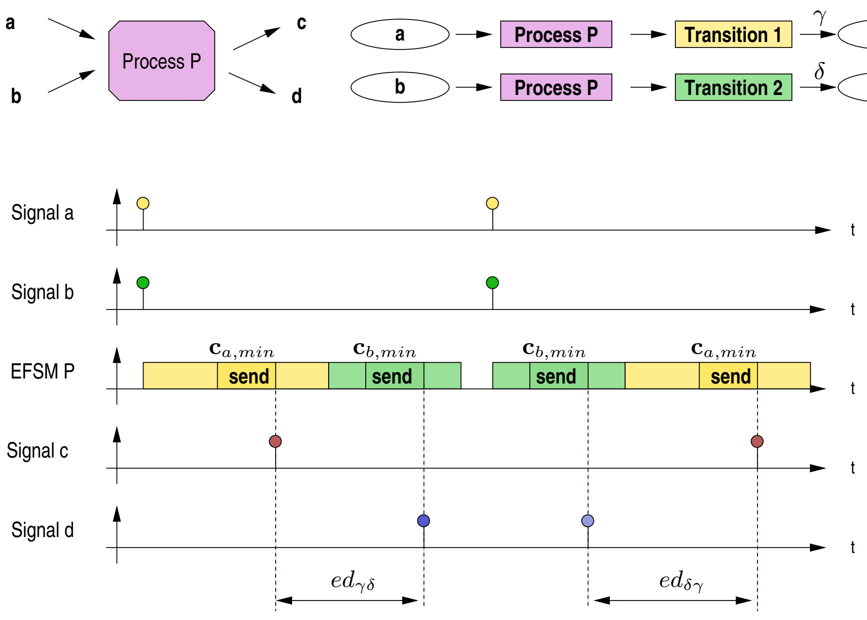


Figure 5.13: Event dependencies between signals from one SDL process (different transitions)

Figure 5.14 shows the situation of two events sent during the same transition. The minimum distance between the new signal sources β and γ, under the assumption that b is sent before c, is given as:

$$\begin{aligned} ed_{\beta\gamma} &= \mathbf{c}_{a\to c,min} + \mathbf{s}_c - \mathbf{c}_{a\to b,max} - \mathbf{s}_b \\ ed_{\gamma\beta} &= \mathbf{c}_{a,min} - \mathbf{c}_{a\to c,max} - \mathbf{s}_c + \mathbf{c}_{a\to b,min} + \mathbf{s}_b \end{aligned} \tag{5.11}$$

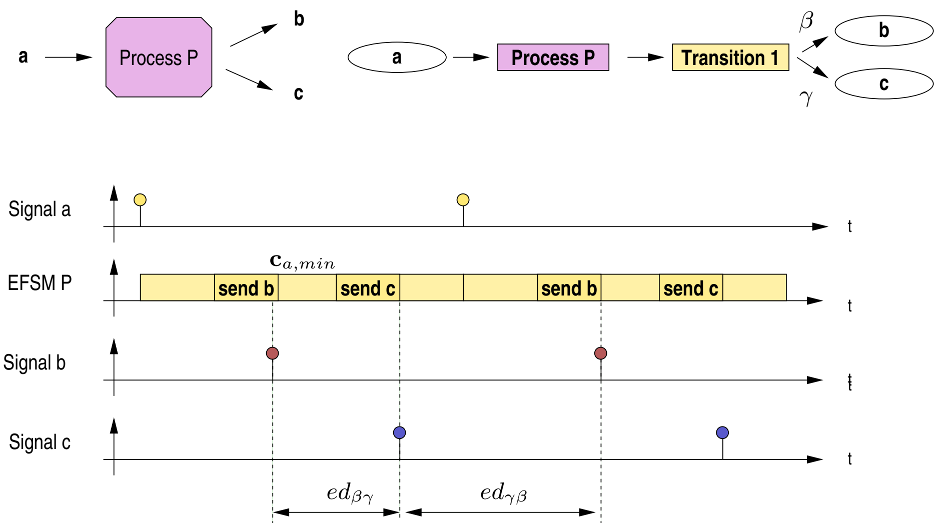


Figure 5.14: Event dependency between signals from one SDL process (one transition)

Propagation of event dependencies between SDL processes Figure 5.15 shows two SDL processes P and Q. The signal e triggers process P to output signal g after at least time $\mathbf{r}_{e,min}$, and at most $\mathbf{r}_{e,max}$. Analogously signal f triggers Q to output signal k in time $\mathbf{r}_f$, with $\mathbf{r}_{f,min} < \mathbf{r}_f < \mathbf{r}_{f,max}$.

Given event dependencies $ed_{\epsilon\varphi}$ and $ed_{\varphi\epsilon}$, under certain circumstances minimum distances between g and k can be guaranteed as well:

$$ed_{\gamma\kappa} = \begin{cases} ed_{\epsilon\varphi} + \mathbf{r}_{f,min} - \mathbf{r}_{e,max} & \text{if} \quad ed_{\epsilon\varphi} + \mathbf{r}_{f,min} \geq \mathbf{r}_{e,max} \\ 0 & \text{else} \end{cases} \tag{5.12}$$

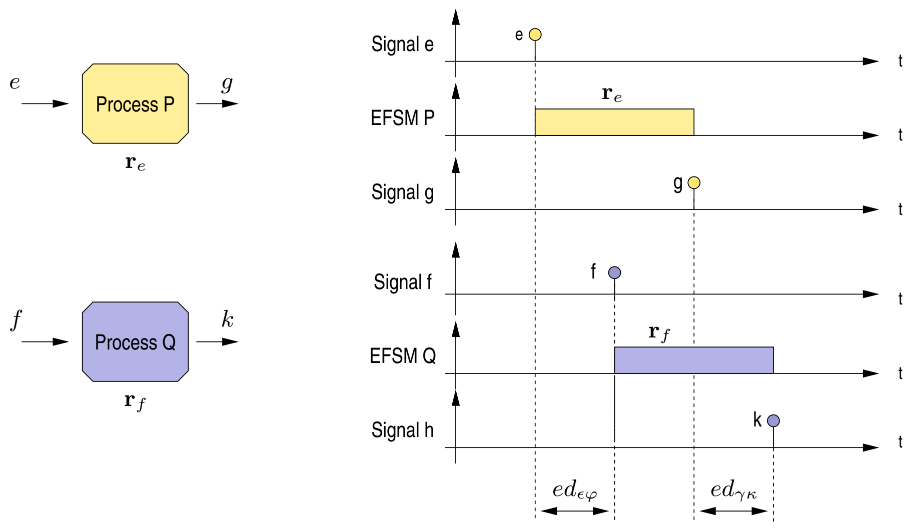


Figure 5.15: Propagation of event dependencies between SDL processes

and

$$ed_{\kappa\gamma} = \begin{cases} ed_{\varphi\epsilon} + \mathbf{r}_{e,min} - \mathbf{r}_{f,max} & \text{if} \quad ed_{\varphi\epsilon} + \mathbf{r}_{e,min} \geq \mathbf{r}_{f,max} \\ 0 & \text{else} \end{cases} \tag{5.13}$$

5.3 Serialized Activity Thread

A serialized activity thread implementation is structurally identical to the implementation of a single process after the server model. All signals are received by a message queue. An independently running finite state machine removes the signals from the queue in first-come-first-serve order, and executes the according activity thread. Therefore, the algorithms presented for the server model in section 5.2 can be directly applied to the serialized activity thread model.

The execution time $\mathbf{c}_e$ triggered by a signal e, which is a central parameter of real-time analysis, consists of the dequeueing time $\mathbf{d}_e$ and the time required to execute the entire activity thread, including any $\mathbf{s}_i$ required to output an external signal i to a channel. The execution time of the activity thread is determined by its VHDL implementation. The execution times of the tasks along the activity thread can be added. Where the activity thread branches due to multiple output statements, the execution times of all branches have to be added. The conditional execution of parts of the activity thread, in contrast, represents a true alternative. The execution times of these parts therefore contribute candidates for the maximum respectively minimum computation time of the activity thread. Therefore, the worst case execution time triggered by e is given as follows:

$$\mathbf{c}_{e,max} = \mathbf{d}_e + max\left(\sum_{t\in R}\mathbf{t}_t + \sum_{i\in E_e}\mathbf{s}_i \qquad \forall R \in B_e\right) \tag{5.14}$$

For the exemplary SDL system used in chapter 4 (see figure 4.6 and 4.8), this results in execution times $c_{at,m} = \mathbf{d}_m + max(\{\mathbf{t}_{t1} + \mathbf{t}_{t4} + \mathbf{t}_{t7}\},\ \{\mathbf{t}_{t2} + \mathbf{t}_{t5} + \mathbf{t}_{t7}\})$ and $c_{at,n} = \mathbf{d}_n + \mathbf{t}_{t3} + \mathbf{t}_{t6} + \mathbf{t}_{t7} + \mathbf{t}_{t8} + \mathbf{s}_t$.

The execution time of an activity thread generally bears no similarity to the execution times of processes involved in the processing of the same signal in a server model implementation. In the activity thread, the sending and receiving of internal signals is missing. It only contains transitions actually involved in processing the signal, which is not reflected in the $\mathbf{c}_{max}$ and $\mathbf{c}_{min}$ of the SDL processes. On the other hand, a possible parallel processing in different SDL processes is executed sequentially in the serialized activity thread model.

5.4 Parallel Activity Thread

The basic architecture of a parallel activity thread implementation process consists of one message queue and one activity thread which processes the incoming signals in first-come-first-serve order. The algorithms presented in section 5.2 are therefore valid as well for this implementation model. The analysis is simplified by the fact that here one implementation process always deals with only one signal type.

The challenge of the parallel activity thread model lies in the determination of the worst case execution time $\mathbf{c}_e$ triggered by signal e. Figure 5.16 shows possible $\mathbf{c}_m$ and $\mathbf{c}_n$ of the activity threads triggered by signals m and n of the example presented in chapter 4.

The execution time of transition t of SDL process P is denoted $\mathbf{t}_t$. Like already outlined in section 4.4, in the activity thread implementation it is necessary to change the execution order of the transition in such a way that the output of a new signal occurs after everything else in the transition. $\mathbf{t}_t$ by definition does not contain any signal outputs. Outputs of internal signals are replaced by the transitions triggered in the receiving processes. The time $\mathbf{s}_i$ to send signals i to the environment has to be considered separately. Before a transition of a SDL process which appears in several activity threads can be

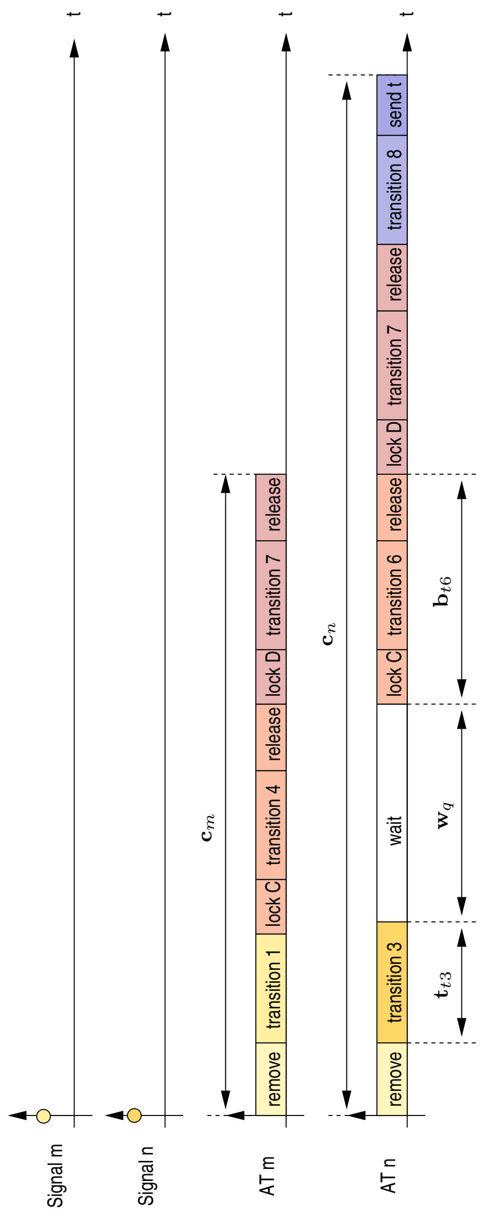


Figure 5.16: Parallel execution of activity threads from figure 4.8

executed, the shared data component has to be locked, which takes locking time $\mathbf{l}$. After the transition $\mathbf{t}_t$, the component is released again in time $\mathbf{u}$. The entire time the shared data component is blocked is denoted $\mathbf{b}_t$. When the shared data component is blocked by a different activity thread, waiting time $\mathbf{w}_i$ elapses before any transition of process P can be executed. Hereby, $i(t)$ denotes the SDL signal i which is the specified trigger for the transition t to be executed[2]. The main task of the real-time analysis for the parallel activity thread model is to determine the worst case waiting times $\mathbf{w}_{i,max}$ for all transitions involved in the activity thread.

The overall maximum execution time of activity thread at which is triggered by signal e is the maximum execution time of the transitions plus the time required to send signals to the environment, plus the overall maximum waiting time, which has to be determined using the methods from the following section.

$$\mathbf{c}_{e,max} = \mathbf{d}_e + max\left(\sum_{t \in R}(\mathbf{l} + \mathbf{t}_t + \mathbf{u}) + \sum_{i(t),t \in R} \mathbf{w}_{at,i,max} + \sum_{i \in E_e} \mathbf{s}_i \qquad \forall R \in B_e\right) \tag{5.15}$$

5.4.1 Waiting Time

The waiting time $\mathbf{w}_i$ before the computation triggered by signal i at process P can be performed is caused by other accesses to the shared data component. An activity thread can contain several transitions belonging to the same process P, e.g. resulting from a branch or loop in the activity thread. One activity thread has however only one active execution at one time, and the execution ordering in the implementation ensures that each transition is finished before the next is started. Therefore, one activity thread can access a shared data component only once at a time.

Each activity thread at from the set Z of all task precedence systems, i.e. activity threads, has a set $A_{P,at}$ of mutual exclusive accesses to the shared data component of process P at the internal signals i. Each signal i can trigger in process P one of the transitions t of the set T_i, causing the execution time $\mathbf{t}_t$. For this transition, the shared data component is blocked during the blocking time $\mathbf{b}_t = \mathbf{l} + \mathbf{t}_t + \mathbf{u}$. Therefore, signal i can block the component for the maximum time

$$\mathbf{b}_i = max(\mathbf{b}_t) \qquad \forall t \in T_i.$$

First-Come-First-Serve The fact that one access to the shared data has to be finished before the next from the same activity thread is initiated, means that only one request from an activity thread can wait for the shared

[2]Throughout this section, the term “signal i” does not stand for the literal sending of a signal, but serves to identify a specific point in the activity thread.

data access at the same time. This special case has already occured several times in section 5.2. Using the results from this section, and assuming event independence, the worst case waiting time is the sum of the maximum blocking time that can be caused by all other activity threads a at process P at the same time. The waiting time is therefore equal for all accesses to process P from within activity thread at and can be written as:

$$\mathbf{w}_{at,e,max} = \sum_{\{a \,|\, a \in Z \wedge a \neq at\}} max\,(\mathbf{b}_f) \qquad \forall f \in A_{P,a} \tag{5.16}$$

Priority-Based Access If several activity threads attempt to lock a shared data component, after this policy the thread with the highest priority is granted the lock to the component. Threads with lower priority are not preempted, however. Here, too, a shared data component can only be accessed from one activity thread once at a time. It is however possible that several subsequent accesses from high priority activity threads contribute to the waiting time of a thread with lower priority. The event streams of internal signals are therefore necessary to calculate the worst case waiting time at a shared data component.

The event streams of the internal signal sources can be derived with the algorithm described in section 5.2.3, using the execution time of the entire activity thread $\mathbf{c}_{e,min}$ and $\mathbf{c}_{e,max}$. In order to resolve the cycle that $\mathbf{c}_{e,max}$ depends on the waiting time still to be determined, it is necessary to assume that each activity thread has the same priority at all shared data components. The waiting time of each thread with lower priority then only depends on the event streams of higher priority threads.

The waiting time of the activity thread with the highest priority at shared data component of process P is then equal to the maximum blocking time of all other possible (non preemptable!) accesses to P:

$$\mathbf{w}_{at,e,max} = max\,(\mathbf{b}_f)_{\{a \,|\, a \in Z \wedge a \neq at\}} \qquad \forall f \in A_{P,a} \tag{5.17}$$

With this, $\mathbf{c}_{e,max}$ and the internal event streams of the high priority activity thread can be calculated, and with that the waiting time of the thread with the next lower priority. For each activity thread at, K_{at} denotes the set of threads with higher priority, and L_{at} the set of lower priority threads. The waiting time of activity thread at now is the longest busy period of all threads with higher priority plus the blocking time of a non preemptable lower priority thread:

$$\mathbf{w}_{at,e,max} = max\left(\sum_{k \in K_{at},\, f \in A_{P,k},\, \varphi \in S_f} E_\varphi(I) \cdot \mathbf{b}_f \right) + max\,(\mathbf{b}_l)_{l \in L_{at}} \qquad \forall I \le E_\varphi(I) \cdot \mathbf{b}_f \tag{5.18}$$

The above equation considers only one access to P from each activity thread. If at contains several signals of P, they inevitably have a minimum time distance and are therefore treated in the following section.

5.4.2 Event Dependencies

The event dependencies of internal signals in the activity thread implementation can be derived using the equations from section 5.2.4, using the execution times of the activity threads and with $\mathbf{s} = 0$ of all internal signals.

The waiting time of a signal e from source ϵ which can be caused by the occurence of a signal f from source φ in a different activity thread is reduced if the minimum event distance $ed_{\varphi\epsilon}$ between the two signals is taken into account. Figure 5.17 illustrates the influence of a minimum event distance between f and e on the maximum blocking time of i.

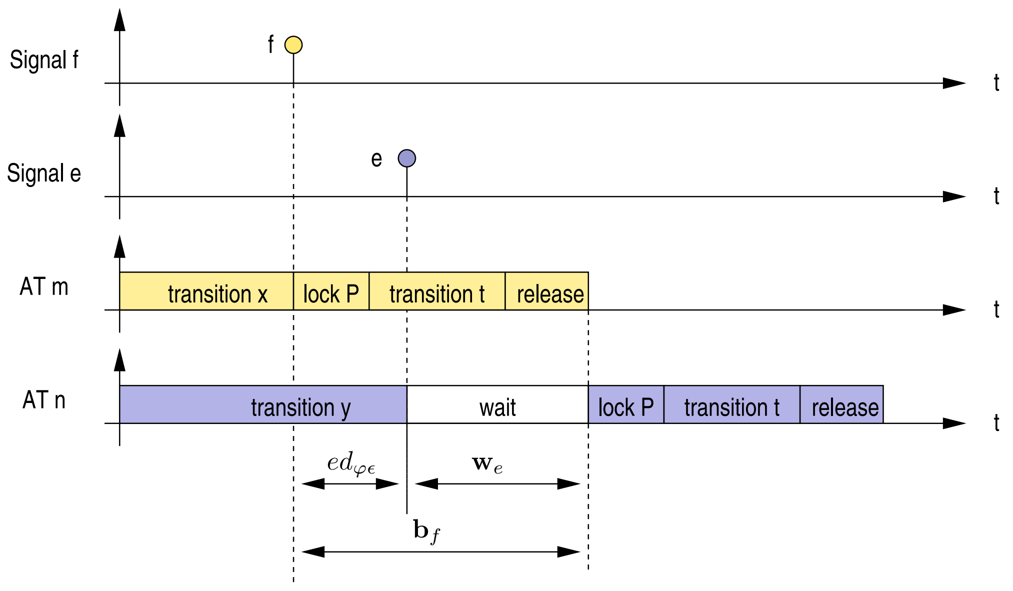


Figure 5.17: Blocking time influenced by event dependency $ed_{\varphi\epsilon}$

First-Come-First-Serve Since only one access of each activity thread has to be considered, the maximum waiting time of signal e at process P in activity thread at for the FCFS-scheme is given as follows:

$$\mathbf{w}_{at,e,max} = \sum_{\{a \,|\, a \in Z \wedge a \neq at\}} (max\,(\mathbf{b}_f) - ed_{\varphi\epsilon}) \qquad \forall f \in A_{P,a} \tag{5.19}$$

For signals without event dependency, $ed_{\varphi\epsilon} = 0$ can be used.

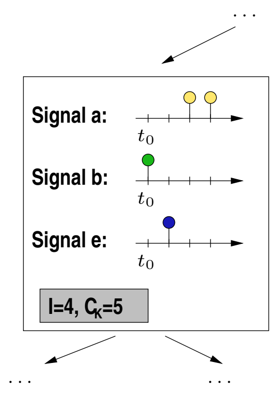
Priority-Based Access As outlined before, in the priority based access scheme it is possible that high priority activity threads influence the waiting time of a signal e more than once. For the determination of $\mathbf{w}_e$ therefore the maximum busy period of the activity threads with higher priority needs to be found. The event streams of the internal signals can be derived with the method described in the previous section. In order to find the worst case event sequence which satisfies both the conditions set by the event streans and the minimum event distance, a variant of the branch-and-bound algorithm presented in section 5.2.2 can be used. To be investigated are all event sequences containing the observed signal e and all signals of activity threads with higher priority which form a busy period. $C_K(I)$ denotes the entire blocking time caused by high priority signals, where here $E_{\varphi,s}(I)$ denotes the number of signal occurences given by the found event sequence s:

$$C_{K,s}(I) = \sum_{k \in K_{at},\, f \in A_{P,k},\, \varphi \in S_f} E_{\varphi,s}(I) \cdot \mathbf{b}_f \tag{5.20}$$

I is a busy period, if $C_{K,s}(I) - I \geq 0$ is given. If signal e would occur at $I = 0$, the waiting time would be maximum, but depending on the minimum event distances this might not be possible. Instead, the found event sequence gives the interval I_e, after which e can occur. This is illustrated in figure 5.18. The waiting time is thus reduced, and can be calculated as:

$$w_{e,s} = C_{K,s}(I) - I_e + max\,(\mathbf{b}_l)_{l \in L_{at}} \tag{5.21}$$

The worst case waiting time for e now is the maximum waiting time of all event sequences s which constitute a busy period of the high priority threads.



$K = \{a, b\}, L = \{\}$, $c_a = 1, c_b = 3$, $ed_{\beta\alpha} = 2$, $ed_{\beta\epsilon} = ed_{\epsilon\alpha} = 1$

$\Rightarrow I_e = 1$; $w_{e,s} = C_{K,s} - I_e = 4$

Figure 5.18: Partial view of Search Tree

5.5 SDL System Analysis

5.5.1 Overall Reaction Time

The entire range of possible reactions of the SDL system to an external trigger e is described by the set B_e, which contains all branches R of the according task precedence system. The sequence of transitions given by one R constitutes one possible reaction to the signal e. The worst case time required to execute this reaction $\mathbf{r}_R$ is to be determined by the analysis of the entire SDL system.

In a pure server model implementation, each transition t of R is contained in the implementation process of the SDL process it belongs to. In a pure activity thread implementation, all transitions which are triggered by one external signal are contained in one implementation process. If a combination of the implementation models has been chosen, the transitions belonging to R can be part of different implementation processes. In order to identify the internal signals which mark the border between the implementation processes, a subset of R named RI is defined. RI denotes the sequence of transitions t which are each the start of a new implementation process. In other words, each signal $i(t)$ which triggers a transition $t \in RI$ is output signal of one implementation process and trigger to the next implementation process. Obviously, in a server model implementation, $R = RI$.

t_k denotes the kth transition of a total of n transitions in RI, and $i_k = i(t_k)$ denotes its triggering signal. The overall time $\mathbf{r}_R$ to finish reaction R is then given as

$$\mathbf{r}_R = \sum_{k=1}^{n-1} \mathbf{r}_{i_k \to i_{k+1}} \qquad \text{with } i_k = i(t_k), \;\; t_k \in RI \tag{5.22}$$

with

$$\mathbf{r}_{i_k \to i_{k+1}} = \mathbf{q}_{i_k} + \mathbf{w}_{i_k} + \mathbf{c}_{i_k \to i_{k+1}}. \tag{5.23}$$

The event streams and event dependencies of the internal signals can be derived with the algorithms described in sections 5.2.3 and 5.2.4. The execution times $\mathbf{t}$, $\mathbf{s}$, $\mathbf{q}$, $\mathbf{d}$, $\mathbf{l}$ and $\mathbf{u}$ are given by the VHDL implementation. With this input, the worst case reaction times $\mathbf{r}_{i_k \to i_{k+1}}$ of the implementation processes, i.e. the time required to output signal i_{k+1} after trigger i_k, can be bound with the methods of section 5.2, 5.3 and 5.4.

5.5.2 Derivation of Internal Event Streams

It becomes clear from the above that the event streams of all internal signals are necessary for the real-time analysis. They can be derived stepwise after the method described in the following:

For each implementation process, which is entirely triggered by external signals, the real-time analysis can be performed and the output event streams and event dependencies determined. In the next step, all implementation processes are analyzed whose input event streams are now completely defined, in turn deriving the output event streams.

It is nevertheless possible that a cycle exists in the form that the analysis of implementation process $P1$ depends on the output event stream of process $P2$. $P2$ in turn can only be analyzed if the output event stream of $P1$ is known. An example of such a situation is depicted in figure 5.19. In such a case the following procedure is proposed: Starting with process $P1$, the unknown event stream of signal b is approximated with the event tuple $z = \infty, a = 0$, and the real-time analysis is performed. With the thus generated output event stream for signal c, process $P2$ is analyzed, deriving a new estimate for event stream ES_β. If with the new event stream the condition $C(0) \leq a_{2,\iota} \;\; \forall\, \iota \in S_i, i \in I_{P1}$ is fulfilled, the special case is given that only one occurence of each signal can influence the reaction time. The real-time analysis performed with $ES_\beta = \left\{ \binom{\infty}{0} \right\}$ then has yielded a correct result and the derived ES and $\mathbf{r}$ are correct. If the condition is not fulfilled, the analysis cycles is calculated again with the new ES_β, until no more changes occur.

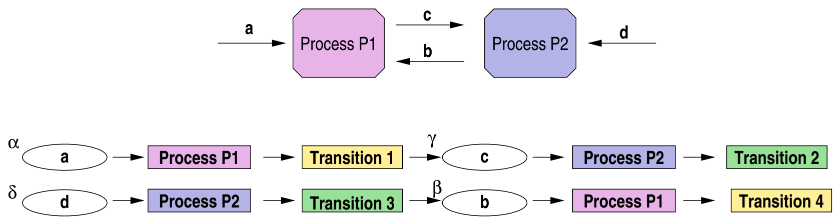


Figure 5.19: Cyclic dependency between ES_γ and ES_β

5.5.3 Inclusion of Additional Information

As outlined in section 5.1.2, it is not possible that the real-time analysis evaluates functional dependencies contained in the SDL processes, which may in some cases lead to overly pessimistic results. One possibility for the designer to smoothly include background information in the real-time analysis without increasing the complexity is with additional event streams and event dependencies. Next to the given event streams of external signals, each internal signal source α can be annotated with a given event stream ES_α and event dependencies, which are used instead of the generated values in the real-time analysis. In such a fashion it is e.g. possible to express that an internal signal is only sent every n occurences of the triggering external signal.

The concept of **operation modes** goes one step further. With their help, the designer can express the knowlege that there exists a number of different operation conditions of the SDL system, with different external conditions and also with distinctive internal functionality. For the real-time analysis, each mode has its own set of analysis variables which characterize the SDL system. In particular, by selecting the signals contained in I_P and A_P, it can be taken into account that certain external or internal signals never occur in one particular mode. Specific temporal correlation of a mode is introduced with event streams and event dependencies. Knowledge on different behaviour in a mode can be put down by selecting the transitions contained in T_e.

5.6 Message Queue Depth

Message queues are expensive to implement in hardware, and the area required significantly increases with the depth of the queue. It is therefore desirable to dimension them as small as possible. A message queue which is undersized, on the other hand, causes unexpected behaviour at run-time. Depending on the implementation, SDL signals might be delayed or lost which leads to higher reaction times than anticipated by the real-time analysis or behaviour which diverges from the SDL specification. For these reasons it is imperative to know the required queue depth as exactly as possible.

5.6.1 Required Message Queue Depth

The required message queue depth of process P can be found by assuming a queue which is deep enough and determining the maximum number m of signals which can wait in the queue at any given time. At first, no event dependencies are considered.

In the already well known special case $C(0) \leq a_{2,\iota}\ \forall\, \iota \in S_i, i \in I_P$, it is guaranteed that a signal is processed before the next of its type arrives. This leads to the trivial solution that the queue maximally needs to hold one signal of each type from each signal source, i.e.

$$m = \sum_{i \in I_P} dim(S_i) \tag{5.24}$$

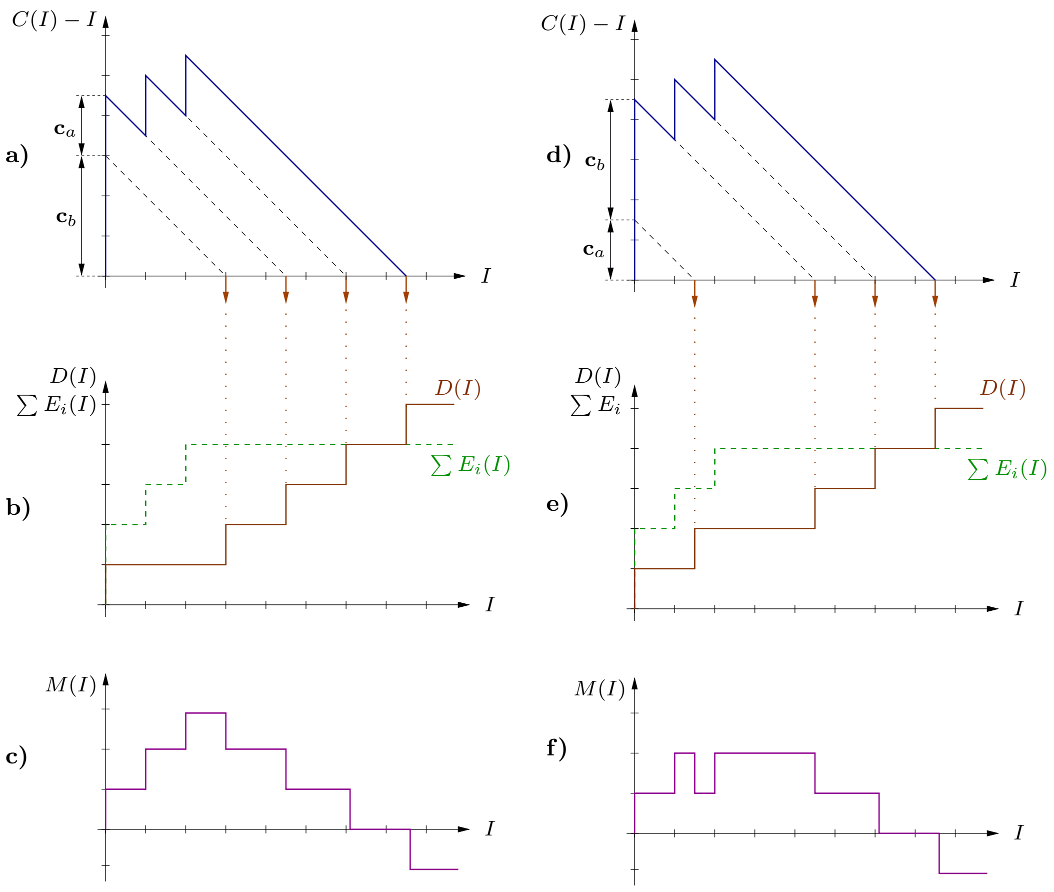
In the general case, the number of signals in the queue at a given time can be determined after the following consideration: During a time interval I, the maximum number of signals which can arrive is given by $E(I)$. Now a departure function $D(I)$ is defined, which denotes the number of signals whose processing is finished during I. During I, $(D(I) + 1)$ signals are removed from the queue. Assuming that the queue is empty at the begin of I, the number of signals in the queue after I is

$$M(I) = \sum_{i \in I_P} \sum_{\iota \in S_i} E_\iota(I) - (D(I) + 1) \tag{5.25}$$

The sought maximum message queue length then is the maximum of all possible $M(I)$.

For the departure function $D(I)$ it is useful to take one more look at the function $C(I) - I$ which has been calculated for the derivation of the reaction time. Like mentioned before, it is identical to the unfinished work in the system $U(t)$ defined in queueing theory. After [Kle75], in a FCFS scheme the departure instants from the system can be derived "by extrapolating the linearly decreasing portion of $U(t)$ down to the horizontal axis; at these intercepts a customer departure occurs and a new customer service begins". Thus, at each of these departure instants $D(I)$ increases by one, starting with $D(0) = 0$. This is illustrated in figure 5.20 (a) and (b).

If several signals can arrive simultanously, the execution order is not defined. All possible execution orders have to be investigated, as they lead to



$ES_\alpha : \left\{ \binom{12}{0}, \binom{12}{1}, \binom{12}{2} \right\};\ \ \mathbf{c}_a = 1.5$

$ES_\beta : \left\{ \binom{10}{0} \right\};\ \ \mathbf{c}_b = 3;$

Figure 5.20: Derivation of queue depth for example from figure 5.7

different departure functions $D(I)$. This is depicted in figure 5.20 (a) and (d). For the different departure functions, the number of messages in the queue $M(I)$ is calculated using equation 5.25 (figure 5.20 (c) and (f)). Here, $M(I) = -1$ means that currently no signal is in the system, i.e. that the queue is empty and the EFSM is idle. The next arriving signal will be processed immediately. When $M(I) = 0$, the queue is empty, but the EFSM is busy, i.e. the next signal will be put in the message queue.

The worst case is given at the earliest possible arrival of signals, which is ensured by the properties of the event streams defined in equation 3.6. As is the case with the worst case reaction time, with equation 3.6 it can be shown that only $I < I_0$ with $C(I_0) - I_0 \leq 0$ have to be investigated. At the same time, this condition ensures that the message queue is empty at $t = 0$. With that, the maximum required message queue depth m is:

$$m = max(M(I)) \qquad \forall I < I_0 \text{ with } C(I_0) - I_0 \leq 0 \tag{5.26}$$

5.6.2 Event Dependencies

The consideration of minimum distances between signals can lead to reduced required message queue depths. Like before, the task is to find the worst case sequence of events which satisfies the constraints given by the event streams and the event dependencies, and which generates the worst case, here regarding the message queue depth. In order to find this worst case event sequence, the branch and bound algorithm described in section 5.2.2 can be used, with one modification. In contrast to the reaction time, for the queue depth it is not ensured that a node with a larger C and at the same time smaller I than a second node will always lead to the worse case. Therefore, only nodes with $C(I) < I$ can be deleted.

After the search tree has been built after this fashion, all leaf nodes are investigated, i.e. the event sequence given by each distinctive branch is considered. For each event sequence, $E(I)$ and $C(I)$ are calculated, starting with the earliest signal as $I = 0$. The fact that for each additional signal $C(I) < I$ is given, means also that the message queue is empty before $I = 0$. Using $C(I)$ and $E(I)$, $D(I)$ and $M(I)$ can be derived like described in the previous section. Figure 5.21 illustrates this for the example already used in figure 5.7 and 5.20, with an additional event distance $ed_{\alpha\beta} = ed_{\beta\alpha} = 1$. This procedure has to be repeated for all possible root nodes starting with signal $e \in I_P$. Like before, the required message queue depth is the maximum of all derived $M(I)$.

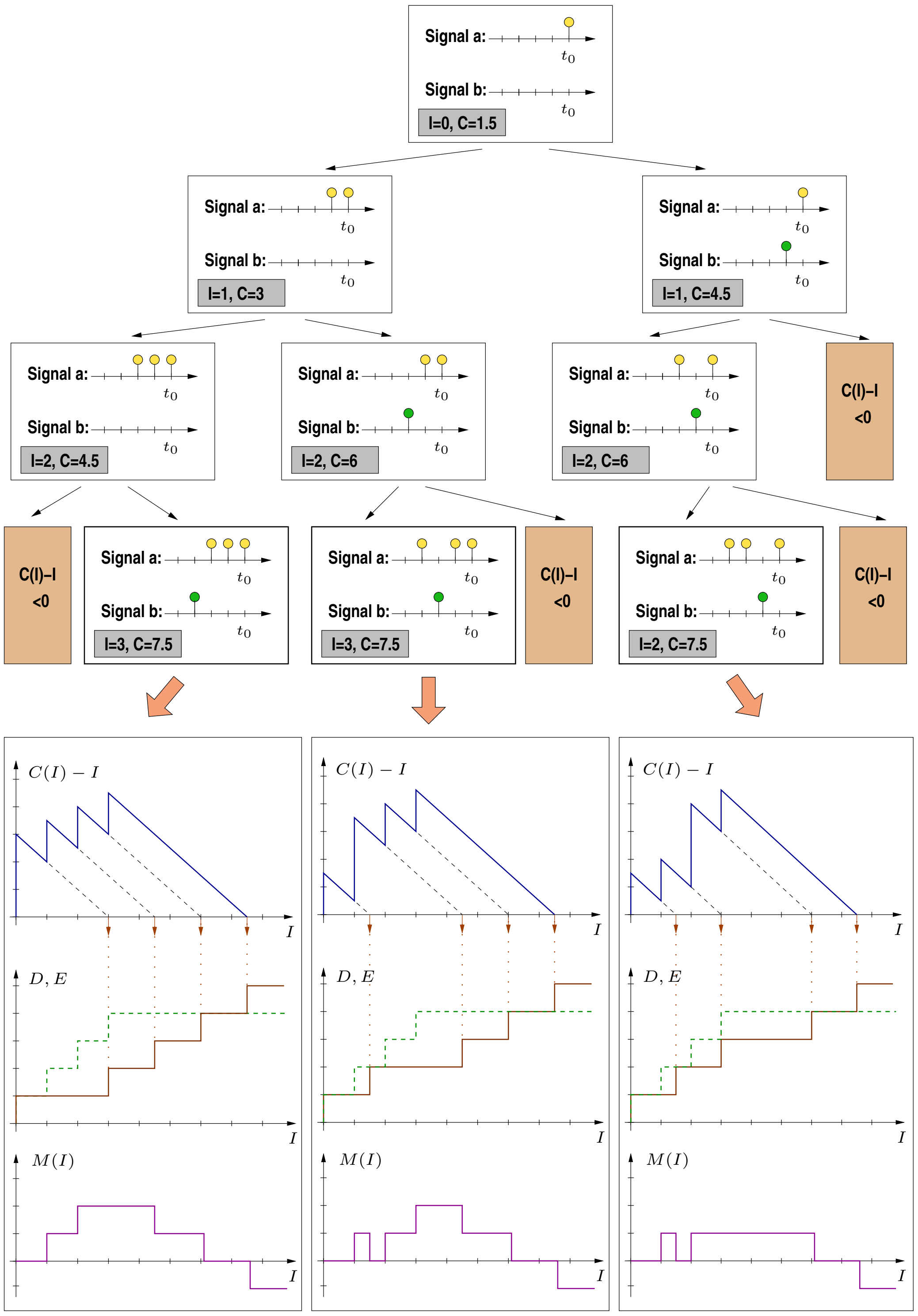


Figure 5.21: Derivation of queue depth for example from figure 5.7 with additional event dependency $ed_{\alpha\beta} = ed_{\beta\alpha} = 1$

5.6.3 Undersized Queues

That a queue has been dimensioned too small means that it is possible for a signal to arrive when the queue is full. The consequences of this depend on the strategy implemented in the signal channel and the queue. If the channel protocol does not provide a handshake or similar, the signal is lost. This leads to a behaviour at run-time which unpredictably deviates from the SDL specification, possibly leading to a completely changed functionality. The second possibility is that the sending of the SDL signal is blocked until the queue is again ready to receive a signal. This, too, is a discrepancy to the non-blocking send assumed in SDL, but it leads only to a temporally and not functionally changed behaviour, which is detailed below.

If a message queue is one place too small, the sending of a signal e to that process can maximally be delayed for the longest execution time of all signals which can occur simultaneously. This delay is termed $\mathbf{z}$. It does not increase the worst case reaction time to this particular signal e, since the waiting time already includes all other simultaneous signals. The execution time of the transition sending the signal, however, is prolongeated by $\mathbf{z}$. This affects the worst case reaction times of all other reactions R which involve the process P sending signal e. Furthermore, it is possible that due to the increased execution time the queue depth calculated for process P is also no longer valid. In that fashion the delay might be propagated throughout the SDL system. These effects, however, can be calculated beforehand during the real-time analysis by calculating new reaction times which take the shortened queue depths into consideration. If the laxity given by the deadlines is large enough, the message queues can deliberately be dimensioned too small such as to save hardware area.

Chapter 6

Automated Design Process

While the implementation models and the real-time analysis presented in the previous chapters are in fact independent of the employed design method, the basic idea has been to automate the process of generating an implementation from the SDL specification. Section 6.1 details the problems which have to be solved during an automated design process. The rapid prototyping framework REAR, which realizes such an automated design from SDL and was used to test the concepts presented in this work, is described in section 6.2.

6.1 Design Tasks

An SDL specification describes the desired functionality of the system under development at a high level of abstraction, with SDL processes communicating over messages. It intentionally contains no implementation details, which have now to be decided during the design process.

The first step is the partitioning and mapping of the SDL specification on the available execution units, which also fixes the mapping to hardware and software. At the same time, a decision on the implementation model has to be made, which in turn influences the granularity of the partitioning.

A central task is the communication refinement, where SDL's abstract communication is mapped to actual channels and protocols. This involves the communication within and between hardware and software based execution units. Additionally, the connection to the environment has to be implemented. From the view of the specification, everything not contained in the SDL system is the environment. Therefore, communication might have to be established to a software task or hardware circuit on the same execution unit, to a different processing node or via sensors and actors to the physical environment of the electronic system itself. For a hardware implementation, the run-time components for the mentioned signal transfers and the message queues have to be chosen and parametrized. The queues have to be dimensioned in their depth and their signal coding needs to be fixed.

Time in SDL is dimensionless; it is left to the implementation to define which amount of actual time corresponds to one time unit in the specification. During the design process, a timer component has to be chosen from the library. The parametrization of its input clock frequency and its data width, together with a possible scaling of the time values in the specification, define the resolution of time in the implementation.

For the implementation of the process behaviour, a description in a hardware language has to be generated from the SDL specification. This can be a VHDL model like described in section 4.2, but naturally the use of other hardware description languages and other finite-state machine specification techniques is equally possible. In the hardware model of the EFSM, the access to the chosen run-time components, message queues, communication channels and timers, has to be inserted.

The SDL processes and run-time components have to connected with each other and with additional predefined components which have been included at SDL level like described in section 3.3. This results in a structural model of the hardware implementing the entire SDL system. Finally, the high level hardware design has to be synthesized into a netlist of components in the target technology, placed and routed.

6.2 Rapid Prototyping Environment REAR

The aim of the rapid prototyping environment REAR[1] ([PMK$^+$00]) is to substantially reduce development times of real–time applications by confirming the functional and temporal requirements at a very early stage of development with the help of an executable prototype. It integrates two complementary tasks: On the one hand it provides an automated design environment for a rapid and facile generation of a working prototype. On the other hand, the design process is extended with real–time requirement specification and analysis in order to prove that the embedded system will meet all timing requirements, and to verify that the timing requirements have been modeled correctly. The concepts and methods presented in this work have been implemented and tested in the REAR environment.

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

6.2.1 Target Architecture

The rapid prototyping target architecture was designed to support real–time analysis in guaranteeing realistic, not–too–pessimistic worst–case execution times. The basis for this is the task classification model presented in [FFKM97], where each type of real–time task corresponds to a best suited type of processing unit, in terms of performance and deterministic execution times. It is a configurable and scalable heterogeneous multiprocessor system consisting of standard off–the–shelf components, which are tightly coupled by a global PCI–bus (figure 6.1).

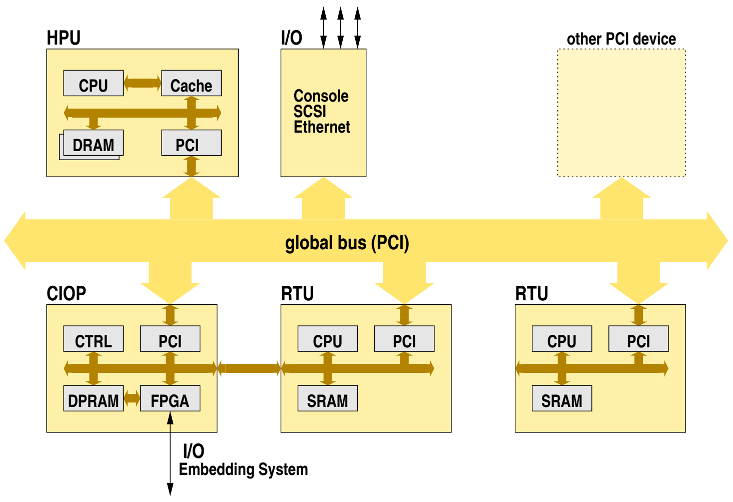


Figure 6.1: Rapid prototyping target architecture – block diagram

The High Performance Unit (HPU), which is used for soft real–time tasks and as the development host, is based on standard computer architectures to benefit from technological advances. The Real–Time Unit (RTU) is optimized for hard real–time tasks with short response times, which do not allow predictions of the cache behaviour. Instead, the slower RAM–access times have to be used for the determination of worst case execution times.

The Configurable I/O-Processors (CIOP) consist of one Xilinx FPGA for application specific hardware and optionally an additional dual ported RAM, which can be used for e.g. message buffers and as temporary memory for the FPGA. The CIOP acts as separate application specific processing unit for tasks with deadlines too short to be met in software and provides a flexible way of linking the prototyping architecture to the embedding process. It is the target of the automated hardware design process described in this chapter. The FPGA is connected to the other processing units over a PCI bus interface, and additionally to the RTU with a direct connection to a CPLD

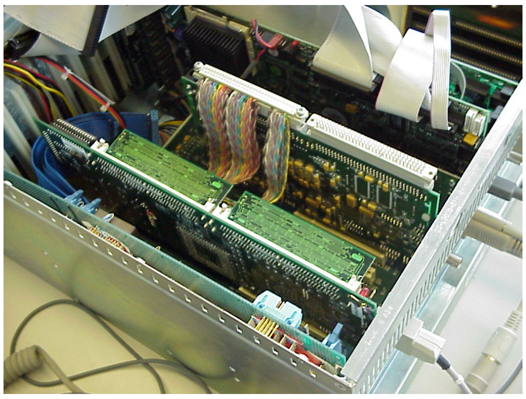


Figure 6.2: Photo of the rapid prototyping target architecture

implementing glue logic which links it to microprocessor's local bus. Signals to and from the embedding process can be directly interconnected with I/O-pins of the FPGA. A more detailed description of the target architecture with a first CIOP from own development is given in [FKMF97]. The experiments referred to in chapter 7 have been conducted on the Spyder-Virtex-X2 rapid-prototyping board developed by FZI, Karlsruhe ([For00]). The FPGA board has been operated with a clock period of 30 ns.

6.2.2 Rapid Prototyping Design Process

Figure 6.3 shows an overview of the rapid prototyping design process. It starts with a specification in SDL, using the Telelogic's CASE tool SDT for editing, syntax and semantic check and for the simulation at functional level ([Tel]). A typical screenshot of SDT with a SDL system diagram, system organizer and simulator is shown in figure 6.4. The timing constraints, i.e. deadlines, event streams and event dependencies, are annotated to the SDL specification.

The prose representation of the SDL specification is output from SDT and is parsed by the SDL-Compiler presented in [BRM$^+$99]. It has been extended in [Lar98] to extract the information relevant for the real-time analysis. This includes the SDL system's task precedence graph and the annotated timing constraints. This real-time analysis model, together with the worst case execution times obtained from the target architecture, is the input of the real-time analysis ([Pet00], [Kol01]).

The SDL model is partitioned by manually mapping entire SDL processes on the processing units of the target architecture REAR, depending on their timing requirements and computational complexity. This mapping is also

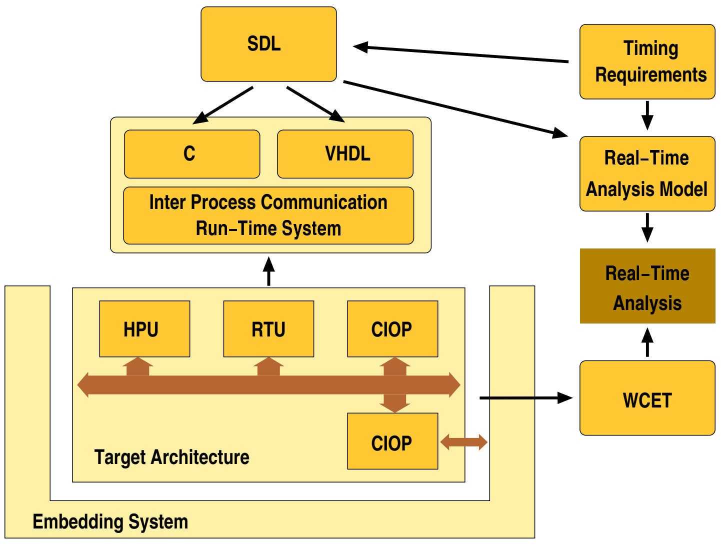
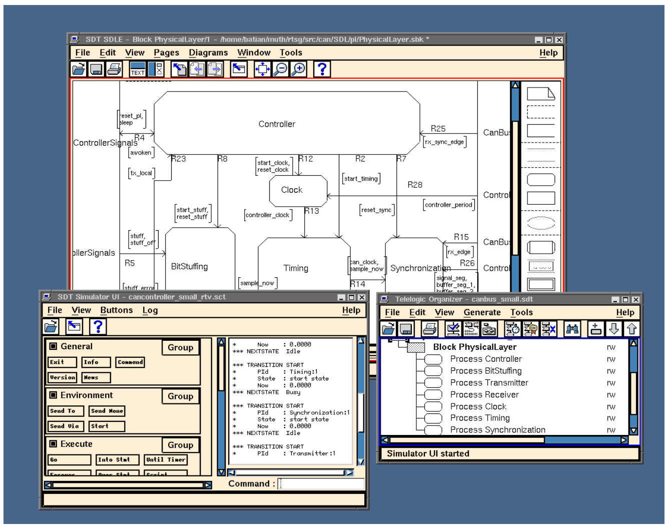


Figure 6.3: Rapid prototyping design process – overview

annotated to the SDL system. For the SW part SDT's CAdvanced code generator is used to generate C-code. Each implementation process is mapped to one task of the freely available real–time operating system RTEMS as target, into which message–based earliest deadline first scheduling has been integrated, as detailed in [Kol01]. A retargetable IPC layer hides the system's heterogeneity and provides high level primitives for local and remote (inter–unit) communication and synchronization ([FMF98]).

VHDL Code Generation Figure 6.5 shows a closer view of the design process targeting application specific hardware. The textual SDL description is parsed by the SDL-Compiler, which is based on JavaCC. The parser builds in several passes of the semantical analysis an internal model of the SDL system, which is then used to output the VHDL implementation for all SDL processes annotated with "map-on-asic". For an implementation after the server model, the internal system model is traversed following the hierarchical structure of the SDL system, i.e. its composition of processes. For each process, the EFSM is output in VHDL. The code generation after the activity thread model, in contrast, has a recursive structure. Starting with the external signals, for each signal the receiving process is determined, and the appropriate transitions in the destination are output in VHDL. Each transition has to be processed twice: In the first pass, the VHDL implementation for the entire transition except the signal outputs is generated. The second pass deals with the transition's signal outputs where in turn the signal



```
*       Now     : 0.0000
*** NEXTSTATE  Idle

*** TRANSITION START
*       PId     : Timing:1
*       State   : start state
*       Now     : 0.0000
*** NEXTSTATE  Busy

*** TRANSITION START
*       PId     : Synchronization:1
*       State   : start state
*       Now     : 0.0000
*** NEXTSTATE  Idle

*** TRANSITION START
*       PId     : Transmitter:1
```

Figure 6.4: CASE-tool SDT – screenshot

destination is determined, starting the next recursion level. This two-step procedure ensures that the transition is finished before the next transition is started, which has already been postulated in section 4.4. [Rei97] gives details on the SDL parser and on the VHDL generation after the server model. The generation of VHDL code for the activity thread model is dealt with in [MK00].

The support of SDL language features in the design process respectively by the SDL-Compiler is summarized in table 6.1. The currently supported SDL subset is sufficient for the specification of a wide range of designs. In future versions, the features listed in the second row could be added. In contrast to this, there is a number of SDL constructs which are not supported because they are either not suitable for implementation in general or specifically for an implementation in hardware, or because they are not covered by the real-time analysis.

Component Integration The inclusion and adaption of the run-time components, which is dependent on the target architecture, the used library and the actual application, is not task of the SDL-Compiler. Instead, the compiler can insert generic procedure calls for "send", "remove-from-queue", etc. in the generated VHDL code, which can later be replaced by interfaces to the run-time components by the high-level synthesis system CADDY-II, like described in [BRM^{+}99]. Figure 6.5 depicts a second option, which is

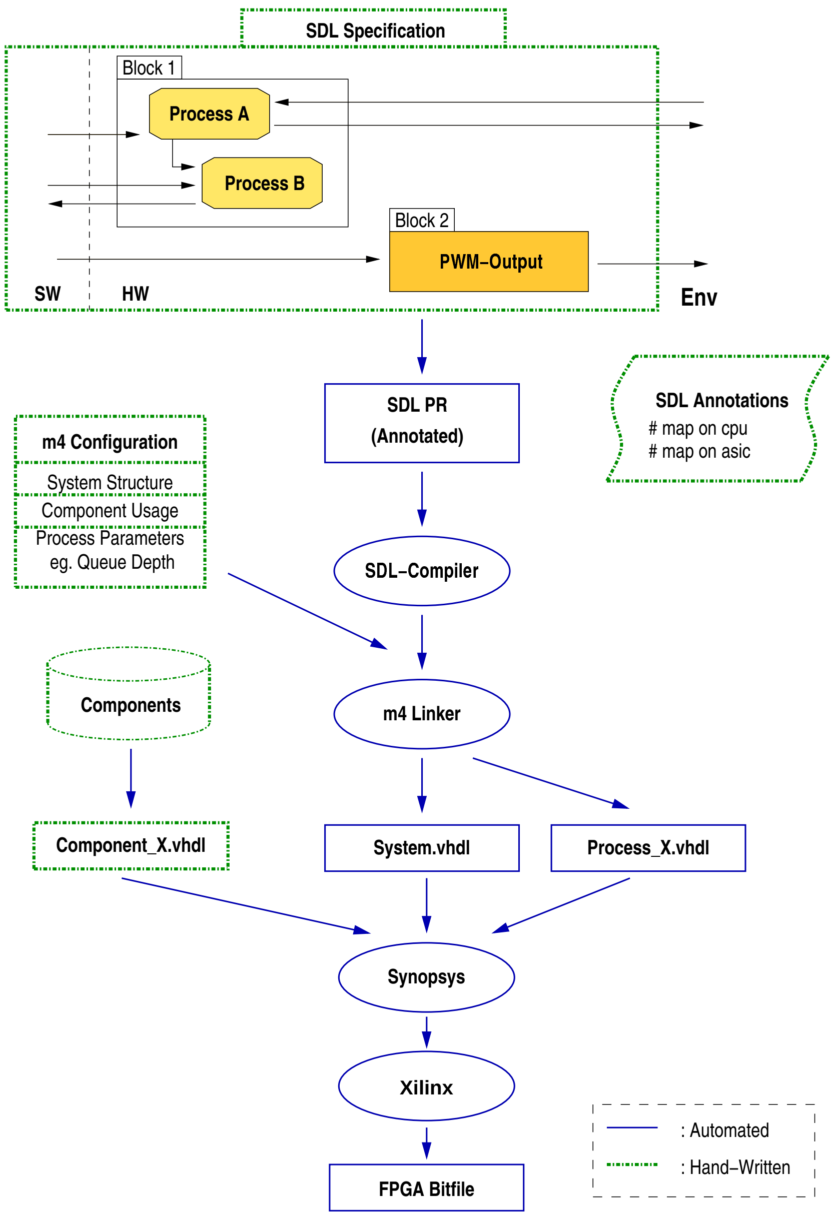


Figure 6.5: Rapid prototyping hardware design process

intended for the RT-level synthesis flow, and was used in the presented experiments. Here, the interfaces to the run-time components are inserted via macro replacement in a link-step after the SDL-Compiler. The generated VHDL-code contains macro calls in all places relevant for the run-time interface, i.e. in the FSM at "send", "remove", "set-timer", but also in the entity and variable declarations. The link step is realized with the power-

Supported	**Support possible**	**Not supported**
FSM control flow: State, Nextstate, Task, Decision, Label, Join, Input, Output, Continous Signals; SDL Data Types; Timer; View/Reveal; Native VHDL code	Procedures, Macros, Export/Import; Object oriented specification	Non-Deteterminisms; Informal Text; Dynamic Process Creation; Abstract Data Types; Save, Priority Input, Enabling Conditions

Table 6.1: Supported SDL Language Features

ful freeware macro processor `m4` ([Ren94]). The macro replacement process is controlled by a set of configuration files, which include the necessary implementation details. They define the run-time components to be used and their parametrization, as well as the communication channels and additional components of the system design. Figure 6.6 depicts the block structure of such a system on the target architecture's CIOP.

The VHDL models of all components are synthesized by the Synopsys design compiler, which generates a gate-level net list specific for the target technology FPGA. The net list is placed and routed on the target FPGA by the Xilinx software, which generates a bit-file which can be downloaded on the FPGA via the board's PCI interface.

As figure 6.5 indicates, the hardware design process is semi-automated: Next to the SDL specification and the run-time components, the `m4` configuration files are to be written by hand. The rest of the design process is tool supported. Design iterations with modified SDL specifications or changed parameters require no manual interference, as long as the implementation details in the configuration are not affected.

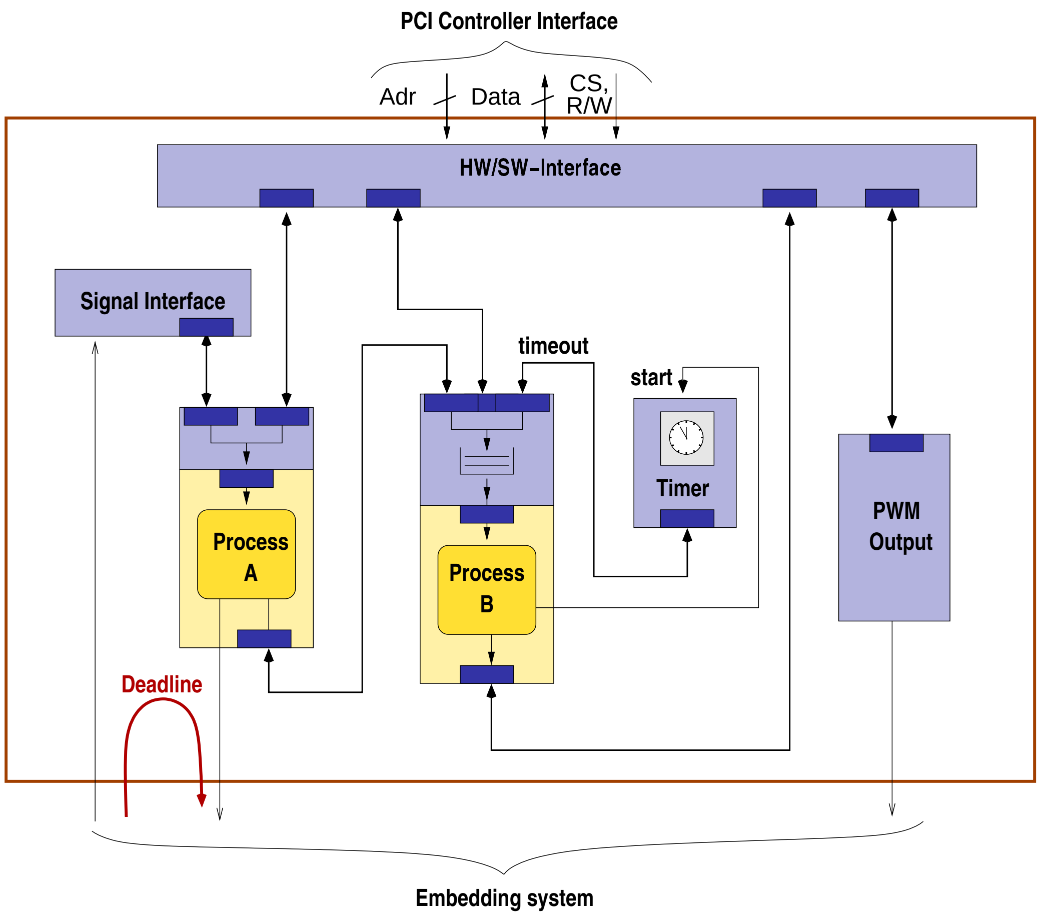


Figure 6.6: Hardware system architecture for SDL example from figure 6.5

Chapter 7

Experimental Results and Evaluation

The rapid prototyping design environment presented in the previous chapter has been used as a test bed for different application examples. The aim here was firstly to demonstrate the feasibility of an automated generation of application specific hardware from SDL. Secondly, the most important properties of the generated hardware, resource usage and timing, and the trade-offs between the different implementation models were to be investigated. This chapter first gives a rough characterization of three used application examples; they are given in detail in the appendix. The resources needed by the generated hardware are fixed during hardware synthesis. Section 7.2 reports the results from the implementation and compares the synthesis results for the application examples and implementation models. The CAN bus physical layer has stringent real-time requirements, and was therefore selected for a comparison of its timing properties in section 7.3. Section 7.4 summarizes and evaluates the results from the application examples.

7.1 Application Examples

7.1.1 CAN-bus physical layer

CAN [E$^+$94] is a serial field bus which was originally developed for communication in vehicles, but has reached by now widespread use in the field of production automation. The CAN bus runs a masterless, message oriented bus protocol with CSMA/CA (Carrier-Sense Multiple Access/Collision Avoidance) access mode. Bus access is granted to each participant by bitwise arbitration using individual message IDs. Several cooperating error detection mechanisms guarantee fast system wide error detection and error recovery. CSMA/CA bus access, in combination with message priorities, the short data block length (max. 8 Byte) and data rates up to 1 Mbit/s lead to very short message latencies.

According to the OSI layer model for communication systems, the physical layer provides the transmission of unstructured bits across the physical medium. For this, it defines the signal coding and timing of the bits, as well as the physical connection to the bus. The CAN bus has a Non-Return-to-Zero (NRZ) signal coding, i.e. the bus level is high or low during the entire bit time. To ensure that the bit stream on the bus contains enough signal edges for a time synchronization of all participants, bit stuffing of width five is employed. This means, that the sender of a message inserts a stuffing bit of the opposite polarity after each sequence of five equal bits. These stuffing bits are removed by the receiver, restoring the original message.

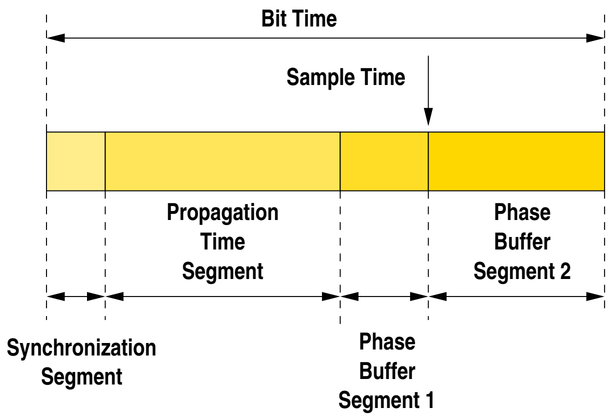


Figure 7.1: Internal bit timing in CAN

The internal bit timing provided in CAN divides each bit time into four non overlapping segments, as depicted in figure 7.1. The *synchronization segment* denotes the beginning of a new bit in the stream. The *propagation time segment* can be dimensioned to accomodate the maximum propagation delay between sender to receiver. The sample time is fixed by the width of the two *phase buffer segments.* While the synchronization segment is one time unit long, the width of the other segments is configurable between one and eight time units, with a total bit length between 8 and 25 time units. CAN avoids the overhead created by explicit start bits like used in asynchronous transmission. Instead, the signal edges contained in the bit stream are used to synchronize the bus participants in the following manner: each receiver expects the next signal edge during the *synchronization segment.* If it apears earlier or later, the receiver adapts the sample time and at the same time the beginning of the next bit time by adjusting the two *phase buffer segments*.

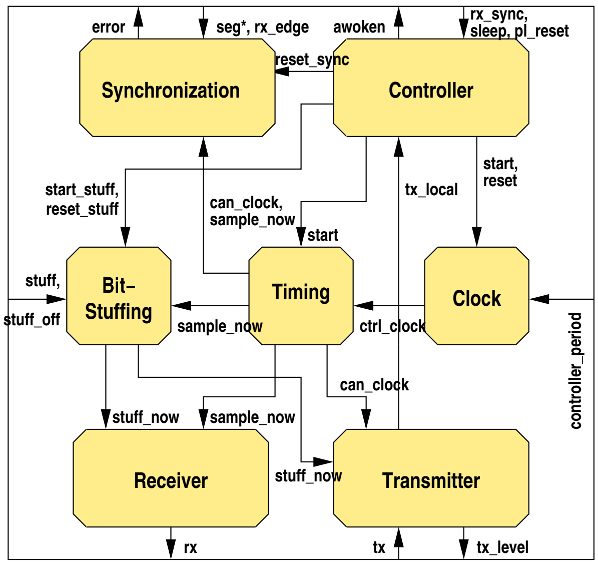


Figure 7.2: SDL specification of the CAN physical layer (SDL processes)

Figure 7.2 shows the process structure of the physical layer's SDL specification. The SDL block physical layer has interfaces to the physical bus and to the CAN data link layer (DLL), which is not further considered here. It implements the following functionality:

- Configurable timing parameters
- Simultaneous sending and receiving
- Communication of bits to send and bits received via SDL messages from/to data link layer
- Start from reset state upon bit to send from DLL or observed edge on bus
- Reset upon signal from DLL

The process Clock outputs the internal time units using a SDL timer. The process Timer counts the time units and delivers internal events at the begin of a new bit interval (signal `can_clock`) and at the sample time (signal `sample_now`). The process Synchronization observes the edges on the bus (`rx_edge`) and adapts if necessary the phase buffer segments, which are made available to the process Timing with the SDL construct view/reveal. The process BitStuffing watches the bit stream on the bus and gives a notification (`stuff_now`) when a stuffing bit is to be inserted respectively to be removed. The process Receiver samples the bus and sends the received bit (signal `rx`) to the data link layer. The process Transmitter gets the bits to send from the DLL (signal `tx`) and outputs them on the bus at the appropriate time (signal `can_clock`). The process Controller keeps track of the overall reset state of the block and starts and resets the other processes. The complete SDL specification of the physical layer can be found in appendix A.

7.1.2 Servo Motor Controller

In a rapid prototyping application, six simple servo motors of the kind used in modelling were to be controlled. The servos require a pulse code modulated control signal, with a TTL-compatible high-signal of 1 to 2 ms duration, repeated every 10 to 30 ms, as depicted in figure 7.3. The length of the impulse determines the angle of the motor between 0° and 90°. The application required a defined position of the motors beginning at power on.

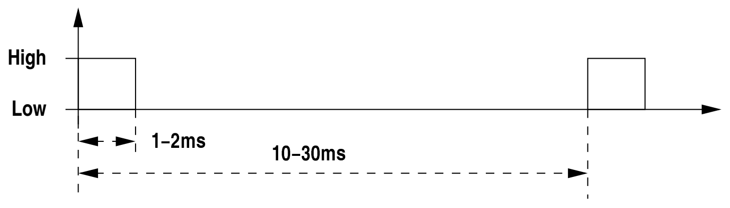


Figure 7.3: Timing of the servo motor control signal

Figure 7.4 depicts the block diagram of the servo motor controller's SDL specification. Each servo is controlled by one SDL process with one SDL timer for the length of the high pulse (position) and one SDL timer measuring the base period. During the start transition, both timers are initialized and started. Upon the output of the position timer, the control signal pin is pulled low. When the base timer is run down, both timers are restarted and the output pin is set high. At any time, a new duration for the position timer can be set via the external signal `pos_x`. The entire SDL specification of the servo motor controller is given in appendix A.

7.1.3 Assembly Line

The example termed assembly line does not have a corresponding physical experimental setup comparable to the CAN-bus or the servo motors. It is a

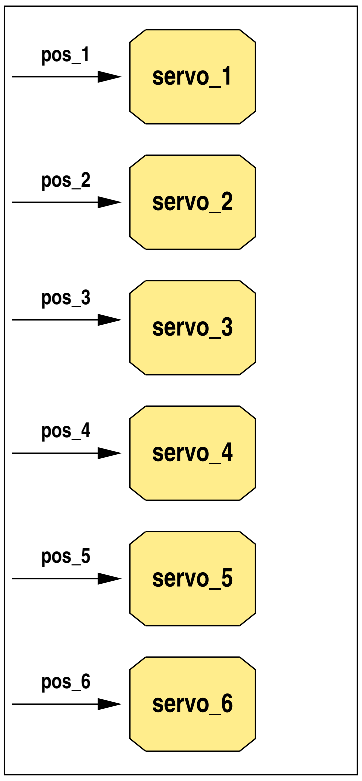


Figure 7.4: SDL specification of the servo motor controller

SDL specification of a manufacturing situation which was intended for test of the code generation after the different implementation models. It was implemented on the rapid prototyping environment REAR and triggered with events from software instead of real sensors.

In the assembly line scenario, two band conveyers bring two different kinds of half-products. Each product passes a photo sensor, which triggers the signal `in1` respectively `in2` in the SDL specification (see figure 7.5 and appendix A). In the process Controller, each half-product is counted, and if the required amount is reached, the motor of the corresponding band conveyor is stopped. The two motors are controlled each by one SDL process (Motor1 and Motor2), and are equipped with a common emergency stop button, which triggers the external signal `emergency_stop`, upon which both motors are stopped. A fourth process ProductCounter keeps count of the succesfully assembled batches of half-products.

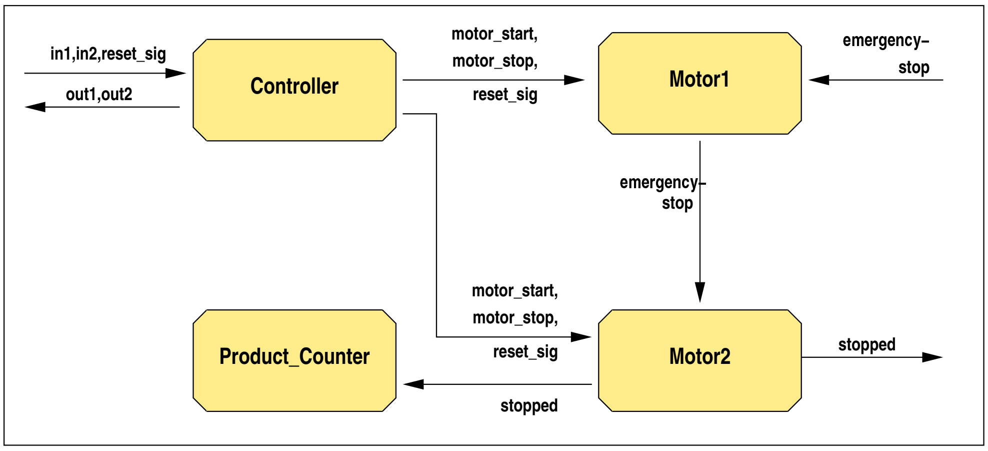


Figure 7.5: SDL specification of the assembly line example

7.2 Resource Usage of the Implementation

A first evaluation of the resource usage of the implementations can be derived from the number of the inferred memory devices. The reports given by Synopsys during synthesis allow a detailed analysis of the use of flip-flops. It will be discussed later to what extent this can be used as an indication of the entire resource requirements.

7.2.1 Resource Estimations

Like described in section 4.2.1, the implementation of a SDL specification on a FPGA consists of the implementation processes (server processes or activity threads), and the necessary run-time components. An analysis of the inferred memory devices which are reported by synopsys showed that it is possible to estimate the resource usage of both the run-time components and the generated processes based on a few implementation decisions and characteristic numbers of the SDL system. These are summarized in table 7.6. In the following, the memory device estimations (N = number of flip-flops) for the different components of the implementation are given.

queue_l0 The message queue of length 0 stores no signals. It only performs a multiplexing between the input channels and interfaces between the channel protocol and the EFSM:

$$N_{\text{queue\_l0}} = c + 2 \cdot \lceil ld(c) \rceil + 1 \qquad (7.1)$$

d_q	width of the message queue (signal ID + data)
l_q	length of the message queue
c	number of input channels
d_v	data width of a SDL processes local variable
d_t	data width of a timer
d_s	width a signal to send (signal ID + data)
s	number of SDL states of a process
i	number of implicit states of a process

Figure 7.6: Variables determining resource usage

queue_l1 The message queue of length 1 can store one signal. It executes the channel protocol at the input, and asynchronously the handshake with the EFSM:

$$N_{\text{queue\_l1}} = d_q + c + \lceil ld(c) \rceil + 4 \tag{7.2}$$

queue_ln The message queue of length $n = l_q$ has the functionality of the `queue_l1`, but additionally contains a FIFO of size $d_q \times l_q$ from a commercial component library (the intermediate input register inferred in the current implementation could be optimized):

$$N_{\text{queue\_ln}} = (l_q + 1) \cdot d_q + c + \lceil ld(c) \rceil + 4 \tag{7.3}$$

timer The SDL timer contains one register of the timer width containing the timer value, plus additional registers for reset and the channel protocol for sending the timer signal. Additionally, a adder/subtractor is required.

$$N_{\text{timer}} = d_t + 4 \tag{7.4}$$

write_to_sdl_signal This component reads a SDL signal from the asynchronous bus, stores it intermediately, and sends it to its destination process using the channel protocol.

$$N_{\text{write\_to\_sdl\_signal}} = max(d_q) + 3 \tag{7.5}$$

hw_sw_queue The hardware-software-queue receives all signals with destination SW from c source processes, and stores them in a FIFO of size $d_q \times l_q$.

It asserts an interrupt when the queue is not empty, and removes the signal from the FIFO after it has detected a read access from the software side:

$$N_{\text{hw\_sw\_queue}} = (l_q + 1) \cdot d_q + c + \lceil ld(c) \rceil + 4 \tag{7.6}$$

edge_to_sdl_signal This component sends a predefined SDL signal to a SDL process when it detects a defined edge on an input pin. Since the signal ID is constant, it is hard wired to '0' and '1':

$$N_{\text{edge\_to\_sdl\_signal}} = 3 \tag{7.7}$$

shared_var The shared variable component for the parallel activity thread model stores the local variables and state information of one SDL process P. It can be accessed from c activity threads:

$$N_{\text{shared\_var}} = \sum_{\text{vars of process P}} (d_v) + \lceil ld(s_P + 1) \rceil + c + \lceil ld(c) \rceil + 1 \tag{7.8}$$

mcsrr and count32 Each FPGA design contains a "master control, status and revision register" which identifies what is loaded in the FPGA, as well as a 32 bit counter helping as a kind of "heartbeat" during debugging.

$$N_{\text{mcsrr+count32}} = 34 \tag{7.9}$$

EFSM server process The EFSM implementing a server process performs a handshake to the queue. It stores the queue input, all local variables and the process state, plus if applicable a timer value and revealed variables. One register is inferred for each destination process of the SDL signals to send. Additional memory elements are necessary for the implicit states contained in the VHDL specification. These are determined during synthesis, and depend mainly on the complexity of the transitions.

$$\begin{aligned} N_{\text{server\_process}} = {} & \lceil ld(s+1) \rceil + d_q + i + 4 + \\ & + \sum_{\text{local vars}} (d_v) + \sum_{\text{revealed vars}} (d_v) + \sum_{\text{timers}} (d_t + 2) + \sum_{\text{destinations}} (d_s + 1) \end{aligned} \tag{7.10}$$

EFSM serial activity thread The EFSM of a serial activity thread implementation, similar to the server process, stores the local variables and states of all involved SDL processes. It contains memory for the queue input, timer variables and for the signals to be sent to the environment.

$$\begin{aligned} N_{\text{serial\_atm}} &= \lceil ld(s+1) \rceil + d_q + i + 4 + \\ &+ \sum_{\text{local vars}} (d_v) + \sum_{\text{timers}} (d_t + 2) + \sum_{\text{destinations}} (d_s + 1) \end{aligned} \tag{7.11}$$

EFSM parallel activity thread The EFSM implementing one activity thread in the parallel activity thread model has the well known interface to the message queue, and the registers for timers and signals to send to the environment. It needs memory for the variables and states of all involved SDL processes. In the case of a modified write of variables to the shared variable component, additional registers are inferred.

$$\begin{aligned} N_{\text{parallel\_atm}} &= d_q + i + 4 + \sum_{\text{SDL processes}} \Big(\lceil ld(s+1) \rceil + \sum_{\text{local vars}} (d_v) \Big) \\ &+ \sum_{\text{modified vars}} (d_v) + \sum_{\text{timers}} (d_t + 2) + \sum_{\text{destinations}} (d_s + 1) \end{aligned} \tag{7.12}$$

7.2.2 CAN-bus physical layer

During synthesis, synopsys logs the memory elements it has inserted in each VHDL entity, shown by name for all VHDL variables, signals and implicit states. It is therefore possible to determine exactly for which parts of the implementation and for which tasks in the EFSM the Flip-Flops (FFs) were used. Figure 7.7 and 7.8 summarize these synthesis results, which allow a comparison between the implementation models. The synthesis has been performed for the clock period of the FPGA board of 30 ns. The synthesis of the serial activity thread implementation however required a higher clock period.

Figure 7.7 shows the FF usage of the entire FPGA design, consisting of run-time components, message queues and the implementation processes (EFSMs). The group run-time components includes the timers, bus interface, SDL signal input and output, revision register and counter, like outlined in the previous section. Here, the serial ATM implementation has slightly lower requirements than server and parallel ATM, since only one EFSM has to be supplied with signals, leading to a lower communication effort. The serial

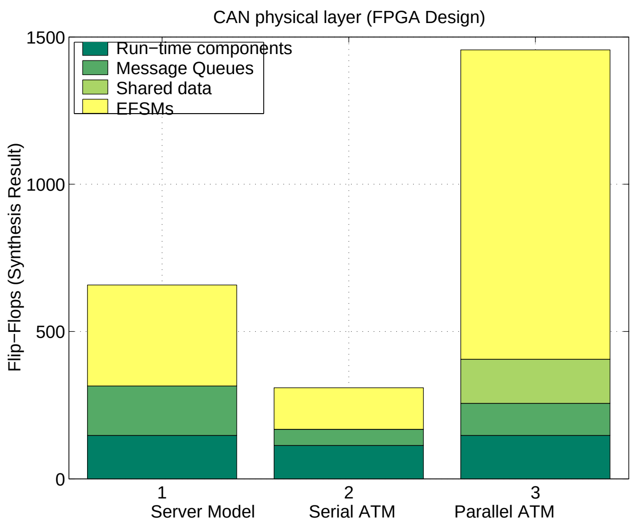


Figure 7.7: CAN bus: memory elements of the entire FPGA design

ATM's message queue portion is smaller for the same reason. The SDL specification contains more external input signals than processes, leading to a larger number of activity threads in the parallel ATM implementation than server processes in the server implementation. The number of queues in the parallel ATM therefore is higher, they do however not require a great depth as one AT only receives one signal type. In the preset case this leads to in a higher resource usage for the message queues of the server model. The shared data components only appear in the parallel ATM, and will be referred to later in this subsection. The implementation of the EFSM is the largest factor distinguishing the implementation models, and is therefore further broken down in figure 7.8.

The number of FFs required for the group "variables and SDL states" in the server model and serial ATM directly corresponds to the number of bits needed to store the variables and states defined in SDL (it is slightly higher in the server model because revealed variables are stored doubly). It is identical to the number of FFs in the shared data component of the parallel activity thread model. As equation 7.12 already shows, the EFSM of the parallel ATM also needs storage place for local copies of the process data. Since in a pure parallel ATM implementation, an activity thread is created for each of the numerous external input signals, the group variables and states is greatly multiplied in the parallel ATM.

According to the specification, the SDL timer is accessed only from one SDL process. Therefore, server model and serial ATM require one register storing the timer value. This is doubled in the parallel ATM, because the

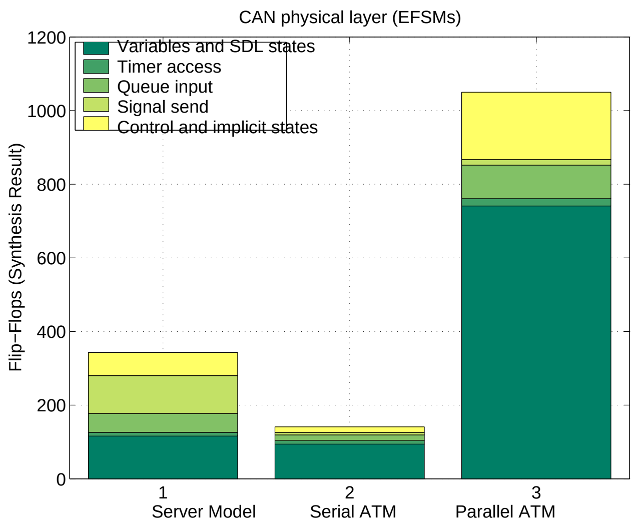


Figure 7.8: CAN bus: memory elements of the EFSM implementation

timer is started from within two ATs.

The number of FFs contained in "queue input" directly corresponds to the number of implementation processes (EFSMs) in the design. Therefore, it is smallest in the serial ATM and largest in the parallel ATM.

Like outlined in the previous subsection, for each destination process a register is necessary to store the signal to send. Since in the activity thread model only signals to the environment are sent, the group "signal send" is small. The server model, in contrast, requires many FFs because there is much communication taking place between the processes.

The FFs summarized in "control and implicit states" are used for the handshake with the queue, the timer, the shared data component and for the channel protocol. Their number therefore firstly depends on the number of processes, being low in the serial ATM and higher in the server model. The parallel ATM here has additional high requirements due to the many EFSMs and additionally numerous accesses to the shared data components.

7.2.3 Servo motor controller

Due to the simple structure of the SDL system, where one external signal corresponds to one SDL process, the parallel ATM implementation of the servo motor controller is identical to the server implementation. Therefore, only server model and serial ATM are compared. Like in the previous subsection, figure 7.9 and 7.10 show the required memory elements of the entire design and the EFSMs.

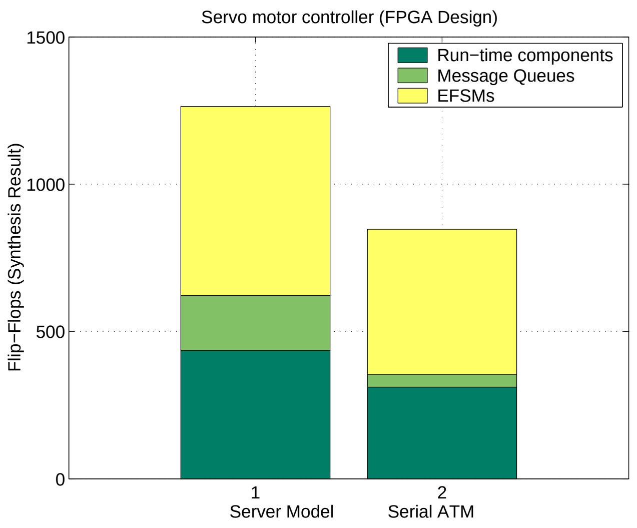


Figure 7.9: Servo controller: memory elements of the entire FPGA design

The large number of required run-time components results from the 12 timers, which are necessary in both implemention models. The signal I/O in the serial ATM is simplified due to the lower number of EFSMs, leading to a slightly smaller “run-time components” group.

The serial ATM requires one queue for the one EFSM, which however needs to be deeper than the 6 queues of the server model. The number of FFs required by “message queues” in the server model is therefore larger by about a factor 4.

Figure 7.10 details the memory elements required by the EFSMs. Like before in the CAN example, there is no difference between server model and serial ATM in the resources needed for variables, states and timer access. The “queue input” group, like to be expected, is larger by a factor 6 in the server model than in the serial ATM. No signals are sent in both implementation models. The server model requires more control registers, but the group “control and implicit states” is increased by less than factor 6 since the serial ATM contains more implicit states.

7.2.4 Assembly line

Figure 7.11 summarizes the resource requirements of the assembly line application example. The group “run-time components” contains no timers. The differences in the signal I/O are due to the structure of the SDL system: Two server processes receive external signals, and one outputs a signal to the environment. Therefore, the parallel ATM and the server model require the

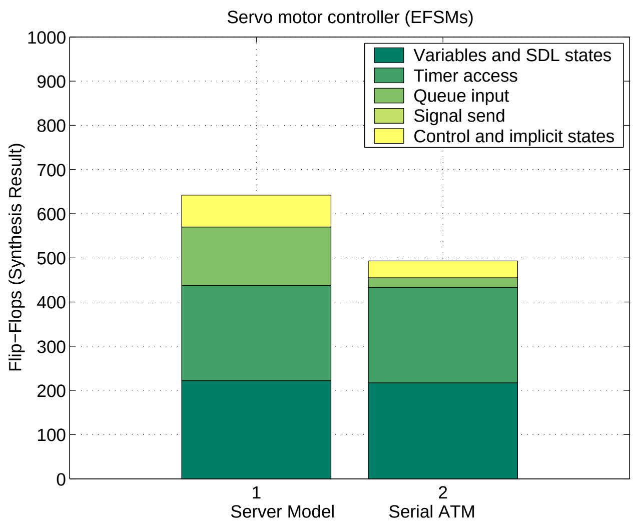


Figure 7.10: Servo controller: memory elements of the EFSM implementation

same signal I/O resources, which are reduced in the serial ATM.

The number of SDL processes in this example is equal to the number of external input signals. Therefore, the FFs needed for the message queues are the same in server model and parallel ATM, while they are lower in the serial ATM.

Here, too, the largest contribution comes from the EFSM implementation, which is shown in figure 7.12. Again, the group "variables and states" is the same in server model and serial ATM, and equals the size of the "shared data" group of the parallel ATM. The local variables of the parallel ATM's EFSMs lead to a very large "variables and state" group. There are no timer accesses. The "queue input" section is like before proportional to the number of implementation processes.

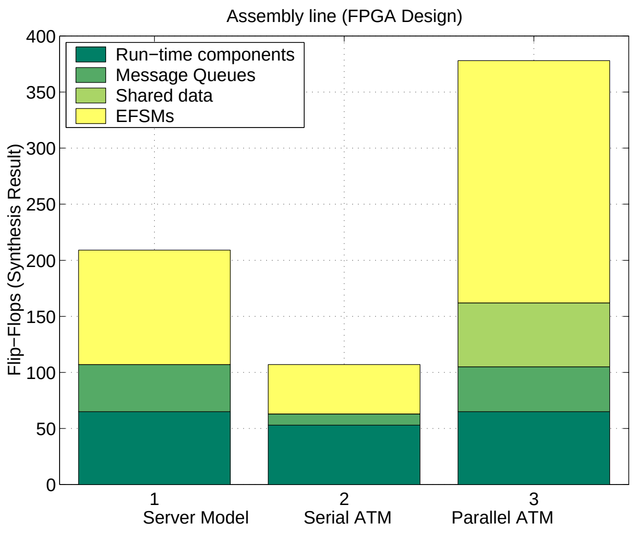


Figure 7.11: Assembly line: memory elements of the entire FPGA design

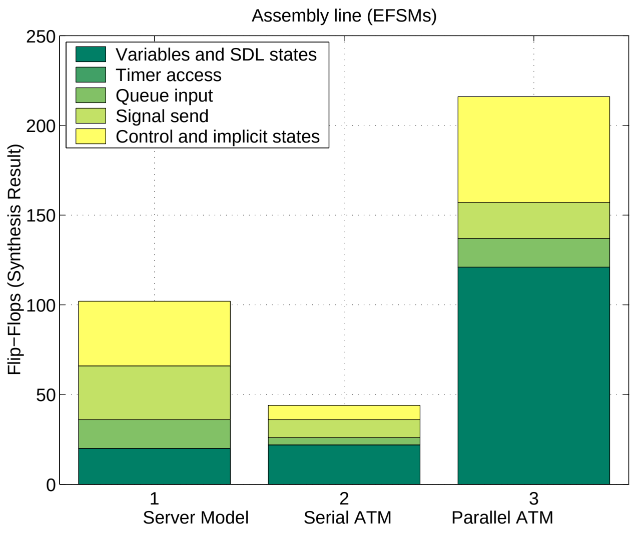


Figure 7.12: Assembly line: memory elements of the EFSM implementation

7.2.5 FPGA Resource Usage

The results shown up to now originated from synthesis and depended solely from the VHDL hardware description. The reports in this subsection come from the implementation on the target Xilinx Virtex FPGA. This means that they give an indication how many resources are actually needed, but they are at the same time technology dependent and not generally transferrable to different target architectures.

The Xilinx Virtex FPGA has a regular architecture that comprises an array of configurable logic blocks (CLBs) surrounded by programmable input/output blocks, interconnected by a hierarchy of routing resources. The CLBs like depicted in figure 7.13 are built of two identical slices, which consist of two 4-input look-up tables (LUTs) and two memory elements. The LUTs can serve as versatile function generators or as RAM. The two memory elements can be configured as either edge-triggerd D-type flip-flops or level-sensitive latches. Each slice contains additional logic for the optimized implementation of arithmetic functions, and multiplexers between the LUTs and the memory elements.

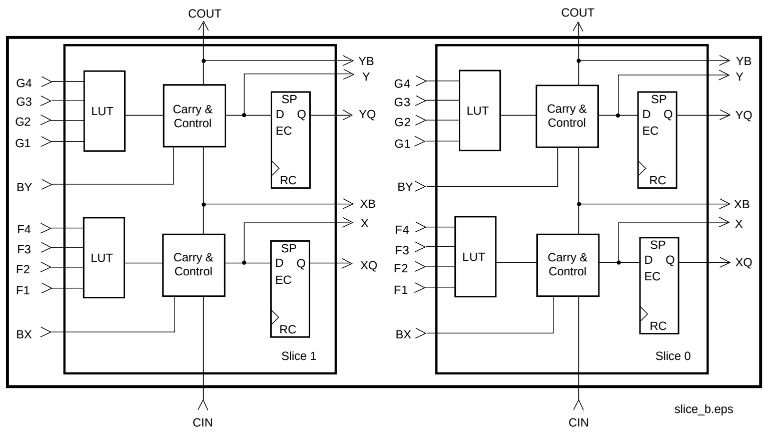


Figure 7.13: Xilinx Virtex configurable logic block consisting of two slices

The tables 7.1, 7.2 and 7.3 summarize the results from the three application examples. The columns "% of FFs/LUTs in slices" indicate how many of the theoretical two FFs/LUTs per slice were actually used, according to the number of FFs and LUTs reported by Xilinx. At first sight it is clear that not nearly 100% of the resources in the allocated slices can be used. The reason for this does not lie, as this was the case in older FPGA generations, in a high usage of CLBs for routing, which was never higher than 2%. Re-

	slices	FFs	% of FFs in slices	LUTs	% of LUTs in slices
Server model	917	653	36%	1413	77%
Serial ATM	592	314	27%	983	83%
% of server model	64%	48%		69%	
Parallel ATM	1734	1365	39%	2161	62%
% of server model	189%	209%		153%	

Table 7.1: Xilinx resource usage CAN-bus physical layer

sponsible instead is presumably the fact that the resources of a slice can not be allocated completely independently, leading to many slices where only a part of the elements are used.

It can be seen that the number of used resources is proportional to the number of memory elements, which have been analyzed in the previous subsections. Comparisons based on the memory elements are therefore valid. The number of required slices however is determined by both memory elements and look-up tables.

The rows "% of server model" indicate which role the combinational logic, multiplexers and arithmetic functions play, which are covered in the LUTs and not the memory elements. The decrease respectively increase in resource usage between the implementation models is different if one regards the overall slices, the flip-flops, or the look-up tables. The serial ATM always requires more slices than indicated by the FFs. The reason for this is the overproportional need for logic and multiplexers caused by the very complex state machine of the one EFSM, which shows an extreme deep nesting of if-else-statements. The resource usage of the parallel ATM, in contrast to this, is rather determined by the multiple storage of the process data. The state machines are generally simpler than those of the server model. Therefore, the increase of required slices is lower than indicated by the FFs.

A final comparison between the implementation models is drawn with the inclusion of the results from the real-time analysis in subsection 7.4.

	slices	FFs	% of FFs in slices	LUTs	% of LUTs in slices
Server model	1178	1108	47%	1559	66%
Serial ATM	835	701	42%	1083	65%
% of server model	71%	63%		69%	

Table 7.2: Xilinx resource usage servo motor controller

	slices	FFs	% of FFs in slices	LUTs	% of LUTs in slices
Server model	345	217	31%	538	78%
Serial ATM	217	101	23%	343	79%
% of server model	63%	47%		64%	
Parallel ATM	485	321	33%	703	72%
% of server model	141%	148%		131%	

Table 7.3: Xilinx resource usage assembly line

7.3 Real-Time Analysis Results

7.3.1 Timing Requirements and Operation Modes

The CAN bus protocol gives a very exact definition of the timing, from which event streams and event dependencies of external and some internal signals, as well as the deadlines can be derived. Figure 7.1 depicts the internal timing of one message bit, while figure 7.14 shows the composition of one CAN message frame.

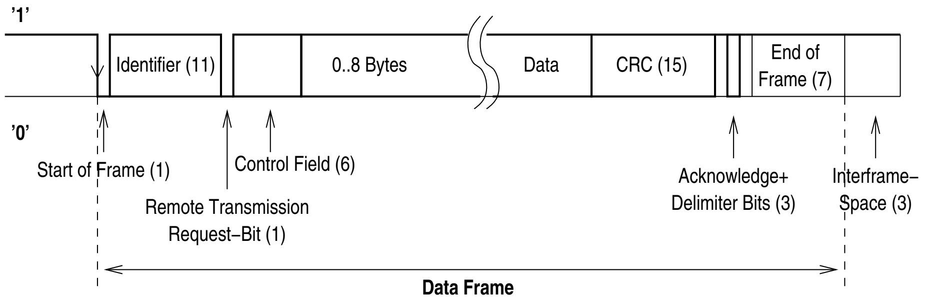


Figure 7.14: CAN message frame

In the following, the 13 external signals of the physical layer specification are listed, together with a short description and their event streams and, if applicable, deadlines. Here, P_{bit} denotes the duration of one message bit of the local physical layer, with $P_{bit} = \frac{1}{f_{CAN}}$. $P_{bit,sender}$ is the duration of one bit on the physical bus, which is determined by the actual sender of the message. When a message is received, P_{bit} and $P_{bit,sender}$ can diverge slightly due to oscillator tolerances. P_{sub} is the period of the sub-bit cycle, with $P_{bit} = 10 \cdot P_{sub}$ in the experiments.

tx is sent from the DLL and contains the next bit to be transmitted on the bus. If the process has been idle, the process Controller is notified via signal `tx_local`, which starts the processes Clock, Timing and Bitstuffing. The signal arrives every bit time, with a jitter j_{tx}.

$$ES_{tx} : \left\{ \begin{pmatrix} \infty \\ 0 \end{pmatrix}, \begin{pmatrix} P_{bit} \\ P_{bit} - j_{tx} \end{pmatrix} \right\}$$

tx_off signals from the DLL that the transmittion of the current SDL message is finished. The minimum distance is therefore the minimum length of a CAN message, i.e. 47 bits. To prevent the sending of a next bit, the triggered state change must be concluded before the next signal `controller_clock`.

$$ES_{tx\_off} : \left\{ \begin{pmatrix} 47 \cdot P_{bit} \\ 0 \end{pmatrix} \right\}$$

stuff, stuff_off mark the portion of the message where bitstuffing is performed, starting with the beginning of the message and ending after the CRC field. The event distance is the minimum length of a CAN message, with a dependency between the two signals: $ed_{stuff,stuff\_off} = 34 \cdot P_{bit}$; $ed_{stuff\_off,stuff} = 13 \cdot P_{bit}$; Both signals must be processed before the next `sample_now`.

$$ES_{stuff} = ES_{stuff\_off} : \left\{ \begin{pmatrix} 47 \cdot P_{bit} \\ 0 \end{pmatrix} \right\}$$

sleep is sent from the DLL at the end of a CAN message, and puts the physical layer in the reset state.

$$ES_{sleep} : \left\{ \begin{pmatrix} 47 \cdot P_{bit} \\ 0 \end{pmatrix} \right\}$$

rx_edge is a signal which is sent to the process Synchronization at every rising or falling edge on the CAN bus. It is used to update the timing parameters of the current bit, and therefore must be processed before the next sub-bit cycle.

$$ES_{rx\_edge} : \left\{ \begin{pmatrix} P_{bit,sender} \\ 0 \end{pmatrix} \right\}$$

rx_sync_edge is sent to the process Controller at every falling edge on the CAN bus. If the process is idle, this edge signifies that a new message is being sent on the bus, and the physical layer is activated in order to receive the message, and the DLL is notified. The start of Clock, Timing and BitStuffing must be fast enough, so that the first bit is sampled correctly.

$$ES_{rx\_sync\_edge} : \left\{ 2 \cdot \begin{pmatrix} P_{bit,sender} \\ 0 \end{pmatrix} \right\}$$

CC is the output of the timer which is started periodically by process Clock with the period P_{sub}. It comprises the internal sub-bit clock of the physical layer, which triggers the processes Timing, BitStuffing, Transmitter and Receiver.

$$ES_{CC} : \left\{ \begin{pmatrix} P_{sub} \\ 0 \end{pmatrix} \right\}$$

can_clock, sample_now are internal signals output by process Timing. They are output with a period of P_{bit}, and have an event distance of $P_1 = \text{synchronization\_segment} + \text{propagation\_segment} + \text{buffer\_segment\_1}$ respectively $P_2 = \text{buffer\_segment\_2}$ between them (see also figure 7.1). This information is additionally given for the real-time analysis.

$$ES_{can\_clock} = ES_{sample\_now} : \left\{ \begin{pmatrix} P_{bit} \\ 0 \end{pmatrix} \right\}$$

controller_period, buffer_seg_1, buffer_seg_2, signal_seg are signals for the configuration of the physical layer, which have to be sent by the DLL before the physical layer can operate.

$$ES_{configuration} : \left\{ \begin{pmatrix} \infty \\ 0 \end{pmatrix} \right\}$$

reset_pl is the central reset signal of the physical layer. It does not occur periodically. Since no actions have to be performed after reset, there is no tight deadline.

$$ES_{reset\_pl} : \left\{ \begin{pmatrix} \infty \\ 0 \end{pmatrix} \right\}$$

It greatly simplifies real-time analysis to introduce operation modes, which define mutually exlusive groups of signals which can be analyzed seperately. Table 7.4 shows three possible operation modes of the physical layer: The group **Configuration** contains signals which do not occur at all during normal operation. The group **Start/Reset** summarizes the signals which are sent at the startup of the physical layer at the beginning of the CAN message, or the reset when the message is finished. The signals in the third group **Run** are used during the transmission or reception of a message. It is useful to separate the two last groups for real-time analysis, because the entire group **Run** is switched on and off by the group **Start/Reset**. This means that no **Run**-signals can occur simultaneously to **Start**-signals, and when the **Reset**-signals appear, the **Run**-signals are stopped and their deadlines are no longer interesting.

Mode	**Configuration**	**Start/Reset**	**Run**
Signals	`reset_pl`, `controller_period`, `buffer_seg_1`, `buffer_seg_2`, `signal_seg`	`sleep`, `tx_local`, `start_clock`, `start_stuff`, `start_timing`, `reset_stuff`, `reset_clock`, `reset_sync`	`CC`, `ctrl_clock`, `sample_now`, `can_clock`, `stuff_now`, `stuff`, `stuff_off`, `tx`, `tx_off`, `rx_edge`, `rx_sync_edge`

Table 7.4: Assignment of SDL signals to operation modes

The real-time analysis is demonstrated on a reaction to the internal timer signal `CC`. The task precedence system (TPS) triggered by signal `CC` is shown in figure 7.15, slightly simplified in order to keep the figure understandable. The reaction to be observed is the output of signal `rx`, which is marked in figure 7.15 by the heavy boxes. Also marked are the parts of the TPS which influence the real-time behaviour of the reaction. The deadline for the reaction `CC`$\rightarrow$`rx` is given by the internal timing of the CAN message bit: If too much time elapses between the timer output and `rx`, the bus is sampled too late. A reasonable value for the deadline is two sub-bit cycles, i.e. $2 \cdot P_{sub}$.

As a simplification of the real-time analysis to be performed, no message, transmission or synchronization errors are considered.

7.3.2 Server Model

The real-time analysis of the server model implementation is performed in two steps. First, the SDL processes are analyzed. For each, the processing times **c**, the reaction times **r** and the output event streams are determined. The processing times **c** are obtained by summarizing the clock cycles in the VHDL implementation. After the simple scheduling applied during code generation, clock cycles are inserted at each SDL task, and at the end of the EFSM transition. To this add the clock cycles contained in the run-time components, which are given in table 7.5. Additionally, the maximum and minimum times **s** for sending the SDL signals have to be determined. In the second step, the task precedence system of the reaction to be analyzed is followed, summarizing the reaction times on the way in order to obtain the overall reaction time. In the following, only the SDL processes involved in the observed task precedence systems are regarded, concentrating on the operation mode **Run**.

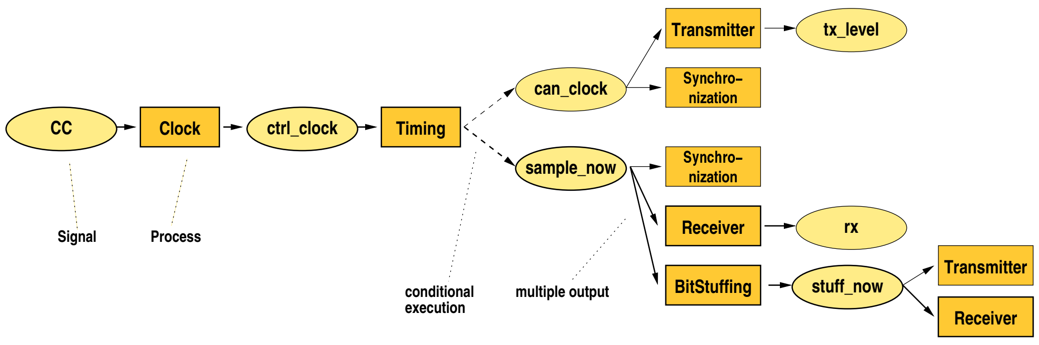


Figure 7.15: Task precedence system triggered by signal CC

Run-time system

The message queue of the utilized run-time components can not receive two SDL signals simultaneously. If it is possible that up to k messages are sent to one SDL process at the same time, the minimum sending time $\mathbf{s}$ has for the worst case to be multiplied by k, i.e. $\mathbf{s}_{max} = k \cdot \mathbf{s}$. Table 7.5 summarizes the execution times required by the used implementations of the run-time components.

Process Clock

1. Input signals in mode **Run**: `CC`
2. Execution times:

$$\begin{aligned} \mathbf{c}_{CC,send,max} &= \mathbf{d} + 2 + \mathbf{s}_{ctrl\_clock,max} = 7 \\ \mathbf{c}_{CC,send,min} &= \mathbf{c}_{CC,send,max} \\ \mathbf{c}_{CC \to ctrl\_clock} &= \mathbf{d} = 3 \end{aligned}$$

3. Reaction time:

- $ES_{CC} : \left\{ \binom{P_{sub}}{0} \right\}$
- In a periodic system, the utilization must be below or equal 100%:

$$\mathbf{c}_{CC,max} \leq P_{sub} \tag{7.13}$$

- If equation (7.13) holds, the maximum reaction time can be found at $I = 0$, like described in equation (5.5):

$$\begin{aligned} \mathbf{r}_{CC,max} &= \mathbf{q} + \mathbf{w} + \mathbf{c} \\ &= \mathbf{q} + C(0) - \mathbf{c}_{CC} + \mathbf{c}_{CC} \\ &= \mathbf{q} + \mathbf{c}_{CC,max} = 9 \end{aligned} \tag{7.14}$$

	clock cycles	simultaneous signals
$\mathbf{s}_{ctrl\_clock}$	$\mathbf{s}$	—
$\mathbf{s}_{can\_clock,tsm}$	$3 \cdot \mathbf{s}$	`tx`, `tx_off`
$\mathbf{s}_{stuff\_now,tsm}$	$3 \cdot \mathbf{s}$	`tx`, `tx_off`
$\mathbf{s}_{sample\_now\_sync,max}$	$2 \cdot \mathbf{s}$	`rx_edge`
$\mathbf{s}_{sample\_now\_rcv,max}$	$2 \cdot \mathbf{s}$	`stuff_now`
$\mathbf{s}_{stuff_now_rcv}$	$2 \cdot \mathbf{s}$	`sample_now`
$\mathbf{s}_{sample\_now\_stuff,max}$	$2 \cdot \mathbf{s}$	`stuff`
$\mathbf{s}_{rx,max}$	$\mathbf{s}$	—
$\mathbf{s}$	2	
$\mathbf{q}$	2	
$\mathbf{d}$	3	

Table 7.5: Execution times required by run-time components

and

$$\mathbf{r}_{CC \rightarrow ctrl\_clock} = \mathbf{q} + \mathbf{c}_{CC \rightarrow ctrl\_clock} = 5; \tag{7.15}$$

4. Output event stream: Since $\mathbf{w} = 0$ and $\mathbf{c}_{CC,send,min} = \mathbf{c}_{CC,send,max}$, equation (5.7) yields $J = 0$. Therefore,

$$ES_{ctrl\_clock} = ES_{CC} : \left\{ \begin{pmatrix} P_{sub} \\ 0 \end{pmatrix} \right\} \tag{7.16}$$

Process Timing

1. Input signals in mode **Run**: `ctrl_clock`

2. Execution times:

$$\mathbf{c}_{ctrl\_clock,send,max} = \mathbf{d} + 2 + \mathbf{s}_{sample\_now\_sync,max} + \mathbf{s}_{sample\_now\_rcv,max}$$
$$+ \mathbf{s}_{sample\_now\_stuff,max} = 3 + 2 + 4 + 4 + 4 = 17$$
$$\mathbf{c}_{ctrl\_clock \rightarrow sample\_now\_rcv} = \mathbf{d} + 1 = 4$$
$$\mathbf{c}_{ctrl\_clock \rightarrow sample\_now\_sync} = \mathbf{d} + 1 + \mathbf{s}_{sample\_now\_rcv,max} = 8$$
$$\mathbf{c}_{ctrl\_clock \rightarrow sample\_now\_stuff} = \mathbf{d} + 1 + \mathbf{s}_{sample\_now\_sync,max} + \mathbf{s}_{sample\_now\_rcv,max} = 12$$

3. Reaction time:

   - $ES_{ctrl\_clock} : \left\{ \binom{P_{sub}}{0} \right\}$

- In a periodic system, the utilization must be below or equal 100%:

$$\mathbf{c}_{ctrl\_clock,max} \leq P_{sub} \tag{7.17}$$

- Again, the maximum reaction time can be found at $I = 0$, like described in equation (5.5):

$$\begin{aligned} \mathbf{w} &= 0 \\ \mathbf{r}_{ctrl\_clock,max} &= \mathbf{q} + \mathbf{c}_{ctrl\_clock} = 19 \end{aligned} \tag{7.18}$$

  and

$$\mathbf{r}_{ctrl\_clock \to sample\_now\_rcv} = \mathbf{q} + \mathbf{c}_{ctrl\_clock \to sample\_now\_rcv} = 2 + 4 = 6; \tag{7.19}$$

4. Output event stream: The additional information from section 7.3.1 is used here.

$$\begin{aligned} ES_{sample\_now} &= ES_{can\_clock} : \left\{ \begin{pmatrix} P_{bit} \\ 0 \end{pmatrix} \right\} \\ ed_{can\_clock,sample\_now} &= P_1 \\ ed_{sample\_now,can\_clock} &= P_2 \end{aligned} \tag{7.20}$$

Process BitStuffing

1. Input signals in mode **Run**: `sample_now`, `stuff`, `stuff_off`
2. Execution times:

$$\mathbf{c}_{stuff} = \mathbf{c}_{stuff\_off} = \mathbf{d} + 2$$
$$\mathbf{c}_{sample\_now,send,max} = \mathbf{d} + 2 + \mathbf{s}_{stuff\_now\_rcv} + \mathbf{s}_{stuff\_now\_tsm} = 3 + 2 + 4 + 6 = 15$$
$$\mathbf{c}_{sample\_now,send,min} = \mathbf{d} + 2 + 2 \cdot \mathbf{s} = 9$$
$$\mathbf{c}_{sample\_now \to stuff\_now\_rcv} = \mathbf{d} + 1 = 4$$
$$\mathbf{c}_{sample\_now \to stuff\_now\_tsm} = \mathbf{d} + 1 + \mathbf{s}_{stuff\_now\_rcv} = 8$$

3. Reaction time:

- Event streams and dependencies:

$$\begin{aligned} &ES_{stuff} = ES_{stuff\_off} : \left\{ \begin{pmatrix} 47 \cdot P_{bit} \\ 0 \end{pmatrix} \right\} \\ &ed_{stuff,stuff\_off} = 34 \cdot P_{bit}; \quad ed_{stuff\_off,stuff} = 13 \cdot P_{bit}; \\ &ES_{sample\_now} : \left\{ \begin{pmatrix} P_{bit} \\ 0 \end{pmatrix} \right\} \end{aligned} \tag{7.21}$$

- In a periodic system, the utilization must be below or equal 100%:

$$\begin{aligned}\frac{\mathbf{c}_{sample\_now,max}}{P_{bit}} + \frac{\mathbf{c}_{stuff} + \mathbf{c}_{stuff\_off}}{47 \cdot P_{bit}} &\leq 1 \\ \Rightarrow \mathbf{c}_{sample\_now,max} + \frac{\mathbf{c}_{stuff} + \mathbf{c}_{stuff\_off}}{47} &\leq P_{bit}\end{aligned} \tag{7.22}$$

- Event dependency:

$$\mathbf{c}_{stuff} = \mathbf{c}_{stuff\_off} << 13 \cdot P_{bit}$$

  $\Rightarrow$ only one signal of both has to be considered

- $I = 0$:

$$C(0) = \mathbf{c}_{sample\_now} + \mathbf{c}_{stuff}$$

- $I > 0$: Since with equation (7.22) $\mathbf{c}_{sample\_now} < P_{bit}$ and $\mathbf{c}_{stuff} < 47 \cdot P_{bit}$, no larger $C(I) - I$ can be reached than at $I = 0$
- Maximum reaction time therefore:

$$\mathbf{r}_{sample\_now,max} = \mathbf{r}_{stuff} = \mathbf{r}_{stuff\_off} = \mathbf{q} + \mathbf{c}_{sample\_now} + \mathbf{c}_{stuff} \tag{7.23}$$

  and

$$\begin{aligned}\mathbf{r}_{sample\_now \rightarrow stuff\_now\_rcv} &= \mathbf{q} + \mathbf{c}_{stuff} + \mathbf{c}_{sample\_now \rightarrow stuff\_now\_rcv} \\ \mathbf{r}_{sample\_now \rightarrow stuff\_now\_tsm} &= \mathbf{q} + \mathbf{c}_{stuff} + \mathbf{c}_{sample\_now \rightarrow stuff\_now\_tsm}\end{aligned} \tag{7.24}$$

4. Output event stream: With equation (5.7) yields $J = \mathbf{c}_{stuff} + \mathbf{c}_{CC,send,max} - \mathbf{c}_{CC,send,min}$. Therefore,

$$ES_{stuff\_now} : \left\{ \begin{pmatrix} \infty \\ 0 \end{pmatrix} \begin{pmatrix} P_{bit} \\ P_{bit} - \mathbf{c}_{stuff} - 6 \end{pmatrix} \right\} \tag{7.25}$$

Process Receiver

1. Input signals in mode **Run**: `sample_now`, `stuff_now`

2. Execution times:

$$\begin{aligned}\mathbf{c}_{stuff\_now} &= \mathbf{d}+1=4\\ \mathbf{c}_{sample\_now,send,max} &= \mathbf{d}+2+\mathbf{s}_{rx,max}=7\\ \mathbf{c}_{sample\_now,send,min} &= \mathbf{c}_{sample\_now,send,max}\\ \mathbf{c}_{sample\_now\rightarrow rx} &= \mathbf{d}+1=4\end{aligned}$$

3. Reaction time:

   - Input event streams:

$$\begin{aligned}ES_{sample\_now} = ES_{can\_clock} &: \left\{\begin{pmatrix}P_{bit}\\0\end{pmatrix}\right\}\\ ES_{stuff\_now} &: \left\{\begin{pmatrix}\infty\\0\end{pmatrix}\begin{pmatrix}P_{bit}\\P_{bit}-\mathbf{c}_{stuff}-6\end{pmatrix}\right\}\end{aligned} \qquad (7.26)$$

   - Utilization condition:

$$\mathbf{c}_{sample\_now}+\mathbf{c}_{stuff\_now}\le P_{bit} \qquad (7.27)$$

   - Event dependencies: Signal `stuff_now` is triggered by signal `sample_now_stuff`, which is output in the same transition as signal `sample_now_rcv`. With equation (5.13) and (5.11) (propagation of event dependencies between SDL processes, event dependencies between signals from one SDL process), therefore, an event dependency can be derived:

$$\begin{aligned}ed_{sample\_now,stuff\_now} &= \mathbf{r}_{sample\_now\rightarrow stuff\_now\_rcv,min}=6\\ ed_{sample\_now\_stuff,sample\_now} &= \mathbf{c}_{a,min}-\mathbf{c}_{a\rightarrow c,max}-\mathbf{s}_c+\mathbf{c}_{a\rightarrow b,min}+\mathbf{s}_b\\ &= 15\\ ed_{stuff\_now,sample\_now} &= ed_{sample\_now\_stuff,sample\_now}\\ &\quad -\mathbf{r}_{sample\_now,min}+\mathbf{r}_{stuff\_now,max}=6\end{aligned} \qquad (7.28)$$

   - Investigating signal `sample_now` at $I=0$: $C(0)=\mathbf{c}_{sample\_now}$; `stuff_now` has no influence because $ed>\mathbf{c}_{stuff\_now}$; no $I>0$ to be considered because $\mathbf{c}_{sample\_now}<P_{bit}$ postulated.

- Reaction time of signal `sample_now`:

$$\begin{aligned}\mathbf{r}_{sample\_now,max} &= \mathbf{q} + \mathbf{w} + \mathbf{c} \\ &= \mathbf{q} + \mathbf{c}_{sample\_now,max} = 9\end{aligned} \tag{7.29}$$

  and

$$\mathbf{r}_{sample\_now \to rx} = \mathbf{q} + \mathbf{c}_{sample\_now \to rx} = 6; \tag{7.30}$$

4. Output event stream: Since $\mathbf{w} = 0$ and $\mathbf{c}_{sample\_now,send,min} = \mathbf{c}_{sample\_now,send,max}$, equation (5.7) yields $J = 0$. Therefore,

$$ES_{rx} = ES_{sample\_now} : \left\{ \begin{pmatrix} P_{bit} \\ 0 \end{pmatrix} \right\} \tag{7.31}$$

Overall Reaction Time

As can be seen in figure 7.15, the reaction `CC→rx` to be investigated involves the processes Clock, Timing and Receiver, with the intermediate signals `ctrl_clock`, `sample_now`. In order to obtain the overall reaction time $\mathbf{r}_{CC \to rx}$, the reaction times of the involved signals have to be added:

$$\begin{aligned}\mathbf{r}_{CC \to rx} &= \mathbf{r}_{CC \to ctrl\_clock} + \mathbf{r}_{ctrl\_clock \to sample\_now} \\ &\quad + \mathbf{r}_{sample\_now \to rx} + \mathbf{s}_{rx} \\ &= 5 + 6 + 6 + 2 = 19 \quad \text{cycles}\end{aligned} \tag{7.32}$$

With the clock period of 30 ns used in the experimental setup, the reaction time which should be shorter than the given deadline therefore is:

$$\mathbf{r}_{CC \to rx} = 0.57\ \mu\text{s} \leq 2 \cdot P_{sub} \tag{7.33}$$

With

$$P_{sub} = \frac{1}{10} \cdot \frac{1}{f_{CAN}} \tag{7.34}$$

this results in a upper bound on the possible CAN bus frequency:

$$f_{CAN} \leq \frac{2}{10 \cdot \mathbf{r}_{CC \to rx}} = 351 \frac{\text{kbit}}{s} \tag{7.35}$$

7.3.3 Serial Activity Thread Model

Execution Times

Like before, the execution times **c** are obtained by summarizing the clock cycles in the VHDL implementation. A more realistic analysis is obtained, when the operation modes are considered also at this step. In that case, branches in the EFSM that are triggered by internal signals not contained in the observed mode are not summarized in the execution time. The execution times of the serial ATM implementation for operation mode **Run** are shown in the following:

$$\begin{gathered}
\texttt{stuff:}\ \mathbf{c}_{stuff} = \mathbf{d} + 2 \\
\texttt{stuff\_off:}\ \mathbf{c}_{stuff\_off} = \mathbf{d} + 2 \\
\texttt{tx:}\ \mathbf{c}_{tx} = \mathbf{d} + 1 \\
\texttt{tx\_off:}\ \mathbf{c}_{tx\_off} = \mathbf{d} + 1 \\
\texttt{rx\_edge:}\ \mathbf{c}_{rx\_edge} = \mathbf{d} + 3 \\
\texttt{rx\_sync\_edge:}\ \mathbf{c}_{rx\_sync\_edge} = \mathbf{d} + 1 \\
\texttt{CC:}\ \mathbf{c}_{CC} = \mathbf{d} + 7 + \mathbf{s}_{rx,max} \\
\mathbf{c}_{CC\rightarrow rx} = \mathbf{d} + 3
\end{gathered}$$

Overall Reaction Time

Between the signals `stuff` and `stuff_off`, as well as between `tx` and `tx_off`, there are the given event dependencies with $ed > \mathbf{c}$. Therefore, only one signal of each pair has to be considered at a time.

The requested computation at $I = 0$ then results in:

$$C(0) = \mathbf{c}_{stuff} + \mathbf{c}_{tx} + \mathbf{c}_{rx\_edge} + \mathbf{c}_{rx\_sync\_edge} + \mathbf{c}_{CC} = 31 \tag{7.36}$$

The worst case waiting time of signal `CC` based on $C(0)$ then is

$$\mathbf{w}_{CC} = C(0) - \mathbf{c}_{CC} = 19; \tag{7.37}$$

with the maximum reaction time

$$\begin{aligned}
\mathbf{r}_{CC\rightarrow rx} &= \mathbf{q} + \mathbf{w}_{CC} + \mathbf{c}_{CC\rightarrow rx} + \mathbf{s}_{rx} \\
&= 2 + 19 + 3 + 3 + 2 = 29;
\end{aligned} \tag{7.38}$$

Now, it has to be shown that this is in fact the worst case. Like before, the deadline for the reaction `CC`→`rx` is given as $2 \cdot P_{sub}$. The minimum sub-bit period P_{sub} then is:

$$\mathbf{r}_{max} \leq 2 \cdot P_{sub} \Rightarrow P_{sub} \geq \frac{\mathbf{r}_{max}}{2} = 14.5 \tag{7.39}$$

Figure 7.16 shows the plot of $C(I) - I$ of the serial ATM implementation for $P_{sub} = 14.5\,cycles$. It can be clearly seen that the maximum value of $C(I) - I$ appears at $I = 0$, like is to be expected in a periodical system.

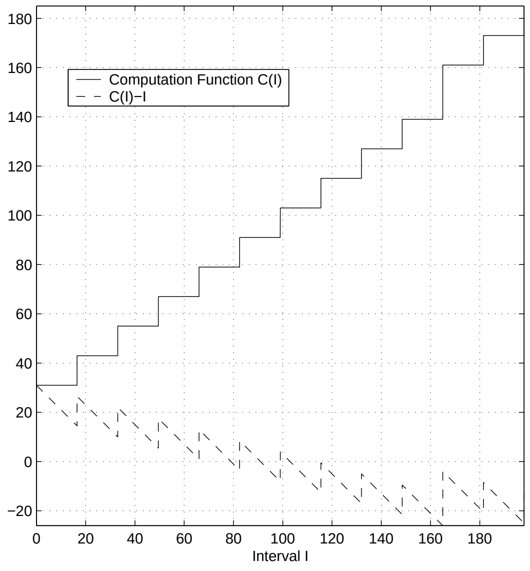


Figure 7.16: $C(I) - I$ for $P_{bit} = 14.5\,cycles$

The obtained minimum value for P_{sub} translates to a maximum CAN bus frequency of:

$$f_{CAN} = \frac{1}{10 \cdot P_{sub}} = \frac{1}{10 \cdot 14.5 \cdot 30 \text{ ns}} = 230\frac{\text{kbit}}{s} \tag{7.40}$$

7.3.4 Parallel Activity Thread Model

The reaction time of the parallel activity thread model consists of the execution time in the activity thread plus additional waiting time at the shared data components. In a first step, therefore, the blocking times caused by the transitions of the activity threads are determined. The access to the shared data in the run-time components used in the experiments is granted based on priorities. For the computation of the waiting time at the shared data component, the event streams and dependencies of the thread with higher priority, and the highest possible blocking time through a lower priority activity thread, need to be known.

		clock cycles
lock	$\mathbf{l}$	2
unlock	$\mathbf{u}$	1
queue length 0:	$\mathbf{q}_0$	2
	$\mathbf{d}_0$	0

Table 7.6: Execution times required by run-time components

In addition to table 7.5, the execution times of the run-time components of the parallel activity thread model given in table 7.6 are required. Again, only signals and activity threads of the observed operation mode **Run** need to be regarded.

Blocking Times

Activity Thread CC

$$\mathbf{b}_{CC,Clock} = \mathbf{l} + \mathbf{u} + 1 = 4$$
$$\mathbf{b}_{CC,Timing} = \mathbf{l} + \mathbf{u} + 1 = 4$$
$$\mathbf{b}_{CC,Bitstuffing} = \mathbf{l} + \mathbf{u} + 1 = 4$$
$$\mathbf{b}_{CC,Receiver} = \mathbf{l} + \mathbf{u} + 1 = 4$$
$$\mathbf{b}_{CC,Transmitter} = \mathbf{l} + \mathbf{u} + 1 = 4$$
$$\mathbf{b}_{CC,Synchronization} = \mathbf{l} + \mathbf{u} = 4$$

Activity Thread stuff

$$\mathbf{b}_{stuff,Bitstuffing} = \mathbf{l} + \mathbf{u} + 1 = 4$$

Activity Thread stuff_off

$$\mathbf{b}_{stuff\_off,Bitstuffing} = \mathbf{l} + \mathbf{u} + 1 = 4$$

Activity Thread tx

$$\mathbf{b}_{tx,Transmitter} = \mathbf{l} + \mathbf{u} = 3$$

Activity Thread tx_off

$$\mathbf{b}_{tx\_off,Transmitter} = \mathbf{l} + \mathbf{u} = 3$$

Activity Thread rx_edge

$$\mathbf{b}_{rx\_edge,Synchronization} = \mathbf{l} + \mathbf{u} + 2 = 5$$

Activity Thread rx_sync_edge

$$\mathbf{b}_{rx\_sync\_edge,Synchronization} = \mathbf{l} + \mathbf{u} = 3$$

Waiting time in activity thread CC

The reaction `CC`→`rx` to be investigated is part of the activity thread `CC`, which has been given the highest priority in the implementation. The reaction involves the processes Clock, Timing and Receiver, with the intermediate signals `ctrl_clock`, `sample_now`. First, the maximum waiting times at these processes has to be determined.

Process Clock Input signals in mode **Run**: `CC` ⇒ no blocking by other ATs

Process Timing Input signals in mode **Run**: `ctrl_clock` ⇒ no blocking by other ATs

Process Receiver Input signals in mode **Run**: `sample_now`, `stuff_now` are part of the same AT ⇒ no blocking by other ATs

Processes Synchronization, Bitstuffing are involved after the sending of signal `rx` and therefore do not contribute to $\mathbf{r}_{CC\rightarrow rx}$.

Overall Reaction Time

$$\begin{aligned}\mathbf{r}_{CC\rightarrow rx} &= \mathbf{q} + \mathbf{d} + \mathbf{b}_{CC,Clock} + \mathbf{b}_{CC,Timing} + \mathbf{b}_{CC,Receiver} + \mathbf{s}_{rx}\\ &= 2 + 4 + 4 + 4 + 2 = 16\end{aligned}$$

$$\mathbf{r}_{max} \leq 2 \cdot P_{sub} \Rightarrow P_{sub} \geq \frac{\mathbf{r}_{max}}{2} = 8 \tag{7.41}$$

This corresponds to a maximum CAN bus frequency of:

$$f_{CAN} = \frac{1}{10 \cdot P_{sub}} = \frac{1}{10 \cdot 8 \cdot 30 \text{ ns}} = 416 \frac{\text{kbit}}{s} \tag{7.42}$$

7.4 Evaluation

The application examples presented in this chapter have been specified in SDL and implemented on the rapid prototyping target architecture using the automated design process presented in chapter 6. The implementations have been tested and found to work properly in their respective environments, i.e. CAN-bus network, servo motors and software interface. This shows that the automated generation of application specific hardware from SDL is feasible.

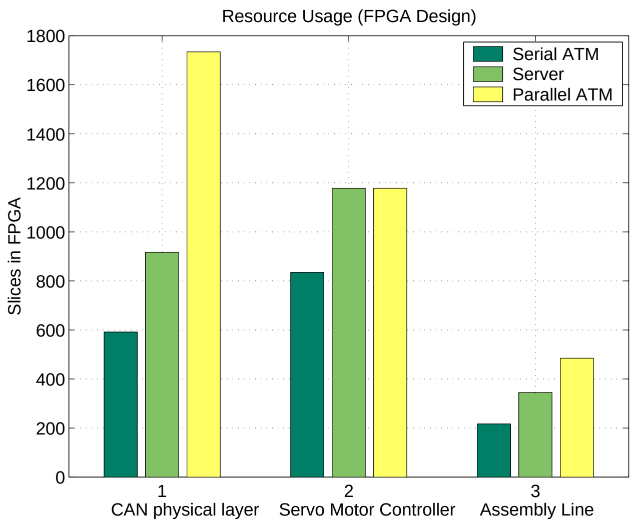


Figure 7.17: Resource usage overview of the application examples

The results from the application examples can be used for a comparison of the three implementation models in terms of required hardware resources and guaranteeable reaction time. Figure 7.17 summarizes the synthesis results already discussed in detail in section 7.2. In all three examples the serial activity thread model has the lowest resource usage, followed by the server model. The parallel activity thread model requires more FPGA resources than the server model, except for the servo motor controller example, where parallel ATM and server model are identical due to the structure of the SDL specification. Figure 7.18 shows the results of the real-time analysis carried out in section 7.3. Here, the order is reversed with the parallel ATM guaranteeing the shortest time for the investigated reaction `CC` $\rightarrow$ `rx`, and the serial ATM the longest.

Obviously, the results gained with the examples, absolute numbers as well as the comparative values, highly depend on many individual features, which can lead to different results. Among those influencing factors are the type of application, the manner of SDL specification, the used VHDL code generator

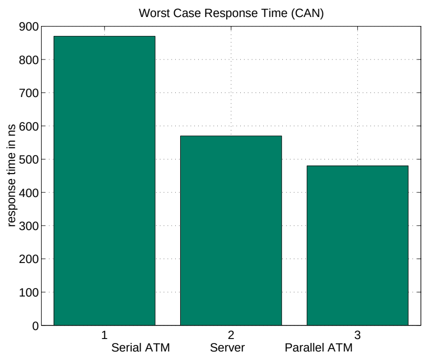


Figure 7.18: Maximum response time for the CAN bus example

and the used run-time components. What the detailed analysis of resource usage and the run-time analysis in this chapter allow, however, are a number of observations on the trade-offs between the implementation models which are based on their fundamental qualities:

1. The server model has the advantage of a parallel execution at SDL process level. Particularly if a task precedence system branches, i.e. when during one transition several signals are sent to different SDL processes, the server model allows a parallel execution. Both variants of the activity thread model execute different branches of one activity thread sequentially. This obviously has an effect on the reaction time. A second effect particular to the activity thread model is the multiple implementation of functionality of one transition, which is triggered in different activity threads. This can lead to an increased resource usage.

2. Parallel execution units which have to interact with each other always increase the hardware resource requirements, because data has to be stored several times. In the server model, in the form of SDL signals, which are stored multiply along the signal flow in the task precedence system: In the message queue, in the EFSM input register, possibly in a SDL variable and finally in the send register for the next internal signal. In a dual manner, the parallel activity thread model centralizes the signal flow, but distributes the local data, i.e. states and variables of the SDL processes. These are stored in the shared data component, and as a local copy in all involved activity threads.

3. The distribution of data storage described in (2.) not only requires hardware resources, but also leads to an overhead in time, since data has to be transferred during execution. In both the server and the parallel activity thread model this happens at identical points of the

SDL specification, always when a SDL signal is sent. In the server implementation, this takes time due to the message transferral between signal channel, queue, and EFSM. In the parallel ATM, the activity thread must obtain the shared data, and write it back after the transition. Since generally the sending of a signal requires more time than to obtain and release the process data, the parallel activity thread model has a temporal advantage, depending on the implementation of the run-time components. Further improvements in the real-time analysis result from the facile implementation of a priority based access in the shared data component.

4. The serial activity thread model stores both the process data and signals only once locally, and internal signals are abolished. Due to the serialization there are no access conflicts. The resource and time effort described in (2.) and (3.) are therefore not relevant here, which leads to minimum resource usage and execution time. However, the serial execution also means that all external signals share one execution unit and the possible parallelity in hardware is unused. This can lead to waiting times, depending on the event streams and dependencies, and therefore to longer worst case reaction times.

Chapter 8

Conclusions and Future Work

The aim of this work is to provide a framework for and to evaluate the automated generation of application specific hardware from SDL specifications, particularly for hard real-time systems, which require the possibility of a before-hand worst case real-time analysis. A useful application of such an environment is rapid prototyping, in which it was integrated and tested with real-world examples.

The presented design method starts with a specification in SDL, extended by annotations, which capture the real-time requirements of the system under development. Event streams and event dependencies are used to describe the temporal behaviour of the system's environment in form of minimum distances between external events. End-to-end deadlines give the maximum time a reaction to an event is allowed to take. In the described automated rapid-prototyping design process, the CASE tool SDT is used for specification and simulation of the SDL model.

The next step towards an electronic circuit realizing the behaviour specified in SDL is a VHDL description of this behaviour. Here, different basic principles for the transformation of SDL into VHDL, the implementation models, are presented. The server model maps each SDL process to its own VHDL entity (implementation process) with its own message queue and asynchronous communication. The activity thread implementation in contrast executes all transitions, which are triggered in the SDL processes by one external signal or timer output, directly one after the other, abolishing the internal communication between the processes. The serial activity thread model joins all of these activity threads in one implementation process with one message queue. After the parallel activity thread model, each activity thread has its own entity. In this case, the local variables and states of the SDL processes must be made accessible to the involved activity threads in a synchronized manner.

In all implementation models it is possible to differentiate between parts of the implementation, which are new every time depending on the specification, and parts which occur in similar fashion every time, like message queues, timers and signal channels. For these reusable system parts, the run-time components are provided. In the rapid-prototyping environment, the VHDL model is generated automatically from SDL using the SDL-Compiler. In the generated code, the interfaces to the run-time components have the form of macro-statements. In a later "link"-step, these are replaced by the necessary access protocols, and the components of the design are integrated. Next, commercial synthesis, place and route tools create the FPGA design for the target architecture.

To be able to cover the worst case in finite analysis time, the real-time analysis takes an abstract view of the implemented system: It observes implementation processes (server processes and activity threads), and their minimum and maximum execution times, which are derived from the VHDL description. Additionally, it uses the SDL systems' task precedence graph, which contains the information which signal is triggered in which transition. The functionality of the specification, i.e. which transition is executed when, can not be included. This proceeding can in some cases lead to overly pessimistic results, which can be improved by the inclusion of event dependencies, of additional system information and of operation modes in the analysis.

The main aim of the real-time analysis is the determination of the worst case reaction time to an event. For this, the maximum waiting time in the message queue of each implementation process must be known. In the used first-come-first-serve execution scheme, the waiting time results from the unfinished work which is still in the queue at the time of arrival of the observed event and which can be computed for the worst case from the event streams and execution times of the signals. If the given event dependencies are to be considered, it is necessary to find at first the sequence of events, which leads to the maximum waiting time in the queue. This is done with a branch-and-bound search algorithm, which due to the break-off rule needs not to be followed too deeply, and gives an on-line search result. For the analysis of an entire SDL system, the event streams of internal signals are also required. For FCFS, a graphical and an algorithmical procedure is presented, which uses the maximum and minimum execution time and the waiting time of the implementation process to determine the output event stream. Next to the event stream, it is also possible to derive event dependencies of internal signals, which result on one hand from a signal origin from the same SDL process, or on the other hand from the propagation of event dependencies between input signals.

The methods described so far are applicable to all implementation models;

for the parallel activity thread model, however, it is additionally necessary to determine the waiting times at the shared data components, which have to be added to the execution time of the implementation process. For this, the access policy to the shared data (FCFS or priority based) and the event dependencies are considered, as well as the fact that one activity thread can access only one shared data component at the same time.

In some cases cyclic dependencies must be resolved before the maximum reaction times of all implementation processes are known. Then, the maximum overall time for a reaction to an external signal which can span across several implementation processes can be calculated. Here, in contrast to the computation of the maximum waiting time, not the complete reaction of each process to the triggering signal must be considered, but instead only up to the output of the next signal in the observed task precedence system.

A second aim of the real-time analysis is to find the maximum necessary message queue length. For this, a departure function is defined which depends on the already known unfinished work in the message queue. Together with the event streams, it is used to determine the maximum filling level of the queue, for independent and for dependent input signals.

The presented methods have been tested with the help of application examples in the rapid prototyping environment. The three examples, the CAN bus physical layer, the servo motor controller and the assembly line example, were specified in SDL. With the automated design process, VHDL code was generated and the components were integrated into a complete VHDL design. The examples were executed on the target architecture, and tested in the real environment. An analysis of the synthesis result reveales that it is possible to make a reasonable estimate of the memory element resources required by the run-time components and the generated implementation processes based on a few parameters of the implementation, like signal and variable width, number of input and output channels and number of states. For the three application examples, the examination of the usage of the memory elements illustrates the differences between the implementation models and the different reasons for high resource requirements in different application types. The technology dependent resource usage finally reported by the place and route tool confirms the first estimates based on the memory elements.

The CAN bus example, which has stringent real-time requirements, is used to demonstrate the real-time analysis, using one of the task precedence systems of the physical layer. First, the event streams, event dependencies and deadlines are determined. In a second step, operation modes are defined, which prove to be very useful during the analysis. The worst case reaction time of the observed signal is calculated for all three implementation models.

The analysis of the resource usage shows that in all implementation models a certain overhead is caused by SDL's model of computation and communication, and also by the automated code generation. One main reason is the effect, that automatically a large number of intermediate storage places is generated, among others queue, queue input of the EFSM, variables, timer values and signals to send. In those registers, often the same value is stored severalfold, which is an effect that would not occur in a hand implementation without the use of SDL. An optimized code generator could improve this drawback only partly. In a comparison of the different implementation models, the serial activity thread model shows the lowest resource requirements, since all variables and signals are stored only once locally. The parallel activity thread shows a high resource usage, particularly if there are many activity threads and process variables.

Considering the achievable response time of the generated hardware, the above said applies analogously: The transferral of data between the multiple storage places requires time, which causes the response times to be relatively high compared to a possible hand implementation. A number of clock cycles could be saved with improved run-time components and code generation, but a certain overhead is bound to remain. Comparing the implementation models, the parallel activity thread model shows a temporal advantage over the server model. Although the serial activity thread model has shorter execution times, the serialization can lead to higher worst-case waiting times, which result in longer reaction times.

Concluding, the results from the implementation show that the automated generation of hardware from SDL is feasible and well integratable in a HW/SW rapid prototyping environment. The presented method shows a number of advantages: high-level specification and simulation, automated code generation, integration with software, good suitability for free partitioning and implementation on distributed systems. If these advantages balance the described overhead in resource usage and response time, depends on the application. In a rapid prototyping environment, this will often be the case.

One possibility for improvement of the current design environment has already been mentioned, and lies in an optimization of the code generator and the run-time components in terms of resource usage and execution times, in order to investigate what the actual minimum requirements for SDL implementations are.

The so far realized design process has put the focus on the functional code generation. The next development could be a user-supported, automated communication refinement and component integration step. This would also facilitate the combination of different implementation models in one system, which like the results indicate could be advantageous.

Further development is possible in the field of the real-time analysis.

The algorithms developed in this work can and should be automated and integrated with the design environment. The first-come-first-serve execution scheme is disadvantageous from a worst-case point of view. A priority-based scheme, like already partly implemented in the parallel activity thread model, does not collide with SDL's semantics. It could be implemented in the server model with the help of several parallel message queues, which are accessed by the EFSM in a fixed priority order. This changed execution mode of course would have to be integrated in the real-time analysis.

Finally, the communication protocol of the signal channels, which depends on the used run-time components, could also be investigated during real-time analysis. It could utilize the results from the already presented analysis of the implementation processes, e.g. internal event streams and dependencies. It could in turn deliver more exact bounds on the sending time of signals than the ones used in this work. The aim would be an integrated real-time analysis of signal channels and implementation processes.

Bibliography

[ABG$^+$99] Clarisse Adida, Michel Boubal, Xavier Granger, Philippe Lamaty, and Jean-Pierre Moreau. Hardware-software codesign of an image processing unit. *integrated system design*, 1999.

[BCG$^+$97] 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.

[BR98] Oliver Bringmann and Wolfgang Rosenstiel. Cross-level hierarchical high-level synthesis. In *Proceedings of Design, Automation, and Test in Europe (D.A.T.E.)*, Paris, France, 1998.

[BRM$^+$99] Oliver Bringmann, Wolfgang Rosenstiel, Annette Muth, Georg Färber, Frank Slomka, and Richard Hofmann. Mixed abstraction level hardware synthesis from SDL for rapid prototyping. In *Proceedings of the 10th IEEE International Workshop on Rapid Systems Prototyping (RSP'99)*, pages 114–119, Clearwater, Florida, USA, June 16–18 1999. IEEE Computer Society Press.

[Cad99a] Cadence Design Systems, Inc. *Cadence Virtual Component Co-Design (VCC) Environment*, 1999.

[Cad99b] Cadence Design Systems, Inc. *Cadence Virtual Component Co-Design (VCC) Links to Implementation*, 1999.

[COH$^+$99] Pai Chou, Ross Ortega, Ken Hines, Kurt Partridge, and Gaetano 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.

[CS98] G. Carle and J. Schiller. Semi–automated Design of High Performance Communication Subsystems. In *31st IEEE Hawaii International Conference on System Sciences, HICSS 98*, Kona, 1998.

[CSMJ00] W. O. Cesario, Z. Sugar, I. Moussa, and A. A. Jerraya. Efficient integration of behavioral synthesis with existing design flows. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS'2000)*, Madrid,Spain, September 20–22 2000. IEEE Computer Society Press.

[CT97] G. Csopaki and K. J. Turner. Modelling digital logic in SDL. In *Proc. Joint Int. Conf. on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'97)*, pages 367 – 382, 1997.

[DMVJ97] 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, April 1997.

[E$^+$94] Konrad Etschberger et al. *CAN Controller–Area–Network, Grundlagen, Protokolle, Bausteine, Anwendungen.* Hanser Verlag, 1994.

[EHB$^+$96] R. Ernst, J. Henkel, Th. 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.

[FFKM97] Georg Färber, Franz Fischer, Thomas Kolloch, and Annette 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, October 27–29 1997.

[FKMF97] Franz Fischer, Thomas Kolloch, Annette Muth, and Georg Färber. A configurable target architecture for rapid prototyping high performance control systems. In Hamid R. Arabnia et al., editors, *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*

(PDPTA'97), volume 3, pages 1382–1390, Las Vegas, Nevada, USA, June 30 – July 3 1997.

[FMF98] Franz Fischer, Annette Muth, and Georg Färber. Towards interprocess communication and interface synthesis for a heterogeneous real–time rapid prototyping environment. In *Proc. of the 6th International Workshop on Hardware/Software Co–Design — Codes/CASHE '98*, pages 35–39, Seattle, Washington, USA, March15–18 1998. IEEE, IEEE Computer Society Press.

[For00] Forschungszentrum Informatik (FZI) at the University of Karslruhe, Karlsruhe, Germany. *SPYDER-VIRTEX-X2 User's Manual, Version 1.11*, July 2000.

[GKRM93] Wolfgang Glunz, Thomas Kruse, T. Rössel, and Dieter 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.

[Glu94] Wolfgang Glunz. *Hardware–Entwurf auf abstrakten Ebenen unter Verwendung von Methoden aus dem Software–Entwurf.* Dissertation, Fachbereich Mathematik/Informatik, Universität–Gesamthochschule Paderborn, Paderborn, München, April 1994.

[Gre93a] Klaus Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme.* Number 268 in Fortschrittsberichte VDI, Reihe 10. VDI–Verlag, Düsseldorf, 1993. Dissertation am Lehrstuhl für Prozessrechner, Technische Universität München.

[Gre93b] Klaus Gresser. *Schedulability Analysis for Event–Driven Real–Time Systems.* Number 268 in Fortschrittsberichte VDI, volume 10. VDI–Verlag, Düsseldorf, 1993. Ph.D. Thesis, Institute for Real–Time Computer Systems, Technische Universität München (in german).

[GVNG94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[HKMT97] 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 8th SDL Forum, SDL'97 Time for Testing SDL,* *MSC and Trends*, pages 397–414, Evry, France, September 1997. Elsevier Science Publishers B.V.

[HSK+99] Wolfgang Horn, Bengt Svantesson, Sashi Kumar, Axel Jantsch, and Ahmet 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, April 8 – 9 1999.

[IEE95] IEEE. *The Institute of Electrical and Electronics Engineers: "IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language" (IEEE-1364-1995*, 1995.

[IEE00] IEEE. *The Institute of Electrical and Electronics Engineers: "IEEE Standard VHDL Language Reference Manual" (IEEE-1076-2000*, 2000.

[ITU94] ITU. *ITU–T Recommendation Z.100: CCITT Specification and Description Language (SDL).* ITU–T, June 1994.

[JRM+99] A. A. Jerraya, M. Romdhani, Ph. Le Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J. M. Daveau, and N.-E. Zergainoh. Multilanguage specification for system design and codesign. In A. A. Jerraya and J. Mermet, editors, *"System-level Synthesis", NATO ASI 1998.* Kluwer Academic Publishers, 1999.

[Kle75] Leonard Kleinrock. *Queueing Systems, Volume 1: Theory.* Wiley, New York, 1975.

[Kol01] Thomas Kolloch. *Automated Real–Time Analysis for SDL Systems.* 2001. Dissertation am Lehrstuhl für Realzeit–Computersysteme, Technische Universität München.

[Lar98] Gunnar Larisch. Integration des Spezifikationswerkzeugs SDT in den Entwicklungsprozeß der *REAR* Rapid Protoyping Umgebung, 1998. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München.

[LWS94] Gunther Lehmann, Bernhard Wunder, and Manfred Selz. *Schaltungsdesign mit VHDL (in german).* Franzis Verlag, Poing, Germany, 1994.

[MdCP01] Jean-Pierre Moreau, Philippe di Crescenzo, and Lloyd Pople. Hardware software system codesign based on SDL/C specifications. *Microelectronic Engineering*, 2001.

[MK00] Thomas Maier-Komor. Implementierung von SDL in Hardware nach dem Activity-Thread-Modell, 2000. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München.

[Ols94] Anders Olsen. *Systems engineering using SDL–92*. Elsevier, Amsterdam, 1994.

[Pet00] Stefan M. Petters. Bounding the execution of real–time tasks on modern processors. In *Proc. of the 7th Int. Conf. on Real–Time Computing Systems and Applications*, Cheju Island, South Korea, December 12–14 2000.

[PK92] O. Pulkkinen and K. Kronlöf. Integration of SDL and VHDL for high level digital design. In *Proc. of the European Design Automation Conference with Euro-VHDL*, September 1992.

[PMK$^+$00] Stefan Petters, Annette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Färber. The REAR framework for emulation and analysis of embedded hard real–time systems. *Design Automation for Embedded Systems*, 5(3):237–250, August 2000.

[Rei97] Dirk Reichelt. Konzeption und Implementierung eines Werkzeugs zur automatischen Generierung einer VHDL-Beschreibung aus einer annotierten SDL-Systembeschreibung, 1997. Diplomarbeit (masters thesis) am Institut für Mathematische Maschinen und Datenverarbeitung, Lehrstuhl für Rechnerarchitektur und -Verkehrstheorie, Friedrich-Alexander-Universität Erlangen-Nürnberg.

[Ren94] Rene Seindal, Free Software Foundation, Inc. *GNU m4, version 1.4*, 1994.

[Rin98] Manfred Ringlstetter. Automatisierte Übersetzung von SDL–Spezifikationen in synthetisierbares VHDL für eine Rapid Prototyping Entwicklungsumgebung, 1998. Diplomarbeit am Lehrstuhl für Prozessrechner, Technische Universität München.

[SDM01] F. Slomka, M. Dörfel, and R. Münzenberger. Generating mixed hardware/software systems from SDL specifications. In *Proc. of the 9th International Symposium on Hardware/Software Co-Design*, Copenhagen, Denmark, 2001.

[SDMH00] F. Slomka, M. Dörfel, R. Münzenberger, and R. Hofmann. Hardware/software codesign and rapid-prototyping of embedded systems. *IEEE Design & Test of Computers*, 17(2), 2000.

[SKH98] Bengt Svantesson, Shashi Kumar, and Ahmed Hemani. A methodology and algorithms for efficient interprocess communication synthesis from system description in SDL. In *Proc. of VLSI Design'98*, pages 78–84, Chennai, India, January 1998.

[Svo89] Liba Svobodova. Implementing OSI systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, 1989.

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

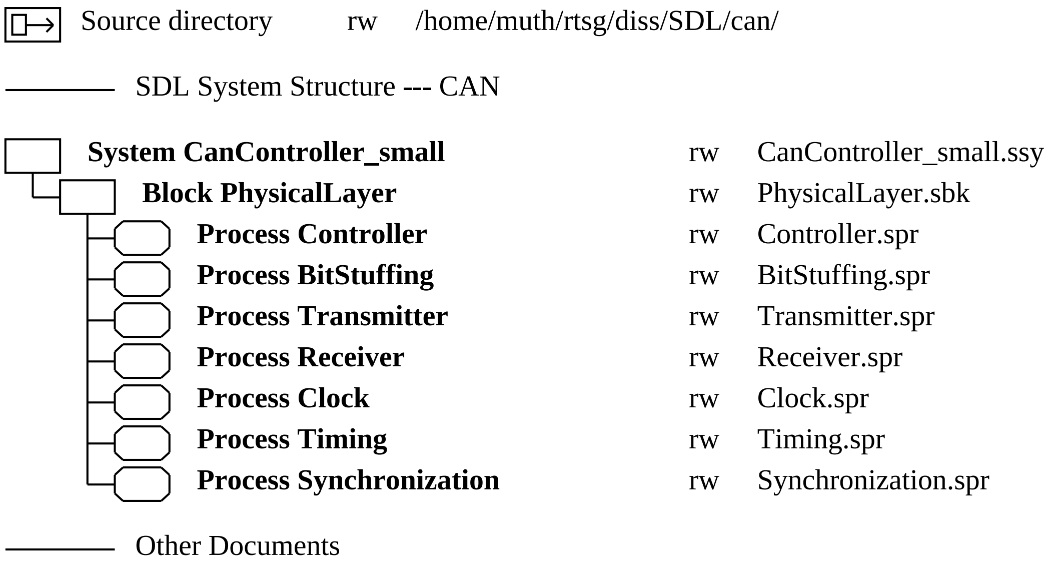
[Wol94] Wayne H. Wolf. Hardware–software co–design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

Appendix A

Application Examples

Appendix A contains the complete SDL specifications of the used application examples, directly printed from the CASE tool SDT.

A.1 CAN Bus Physical Layer



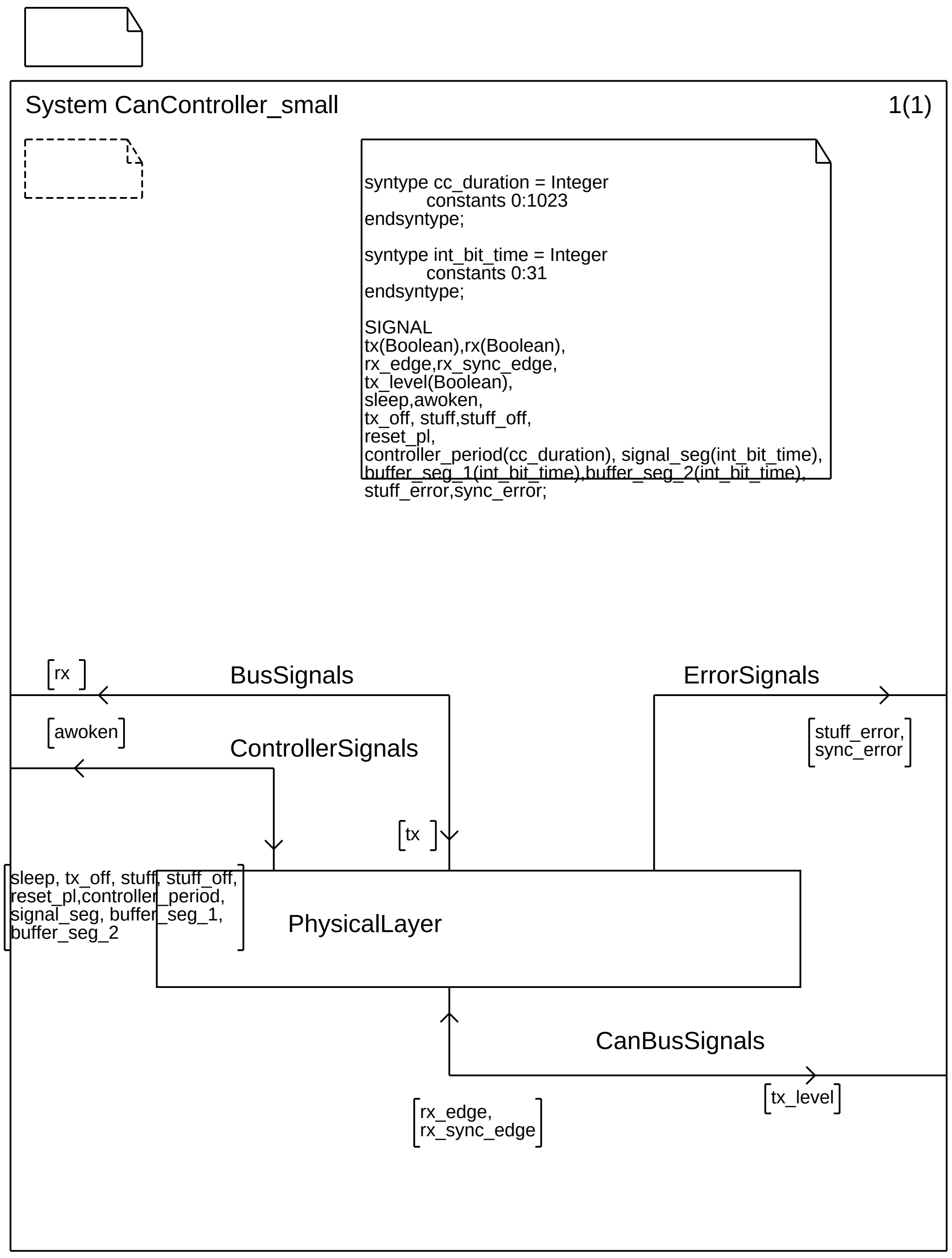
Source directory rw /home/muth/rtsg/diss/SDL/can/

SDL System Structure --- CAN

System CanController_small	rw	CanController_small.ssy
Block PhysicalLayer	rw	PhysicalLayer.sbk
Process Controller	rw	Controller.spr
Process BitStuffing	rw	BitStuffing.spr
Process Transmitter	rw	Transmitter.spr
Process Receiver	rw	Receiver.spr
Process Clock	rw	Clock.spr
Process Timing	rw	Timing.spr
Process Synchronization	rw	Synchronization.spr

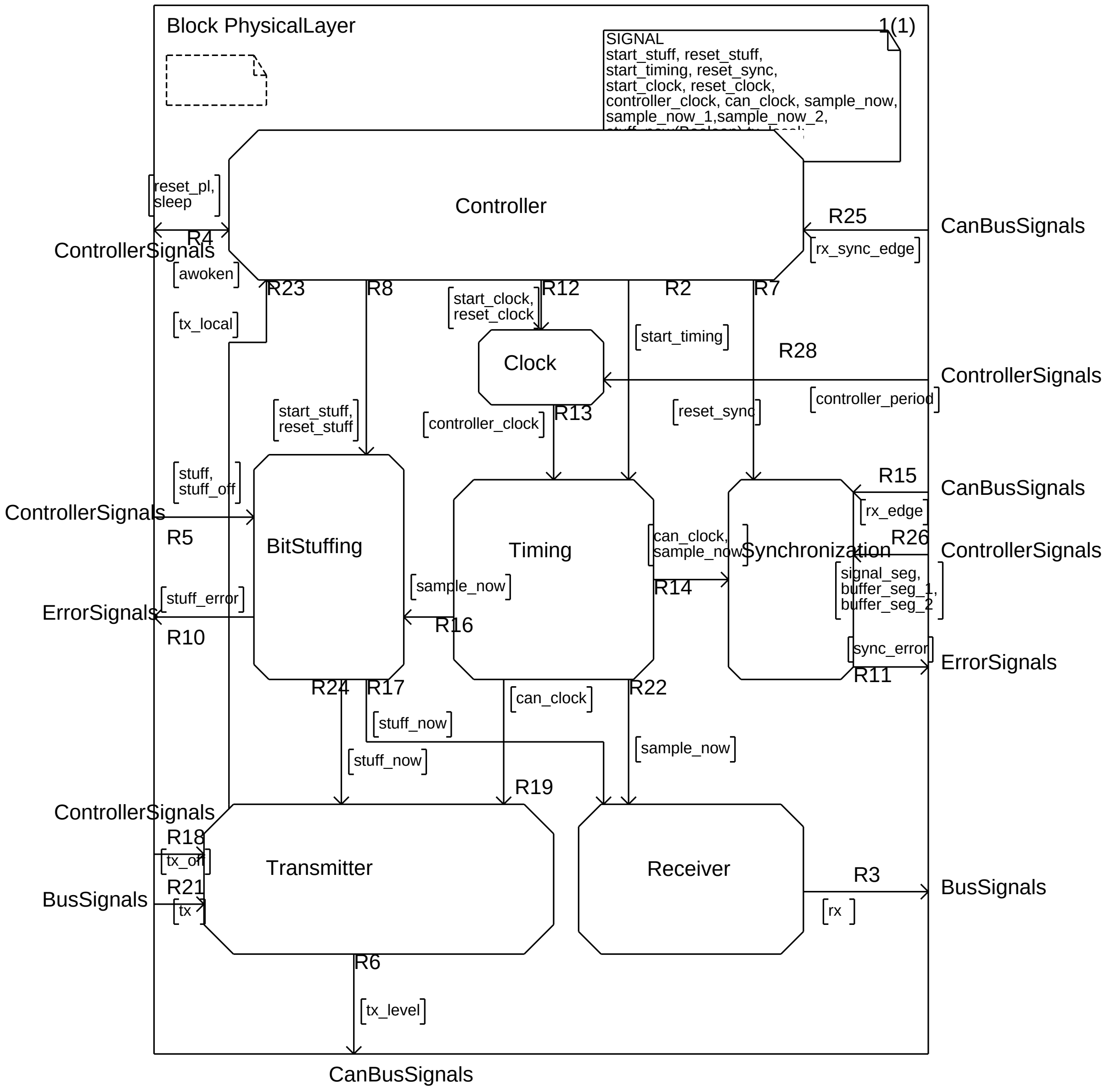
Other Documents

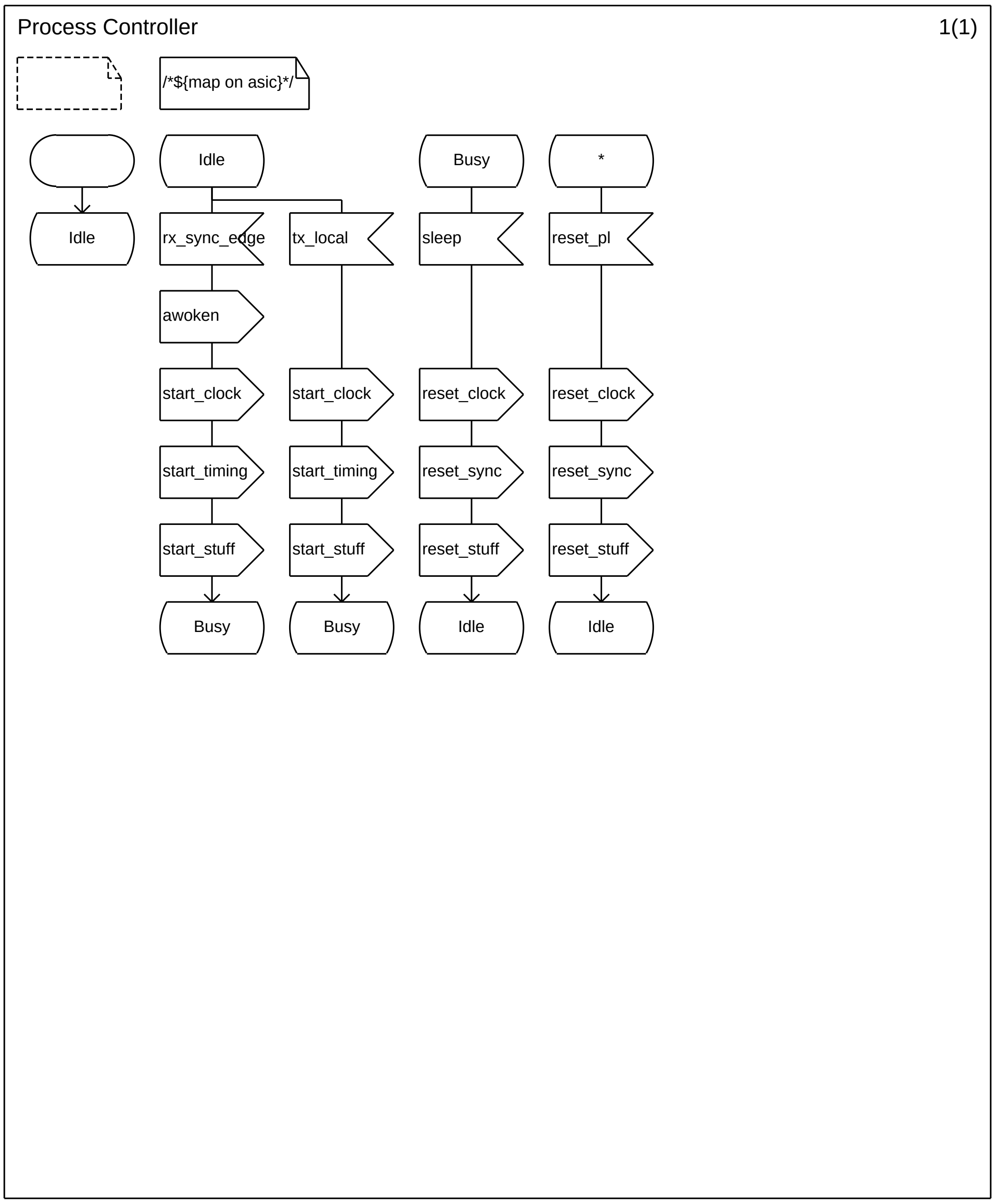
Figure A.1: SDL specification of the CAN physical layer

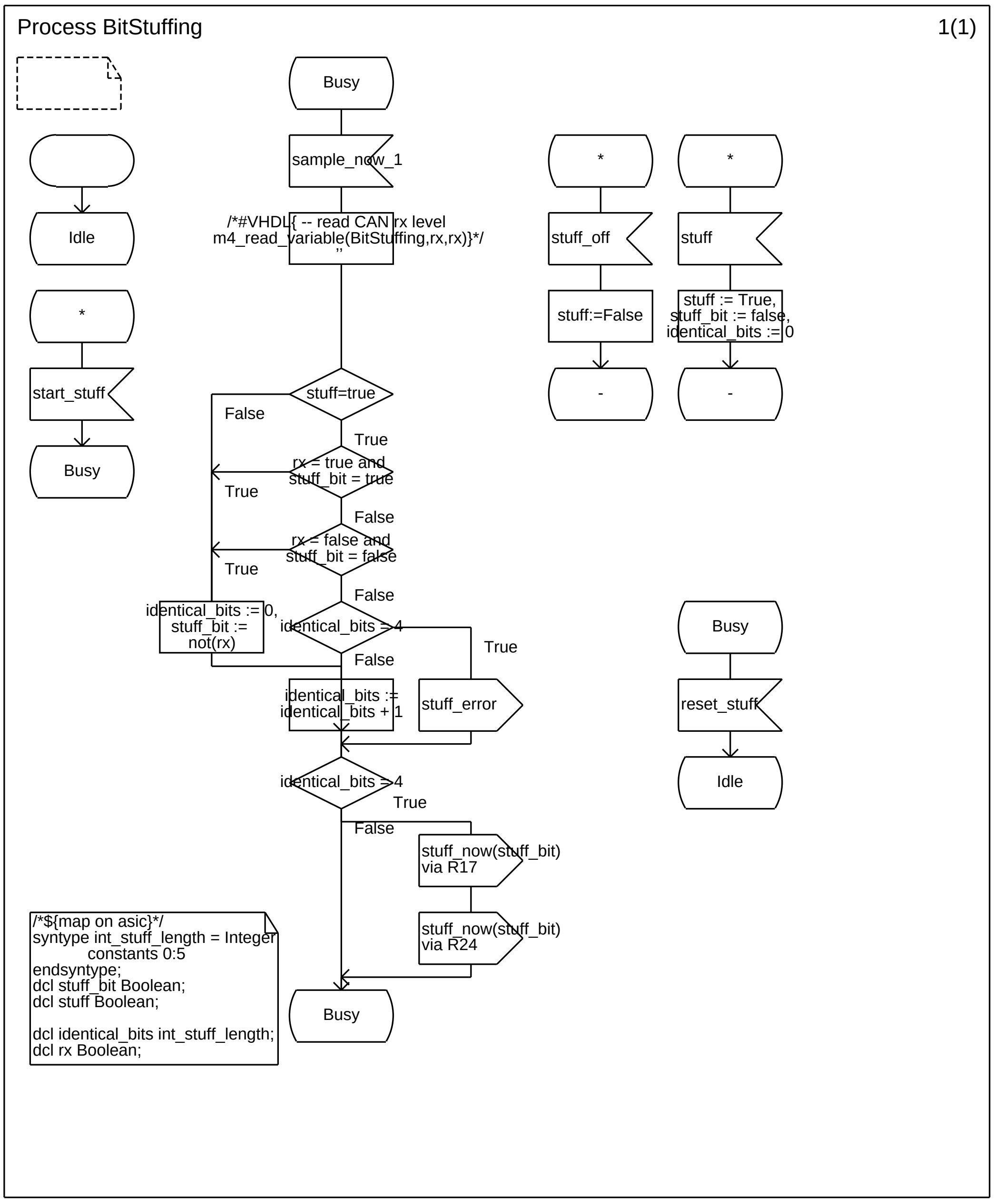


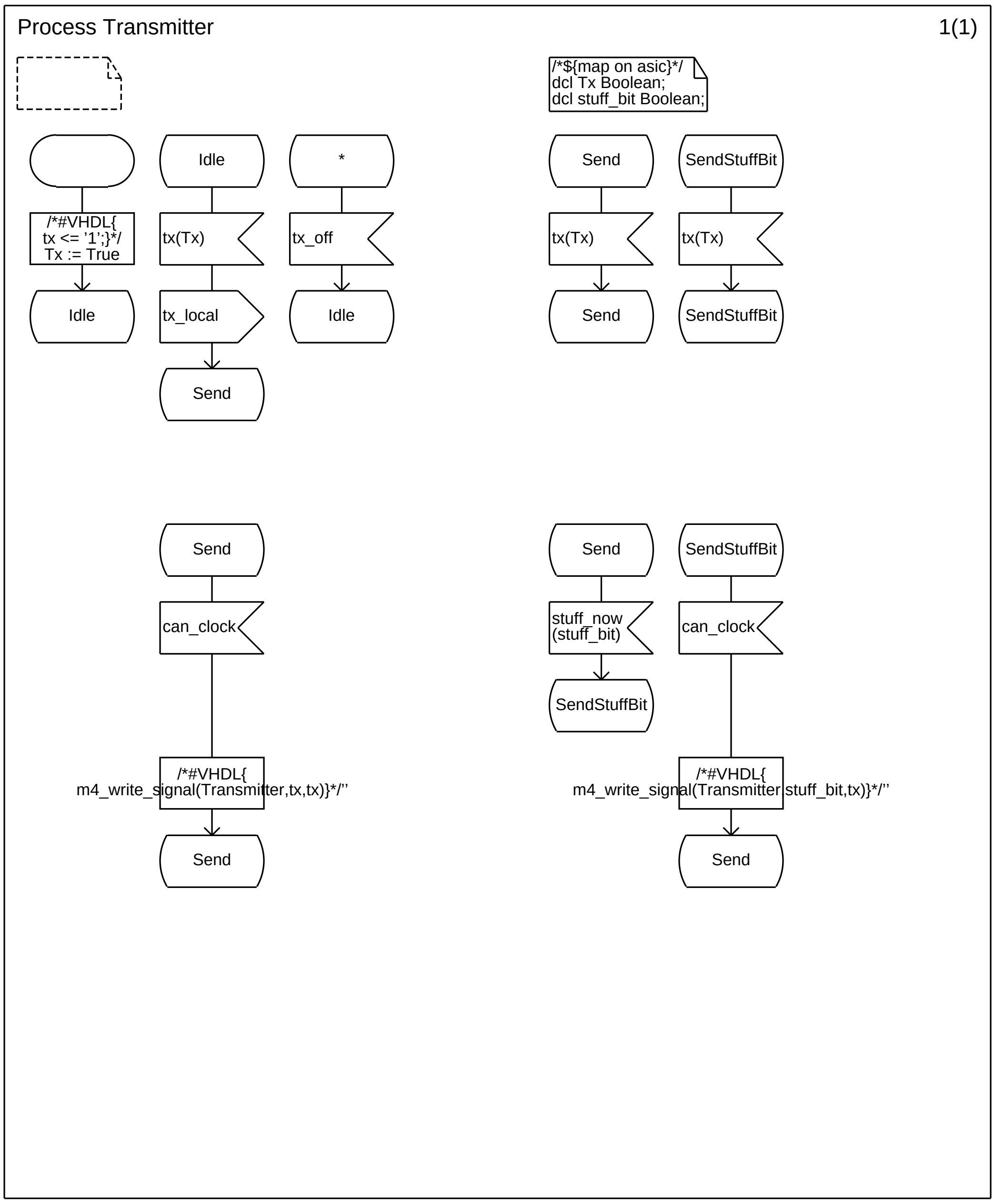
```
syntype cc_duration = Integer
          constants 0:1023
endsyntype;

syntype int_bit_time = Integer
          constants 0:31
endsyntype;

SIGNAL
tx(Boolean),rx(Boolean),
rx_edge,rx_sync_edge,
tx_level(Boolean),
sleep,awoken,
tx_off, stuff,stuff_off,
reset_pl,
controller_period(cc_duration), signal_seg(int_bit_time),
buffer_seg_1(int_bit_time),buffer_seg_2(int_bit_time),
stuff_error,sync_error;
```

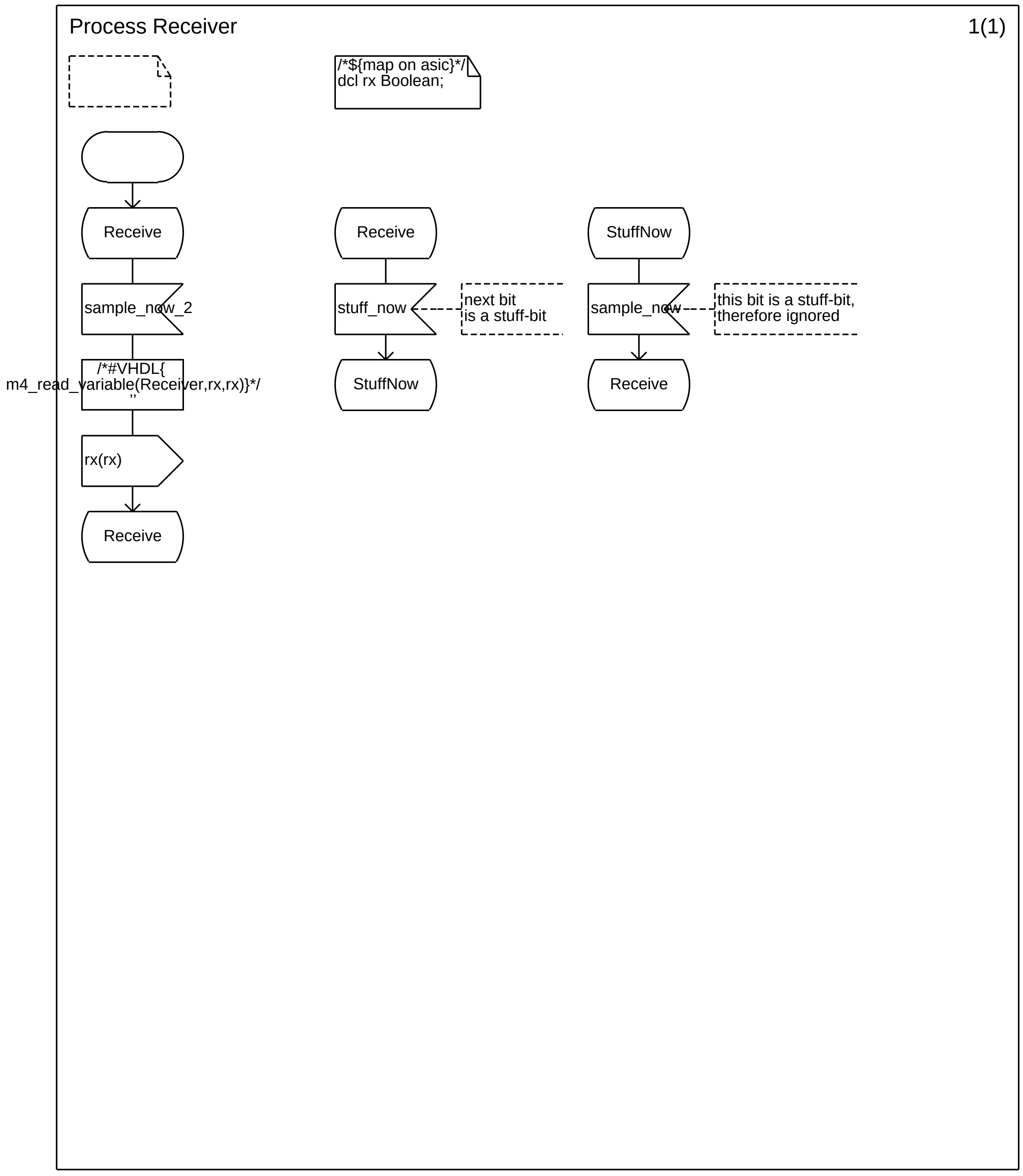
System CanController_small
1(1)
[rx]
BusSignals
ErrorSignals
[awoken]
ControllerSignals
[stuff_error, sync_error]
[tx]
[sleep, tx_off, stuff, stuff_off, reset_pl,controller_period, signal_seg, buffer_seg_1, buffer_seg_2]
PhysicalLayer
CanBusSignals
[tx_level]
[rx_edge, rx_sync_edge]

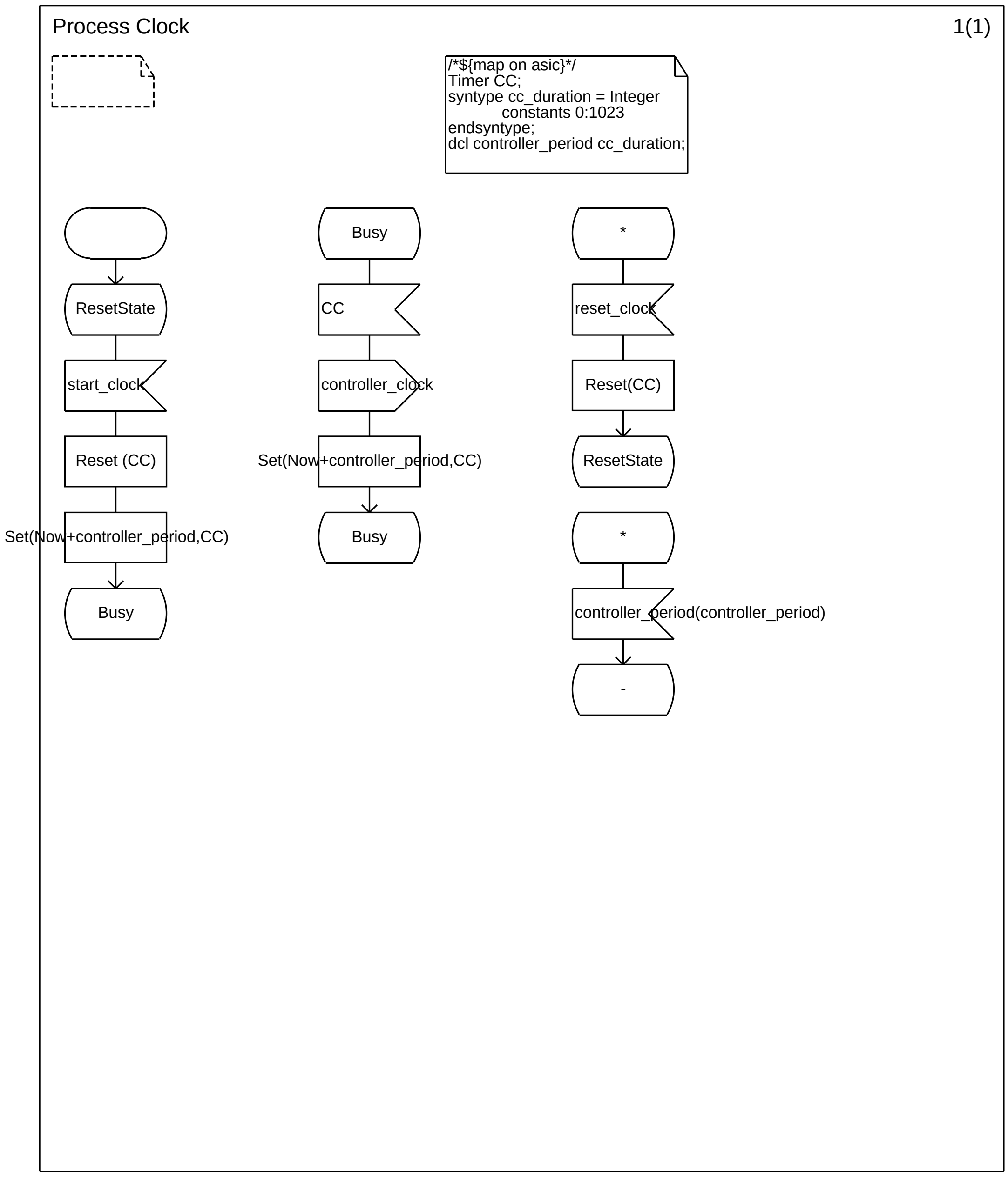

Block PhysicalLayer
1(1)
SIGNAL
start_stuff, reset_stuff,
start_timing, reset_sync,
start_clock, reset_clock,
controller_clock, can_clock, sample_now,
sample_now_1,sample_now_2,
Controller
Clock
BitStuffing
Timing
Synchronization
Transmitter
Receiver
reset_pl, sleep
R4
ControllerSignals
awoken
R23
tx_local
R8
start_clock, reset_clock
R12
R2
R7
R25
CanBusSignals
rx_sync_edge
start_timing
R28
ControllerSignals
controller_period
start_stuff, reset_stuff
controller_clock
R13
reset_sync
stuff, stuff_off
ControllerSignals
R5
R15
CanBusSignals
rx_edge
R26
ControllerSignals
can_clock, sample_now
R14
signal_seg, buffer_seg_1, buffer_seg_2
sample_now
R16
stuff_error
ErrorSignals
R10
sync_error
ErrorSignals
R11
R24
R17
stuff_now
can_clock
R22
sample_now
stuff_now
R19
ControllerSignals
R18
tx_off
R21
BusSignals
tx
R3
BusSignals
rx
R6
tx_level
CanBusSignals

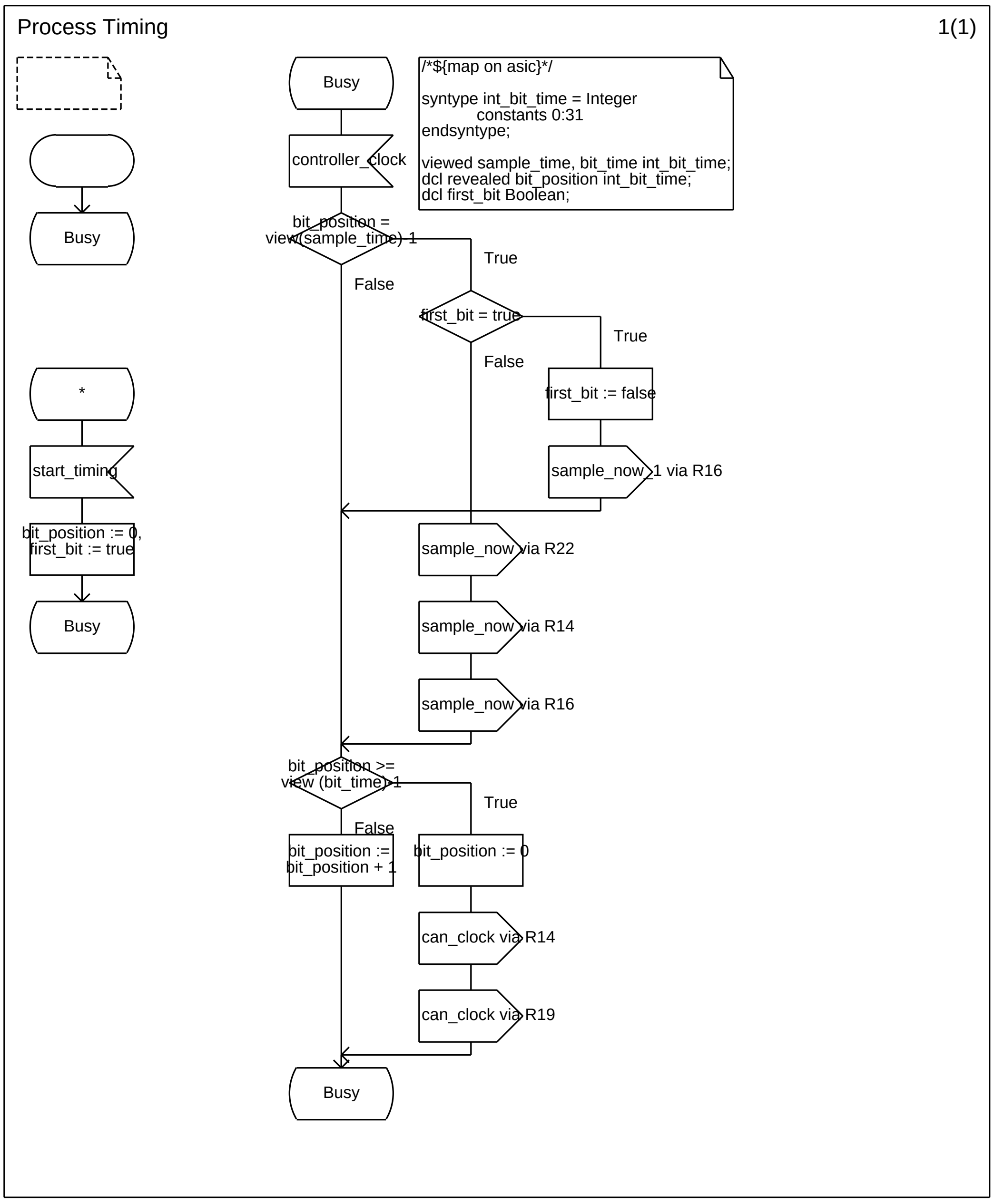

Process Controller
1(1)
/*${map on asic}*/
Idle
Idle
Busy
*
rx_sync_edge
tx_local
sleep
reset_pl
awoken
start_clock
start_clock
reset_clock
reset_clock
start_timing
start_timing
reset_sync
reset_sync
start_stuff
start_stuff
reset_stuff
reset_stuff
Busy
Busy
Idle
Idle


Process BitStuffing
1(1)
Idle
*
start_stuff
Busy
Busy
sample_now_1
/*#VHDL{ -- read CAN rx level
m4_read_variable(BitStuffing,rx,rx)}*/
''
stuff=true
False
True
rx = true and
stuff_bit = true
True
False
rx = false and
stuff_bit = false
True
False
identical_bits := 0,
stuff_bit :=
not(rx)
identical_bits = 4
True
False
identical_bits :=
identical_bits + 1
stuff_error
identical_bits = 4
True
False
stuff_now(stuff_bit)
via R17
stuff_now(stuff_bit)
via R24
Busy
*
stuff_off
stuff:=False
-
*
stuff
stuff := True,
stuff_bit := false,
identical_bits := 0
-
Busy
reset_stuff
Idle
/*${map on asic}*/
syntype int_stuff_length = Integer
constants 0:5
endsyntype;
dcl stuff_bit Boolean;
dcl stuff Boolean;
dcl identical_bits int_stuff_length;
dcl rx Boolean;

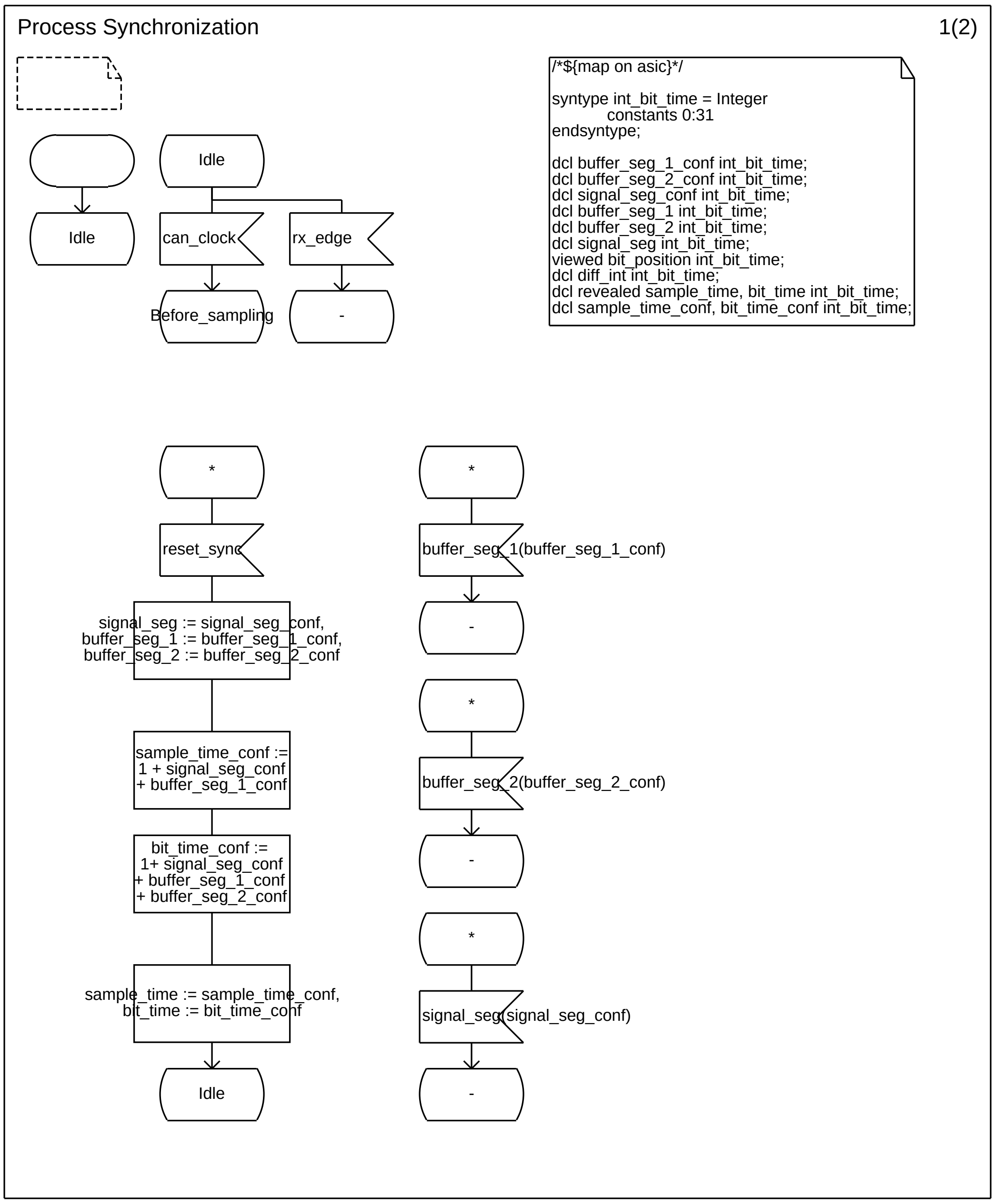

Process Transmitter
1(1)
/*${map on asic}*/
dcl Tx Boolean;
dcl stuff_bit Boolean;
/*#VHDL{
tx <= '1';}*/
Tx := True
Idle
Idle
tx(Tx)
tx_local
Send
*
tx_off
Idle
Send
tx(Tx)
Send
SendStuffBit
tx(Tx)
SendStuffBit
Send
can_clock
/*#VHDL{
m4_write_signal(Transmitter,tx,tx)}*/''
Send
Send
stuff_now
(stuff_bit)
SendStuffBit
SendStuffBit
can_clock
/*#VHDL{
m4_write_signal(Transmitter,stuff_bit,tx)}*/''
Send


Process Receiver
1(1)
/*${map on asic}*/
dcl rx Boolean;
Receive
sample_now_2
/*#VHDL{
m4_read_variable(Receiver,rx,rx)}*/
''
rx(rx)
Receive
Receive
stuff_now
next bit
is a stuff-bit
StuffNow
StuffNow
sample_now
this bit is a stuff-bit,
therefore ignored
Receive

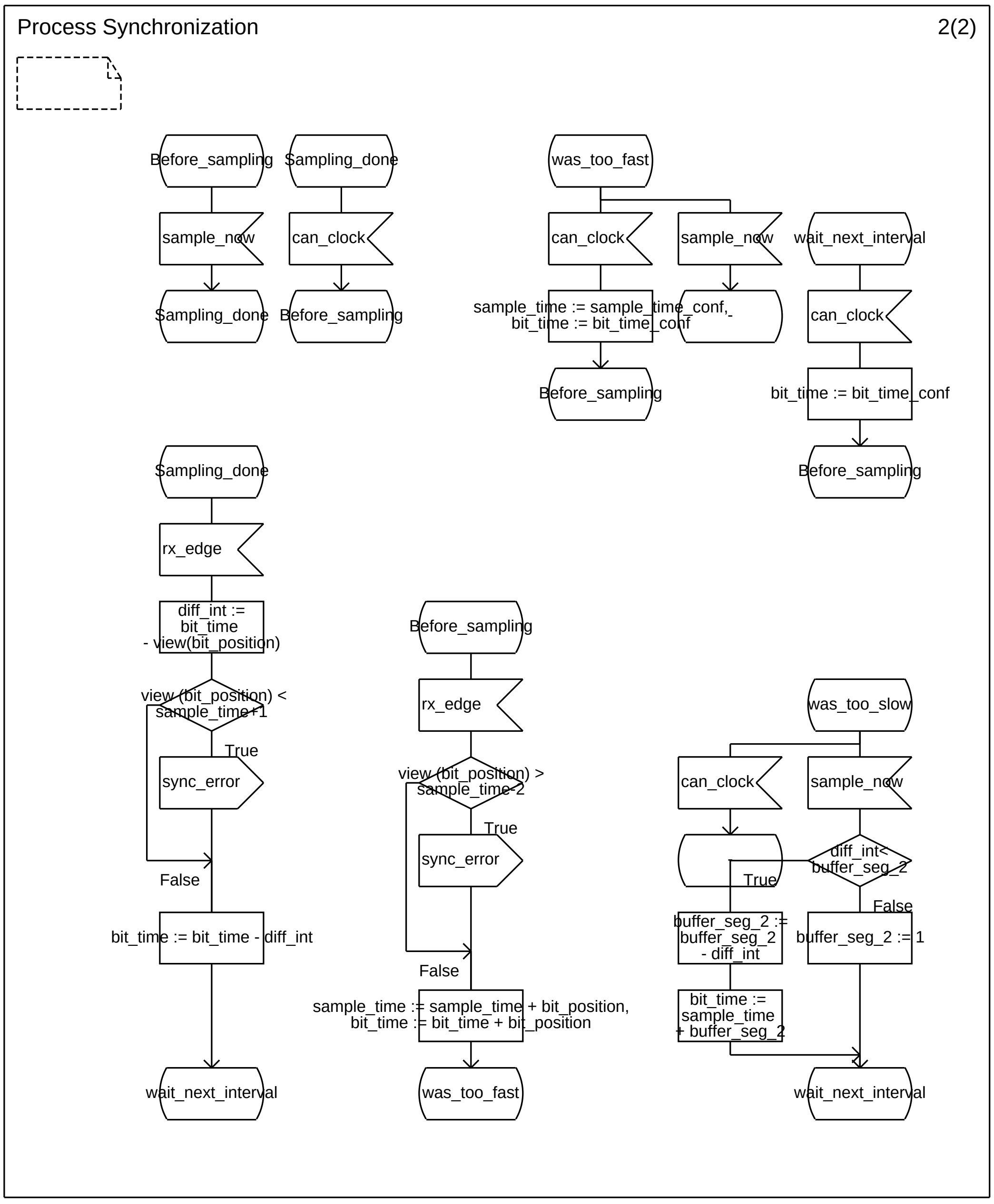

Process Clock
1(1)
/*${map on asic}*/
Timer CC;
syntype cc_duration = Integer
constants 0:1023
endsyntype;
dcl controller_period cc_duration;
ResetState
start_clock
Reset (CC)
Set(Now+controller_period,CC)
Busy
Busy
CC
controller_clock
Set(Now+controller_period,CC)
Busy
*
reset_clock
Reset(CC)
ResetState
*
controller_period(controller_period)
-



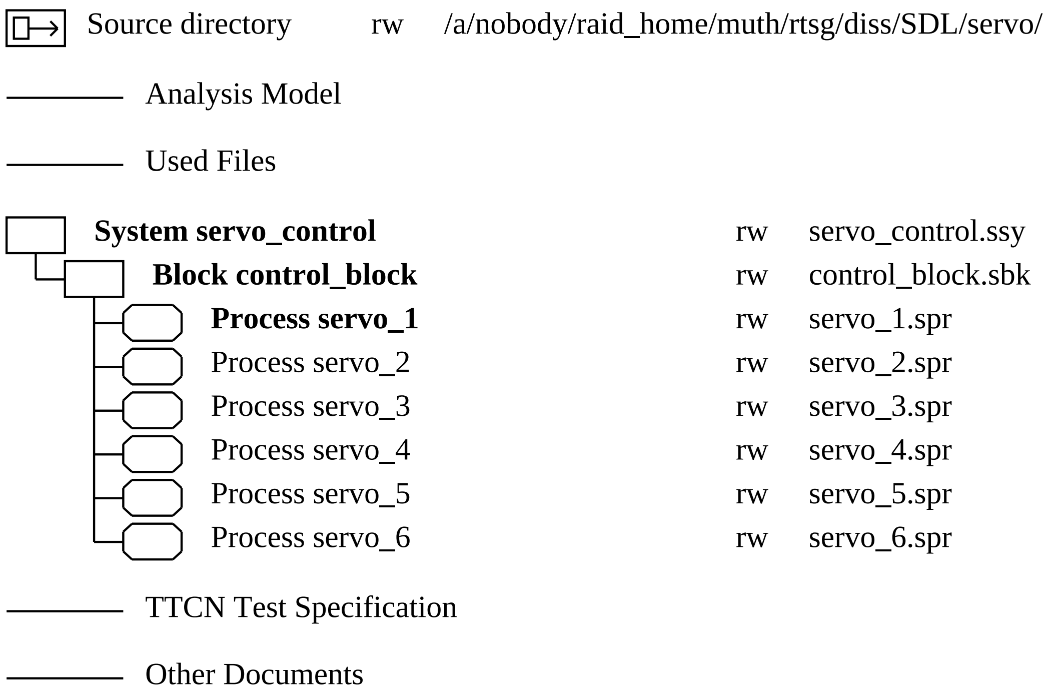
```
/*${map on asic}*/

syntype int_bit_time = Integer
          constants 0:31
endsyntype;

viewed sample_time, bit_time int_bit_time;
dcl revealed bit_position int_bit_time;
dcl first_bit Boolean;
```


Process Synchronization
1(2)
Idle
Idle
can_clock
rx_edge
Before_sampling
-
*
reset_sync
signal_seg := signal_seg_conf,
buffer_seg_1 := buffer_seg_1_conf,
buffer_seg_2 := buffer_seg_2_conf
sample_time_conf :=
1 + signal_seg_conf
+ buffer_seg_1_conf
bit_time_conf :=
1+ signal_seg_conf
+ buffer_seg_1_conf
+ buffer_seg_2_conf
sample_time := sample_time_conf,
bit_time := bit_time_conf
Idle
*
buffer_seg_1(buffer_seg_1_conf)
-
*
buffer_seg_2(buffer_seg_2_conf)
-
*
signal_seg(signal_seg_conf)
-

```
/*${map on asic}*/

syntype int_bit_time = Integer
        constants 0:31
endsyntype;

dcl buffer_seg_1_conf int_bit_time;
dcl buffer_seg_2_conf int_bit_time;
dcl signal_seg_conf int_bit_time;
dcl buffer_seg_1 int_bit_time;
dcl buffer_seg_2 int_bit_time;
dcl signal_seg int_bit_time;
viewed bit_position int_bit_time;
dcl diff_int int_bit_time;
dcl revealed sample_time, bit_time int_bit_time;
dcl sample_time_conf, bit_time_conf int_bit_time;
```

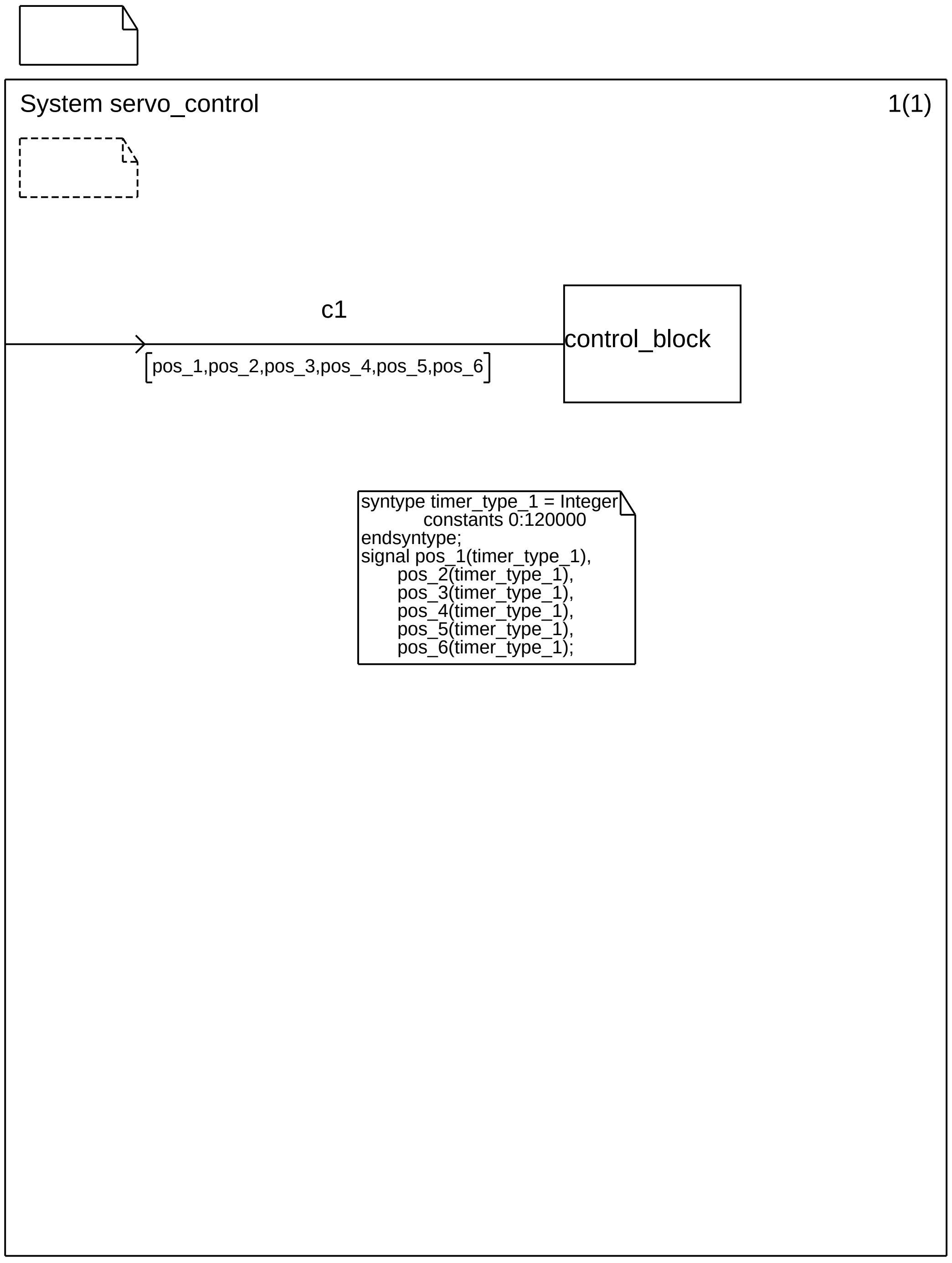

Process Synchronization
2(2)
Before_sampling
Sampling_done
sample_now
can_clock
Sampling_done
Before_sampling
was_too_fast
can_clock
sample_now
wait_next_interval
sample_time := sample_time_conf,
bit_time := bit_time_conf
-
can_clock
Before_sampling
bit_time := bit_time_conf
Before_sampling
Sampling_done
rx_edge
diff_int :=
bit_time
- view(bit_position)
view (bit_position) <
sample_time+1
True
sync_error
False
bit_time := bit_time - diff_int
wait_next_interval
Before_sampling
rx_edge
view (bit_position) >
sample_time-2
True
sync_error
False
sample_time := sample_time + bit_position,
bit_time := bit_time + bit_position
was_too_fast
was_too_slow
can_clock
sample_now
diff_int<
buffer_seg_2
True
False
buffer_seg_2 :=
buffer_seg_2
- diff_int
buffer_seg_2 := 1
bit_time :=
sample_time
+ buffer_seg_2
wait_next_interval

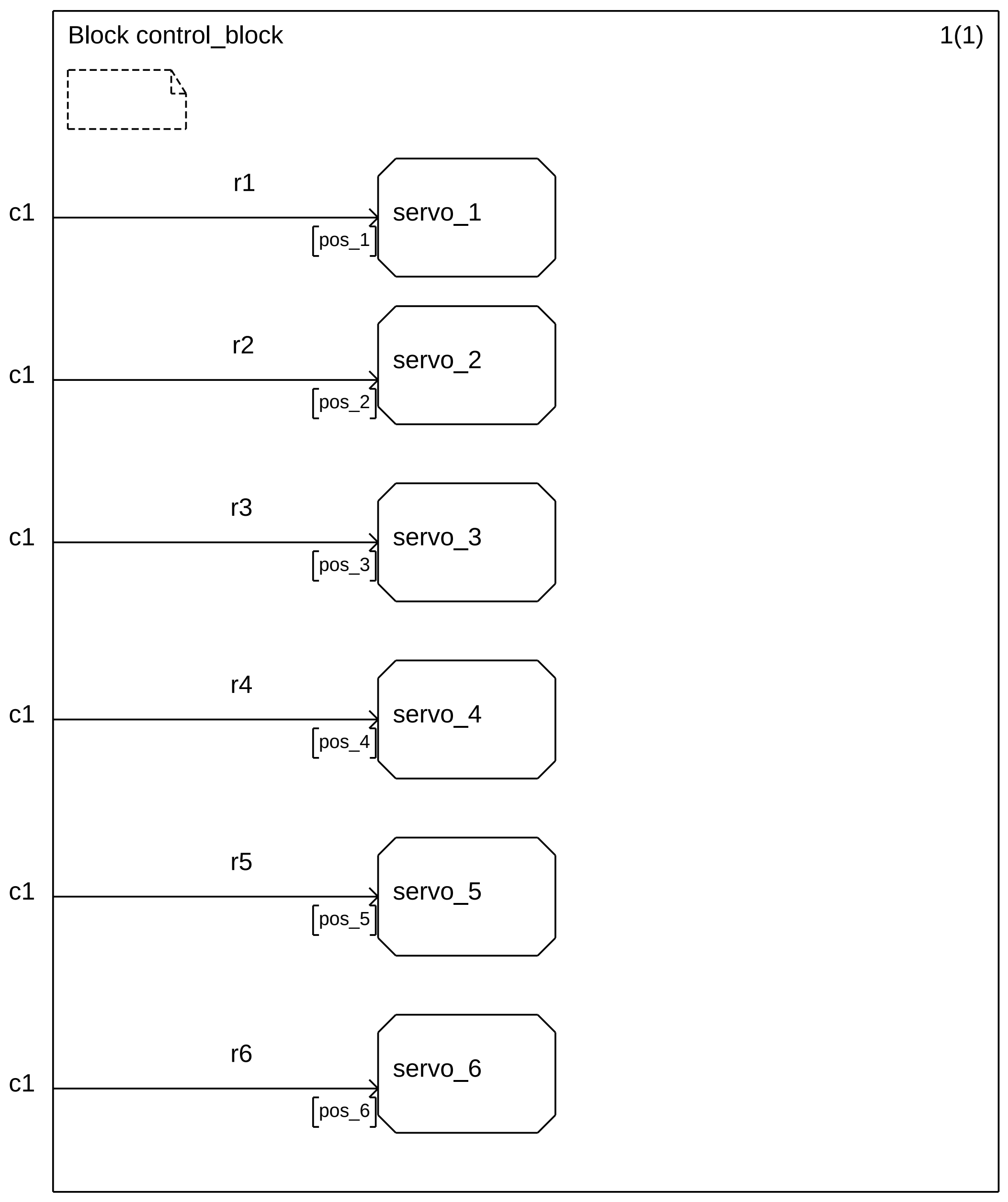
A.2 Servo Motor Controller

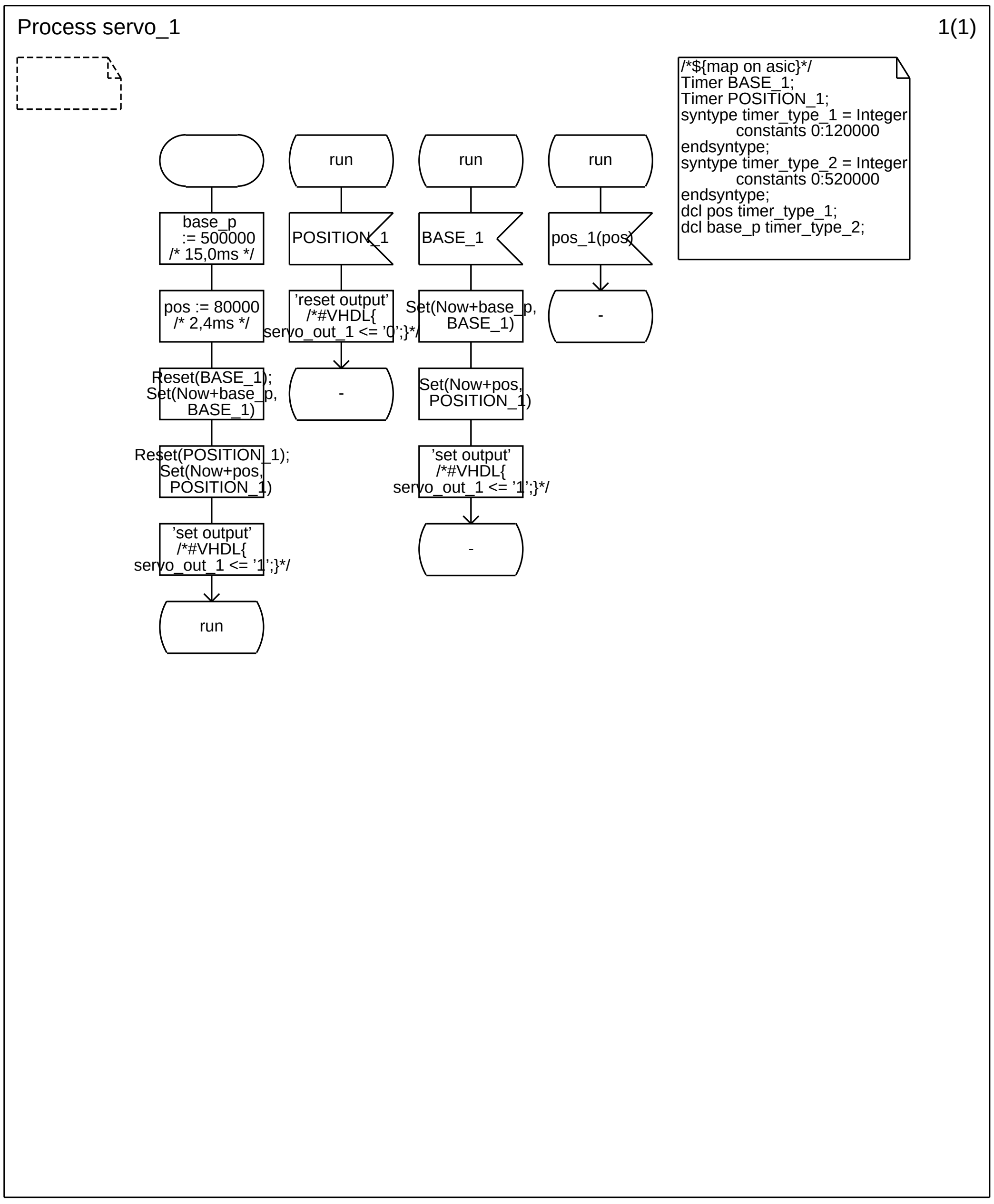


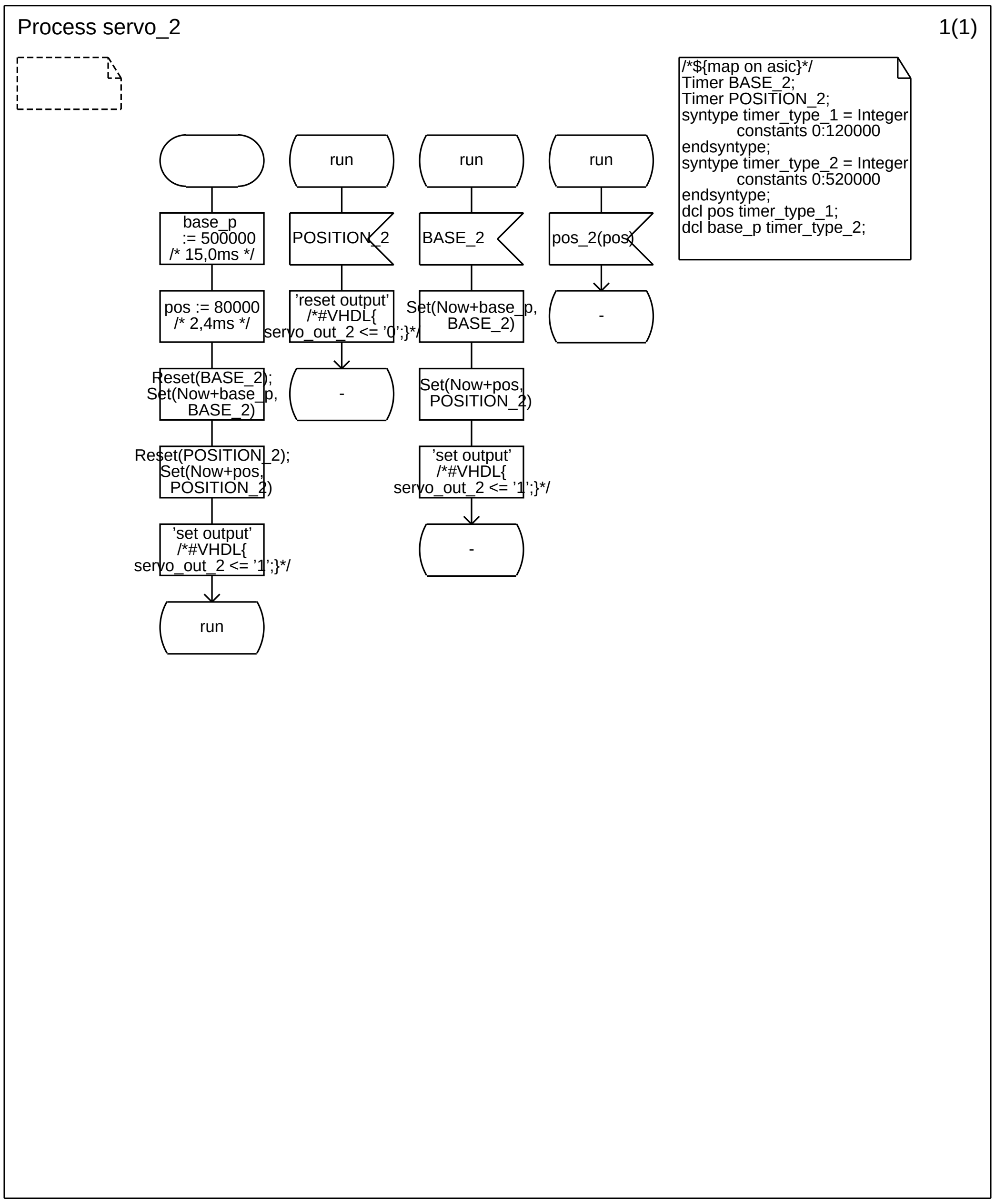
Source directory	rw	/a/nobody/raid_home/muth/rtsg/diss/SDL/servo/
Analysis Model		
Used Files		
System servo_control	rw	servo_control.ssy
Block control_block	rw	control_block.sbk
Process servo_1	rw	servo_1.spr
Process servo_2	rw	servo_2.spr
Process servo_3	rw	servo_3.spr
Process servo_4	rw	servo_4.spr
Process servo_5	rw	servo_5.spr
Process servo_6	rw	servo_6.spr
TTCN Test Specification		
Other Documents		

Figure A.2: SDL specification of the servo motor controller

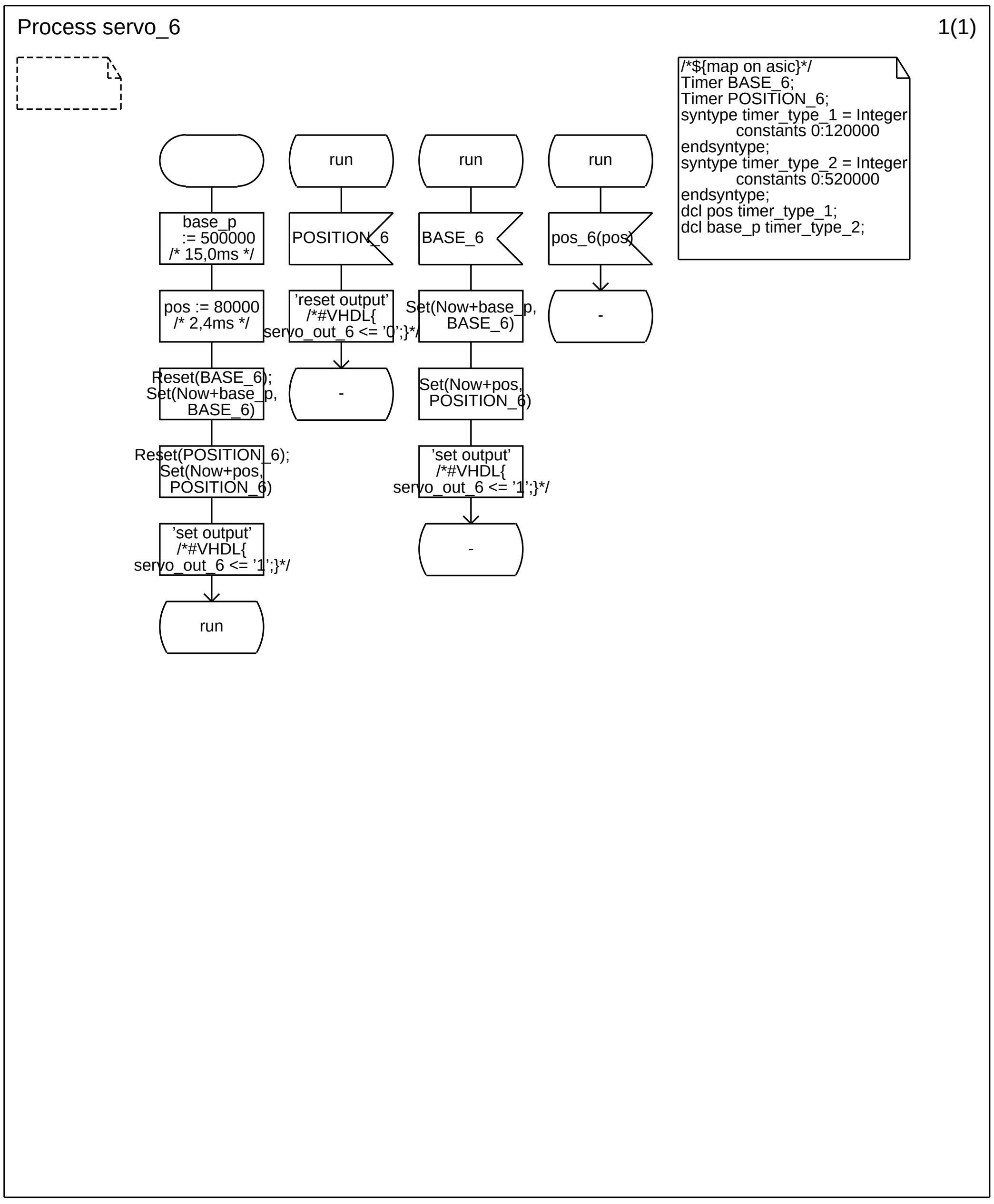

System servo_control
1(1)
c1
[pos_1,pos_2,pos_3,pos_4,pos_5,pos_6]
control_block
syntype timer_type_1 = Integer
constants 0:120000
endsyntype;
signal pos_1(timer_type_1),
pos_2(timer_type_1),
pos_3(timer_type_1),
pos_4(timer_type_1),
pos_5(timer_type_1),
pos_6(timer_type_1);


Block control_block
1(1)
c1
r1
servo_1
pos_1
c1
r2
servo_2
pos_2
c1
r3
servo_3
pos_3
c1
r4
servo_4
pos_4
c1
r5
servo_5
pos_5
c1
r6
servo_6
pos_6

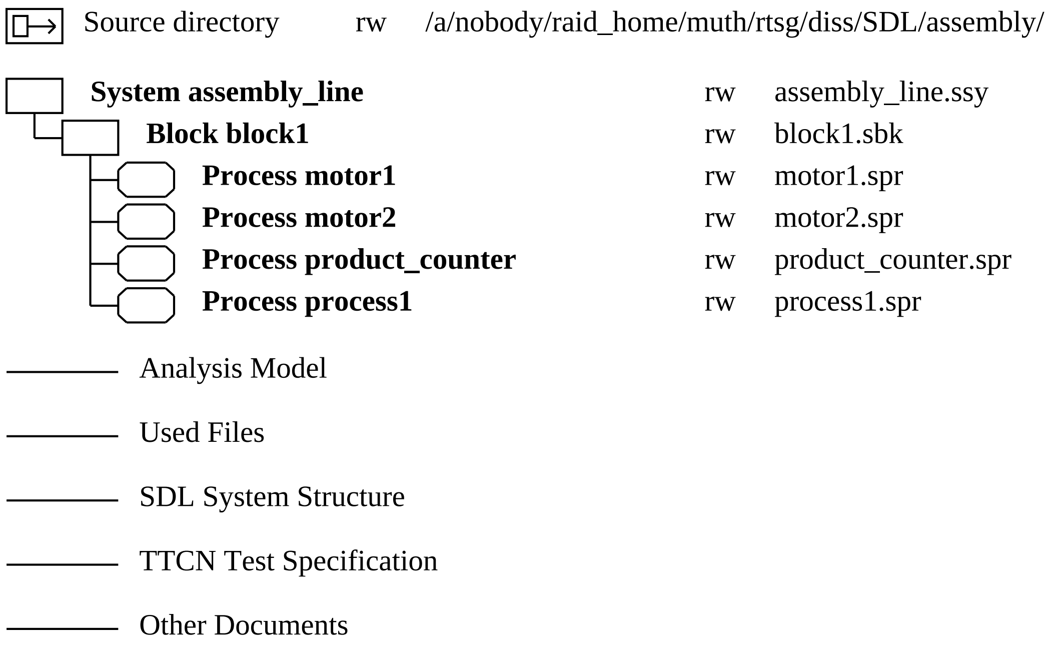

Process servo_1
1(1)
/*${map on asic}*/
Timer BASE_1;
Timer POSITION_1;
syntype timer_type_1 = Integer
constants 0:120000
endsyntype;
syntype timer_type_2 = Integer
constants 0:520000
endsyntype;
dcl pos timer_type_1;
dcl base_p timer_type_2;
base_p := 500000 /* 15,0ms */
pos := 80000 /* 2,4ms */
Reset(BASE_1); Set(Now+base_p, BASE_1)
Reset(POSITION_1); Set(Now+pos, POSITION_1)
'set output' /*#VHDL{ servo_out_1 <= '1';}*/
run
run
POSITION_1
'reset output' /*#VHDL{ servo_out_1 <= '0';}*/
-
run
BASE_1
Set(Now+base_p, BASE_1)
Set(Now+pos, POSITION_1)
'set output' /*#VHDL{ servo_out_1 <= '1';}*/
-
run
pos_1(pos)
-


Process servo_2
1(1)
base_p := 500000 /* 15,0ms */
pos := 80000 /* 2,4ms */
Reset(BASE_2); Set(Now+base_p, BASE_2)
Reset(POSITION_2); Set(Now+pos, POSITION_2)
'set output' /*#VHDL{ servo_out_2 <= '1';}*/
run
run
POSITION_2
'reset output' /*#VHDL{ servo_out_2 <= '0';}*/
-
run
BASE_2
Set(Now+base_p, BASE_2)
Set(Now+pos, POSITION_2)
'set output' /*#VHDL{ servo_out_2 <= '1';}*/
-
run
pos_2(pos)
-

```
/*${map on asic}*/
Timer BASE_2;
Timer POSITION_2;
syntype timer_type_1 = Integer
        constants 0:120000
endsyntype;
syntype timer_type_2 = Integer
        constants 0:520000
endsyntype;
dcl pos timer_type_1;
dcl base_p timer_type_2;
```


Process servo_6
1(1)
/*${map on asic}*/
Timer BASE_6;
Timer POSITION_6;
syntype timer_type_1 = Integer
constants 0:120000
endsyntype;
syntype timer_type_2 = Integer
constants 0:520000
endsyntype;
dcl pos timer_type_1;
dcl base_p timer_type_2;
base_p := 500000 /* 15,0ms */
pos := 80000 /* 2,4ms */
Reset(BASE_6); Set(Now+base_p, BASE_6)
Reset(POSITION_6); Set(Now+pos, POSITION_6)
'set output' /*#VHDL{ servo_out_6 <= '1';}*/
run
run
POSITION_6
'reset output' /*#VHDL{ servo_out_6 <= '0';}*/
-
run
BASE_6
Set(Now+base_p, BASE_6)
Set(Now+pos, POSITION_6)
'set output' /*#VHDL{ servo_out_6 <= '1';}*/
-
run
pos_6(pos)
-

A.3 Assembly Line



Source directory	rw	/a/nobody/raid_home/muth/rtsg/diss/SDL/assembly/
System assembly_line	rw	assembly_line.ssy
Block block1	rw	block1.sbk
Process motor1	rw	motor1.spr
Process motor2	rw	motor2.spr
Process product_counter	rw	product_counter.spr
Process process1	rw	process1.spr

Analysis Model

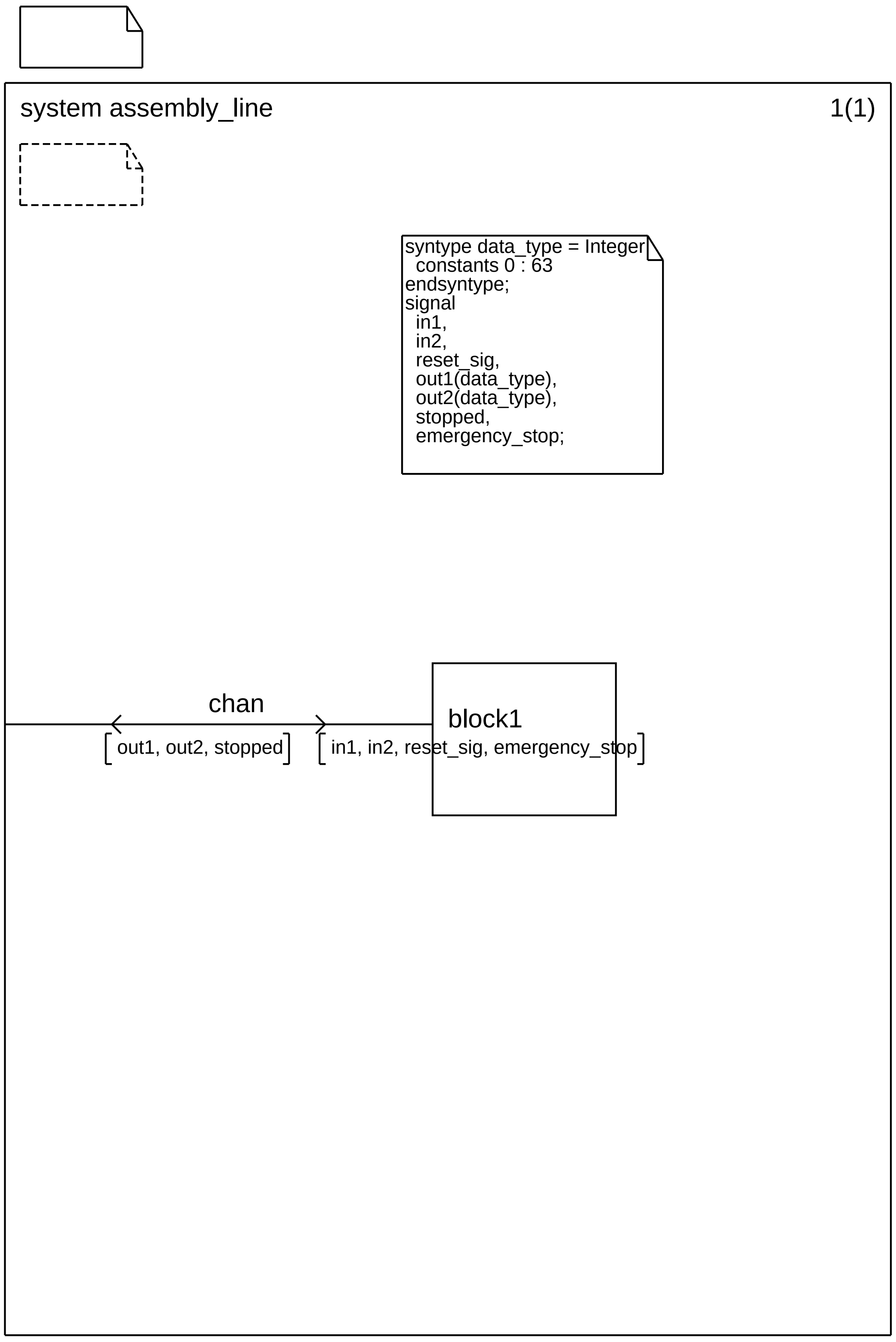
Used Files

SDL System Structure

TTCN Test Specification

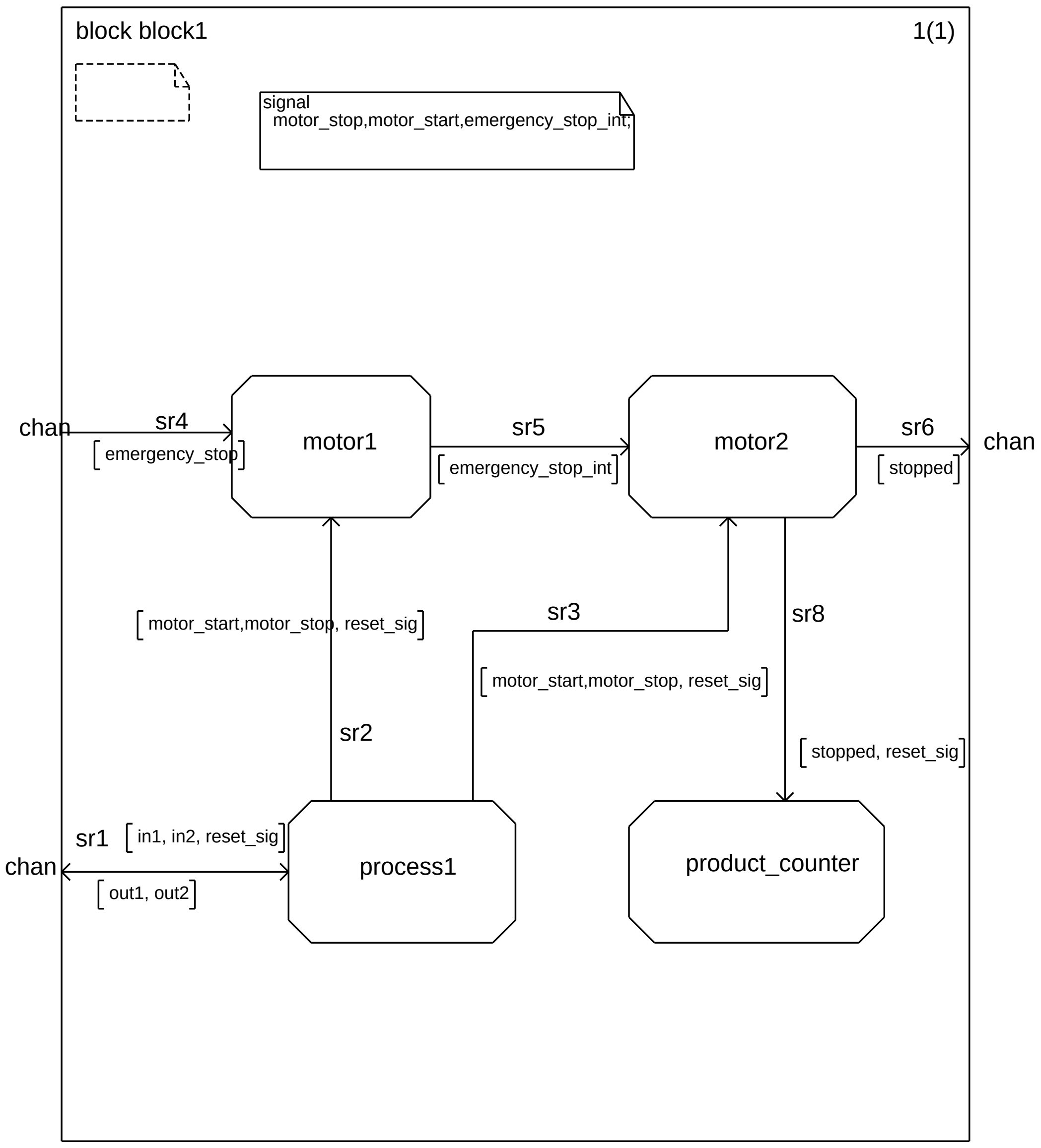
Other Documents

Figure A.3: SDL specification of the assembly line example

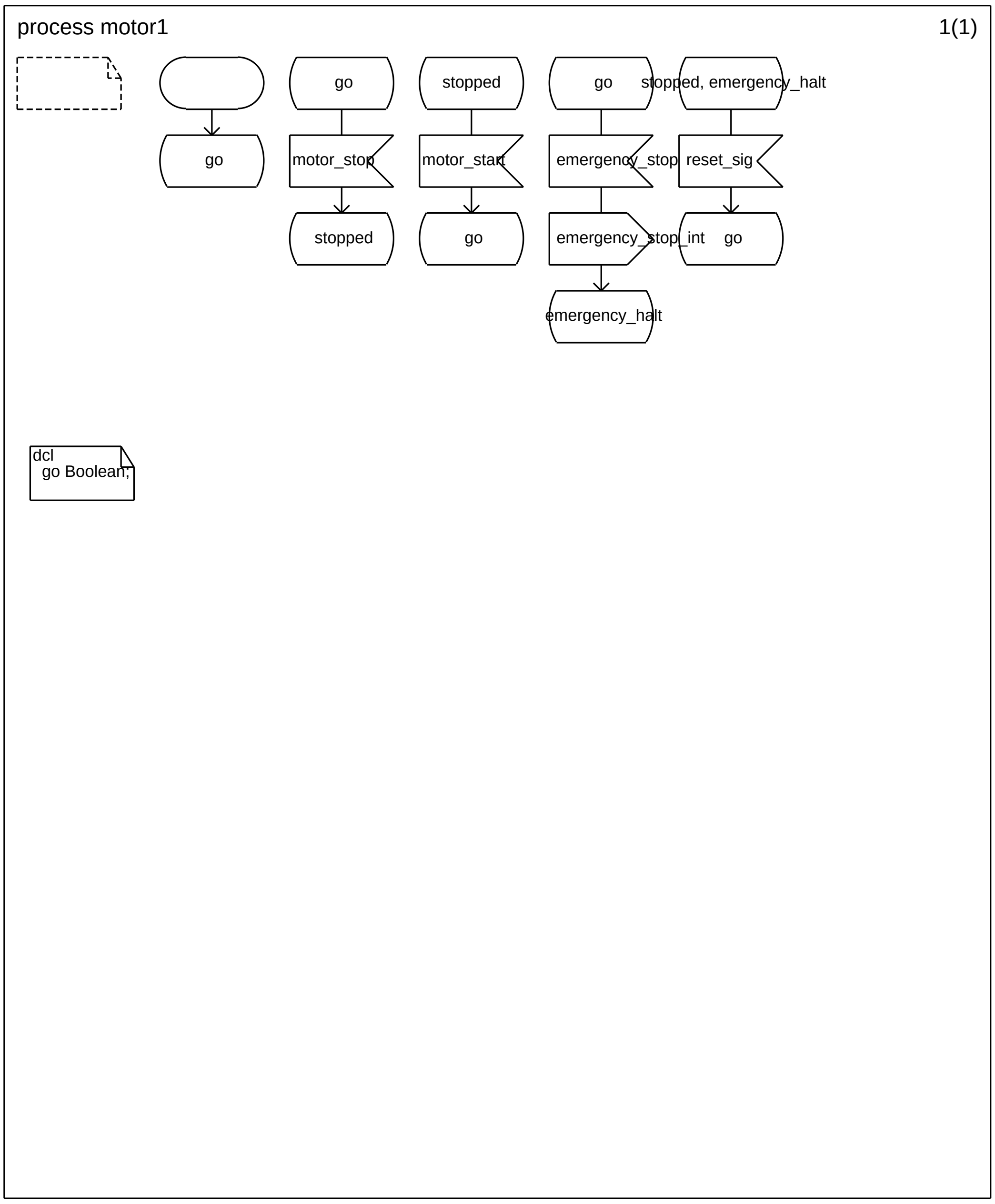


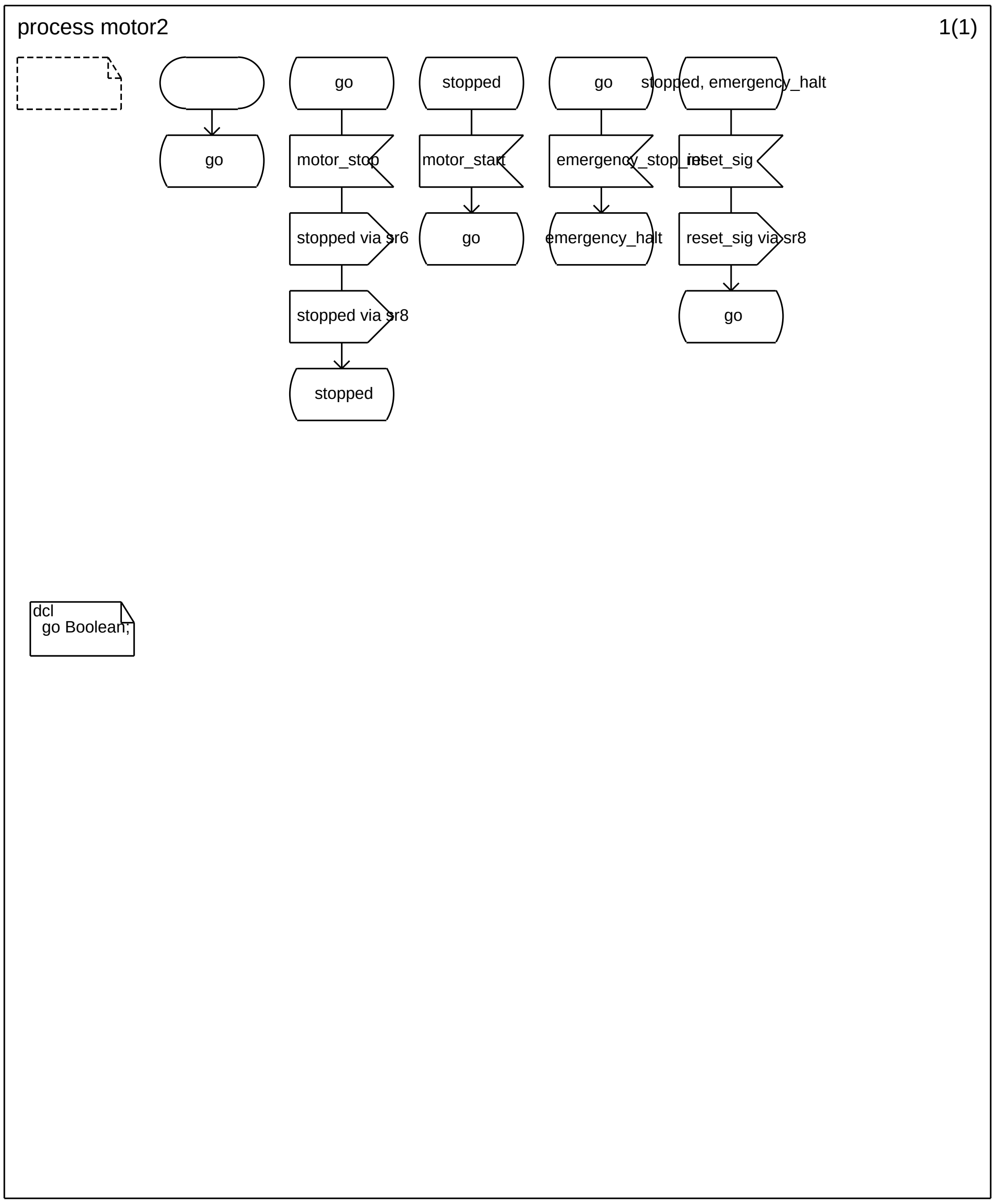
```
syntype data_type = Integer
  constants 0 : 63
endsyntype;
signal
  in1,
  in2,
  reset_sig,
  out1(data_type),
  out2(data_type),
  stopped,
  emergency_stop;
```

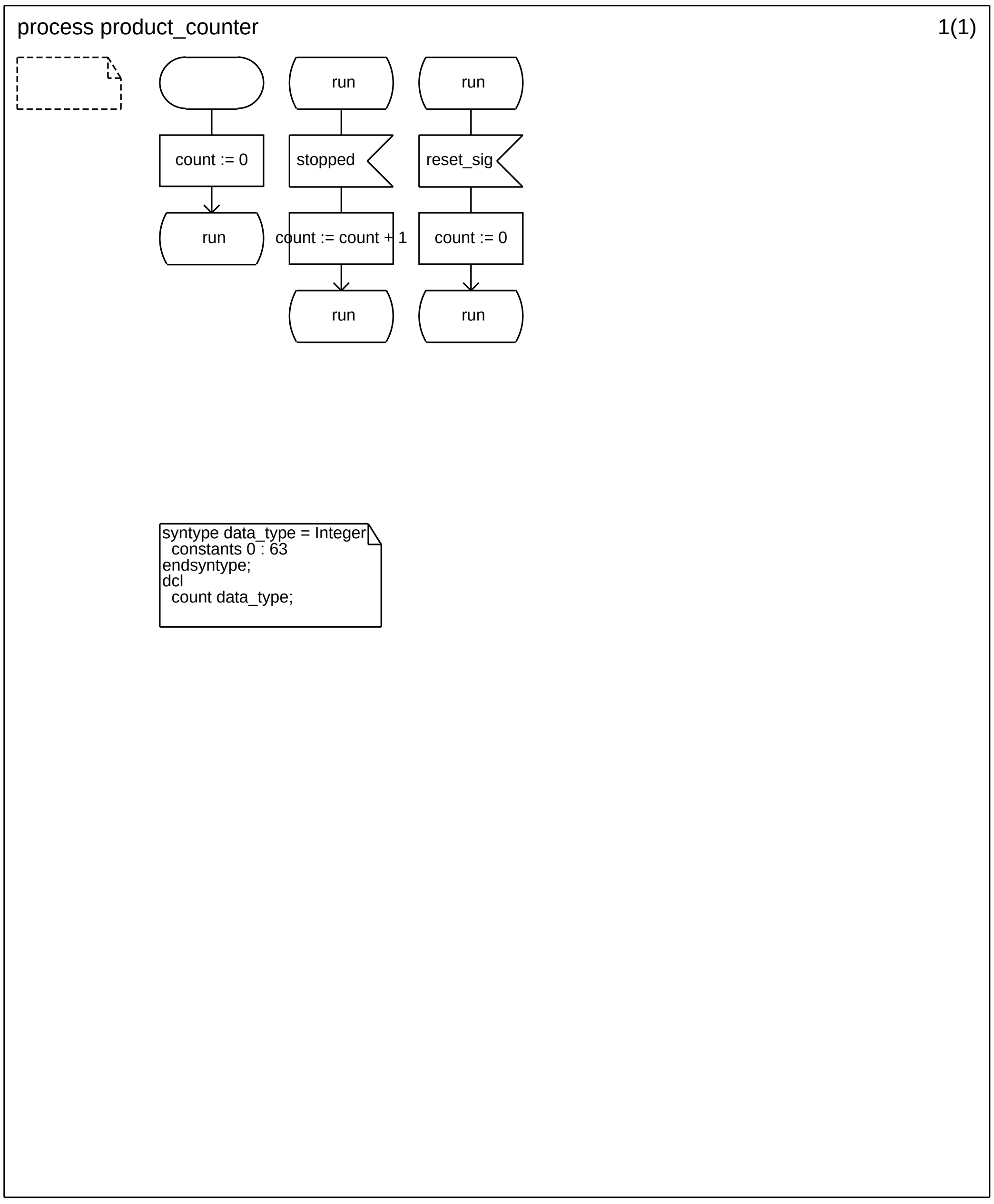
1.

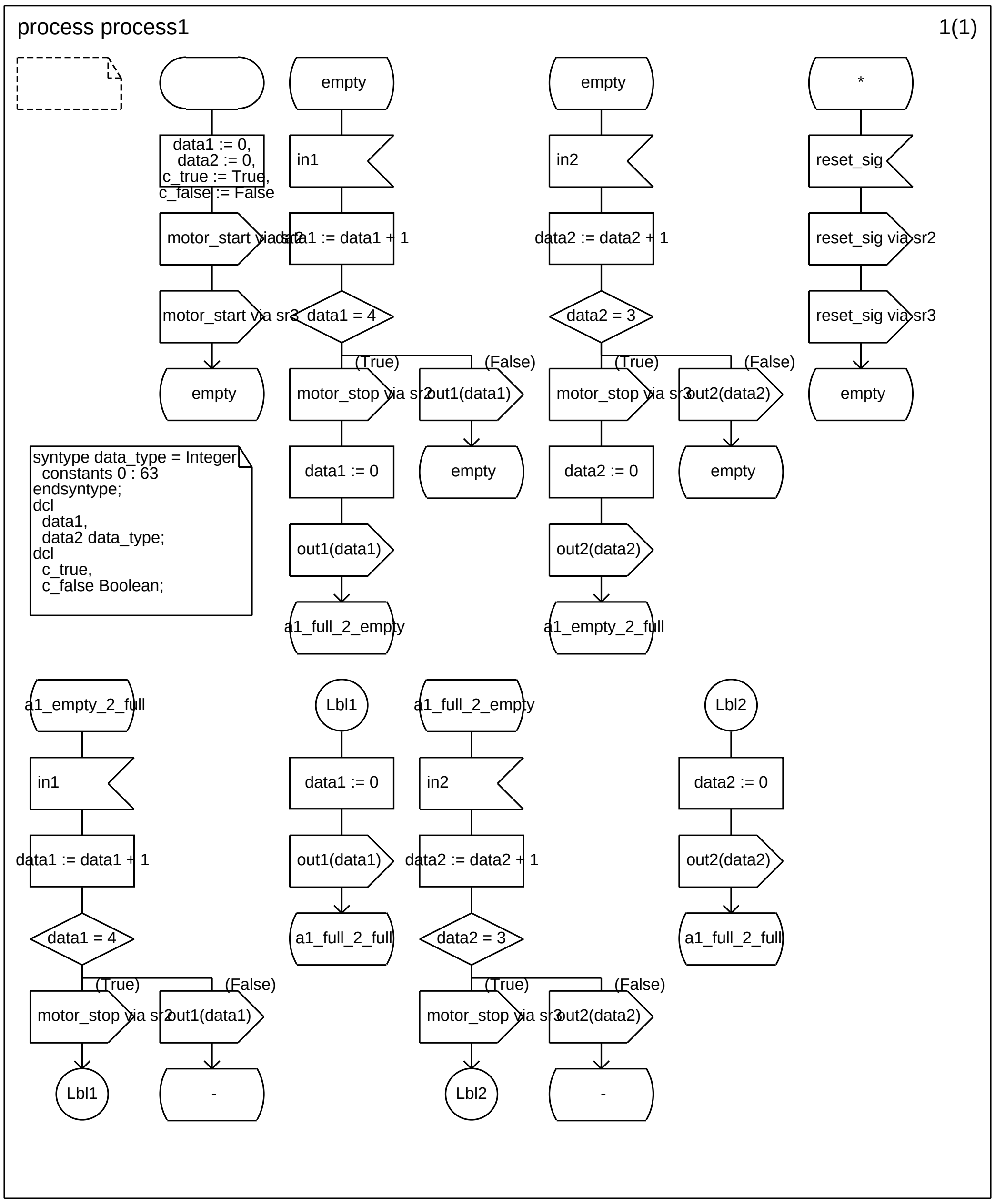


block block1
1(1)
signal
motor_stop,motor_start,emergency_stop_int;
chan
sr4
emergency_stop
motor1
sr5
emergency_stop_int
motor2
sr6
stopped
chan
motor_start,motor_stop, reset_sig
sr3
motor_start,motor_stop, reset_sig
sr8
sr2
stopped, reset_sig
sr1
in1, in2, reset_sig
chan
out1, out2
process1
product_counter


process motor1
1(1)
go
go
motor_stop
stopped
stopped
motor_start
go
go
emergency_stop
emergency_stop_int
emergency_halt
stopped, emergency_halt
reset_sig
go
dcl
go Boolean;


process motor2
1(1)
go
go
motor_stop
stopped via sr6
stopped via sr8
stopped
stopped
motor_start
go
go
emergency_stop_in
emergency_halt
stopped, emergency_halt
reset_sig
reset_sig via sr8
go
dcl
go Boolean;


process product_counter
1(1)
count := 0
run
run
stopped
count := count + 1
run
run
reset_sig
count := 0
run
syntype data_type = Integer
constants 0 : 63
endsyntype;
dcl
count data_type;


process process1
1(1)
data1 := 0,
data2 := 0,
c_true := True,
c_false := False
motor_start via sr2
motor_start via sr3
empty
syntype data_type = Integer
constants 0 : 63
endsyntype;
dcl
data1,
data2 data_type;
dcl
c_true,
c_false Boolean;
in1
data1 := data1 + 1
data1 = 4
(True)
(False)
motor_stop via sr2
out1(data1)
data1 := 0
a1_full_2_empty
in2
data2 := data2 + 1
data2 = 3
motor_stop via sr3
out2(data2)
data2 := 0
a1_empty_2_full
*
reset_sig
reset_sig via sr2
reset_sig via sr3
Lbl1
Lbl2
a1_full_2_full
-

Appendix B

Generated VHDL Code

To print the generated VDHL code of all application examples, or even the complete code of one example, would require an unreasonable amount of space. To give an impression of the generated hardware, a few of the generated VHDL files of the application example "assembly line" are printed here:

- Top-level VHDL file of the server model implementation (figure B.1)
- SDL process motor1, server implementation, before `m4` macro replacement (figure B.2)
- SDL process motor2, server implementation, after `m4` macro replacement (figure B.3)
- SDL process process1, server implementation (figure B.4)
- EFSM Parallel activity thread implementation of AT `emergency_stop` (figure B.5)
- EFSM Serial activity thread implementation (figure B.6)

```
LIBRARY WORK;
LIBRARY CIOP;
LIBRARY SDL;
LIBRARY UNISIM;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE WORK.ALL;
USE WORK.config.ALL;
USE WORK.signallist.ALL;
USE CIOP.ALL;
USE SDL.ALL;
USE UNISIM.ALL;


ENTITY assembly_line IS
    PORT (
        -- PADS ...
        SYS_CLK:        IN      std_logic; -- clk, reset
        GNET2_GRESET_0 : IN std_logic;
        FRD_0: in std_logic;
        FWR_0: in std_logic;
        -- Spyder local bus interface
        LA: in std_logic_vector (31 downto 2);
        LBE0_0: in std_logic;
        LBE1_0: in std_logic;
        LBE2_0: in std_logic;
        LBE3_0: in std_logic;
        LD: inout std_logic_vector (31 downto 0);
        -- galileo local bus interface
        ADDR:           IN      std_logic_vector( 4 DOWNTO 0);
        CSB, RWB, RDB:  IN      std_logic; -- csel,
        DATA:           INOUT   std_logic_vector(31 DOWNTO 0);
        -- galileo interrupt
        BINT5:          OUT     std_logic;
        -- LEDs
        FPGA_LED_0 : out std_logic;
        FPGA_LED_1 : out std_logic;
        -- ... io-ports
           -- driver interface
        OE_A, DIR_A             : OUT   std_logic;
        OE_B, DIR_B             : OUT   std_logic;
        OE_C, DIR_C             : OUT   std_logic;
        OE_D, DIR_D             : OUT   std_logic;
        OE_EF, DIR_E, DIR_F     : OUT   std_logic;

        D:              OUT     std_logic_vector(8-1 DOWNTO 0)

         );
end assembly_line;

ARCHITECTURE assembly_line_arch OF assembly_line IS



-- spyder adaption:
SIGNAL  CLOCK, GSETRESET:       std_logic; -- clk, reset
SIGNAL  LED:            std_logic;
-- general signals:
SIGNAL idin:            std_logic_vector(31 DOWNTO 0);
SIGNAL idout2, idout3,
       idout4, idout5,
       idout6, idout7:  std_logic_vector(31 DOWNTO 0);
SIGNAL zero32:          std_logic_vector(31 DOWNTO 0);
SIGNAL regread:         std_logic;
SIGNAL regwrite:        std_logic_vector( 7 DOWNTO 0);
```

```
-- REGISTERS:
-- MCSRR (0):
SIGNAL mcsrrctrl:       std_logic_vector(1 DOWNTO 0);
SIGNAL imcsrrout:       std_logic_vector(31 DOWNTO 0);
SIGNAL mcsrrout:        std_logic_vector(31 DOWNTO 0);
SIGNAL reset:           std_logic;
SIGNAL softreset:       std_logic;      -- alias for mcsrrctrl(0)
SIGNAL mien:            std_logic;      -- alias for mcsrrctrl(1)
-- COUNT (1):
SIGNAL countclr:        std_logic;
SIGNAL countout:        std_logic_vector(31 DOWNTO 0);




   SIGNAL   hwswqueue_channel: std_logic_vector(19 downto 0);
   SIGNAL   send_hwswqueue: std_logic_vector(2-1 downto 0);
   SIGNAL   ack_hwswqueue: std_logic_vector(2-1 downto 0);
   SIGNAL   not_empty_int,intpend: std_logic;


   -- input signals process process1
   SIGNAL   process1_channel: std_logic_vector(num_in_channels_process1*queue_wi
dth_process1-1 downto 0);
   SIGNAL   send_process1:   std_logic_vector(num_in_channels_process1-1 downto
0);
   SIGNAL   ack_process1:   std_logic_vector(num_in_channels_process1-1 downto 0
);


   SIGNAL   process1_env_channel: std_logic_vector(3 downto 0);
   SIGNAL   process1_env_send: std_logic;
   SIGNAL   process1_env_ack: std_logic;


   -- input signals process motor1
   SIGNAL   motor1_channel: std_logic_vector(num_in_channels_motor1*queue_width_
motor1-1 downto 0);
   SIGNAL   send_motor1:   std_logic_vector(num_in_channels_motor1-1 downto 0);
   SIGNAL   ack_motor1:   std_logic_vector(num_in_channels_motor1-1 downto 0);


   SIGNAL   motor1_env_channel: std_logic_vector(3 downto 0);
   SIGNAL   motor1_env_send: std_logic;
   SIGNAL   motor1_env_ack: std_logic;


   -- input signals process motor2
   SIGNAL   motor2_channel: std_logic_vector(num_in_channels_motor2*queue_width_
motor2-1 downto 0);
   SIGNAL   send_motor2:   std_logic_vector(num_in_channels_motor2-1 downto 0);
   SIGNAL   ack_motor2:   std_logic_vector(num_in_channels_motor2-1 downto 0);


   SIGNAL   motor2_env_channel: std_logic_vector(3 downto 0);
   SIGNAL   motor2_env_send: std_logic;
   SIGNAL   motor2_env_ack: std_logic;


   -- input signals process product_counter
   SIGNAL   product_counter_channel: std_logic_vector(num_in_channels_product_co
unter*queue_width_product_counter-1 downto 0);
   SIGNAL   send_product_counter:   std_logic_vector(num_in_channels_product_cou
nter-1 downto 0);
   SIGNAL   ack_product_counter:   std_logic_vector(num_in_channels_product_coun
ter-1 downto 0);
```

Figure B.1: Top-level VHDL file of application example assembly line

```
   SIGNAL   product_counter_env_channel: std_logic_vector(3 downto 0);
   SIGNAL   product_counter_env_send: std_logic;
   SIGNAL   product_counter_env_ack: std_logic;




COMPONENT rrwdecode -- Galileo Local Bus R/W Decoder

    PORT (
        clock, reset:   IN      std_logic;
        a:              IN      std_logic_vector(4 DOWNTO 0);
        csel:           IN      std_logic;      -- CS# chip select
        rw:             IN      std_logic;      -- R/W#
        rdb:            IN      std_logic;      -- RD#
        read:           OUT     std_logic;
        write:          OUT     std_logic_vector(7 DOWNTO 0)
    );
END COMPONENT;

COMPONENT srwdecode -- Spyder Local Bus R/W Decoder
    PORT (
        clock, reset:   IN      std_logic;
        a:              IN      std_logic_vector(31 DOWNTO 2);
        rw:             IN      std_logic;      -- R/W#
        rdb:            IN      std_logic;      -- RD#
        read:           OUT     std_logic;
        write:          OUT     std_logic_vector(7 DOWNTO 0)
    );
END COMPONENT;

COMPONENT vdoutmux8  -- Virtex Data In/Out Mux
    PORT (
        clock, reset:   IN      std_logic;      -- unused
        read:           IN      std_logic;
        addr:           IN      std_logic_vector( 4 DOWNTO 0);
        dinout:         INOUT   std_logic_vector(31 DOWNTO 0);
        din0, din1, din2, din3, din4, din5, din6, din7:
                        IN      std_logic_vector(31 DOWNTO 0);
        dout:           OUT     std_logic_vector(31 DOWNTO 0)
    );
END COMPONENT;

COMPONENT mcsrr -- Master Control/Status + Revision Register
    GENERIC (
        design:         INTEGER := 10;          -- refer to ciop.h
        rev:            INTEGER := 3;
        nctrl:          INTEGER := 2
    );
    PORT (
        clock, reset:   IN      std_logic;
        write:          IN      std_logic;
        din:            IN      std_logic_vector(31 DOWNTO 0);
        ctrl:           BUFFER  std_logic_vector(nctrl-1 DOWNTO 0);
        dout:           OUT     std_logic_vector(31 DOWNTO 0)
    );
END COMPONENT;

COMPONENT count32
    PORT (
        clock, clear:   IN      std_logic;
        cout:           OUT     std_logic_vector (31 DOWNTO 0)
    );
END COMPONENT;
```

```
COMPONENT write_to_sdl_signal
    GENERIC (data_width:    integer := 32);
    PORT (
    clock:      IN      std_logic;
    reset:      IN      std_logic;
    write:      IN      std_logic;
    data_in:    IN      std_logic_vector(data_width-1 downto 0);
    send:       OUT     std_logic;
    ack:        IN      std_logic;
    data_out:   BUFFER  std_logic_vector(data_width-1 downto 0)
    );
END COMPONENT;


COMPONENT sdl_signal_to_env
    GENERIC ( data_width:     integer := 8);
    PORT (
    clock:      IN  std_logic;
    reset:      IN  std_logic;
    send:       IN  std_logic;
    ack:        OUT std_logic;
    data_in:    IN  std_logic_vector(data_width-1 downto 0)
--    new_signal: OUT std_logic;
--   signal_read:IN  std_logic;
   ; data_out:   OUT std_logic_vector(data_width-1 downto 0)
    );
END COMPONENT;



COMPONENT hw_sw_queue_ln_dn_cn -- HW->SW-Queue
   GENERIC (
      queue_width,queue_depth,num_channels: Integer);
   PORT (
      clk:         IN  std_logic;
      reset:       IN  std_logic;
-- Kanaleingang
      send:        IN  std_logic_vector(num_channels-1 downto 0);
      ack:         OUT std_logic_vector(num_channels-1 downto 0);
      data_in:     IN  std_logic_vector(queue_width*num_channels-1 downto 0);
-- Schnittstelle zum HW/SW-Interface
      not_empty:   OUT std_logic;
      pop:         IN  std_logic;
      data_out:    OUT std_logic_vector(queue_width-1 DOWNTO 0)
      );
END COMPONENT;


   COMPONENT sdl_timer
     GENERIC(width : Integer;
          signal_id_width:      integer;
          signal_id:            integer);
     PORT (
     clk,reset  : in std_logic;
     start      : in std_logic;
     ticks      : in std_logic_vector(width - 1 downto 0);
     send       : OUT std_logic;
     ack        : IN std_logic;
     signal_out : OUT std_logic_vector(signal_id_width-1 downto 0));
   end COMPONENT;


COMPONENT edge_to_sdl_signal
    GENERIC (
        signal_id_width:       integer := 10;
        signal_id:             integer := 0;
        kind_of_edge:           integer := 0
        -- 0: send on falling edge
        -- 1: send on rising edge
        -- 2: send on both rising and falling edge
    );
    PORT (
        clock:          IN      std_logic;
        reset:          IN      std_logic;
        signal_in:      IN      std_logic;
        send:           OUT     std_logic;
        ack:            IN      std_logic;
        data_out:       OUT     std_logic_vector(signal_id_width-1 downto 0)
    );
END COMPONENT;


COMPONENT shared_var
   GENERIC (
      data_width,num_channels: Integer);
   PORT (
      clk:              IN      std_logic;
      reset:            IN      std_logic;
-- Kanaleingang
      request:          IN      std_logic_vector(num_channels-1 downto 0);
      grant:            OUT     std_logic_vector(num_channels-1 downto 0);
      write:            IN      std_logic_vector(num_channels-1 downto 0);
      data_in:          IN      std_logic_vector(data_width*num_channels-1 downt
o 0);
      data_out:         OUT     std_logic_vector(data_width-1 DOWNTO 0)
      );
END COMPONENT;


   COMPONENT fsm_process_process1
      port
      (
      clk:      IN      std_logic;
      reset:    IN      std_logic;

      -- Direct Signals

      -- Out-Channels:
      signal_out_motor1:        OUT     std_logic_vector(4-1 DOWNTO 0);
      signal_out_motor2:        OUT     std_logic_vector(4-1 DOWNTO 0);
      signal_out_hwswqueue:     OUT     std_logic_vector(4-1 DOWNTO 0);

      data_out_hwswqueue:   OUT   std_logic_vector(6-1 DOWNTO 0);

      send_out:   OUT   std_logic_vector(num_out_channels_process1-1 downto 0);
      ack_out:   IN   std_logic_vector(num_out_channels_process1-1 downto 0);


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_process1*queue_widt
h_process1-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_process1-1 downto 0);
      ack_in:   OUT   std_logic_vector(num_in_channels_process1-1 downto 0)



       );
   END COMPONENT;


   COMPONENT fsm_process_motor1
      port
      (
      clk:      IN      std_logic;
      reset:    IN      std_logic;

      -- Direct Signals

      -- Out-Channels:
      signal_out_motor2:        OUT     std_logic_vector(4-1 DOWNTO 0);


      send_out:   OUT   std_logic_vector(num_out_channels_motor1-1 downto 0);
      ack_out:   IN   std_logic_vector(num_out_channels_motor1-1 downto 0);


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_motor1*queue_width_
motor1-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_motor1-1 downto 0);
      ack_in:   OUT   std_logic_vector(num_in_channels_motor1-1 downto 0)



       );
   END COMPONENT;


   COMPONENT fsm_process_motor2
      port
      (
      clk:      IN      std_logic;
      reset:    IN      std_logic;

      -- Direct Signals

      -- Out-Channels:
      signal_out_product_counter:       OUT     std_logic_vector(4-1 DOWNTO 0);
      signal_out_hwswqueue:     OUT     std_logic_vector(4-1 DOWNTO 0);


      send_out:   OUT   std_logic_vector(num_out_channels_motor2-1 downto 0);
      ack_out:   IN   std_logic_vector(num_out_channels_motor2-1 downto 0);


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_motor2*queue_width_
motor2-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_motor2-1 downto 0);
      ack_in:   OUT   std_logic_vector(num_in_channels_motor2-1 downto 0)



       );
   END COMPONENT;


   COMPONENT fsm_process_product_counter
      port
      (
      clk:      IN      std_logic;
      reset:    IN      std_logic;

      -- Direct Signals

      -- Out-Channels:


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_product_counter*que
ue_width_product_counter-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_product_counter-1 downto 0
);
      ack_in:   OUT   std_logic_vector(num_in_channels_product_counter-1 downto
0)



       );
   END COMPONENT;



BEGIN



    --------------------------------------------------------------------
    -- SPYDER ADAPTION:
    --------------------------------------------------------------------

    CLOCK <= SYS_CLK;
    GSETRESET <= NOT GNET2_GRESET_0;
    FPGA_LED_0 <= LED;


    LED <= NOT (countout(24) AND countout(23) AND
                countout(22) AND countout(21));

    zero32 <= (OTHERS => '0');

    --------------------------------------------------------------------
    -- Drivers
    --------------------------------------------------------------------

    OE_A  <= '1'; -- disabled
    DIR_A <= '0';

    OE_B  <= '0'; -- enabled
    DIR_B <= '0'; -- input

    OE_C  <= '1'; -- disabled
    DIR_C <= '0';

    OE_D  <= '0'; -- enabled
    DIR_D <= '1'; -- output

    OE_EF  <= '1'; -- disabled
    DIR_E <= '0';
    DIR_F <= '0';

    --------------------------------------------------------------------
    -- Debug Signals
    --------------------------------------------------------------------



    D(0) <= regwrite(2);
    D(1) <= send_process1(0);
    D(2) <= ack_process1(0);
    D(3) <= send_hwswqueue(0);
    D(4) <= ack_hwswqueue(0);

    D(7 downto 5) <= (OTHERS => '0');


    --------------------------------------------------------------------
```

```
    -- REGISTER ACCESS:
    --------------------------------------------------------------------
    -- instantiate the read/write decoder and data in/out multiplexer


    rrwdecodei: rrwdecode
        PORT MAP (
            clock => CLOCK, reset => GSETRESET,
            a => ADDR, csel => CSB, rw => RWB, rdb => RDB,
            read => regread, write => regwrite
        );

    doutmux8i: vdoutmux8
        PORT MAP (
            clock => CLOCK, reset => GSETRESET, -- unused
            read => regread,
            addr => ADDR, dinout => DATA,
            din0 => mcsrrout, din1 => countout,
            din2 => idout2, din3 => idout3,
            din4 => idout4, din5 => idout5,
            din6 => idout6, din7 => idout7,
            dout => idin                        -- internal data in
        );


    --------------------------------------------------------------------
    -- REGISTERS
    --------------------------------------------------------------------
    -- instantiate the master control/status/revision register
    -- with interrupt pending (IP), master interrupt enable (MIE)
    -- and soft reset (SR) signals:




    mcsrrout(31) <= intpend;
    mcsrrout(30) <= not_empty_int;
    mcsrrout(29 DOWNTO 0) <= imcsrrout(29 DOWNTO 0);

    intpend <= not_empty_int; -- alle Interrupts OR
    BINT5 <= NOT (mien AND intpend);    -- assert the interrupt pin


    mien        <= mcsrrctrl(1);
    softreset   <= mcsrrctrl(0);

    mcsrri:     mcsrr
        PORT MAP (CLOCK, GSETRESET, regwrite(0), idin, mcsrrctrl,
        imcsrrout);


    reset <= softreset OR GSETRESET;


    -- instantiate the 32 bit counter (cleared on (soft) reset):
    countclr <= reset;
    counti:     count32
        PORT MAP (CLOCK, countclr, countout);





hwswqueue_i: hw_sw_queue_ln_dn_cn
   GENERIC MAP(
      queue_width => 10,
      queue_depth => 2,
      num_channels => 2)
   PORT MAP(
      clk => CLOCK,
      reset => reset,
-- Kanaleingang
      send => send_hwswqueue,
      ack => ack_hwswqueue,
      data_in => hwswqueue_channel,
-- Schnittstelle zum HW/SW-Interface
      not_empty => not_empty_int,
      pop => regwrite(4),
      data_out => idout4(10-1 downto 0)
      );



   gal_input_process1: write_to_sdl_signal
    GENERIC MAP (data_width => 4)
    PORT MAP(
    clock=>CLOCK,
    reset=>reset,
    write=>regwrite(2),
    data_in=>idin(4-1 downto 0),
    send=>send_process1(0),
    ack=>ack_process1(0),
    data_out => idout2 (4-1 downto 0));

    process1_channel(4-1 downto 0) <= idout2 (4-1 downto 0);





   process1_i: fsm_process_process1
   PORT MAP
   (
      -- input signals
      send_in => send_process1,
      ack_in => ack_process1,
      signal_and_data_in=> process1_channel,
      -- direct signals
        -- timer

      -- output signals


      send_out(0) => send_motor1(0),

      send_out(1) => send_motor2(0),

      send_out(2) => send_hwswqueue(0),

      ack_out(0) => ack_motor1(0),

      ack_out(1) => ack_motor2(0),

      ack_out(2) => ack_hwswqueue(0),


      signal_out_motor1 => motor1_channel(3 downto 0),



      signal_out_motor2 => motor2_channel(3 downto 0),


      data_out_hwswqueue => hwswqueue_channel(5 downto 0),
```

```
      signal_out_hwswqueue => hwswqueue_channel(9 downto 6),



      -- clock, reset
      reset => reset,
      clk => CLOCK
   );





   gal_input_motor1: write_to_sdl_signal
    GENERIC MAP (data_width => 4)
    PORT MAP(
    clock=>CLOCK,
    reset=>reset,
    write=>regwrite(3),
    data_in=>idin(4-1 downto 0),
    send=>send_motor1(1),
    ack=>ack_motor1(1),
    data_out => idout3 (4-1 downto 0));

    motor1_channel(8-1 downto 4) <= idout3 (4-1 downto 0);





   motor1_i: fsm_process_motor1
   PORT MAP
   (
      -- input signals
      send_in => send_motor1,
      ack_in => ack_motor1,
      signal_and_data_in=> motor1_channel,
      -- direct signals
        -- timer

      -- output signals


      send_out(0) => send_motor2(1),

      ack_out(0) => ack_motor2(1),


      signal_out_motor2 => motor2_channel(7 downto 4),



      -- clock, reset
      reset => reset,
      clk => CLOCK
   );






   motor2_i: fsm_process_motor2
   PORT MAP
   (
      -- input signals
      send_in => send_motor2,
      ack_in => ack_motor2,
      signal_and_data_in=> motor2_channel,
      -- direct signals
        -- timer

      -- output signals


      send_out(0) => send_product_counter(0),

      send_out(1) => send_hwswqueue(1),

      ack_out(0) => ack_product_counter(0),

      ack_out(1) => ack_hwswqueue(1),


      signal_out_product_counter => product_counter_channel(3 downto 0),



      signal_out_hwswqueue => hwswqueue_channel(19 downto 16),



      -- clock, reset
      reset => reset,
      clk => CLOCK
   );




   product_counter_i: fsm_process_product_counter
   PORT MAP
   (
      -- input signals
      send_in => send_product_counter,
      ack_in => ack_product_counter,
      signal_and_data_in=> product_counter_channel,
      -- direct signals
        -- timer

      -- output signals



      -- clock, reset
      reset => reset,
      clk => CLOCK
   );



END assembly_line_arch;
```

```
-- Generated by SDL2VHDL Version 0.3pre
-- from source ../../../SDL/assembly_line.sdl at Apr 3, 2001 12:49:46 PM

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
m4_libraries

-- This code is generated for Register-Transfer-Synthesis (RTS)


entity fsm_process_motor1 is
  port
  (
    clk : in Std_Logic;
    done_out : out Std_Logic
    ;
    reset : in Std_Logic;
    m4_port_declarations(m4_process_name)
  );
end fsm_process_motor1;

architecture fsm_process_motor1_arch of fsm_process_motor1 is
m4_signal_declarations
m4_component_declaration
begin
  m4_component_instantiation
  someProcess: process
    type Signed_Vector is array(Integer range <>) of Signed(9 downto 0);
    m4_variable_declaration
    variable dataIn : Std_Logic_Vector(m4_queue_width downto 0);
    variable readyIn : Std_Logic;
    variable done : Std_Logic;
    variable initPhase : Boolean;
    variable currentProcessOrServiceId : Integer range 2 downto 0 ;
    variable transitionPartId : Integer range 0 downto 0;
    variable state_1 : Integer range 3 downto 0;
    variable go_1_b : Std_Logic;
  begin
    m4_every_time
    wait until clk'event and clk = '1';

    if (reset = '1') then
      initPhase := true;
      currentProcessOrServiceId := 1;
      transitionPartId := 0;
      m4_signal_initializations
    elsif (initPhase = true) then
      initPhase := false;
      state_1 := 1;
    else
      m4_receive(readyIn, dataIn);
      if readyIn = '1' then
        case dataIn (m4_header_range) is
          when "1000" =>
            -- Receive signal motor_stop
            case state_1 is
              when 1 =>
                state_1 := 2;
              when 2 | 3 =>
              when others =>
                null;
            end case;
          when "1001" =>
            -- Receive signal motor_start
            case state_1 is
              when 2 =>
```

```
                state_1 := 1;
              when 1 | 3 =>
              when others =>
                null;
            end case;
          when "0111" =>
            -- Receive signal emergency_stop
            case state_1 is
              when 1 =>
                -- Send signal emergency_stop_int with Id 10 to motor2
                m4_send(emergency_stop_int, motor2, 1010);
                state_1 := 3;
              when 2 | 3 =>
              when others =>
                null;
            end case;
          when "0011" =>
            -- Receive signal reset_sig
            case state_1 is
              when 2 | 3 =>
                state_1 := 1;
              when 1 =>
              when others =>
                null;
            end case;
          when others =>
            null;
        end case;
      end if;
      done_out <= done;
    end if;
  end process someProcess;
end fsm_process_motor1_arch;
```

Figure B.2: Server implementation SDL process motor1 (before m4 macro replacement)

```
-- Generated by SDL2VHDL Version 0.3pre
-- from source ../../../SDL/assembly_line.sdl at Apr 3, 2001 12:49:46 PM

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

library SDL;

USE WORK.config.all;
USE SDL.all;
USE ieee.std_logic_unsigned.all;


-- This code is generated for Register-Transfer-Synthesis (RTS)


entity fsm_process_motor2 is
  port
  (
    clk : in Std_Logic;
    done_out : out Std_Logic
    ;
    reset : in Std_Logic;

      -- Direct Signals

      -- Out-Channels:
      signal_out_product_counter:       OUT     std_logic_vector(4-1 DOWNTO 0);
      signal_out_hwswqueue:     OUT     std_logic_vector(4-1 DOWNTO 0);


      send_out:   OUT   std_logic_vector(num_out_channels_motor2-1 downto 0);
      ack_out:   IN   std_logic_vector(num_out_channels_motor2-1 downto 0);


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_motor2*queue_width_
motor2-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_motor2-1 downto 0);
      ack_in:   OUT   std_logic_vector(num_in_channels_motor2-1 downto 0)



  );
end fsm_process_motor2;

architecture fsm_process_motor2_arch of fsm_process_motor2 is

SIGNAL get_sig:         std_logic;
SIGNAL sig_ready:       std_logic;
SIGNAL data:            std_logic_vector(4-1 DOWNTO 0);


COMPONENT queue_l1_dn_cn
   GENERIC (
      queue_width, queue_depth,num_channels: Integer);
   PORT (
      clk:       IN      std_logic;
      reset:     IN      std_logic;
-- Kanaleingang
      send:      IN     std_logic_vector(num_channels-1 downto 0);
      ack:       OUT    std_logic_vector(num_channels-1 downto 0);
      data_in:   IN     std_logic_vector(num_channels*queue_width-1 downto 0);
-- Schnittstelle zum Prozess
      get_sig:   IN      std_logic;
      sig_ready: OUT     std_logic;
```

```
      data:      OUT  std_logic_vector(queue_width-1 DOWNTO 0)
      );
END COMPONENT;

begin

  this_queue: queue_l1_dn_cn
     GENERIC MAP (queue_width=>4,
                  queue_depth=>1,
                  num_channels=>num_in_channels_motor2)
     PORT MAP (clk=>clk, reset=>reset, send=>send_in, ack=>ack_in,
                data_in=>signal_and_data_in,get_sig=>get_sig,
                sig_ready=>sig_ready,data=>data);

  someProcess: process
    type Signed_Vector is array(Integer range <>) of Signed(9 downto 0);

    variable dataIn : Std_Logic_Vector(4 downto 0);
    variable readyIn : Std_Logic;
    variable done : Std_Logic;
    variable initPhase : Boolean;
    variable currentProcessOrServiceId : Integer range 2 downto 0 ;
    variable transitionPartId : Integer range 0 downto 0;
    variable state_1 : Integer range 3 downto 0;
    variable go_1_b : Std_Logic;
  begin

    wait until clk'event and clk = '1';

    if (reset = '1') then
      initPhase := true;
      currentProcessOrServiceId := 1;
      transitionPartId := 0;



    elsif (initPhase = true) then
      initPhase := false;
      state_1 := 1;
    else


      get_sig <= '1';
      if sig_ready = '1' then
         dataIn(4-1 downto 0):= data;
         readyIn := '1';
         get_sig <= '0';
         while sig_ready = '1' loop
            wait until clk='1' and clk'event;
         end loop;
      else
         readyIn := '0';
      end if;
      if readyIn = '1' then
        case dataIn (4-1 downto 0) is
          when "1000" =>
            -- Receive signal motor_stop
            case state_1 is
              when 1 =>
                -- Send signal stopped with Id 6 to chan
                signal_out_hwswqueue(4-1 downto 0) <= "0110";
               send_out(1) <= '1';
               while ack_out(1) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(1) <= '0';
               signal_out_hwswqueue <= "0000";
```

Figure B.3: Server implementation SDL process motor2

```
               wait until clk'event and clk='1';
                -- Send signal stopped with Id 6 to product_counter
                signal_out_product_counter(4-1 downto 0) <= "0110";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
               signal_out_product_counter <= "0000";
               wait until clk'event and clk='1';
                state_1 := 2;
              when 2 | 3 =>
              when others =>
                null;
            end case;
          when "1001" =>
            -- Receive signal motor_start
            case state_1 is
              when 2 =>
                state_1 := 1;
              when 1 | 3 =>
              when others =>
                null;
            end case;
          when "1010" =>
            -- Receive signal emergency_stop_int
            case state_1 is
              when 1 =>
                state_1 := 3;
              when 2 | 3 =>
              when others =>
                null;
            end case;
          when "0011" =>
            -- Receive signal reset_sig
            case state_1 is
              when 2 | 3 =>
                -- Send signal reset_sig with Id 3 to product_counter
                signal_out_product_counter(4-1 downto 0) <= "0011";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
               signal_out_product_counter <= "0000";
               wait until clk'event and clk='1';
                state_1 := 1;
              when 1 =>
              when others =>
                null;
            end case;
          when others =>
            null;
        end case;
      end if;
      done_out <= done;
    end if;
  end process someProcess;
end fsm_process_motor2_arch;
```

```
-- Generated by SDL2VHDL Version 0.3pre
-- from source ../../../SDL/assembly_line.sdl at Apr 3, 2001 12:49:46 PM

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

library SDL;

USE WORK.config.all;
USE SDL.all;
USE ieee.std_logic_unsigned.all;


-- This code is generated for Register-Transfer-Synthesis (RTS)


entity fsm_process_process1 is
  port
  (
    clk : in Std_Logic;
    done_out : out Std_Logic
    ;
    reset : in Std_Logic;

      -- Direct Signals

      -- Out-Channels:
      signal_out_motor1:        OUT     std_logic_vector(4-1 DOWNTO 0);
      signal_out_motor2:        OUT     std_logic_vector(4-1 DOWNTO 0);
      signal_out_hwswqueue:     OUT     std_logic_vector(4-1 DOWNTO 0);

      data_out_hwswqueue:   OUT   std_logic_vector(6-1 DOWNTO 0);

      send_out:   OUT   std_logic_vector(num_out_channels_process1-1 downto 0);
      ack_out:   IN   std_logic_vector(num_out_channels_process1-1 downto 0);


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_process1*queue_widt
h_process1-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_process1-1 downto 0);
      ack_in:   OUT   std_logic_vector(num_in_channels_process1-1 downto 0)



  );
end fsm_process_process1;

architecture fsm_process_process1_arch of fsm_process_process1 is

SIGNAL get_sig:         std_logic;
SIGNAL sig_ready:       std_logic;
SIGNAL data:            std_logic_vector(4-1 DOWNTO 0);


COMPONENT queue_l1_dn_cn
   GENERIC (
      queue_width, queue_depth,num_channels: Integer);
   PORT (
      clk:       IN      std_logic;
      reset:     IN      std_logic;
-- Kanaleingang
      send:      IN     std_logic_vector(num_channels-1 downto 0);
      ack:       OUT    std_logic_vector(num_channels-1 downto 0);
      data_in:   IN     std_logic_vector(num_channels*queue_width-1 downto 0);
-- Schnittstelle zum Prozess
```

```
      get_sig:   IN      std_logic;
      sig_ready: OUT     std_logic;
      data:      OUT  std_logic_vector(queue_width-1 DOWNTO 0)
      );
END COMPONENT;

begin

  this_queue: queue_l1_dn_cn
     GENERIC MAP (queue_width=>4,
                  queue_depth=>1,
                  num_channels=>num_in_channels_process1)
     PORT MAP (clk=>clk, reset=>reset, send=>send_in, ack=>ack_in,
               data_in=>signal_and_data_in,get_sig=>get_sig,
              sig_ready=>sig_ready,data=>data);

  someProcess: process
    type Signed_Vector is array(Integer range <>) of Signed(9 downto 0);

    variable dataIn : Std_Logic_Vector(4 downto 0);
    variable readyIn : Std_Logic;
    variable done : Std_Logic;
    variable initPhase : Boolean;
    variable currentProcessOrServiceId : Integer range 2 downto 0 ;
    variable transitionPartId : Integer range 0 downto 0;
    variable state_1 : Integer range 4 downto 0;
    variable data1_1_i : Unsigned(5 downto 0);
    variable data2_1_i : Unsigned(5 downto 0);
    variable c_true_1_b : Std_Logic;
    variable c_false_1_b : Std_Logic;
  begin

    wait until clk'event and clk = '1';

    if (reset = '1') then
      initPhase := true;
      currentProcessOrServiceId := 1;
      transitionPartId := 0;



    elsif (initPhase = true) then
      initPhase := false;
      data1_1_i := conv_unsigned(0, 6);
      wait until clk'event and clk = '1';
      data2_1_i := conv_unsigned(0, 6);
      wait until clk'event and clk = '1';
      c_true_1_b := '1';
      wait until clk'event and clk = '1';
      c_false_1_b := '0';
      wait until clk'event and clk = '1';
      -- Send signal motor_start with Id 9 to motor1
      signal_out_motor1(4-1 downto 0) <= "1001";
              send_out(0) <= '1';
              while ack_out(0) /= '1' loop
                 wait until clk='1' and clk'event;
              end loop;
              send_out(0) <= '0';
              signal_out_motor1 <= "0000";
              wait until clk'event and clk='1';
      -- Send signal motor_start with Id 9 to motor2
      signal_out_motor2(4-1 downto 0) <= "1001";
              send_out(1) <= '1';
              while ack_out(1) /= '1' loop
                 wait until clk='1' and clk'event;
              end loop;
              send_out(1) <= '0';
```

Figure B.4: Server implementation SDL process process1

```
               signal_out_motor2 <= "0000";
               wait until clk'event and clk='1';
      state_1 := 1;
    else


      get_sig <= '1';
      if sig_ready = '1' then
         dataIn(4-1 downto 0):= data;
         readyIn := '1';
         get_sig <= '0';
         while sig_ready = '1' loop
            wait until clk='1' and clk'event;
         end loop;
      else
         readyIn := '0';
      end if;
      if readyIn = '1' then
        case dataIn (4-1 downto 0) is
          when "0001" =>
            -- Receive signal in1
            case state_1 is
              when 1 =>
                data1_1_i := data1_1_i + conv_unsigned(1, 6);
                wait until clk'event and clk = '1';
                if data1_1_i = conv_unsigned(4, 6) then
                  -- Send signal motor_stop with Id 8 to motor1
                  signal_out_motor1(4-1 downto 0) <= "1000";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
               signal_out_motor1 <= "0000";
               wait until clk'event and clk='1';
                  data1_1_i := conv_unsigned(0, 6);
                  wait until clk'event and clk = '1';
                  -- Send signal out1 with Id 4 to chan
                  signal_out_hwswqueue(4-1 downto 0) <= "0100";
               data_out_hwswqueue(6-1 downto 0)
                  <= conv_std_logic_vector(data1_1_i, 6);
               send_out(2) <= '1';
               while ack_out(2) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(2) <= '0';
               data_out_hwswqueue <= "000000";
               wait until clk'event and clk='1';
                  state_1 := 2;
                else
                  -- Send signal out1 with Id 4 to chan
                  signal_out_hwswqueue(4-1 downto 0) <= "0100";
               data_out_hwswqueue(6-1 downto 0)
                  <= conv_std_logic_vector(data1_1_i, 6);
               send_out(2) <= '1';
               while ack_out(2) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(2) <= '0';
               data_out_hwswqueue <= "000000";
               wait until clk'event and clk='1';
                  state_1 := 1;
                end if;
              when 3 =>
                data1_1_i := data1_1_i + conv_unsigned(1, 6);
                wait until clk'event and clk = '1';
                if data1_1_i = conv_unsigned(4, 6) then
```

```
                  -- Send signal motor_stop with Id 8 to motor1
                  signal_out_motor1(4-1 downto 0) <= "1000";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
               signal_out_motor1 <= "0000";
               wait until clk'event and clk='1';
                  data1_1_i := conv_unsigned(0, 6);
                  wait until clk'event and clk = '1';
                  -- Send signal out1 with Id 4 to chan
                  signal_out_hwswqueue(4-1 downto 0) <= "0100";
               data_out_hwswqueue(6-1 downto 0)
                  <= conv_std_logic_vector(data1_1_i, 6);
               send_out(2) <= '1';
               while ack_out(2) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(2) <= '0';
               data_out_hwswqueue <= "000000";
               wait until clk'event and clk='1';
                  state_1 := 4;
                else
                  -- Send signal out1 with Id 4 to chan
                  signal_out_hwswqueue(4-1 downto 0) <= "0100";
               data_out_hwswqueue(6-1 downto 0)
                  <= conv_std_logic_vector(data1_1_i, 6);
               send_out(2) <= '1';
               while ack_out(2) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(2) <= '0';
               data_out_hwswqueue <= "000000";
               wait until clk'event and clk='1';
                end if;
              when 2 | 4 =>
              when others =>
                null;
            end case;
          when "0010" =>
            -- Receive signal in2
            case state_1 is
              when 1 =>
                data2_1_i := data2_1_i + conv_unsigned(1, 6);
                wait until clk'event and clk = '1';
                if data2_1_i = conv_unsigned(3, 6) then
                  -- Send signal motor_stop with Id 8 to motor2
                  signal_out_motor2(4-1 downto 0) <= "1000";
               send_out(1) <= '1';
               while ack_out(1) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(1) <= '0';
               signal_out_motor2 <= "0000";
               wait until clk'event and clk='1';
                  data2_1_i := conv_unsigned(0, 6);
                  wait until clk'event and clk = '1';
                  -- Send signal out2 with Id 5 to chan
                  signal_out_hwswqueue(4-1 downto 0) <= "0101";
               data_out_hwswqueue(6-1 downto 0)
                  <= conv_std_logic_vector(data2_1_i, 6);
               send_out(2) <= '1';
               while ack_out(2) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(2) <= '0';
```

```
         data_out_hwswqueue <= "000000";
         wait until clk'event and clk='1';
            state_1 := 3;
          else
            -- Send signal out2 with Id 5 to chan
            signal_out_hwswqueue(4-1 downto 0) <= "0101";
         data_out_hwswqueue(6-1 downto 0)
            <= conv_std_logic_vector(data2_1_i, 6);
         send_out(2) <= '1';
         while ack_out(2) /= '1' loop
            wait until clk='1' and clk'event;
         end loop;
         send_out(2) <= '0';
         data_out_hwswqueue <= "000000";
         wait until clk'event and clk='1';
            state_1 := 1;
          end if;
        when 2 =>
          data2_1_i := data2_1_i + conv_unsigned(1, 6);
          wait until clk'event and clk = '1';
          if data2_1_i = conv_unsigned(3, 6) then
            -- Send signal motor_stop with Id 8 to motor2
            signal_out_motor2(4-1 downto 0) <= "1000";
         send_out(1) <= '1';
         while ack_out(1) /= '1' loop
            wait until clk='1' and clk'event;
         end loop;
         send_out(1) <= '0';
         signal_out_motor2 <= "0000";
         wait until clk'event and clk='1';
            data2_1_i := conv_unsigned(0, 6);
            wait until clk'event and clk = '1';
            -- Send signal out2 with Id 5 to chan
            signal_out_hwswqueue(4-1 downto 0) <= "0101";
         data_out_hwswqueue(6-1 downto 0)
            <= conv_std_logic_vector(data2_1_i, 6);
         send_out(2) <= '1';
         while ack_out(2) /= '1' loop
            wait until clk='1' and clk'event;
         end loop;
         send_out(2) <= '0';
         data_out_hwswqueue <= "000000";
         wait until clk'event and clk='1';
            state_1 := 4;
          else
            -- Send signal out2 with Id 5 to chan
            signal_out_hwswqueue(4-1 downto 0) <= "0101";
         data_out_hwswqueue(6-1 downto 0)
            <= conv_std_logic_vector(data2_1_i, 6);
         send_out(2) <= '1';
         while ack_out(2) /= '1' loop
            wait until clk='1' and clk'event;
         end loop;
         send_out(2) <= '0';
         data_out_hwswqueue <= "000000";
         wait until clk'event and clk='1';
          end if;
        when 3 | 4 =>
        when others =>
          null;
      end case;
    when "0011" =>
      -- Receive signal reset_sig
      case state_1 is
        when 1 | 2 | 3 | 4 =>
          -- Send signal reset_sig with Id 3 to motor1
          signal_out_motor1(4-1 downto 0) <= "0011";
```

```
              send_out(0) <= '1';
              while ack_out(0) /= '1' loop
                 wait until clk='1' and clk'event;
              end loop;
              send_out(0) <= '0';
              signal_out_motor1 <= "0000";
              wait until clk'event and clk='1';
               -- Send signal reset_sig with Id 3 to motor2
               signal_out_motor2(4-1 downto 0) <= "0011";
              send_out(1) <= '1';
              while ack_out(1) /= '1' loop
                 wait until clk='1' and clk'event;
              end loop;
              send_out(1) <= '0';
              signal_out_motor2 <= "0000";
              wait until clk'event and clk='1';
               state_1 := 1;
             when others =>
               null;
           end case;
         when others =>
           null;
       end case;
     end if;
     done_out <= done;
   end if;
  end process someProcess;
end fsm_process_process1_arch;
```

```
-- Generated by SDL2VHDL Version 0.3pre
-- from source ../../../SDL/assembly_line.sdl at Feb 1, 2001 7:50:28 PM

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

library SDL;

USE WORK.config.all;
USE SDL.all;
USE WORK.assembly_line_package.all;
USE ieee.std_logic_unsigned.all;

entity at_emergency_stop is
  port
    (
     clk,reset          : in std_logic;

      -- Direct Signals


      -- Shared Data Access


      request_process1:  OUT   std_logic;
      grant_process1:  IN   std_logic;
      write_process1:  OUT   std_logic;
      shared_data_in_process1: IN std_logic_vector(17-1 DOWNTO 0);
      shared_data_out_process1: OUT std_logic_vector(17-1 DOWNTO 0);


      request_motor1:  OUT   std_logic;
      grant_motor1:  IN   std_logic;
      write_motor1:  OUT   std_logic;
      shared_data_in_motor1: IN std_logic_vector(3-1 DOWNTO 0);
      shared_data_out_motor1: OUT std_logic_vector(3-1 DOWNTO 0);


      request_motor2:  OUT   std_logic;
      grant_motor2:  IN   std_logic;
      write_motor2:  OUT   std_logic;
      shared_data_in_motor2: IN std_logic_vector(3-1 DOWNTO 0);
      shared_data_out_motor2: OUT std_logic_vector(3-1 DOWNTO 0);


      request_product_counter:  OUT   std_logic;
      grant_product_counter:  IN   std_logic;
      write_product_counter:  OUT   std_logic;
      shared_data_in_product_counter: IN std_logic_vector(7-1 DOWNTO 0);
      shared_data_out_product_counter: OUT std_logic_vector(7-1 DOWNTO 0);




      -- Out-Channels:


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_at_emergency_stop*q
ueue_width_at_emergency_stop-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_at_emergency_stop-1 downto
 0);
      ack_in:   OUT   std_logic_vector(num_in_channels_at_emergency_stop-1 downt
o 0)
```

```
     );
end at_emergency_stop;


architecture assembly_line_arch of at_emergency_stop is


SIGNAL get_sig:         std_logic;
SIGNAL sig_ready:       std_logic;
SIGNAL data:            std_logic_vector(4-1 DOWNTO 0);


COMPONENT queue_l1_dn_cn
   GENERIC (
      queue_width, queue_depth,num_channels: Integer);
   PORT (
      clk:       IN      std_logic;
      reset:     IN      std_logic;
-- Kanaleingang
      send:      IN     std_logic_vector(num_channels-1 downto 0);
      ack:       OUT    std_logic_vector(num_channels-1 downto 0);
      data_in:   IN     std_logic_vector(num_channels*queue_width-1 downto 0);
-- Schnittstelle zum Prozess
      get_sig:   IN      std_logic;
      sig_ready: OUT     std_logic;
      data:      OUT  std_logic_vector(queue_width-1 DOWNTO 0)
      );
END COMPONENT;



begin

  this_queue: queue_l1_dn_cn
     GENERIC MAP (queue_width=>4,
                  queue_depth=>1,
                  num_channels=>num_in_channels_at_emergency_stop)
     PORT MAP (clk=>clk, reset=>reset, send=>send_in, ack=>ack_in,
                data_in=>signal_and_data_in,get_sig=>get_sig,
               sig_ready=>sig_ready,data=>data);


  main: process

VARIABLE tmp:           std_logic_vector (32-1 downto 0);


    variable dataIn : Std_Logic_Vector(4 downto 0);
    variable readyIn : Std_Logic;
    variable initPhase : Boolean;

    -- sdl states
    variable motor1State        : Integer range 3 downto 0;
    variable motor2State        : Integer range 3 downto 0;
    variable process1State      : Integer range 4 downto 0;
    variable product_counterState       : Integer range 1 downto 0;
    -- sdl variables
    variable motor1vars : motor1VarsType;
    variable motor2vars : motor2VarsType;
    variable process1vars       : process1VarsType;
    variable product_countervars        : product_counterVarsType;
    -- temp variables for conditions
    variable condition_1: Boolean;
    variable condition_2: Boolean;
```

Figure B.5: Parallel activity thread implementation of AT emergency_stop

```
    variable condition_3: Boolean;
    variable condition_4: Boolean;
  begin
    if (reset = '1') then
      initPhase := true;

tmp := (others => '0');

    elsif (initPhase = true) then
    initPhase := false;
      motor1State := 1;
      -- release process data

                shared_data_out_motor1(2) <= motor1Vars.go_b;
                tmp := conv_std_logic_vector(motor1State,32);
                shared_data_out_motor1(1 downto 0) <= tmp(2-1 downto 0);



                write_motor1 <= '1';
                wait until clk='1' and clk'event;
                write_motor1 <= '0';
                request_motor1 <= '0';
      -- Now process output statements
      motor2State := 1;
      -- release process data

                shared_data_out_motor2(2) <= motor2Vars.go_b;
                tmp := conv_std_logic_vector(motor2State,32);
                shared_data_out_motor2(1 downto 0) <= tmp(2-1 downto 0);



                write_motor2 <= '1';
                wait until clk='1' and clk'event;
                write_motor2 <= '0';
                request_motor2 <= '0';
      -- Now process output statements
      process1Vars.data1_i := 0;
      process1Vars.data2_i := 0;
      process1Vars.c_true_b := '1';
      process1Vars.c_false_b := '0';
      process1State := 1;
      -- release process data

                tmp := conv_std_logic_vector(process1Vars.data1_i,32);
                shared_data_out_process1(16 downto 11) <= tmp(6-1 downto 0);
                tmp := conv_std_logic_vector(process1Vars.data2_i,32);
                shared_data_out_process1(10 downto 5) <= tmp(6-1 downto 0);
                shared_data_out_process1(4) <= process1Vars.c_true_b;
                shared_data_out_process1(3) <= process1Vars.c_false_b;
                tmp := conv_std_logic_vector(process1State,32);
                shared_data_out_process1(2 downto 0) <= tmp(3-1 downto 0);






                write_process1 <= '1';
                wait until clk='1' and clk'event;
                write_process1 <= '0';
                request_process1 <= '0';
      -- Now process output statements
      -- process1 sends signal motor_start to process motor1
      -- Trace of Process "motor1":
      -- obtain process data
```

```
          request_motor1 <= '1';
          while grant_motor1 /= '1' loop
             wait until clk='1' and clk'event;
          end loop;

          motor1Vars.go_b := shared_data_in_motor1(2);

          motor1State := conv_integer(shared_data_in_motor1(1 downto 0));



    -- Process motor1 accepts signal motor_start in state stopped
    if ((motor1State = 2)) then
      motor1State := 1;
      -- release process data

          shared_data_out_motor1(2) <= motor1Vars.go_b;
          tmp := conv_std_logic_vector(motor1State,32);
          shared_data_out_motor1(1 downto 0) <= tmp(2-1 downto 0);



          write_motor1 <= '1';
          wait until clk='1' and clk'event;
          write_motor1 <= '0';
          request_motor1 <= '0';
      -- now process output statements
    else
      -- release process data

          shared_data_out_motor1(2) <= motor1Vars.go_b;
          tmp := conv_std_logic_vector(motor1State,32);
          shared_data_out_motor1(1 downto 0) <= tmp(2-1 downto 0);



          write_motor1 <= '1';
          wait until clk='1' and clk'event;
          write_motor1 <= '0';
          request_motor1 <= '0';
    end if;
    -- end of trace of process motor1
    -- process1 sends signal motor_start to process motor2
    -- Trace of Process "motor2":
    -- obtain process data

          request_motor2 <= '1';
          while grant_motor2 /= '1' loop
             wait until clk='1' and clk'event;
          end loop;

          motor2Vars.go_b := shared_data_in_motor2(2);

          motor2State := conv_integer(shared_data_in_motor2(1 downto 0));



    -- Process motor2 accepts signal motor_start in state stopped
    if ((motor2State = 2)) then
      motor2State := 1;
      -- release process data

          shared_data_out_motor2(2) <= motor2Vars.go_b;
          tmp := conv_std_logic_vector(motor2State,32);
          shared_data_out_motor2(1 downto 0) <= tmp(2-1 downto 0);
    motor1Vars.go_b := shared_data_in_motor1(2);

    motor1State := conv_integer(shared_data_in_motor1(1 downto 0));



-- Process motor1 accepts signal emergency_stop in state go
if ((motor1State = 1)) then
  -- Signal parameter "getSignalParameter" is ignored.
  motor1State := 3;
  -- release process data

    shared_data_out_motor1(2) <= motor1Vars.go_b;
    tmp := conv_std_logic_vector(motor1State,32);
    shared_data_out_motor1(1 downto 0) <= tmp(2-1 downto 0);



    write_motor1 <= '1';
    wait until clk='1' and clk'event;
    write_motor1 <= '0';
    request_motor1 <= '0';
  -- now process output statements
  -- motor1 sends signal emergency_stop_int to process motor2
  -- Trace of Process "motor2":
  -- obtain process data

    request_motor2 <= '1';
    while grant_motor2 /= '1' loop
       wait until clk='1' and clk'event;
    end loop;

    motor2Vars.go_b := shared_data_in_motor2(2);

    motor2State := conv_integer(shared_data_in_motor2(1 downto 0));



  -- Process motor2 accepts signal emergency_stop_int in state go
  if ((motor2State = 1)) then
    motor2State := 3;
    -- release process data

    shared_data_out_motor2(2) <= motor2Vars.go_b;
    tmp := conv_std_logic_vector(motor2State,32);
    shared_data_out_motor2(1 downto 0) <= tmp(2-1 downto 0);



    write_motor2 <= '1';
    wait until clk='1' and clk'event;
    write_motor2 <= '0';
    request_motor2 <= '0';
    -- now process output statements
  else
    -- release process data

    shared_data_out_motor2(2) <= motor2Vars.go_b;
    tmp := conv_std_logic_vector(motor2State,32);
    shared_data_out_motor2(1 downto 0) <= tmp(2-1 downto 0);



    write_motor2 <= '1';
    wait until clk='1' and clk'event;
    write_motor2 <= '0';
    request_motor2 <= '0';
```

```

                write_motor2 <= '1';
                wait until clk='1' and clk'event;
                write_motor2 <= '0';
                request_motor2 <= '0';
        -- now process output statements
      else
        -- release process data

                shared_data_out_motor2(2) <= motor2Vars.go_b;
                tmp := conv_std_logic_vector(motor2State,32);
                shared_data_out_motor2(1 downto 0) <= tmp(2-1 downto 0);



                write_motor2 <= '1';
                wait until clk='1' and clk'event;
                write_motor2 <= '0';
                request_motor2 <= '0';
      end if;
      -- end of trace of process motor2
      product_counterVars.count_i := 0;
      product_counterState := 1;
      -- release process data

                tmp := conv_std_logic_vector(product_counterVars.count_i,32);
                shared_data_out_product_counter(6 downto 1) <= tmp(6-1 downto 0)
;
                tmp := conv_std_logic_vector(product_counterState,32);
                shared_data_out_product_counter(0 downto 0) <= tmp(1-1 downto 0)
;



                write_product_counter <= '1';
                wait until clk='1' and clk'event;
                write_product_counter <= '0';
                request_product_counter <= '0';
      -- Now process output statements
    else


      get_sig <= '1';
      if sig_ready = '1' then
         dataIn(4-1 downto 0):= data;
         readyIn := '1';
         get_sig <= '0';
         while sig_ready = '1' loop
            wait until clk='1' and clk'event;
         end loop;
      else
         readyIn := '0';
      end if;
      if readyIn = '1' then
        case conv_integer(dataIn (4-1 downto 0)) is
          -- Trace of Signal "emergency_stop" with id 7:
          -- Destination is motor1
          when emergency_stopSignal =>
            -- Trace of Process "motor1":
            -- obtain process data

                request_motor1 <= '1';
                while grant_motor1 /= '1' loop
                   wait until clk='1' and clk'event;
                end loop;
```

```
              end if;
              -- end of trace of process motor2
            else
              -- release process data

                shared_data_out_motor1(2) <= motor1Vars.go_b;
                tmp := conv_std_logic_vector(motor1State,32);
                shared_data_out_motor1(1 downto 0) <= tmp(2-1 downto 0);



                write_motor1 <= '1';
                wait until clk='1' and clk'event;
                write_motor1 <= '0';
                request_motor1 <= '0';
            end if;
            -- end of trace of process motor1
          when others =>
        end case;
      end if;
    end if;
    wait until clk = '1' and clk'event;
  end process main;

end assembly_line_arch;
```

```
-- Generated by SDL2VHDL Version 0.3pre
-- from source ../../../SDL/assembly_line.sdl at Feb 1, 2001 10:45:15 AM

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

library SDL;

USE WORK.config.all;
USE SDL.all;
USE WORK.assembly_line_package.all;
USE ieee.std_logic_unsigned.all;

entity at_assembly_line is
  port
    (
     clk,reset          : in std_logic;

      -- Direct Signals

      -- Out-Channels: Env
      signal_out_Env:   OUT     std_logic_vector(4-1 DOWNTO 0);

      data_out_Env:   OUT   std_logic_vector(6-1 DOWNTO 0);

      send_out:   OUT   std_logic_vector(num_out_channels_at_assembly_line-1 dow
nto 0);
      ack_out:   IN   std_logic_vector(num_out_channels_at_assembly_line-1 downt
o 0);


      -- In-Channels
      signal_and_data_in:   std_logic_vector(num_in_channels_at_assembly_line*qu
eue_width_at_assembly_line-1 downto 0);
      send_in:  IN   std_logic_vector(num_in_channels_at_assembly_line-1 downto
0);
      ack_in:   OUT   std_logic_vector(num_in_channels_at_assembly_line-1 downto
 0)




     );
end at_assembly_line;


architecture assembly_line_arch of at_assembly_line is


SIGNAL get_sig:         std_logic;
SIGNAL sig_ready:       std_logic;
SIGNAL data:            std_logic_vector(4-1 DOWNTO 0);


COMPONENT queue_l1_dn_cn
   GENERIC (
      queue_width, queue_depth,num_channels: Integer);
   PORT (
      clk:       IN      std_logic;
      reset:     IN      std_logic;
-- Kanaleingang
      send:      IN     std_logic_vector(num_channels-1 downto 0);
      ack:       OUT    std_logic_vector(num_channels-1 downto 0);
      data_in:   IN     std_logic_vector(num_channels*queue_width-1 downto 0);
-- Schnittstelle zum Prozess
      get_sig:   IN      std_logic;
```

```
      sig_ready: OUT     std_logic;
      data:      OUT  std_logic_vector(queue_width-1 DOWNTO 0)
      );
END COMPONENT;



begin

  this_queue: queue_l1_dn_cn
     GENERIC MAP (queue_width=>4,
                  queue_depth=>1,
                  num_channels=>num_in_channels_at_assembly_line)
     PORT MAP (clk=>clk, reset=>reset, send=>send_in, ack=>ack_in,
               data_in=>signal_and_data_in,get_sig=>get_sig,
              sig_ready=>sig_ready,data=>data);


  main: process

VARIABLE tmp:           std_logic_vector (32-1 downto 0);


    variable dataIn : Std_Logic_Vector(4 downto 0);
    variable readyIn : Std_Logic;
    variable initPhase : Boolean;

    -- sdl states
    variable motor1State        : Integer range 3 downto 0;
    variable motor2State        : Integer range 3 downto 0;
    variable process1State      : Integer range 4 downto 0;
    variable product_counterState       : Integer range 1 downto 0;
    -- sdl variables
    variable motor1vars : motor1VarsType;
    variable motor2vars : motor2VarsType;
    variable process1vars       : process1VarsType;
    variable product_countervars        : product_counterVarsType;
    -- temp variables for conditions
    variable condition_1: Boolean;
    variable condition_2: Boolean;
    variable condition_3: Boolean;
    variable condition_4: Boolean;
  begin
    if (reset = '1') then
      initPhase := true;

tmp := (others => '0');

    elsif (initPhase = true) then
    initPhase := false;
      motor1State := 1;
      -- Now process output statements
      motor2State := 1;
      -- Now process output statements
      process1Vars.data1_i := 0;
      process1Vars.data2_i := 0;
      process1Vars.c_true_b := '1';
      process1Vars.c_false_b := '0';
      process1State := 1;
      -- Now process output statements
      -- process1 sends signal motor_start to process motor1
      -- Trace of Process "motor1":
      -- Process motor1 accepts signal motor_start in state stopped
      if ((motor1State = 2)) then
        motor1State := 1;
        -- now process output statements
      else
```

Figure B.6: Serial activity thread implementation (EFSM)

```
    end if;
    -- end of trace of process motor1
    -- process1 sends signal motor_start to process motor2
    -- Trace of Process "motor2":
    -- Process motor2 accepts signal motor_start in state stopped
    if ((motor2State = 2)) then
      motor2State := 1;
      -- now process output statements
    else
    end if;
    -- end of trace of process motor2
    product_counterVars.count_i := 0;
    product_counterState := 1;
    -- Now process output statements
  else


    get_sig <= '1';
    if sig_ready = '1' then
       dataIn(4-1 downto 0):= data;
       readyIn := '1';
       get_sig <= '0';
       while sig_ready = '1' loop
          wait until clk='1' and clk'event;
       end loop;
    else
       readyIn := '0';
    end if;
    if readyIn = '1' then
      case conv_integer(dataIn (4-1 downto 0)) is
        -- Trace of Signal "emergency_stop" with id 7:
        -- Destination is motor1
        when emergency_stopSignal =>
          -- Trace of Process "motor1":
          -- Process motor1 accepts signal emergency_stop in state go
          if ((motor1State = 1)) then
            -- Signal parameter "getSignalParameter" is ignored.
            motor1State := 3;
            -- now process output statements
            -- motor1 sends signal emergency_stop_int to process motor2
            -- Trace of Process "motor2":
            -- Process motor2 accepts signal emergency_stop_int in state go
            if ((motor2State = 1)) then
              motor2State := 3;
              -- now process output statements
            else
            end if;
            -- end of trace of process motor2
          else
          end if;
          -- end of trace of process motor1

        -- Trace of Signal "in1" with id 1:
        -- Destination is process1
        when in1Signal =>
          -- Trace of Process "process1":
          -- Process process1 accepts signal in1 in state empty
          if ((process1State = 1)) then
            -- Signal parameter "getSignalParameter" is ignored.
            process1Vars.data1_i := process1Vars.data1_i + 1;
            condition_1 := process1Vars.data1_i = 4;
            case condition_1 is
              when true =>
                process1Vars.data1_i := 0;
                process1State := 2;
              when false =>
                process1State := 1;
```

```
                when others =>
              end case;
              -- now process output statements
              case condition_1 is
                when true =>
                  -- process1 sends signal motor_stop to process motor1
                  -- Trace of Process "motor1":
                  -- Process motor1 accepts signal motor_stop in state go
                  if ((motor1State = 1)) then
                    motor1State := 2;
                    -- now process output statements
                  else
                  end if;
                  -- end of trace of process motor1
                  -- Signal out1 is sent to environment.
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data1_i, 6);
               signal_out_Env(4-1 downto 0) <= "0100";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when false =>
                  -- Signal out1 is sent to environment.
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data1_i, 6);
               signal_out_Env(4-1 downto 0) <= "0100";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when others =>
              end case;
            -- Process process1 accepts signal in1 in state a1_empty_2_full
            elsif ((process1State = 3)) then
              -- Signal parameter "getSignalParameter" is ignored.
              process1Vars.data1_i := process1Vars.data1_i + 1;
              condition_3 := process1Vars.data1_i = 4;
              case condition_3 is
                when true =>
                  process1Vars.data1_i := 0;
                  process1State := 4;
                when false =>
                  -- no state change for process process1
                when others =>
              end case;
              -- now process output statements
              case condition_3 is
                when true =>
                  -- process1 sends signal motor_stop to process motor1
                  -- Trace of Process "motor1":
                  -- Process motor1 accepts signal motor_stop in state go
                  if ((motor1State = 1)) then
                    motor1State := 2;
                    -- now process output statements
                  else
                  end if;
                  -- end of trace of process motor1
                  -- Signal out1 is sent to environment.
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data1_i, 6);
               signal_out_Env(4-1 downto 0) <= "0100";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when false =>
                  -- Signal out1 is sent to environment.
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data1_i, 6);
               signal_out_Env(4-1 downto 0) <= "0100";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when others =>
              end case;
            else
            end if;
            -- end of trace of process process1

          -- Trace of Signal "in2" with id 2:
          -- Destination is process1
          when in2Signal =>
            -- Trace of Process "process1":
            -- Process process1 accepts signal in2 in state empty
            if ((process1State = 1)) then
              -- Signal parameter "getSignalParameter" is ignored.
              process1Vars.data2_i := process1Vars.data2_i + 1;
              condition_2 := process1Vars.data2_i = 3;
              case condition_2 is
                when true =>
                  process1Vars.data2_i := 0;
                  process1State := 3;
                when false =>
                  process1State := 1;
                when others =>
              end case;
              -- now process output statements
              case condition_2 is
                when true =>
                  -- process1 sends signal motor_stop to process motor2
                  -- Trace of Process "motor2":
                  -- Process motor2 accepts signal motor_stop in state go
                  if ((motor2State = 1)) then
                    motor2State := 2;
                    -- now process output statements
                    -- Signal stopped is sent to environment.
                    signal_out_Env(4-1 downto 0) <= "0110";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                    -- motor2 sends signal stopped to process product_counter
                    -- Trace of Process "product_counter":
                    -- Process product_counter accepts signal stopped in state r
un
                    if ((product_counterState = 1)) then
                      product_counterVars.count_i := product_counterVars.count_i
 + 1;
                      product_counterState := 1;
                      -- now process output statements
                    else
                    end if;
                    -- end of trace of process product_counter
                  else
                  end if;
                  -- end of trace of process motor2
                  -- Signal out2 is sent to environment.
```

```
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data2_i, 6);
               signal_out_Env(4-1 downto 0) <= "0101";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when false =>
                  -- Signal out2 is sent to environment.
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data2_i, 6);
               signal_out_Env(4-1 downto 0) <= "0101";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when others =>
              end case;
            -- Process process1 accepts signal in2 in state a1_full_2_empty
            elsif ((process1State = 2)) then
              -- Signal parameter "getSignalParameter" is ignored.
              process1Vars.data2_i := process1Vars.data2_i + 1;
              condition_4 := process1Vars.data2_i = 3;
              case condition_4 is
                when true =>
                  process1Vars.data2_i := 0;
                  process1State := 4;
                when false =>
                  -- no state change for process process1
                when others =>
              end case;
              -- now process output statements
              case condition_4 is
                when true =>
                  -- process1 sends signal motor_stop to process motor2
                  -- Trace of Process "motor2":
                  -- Process motor2 accepts signal motor_stop in state go
                  if ((motor2State = 1)) then
                    motor2State := 2;
                    -- now process output statements
                    -- Signal stopped is sent to environment.
                    signal_out_Env(4-1 downto 0) <= "0110";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                    -- motor2 sends signal stopped to process product_counter
                    -- Trace of Process "product_counter":
                    -- Process product_counter accepts signal stopped in state r
un
                    if ((product_counterState = 1)) then
                      product_counterVars.count_i := product_counterVars.count_i
 + 1;
                      product_counterState := 1;
                      -- now process output statements
                    else
                    end if;
                    -- end of trace of process product_counter
                  else
                  end if;
                  -- end of trace of process motor2
                  -- Signal out2 is sent to environment.
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data2_i, 6);
```

```
               signal_out_Env(4-1 downto 0) <= "0101";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when false =>
                  -- Signal out2 is sent to environment.
                  data_out_Env(6-1 downto 0) <= conv_std_logic_vector(process1Va
rs.data2_i, 6);
               signal_out_Env(4-1 downto 0) <= "0101";
               send_out(0) <= '1';
               while ack_out(0) /= '1' loop
                  wait until clk='1' and clk'event;
               end loop;
               send_out(0) <= '0';
                when others =>
              end case;
            else
            end if;
            -- end of trace of process process1

          -- Trace of Signal "reset_sig" with id 3:
          -- Destination is process1
          when reset_sigSignal =>
            -- Trace of Process "process1":
            -- Process process1 accepts signal reset_sig in state *
            -- signal is accepted in any state
            -- Signal parameter "getSignalParameter" is ignored.
            process1State := 1;
            -- now process output statements
            -- process1 sends signal reset_sig to process motor1
            -- Trace of Process "motor1":
            -- Process motor1 accepts signal reset_sig in state stopped
            if ((motor1State = 2) OR (motor1State = 3)) then
              motor1State := 1;
              -- now process output statements
            else
            end if;
            -- end of trace of process motor1
            -- process1 sends signal reset_sig to process motor2
            -- Trace of Process "motor2":
            -- Process motor2 accepts signal reset_sig in state stopped
            if ((motor2State = 2) OR (motor2State = 3)) then
              motor2State := 1;
              -- now process output statements
              -- motor2 sends signal reset_sig to process product_counter
              -- Trace of Process "product_counter":
              -- Process product_counter accepts signal reset_sig in state run
              if ((product_counterState = 1)) then
                product_counterVars.count_i := 0;
                product_counterState := 1;
                -- now process output statements
              else
              end if;
              -- end of trace of process product_counter
            else
            end if;
            -- end of trace of process motor2
            -- end of trace of process process1

          when others =>
        end case;
      end if;
    end if;
    wait until clk = '1' and clk'event;
  end process main;
```

```
end assembly_line_arch;
```

Appendix C

SDL Run-Time Components

Here, the VHDL source code of the run-time components used in the application examples is shown:

- Message queue length 0 (figure C.1)
- Message queue length 1 (figure C.2)
- Message queue length n (figure C.3)
- FPGA bus interface (figure C.4)
- SDL signal input components (figure C.5)
- SDL signal output component (figure C.6)
- SDL timer (figure C.7)
- Shared data component for parallel ATM (figure C.8)

```
------------------------------------------------------------------------------
-----------------------------------------------------------------------
-- Message queue length 0
--
-- Adapts the signal channel protocol to the handshake expected
-- by the EFSM; multiplexes the data from the signal channels
--
-----------------------------------------------------------------------



LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY queue_l0_dn_cn IS
   GENERIC (
      queue_width,queue_depth,num_channels: Integer);
   PORT (
      clk:              IN      std_logic;
      reset:            IN      std_logic;
-- Kanaleingang
      send:             IN      std_logic_vector(num_channels-1 downto 0);
      ack:              OUT     std_logic_vector(num_channels-1 downto 0);
      data_in:          IN      std_logic_vector(queue_width*num_channels-1 down
to 0);
-- Schnittstelle zum Prozess
      get_sig:          IN      std_logic;
      sig_ready:        OUT     std_logic;
      data:             OUT     std_logic_vector(queue_width-1 DOWNTO 0)
      );
END queue_l0_dn_cn;

ARCHITECTURE queue_a OF queue_l0_dn_cn IS
   TYPE local_array is array (natural range num_channels-1 downto 0)
                       of std_logic_vector(queue_width-1 downto 0);
   SIGNAL din_int:      local_array;
   SIGNAL ack_internal: std_logic_vector(num_channels-1 downto 0);
   SIGNAL sig_ready_internal: std_logic;
   SIGNAL channel: natural range num_channels-1 DOWNTO 0;
BEGIN

   conv_data_in: FOR i in 0 TO num_channels-1 GENERATE
               din_int(i) <= data_in((i+1)*queue_width-1 downto i*queue_width);

   END GENERATE;

   ack <= ack_internal;

   sig_ready <= send(channel);
   data <= din_int(channel);

   receive_process: PROCESS (clk, reset)
      VARIABLE next_channel: boolean;
      VARIABLE channel_tmp: natural range num_channels-1 DOWNTO 0;
   BEGIN

    IF reset = '1' THEN

      ack_internal <= (OTHERS => '0');
      next_channel := true;
      channel <= 0;
      channel_tmp := 0;
      sig_ready_internal <= '0';

   ELSIF clk'event and clk= '1' THEN

         IF next_channel = true THEN
            FOR i IN 0 TO num_channels-1 LOOP
```

```
               IF send(i) = '1' THEN
                   channel <= i;
                   channel_tmp := i;
                   next_channel := false;
                   EXIT;
               END IF;
            END LOOP;
         END IF;

         IF next_channel = false THEN
            ack_internal(channel_tmp) <= '1';
         END IF;

         IF ack_internal(channel_tmp) = '1' THEN
             ack_internal(channel_tmp) <= '0';
             next_channel := true;
          END IF;


   END IF; -- clk'event
   END PROCESS receive_process;

END queue_a;
```

Figure C.1: Message queue length 0

```
---------------------------------------------------------------------------
---------------------------------------------------------------------
-- Message queue length 1
--
---------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY queue_l1_dn_cn IS
   GENERIC (
      queue_width,queue_depth,num_channels: Integer);
   PORT (
      clk:             IN      std_logic;
      reset:           IN      std_logic;
-- Kanaleingang
      send:            IN      std_logic_vector(num_channels-1 downto 0);
      ack:             OUT     std_logic_vector(num_channels-1 downto 0);
      data_in:         IN      std_logic_vector(queue_width*num_channels-1 down
to 0);
-- Schnittstelle zum Prozess
      get_sig:         IN      std_logic;
      sig_ready:       OUT     std_logic;
      data:            OUT     std_logic_vector(queue_width-1 DOWNTO 0)
      );
END queue_l1_dn_cn;

ARCHITECTURE queue_a OF queue_l1_dn_cn IS
   TYPE local_array is array (natural range num_channels-1 downto 0)
                    of std_logic_vector(queue_width-1 downto 0);
   SIGNAL din_int:      local_array;
   SIGNAL ack_internal: std_logic_vector(num_channels-1 downto 0);
   SIGNAL queue_reg: std_logic_vector(queue_width-1 DOWNTO 0);
   SIGNAL write,read: std_logic;
   SIGNAL sig_ready_internal, inv_reset: std_logic;
BEGIN

   conv_data_in: FOR i in 0 TO num_channels-1 GENERATE
            din_int(i) <= data_in((i+1)*queue_width-1 downto i*queue_width);

   END GENERATE;

   sig_ready <= sig_ready_internal;
   ack <= ack_internal;

   data <= queue_reg;

   receive_process: PROCESS (clk, reset)
      TYPE state_type IS (ready, writing_fifo);
      VARIABLE channel: natural range num_channels-1 DOWNTO 0;
      VARIABLE state: state_type;
      VARIABLE next_channel: boolean;
   BEGIN

    IF reset = '1' THEN

      ack_internal <= (OTHERS => '0');
      next_channel := true;
      channel := 0;
      write <= '0';
      queue_reg  <= (OTHERS => '0');

   ELSIF clk'event and clk= '1' THEN

          IF read = '1' THEN
             write <= '0';
          END IF;
```

```
          IF next_channel = true THEN
             FOR i IN 0 TO num_channels-1 LOOP
                IF send(i) = '1' THEN
                    channel := i;
                    next_channel := false;
                    EXIT;
                END IF;
             END LOOP;
          END IF;

          IF next_channel = false THEN
             IF write = '0' THEN

                queue_reg <= din_int(channel);
                ack_internal(channel) <= '1';
                write <= '1';

             ELSE
                -- sendender Prozess bekommt kein ack -> blockiert,
                -- bis wieder Platz in queue
             END IF;
          END IF;

         IF ack_internal(channel) = '1' THEN
            ack_internal(channel) <= '0';
            next_channel := true;
         END IF;

   END IF; -- clk'event
   END PROCESS receive_process;

   queue_process: PROCESS (clk, reset)
   BEGIN

      IF (reset='1') THEN
         sig_ready_internal <= '0';
         read <= '0';
      ELSIF clk'event and clk= '1' THEN
         IF get_sig = '1' THEN
            IF write = '1' THEN
               read <= '1';
               sig_ready_internal <= '1';
            ELSE
               sig_ready_internal <= '0';
            END IF;
         END IF;

         IF write = '0' THEN
             read <= '0';
          END IF;

         IF sig_ready_internal = '1' THEN
            sig_ready_internal <= '0';
         END IF;

      END IF;
   END PROCESS queue_process;

END queue_a;
```

Figure C.2: Message queue length 1

```
--------------------------------------------------------------------
-- Message queue length n
--
--------------------------------------------------------------------

LIBRARY ieee;
library DWARE,DW06;
USE ieee.std_logic_1164.all;
use DW06.DW06_components.all;
use DWARE.DWpackages.all;


ENTITY queue_ln_dn_cn IS
   GENERIC (
      queue_width,queue_depth,num_channels: Integer);
   PORT (
      clk:              IN      std_logic;
      reset:            IN      std_logic;
-- Kanaleingang
      send:             IN      std_logic_vector(num_channels-1 downto 0);
      ack:              OUT     std_logic_vector(num_channels-1 downto 0);
      data_in:          IN      std_logic_vector(queue_width*num_channels-1 down
to 0);
-- Schnittstelle zum Prozess
      get_sig:          IN      std_logic;
      sig_ready:        OUT     std_logic;
      data:             OUT     std_logic_vector(queue_width-1 DOWNTO 0)
      );
END queue_ln_dn_cn;

ARCHITECTURE queue_a OF queue_ln_dn_cn IS
   TYPE local_array is array (natural range num_channels-1 downto 0)
                       of std_logic_vector(queue_width-1 downto 0);
   SIGNAL din_int:      local_array;
   SIGNAL   ack_internal:       std_logic_vector(num_channels-1 downto 0);
   SIGNAL ff_in, ff_out: std_logic_vector(queue_width-1 DOWNTO 0);
   SIGNAL ff_write, ff_read, ff_full, ff_empty: std_logic;
   SIGNAL one, sig_ready_internal, inv_reset: std_logic;
BEGIN

  fifo: DW_fifo_s1_sf
  generic map (depth => queue_depth, width => queue_width,
               ae_level => 1, af_level => queue_depth-1,
               err_mode => 0, rst_mode => 0)
  port map (data_in => ff_in,
            push_req_n => ff_write,
            pop_req_n => ff_read,
            diag_n => one,
            clk => clk,
            rst_n  => inv_reset,
            data_out => data,
            full => ff_full,
            empty => ff_empty);

   conv_data_in: FOR i in 0 TO num_channels-1 GENERATE
               din_int(i) <= data_in((i+1)*queue_width-1 downto i*queue_width);

   END GENERATE;


   inv_reset <= NOT (reset);
   one <= '1';
   sig_ready <= sig_ready_internal;
   ack <= ack_internal;

   receive_process: PROCESS (clk, reset)
      TYPE state_type IS (ready, writing_fifo);
      VARIABLE channel: natural range num_channels-1 DOWNTO 0;
      VARIABLE state: state_type;
      VARIABLE next_channel: boolean;
   BEGIN

    IF reset = '1' THEN

      ff_write <= '1';
      ack_internal <= (OTHERS => '0');
      next_channel := true;
      channel := 0;
      ff_in  <= (OTHERS => '0');

   ELSIF clk'event and clk= '1' THEN

            IF next_channel = true THEN
               FOR i IN 0 TO num_channels-1 LOOP
                  IF send(i) = '1' THEN
                      channel := i;
                      next_channel := false;
                      EXIT;
                  END IF;
               END LOOP;
            END IF;

            IF next_channel = false THEN
               IF ff_full /= '1' THEN

                  ff_in <= din_int(channel);
                  ff_write <= '0';
                  ack_internal(channel) <= '1';

               ELSE
                  -- sendender Prozess bekommt kein ack -> blockiert,
                  -- bis wieder Platz in queue
               END IF;
           END IF;

           IF ack_internal(channel) = '1' THEN
              ack_internal(channel) <= '0';
              next_channel := true;
           END IF;

           IF ff_write = '0' THEN
              ff_write <= '1';
           END IF;


   END IF; -- clk'event
   END PROCESS receive_process;

   queue_process: PROCESS (clk, reset)
   BEGIN

      IF (reset='1') THEN
         sig_ready_internal <= '0';
         ff_read <= '1';
      ELSIF clk'event and clk= '1' THEN
         IF get_sig = '1' THEN
            IF ff_empty /= '1' THEN
               ff_read <= '0';
               sig_ready_internal <= '1';
            ELSE
               sig_ready_internal <= '0';
            END IF;
         END IF;
```

Figure C.3: Message queue length n

```
         IF ff_read = '0' THEN
            ff_read <= '1';
         END IF;

         IF sig_ready_internal = '1' THEN
            sig_ready_internal <= '0';
         END IF;

      END IF;
   END PROCESS queue_process;

END queue_a;

--pragma translate_off
library DW06;
CONFIGURATION conf_queue OF queue_ln_dn_cn IS
   FOR queue_a
         FOR fifo:
          DW_fifo_s1_sf USE configuration DW06.DW_fifo_s1_sf_cfg_sim;
       END FOR;
   END FOR;
END conf_queue;
--pragma translate_on
```

```
------------------------------------------------------------------
-- FPGA Bus interface
------------------------------------------------------------------
------------------------------------------------------------------
-- 8 register multiplexer and access logic for Spyder Virtex FPGA
------------------------------------------------------------------

LIBRARY ieee;
LIBRARY CIOP;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE CIOP.ALL;

ENTITY vdoutmux8 IS
    PORT (
        clock:          IN      std_logic;      -- unused
        reset:          IN      std_logic;      -- unused
        read:           IN      std_logic;
        addr:           IN      std_logic_vector(4 DOWNTO 0);
        dinout:         INOUT   std_logic_vector(31 DOWNTO 0);
        din0, din1, din2, din3, din4, din5, din6, din7:
                        IN      std_logic_vector(31 DOWNTO 0);
        dout:           OUT     std_logic_vector(31 DOWNTO 0)
    );
END vdoutmux8;

ARCHITECTURE doutmux8a OF vdoutmux8 IS
BEGIN

dout <= dinout;

readbuf: process (read,addr,din0,din1,din2,din3,din4,din5,din6,din7)
begin
  if (read = '1') then
    case ADDR (4 downto 0) is
      when "00000" => dinout <= din0;
      when "00001" => dinout <= din1;
      when "00010" => dinout <= din2;
      when "00011" => dinout <= din3;
      when "00100" => dinout <= din4;
      when "00101" => dinout <= din5;
      when "00110" => dinout <= din6;
      when "00111" => dinout <= din7;
      when others => dinout <= (others => 'Z');
    end case;
  else
    dinout <= (others => 'Z');
  end if;
end process; -- readbuf
END doutmux8a;

------------------------------------------------------------------
-- RTU read/write access decoder
------------------------------------------------------------------
LIBRARY synopsys;
LIBRARY IEEE;
USE synopsys.bv_arithmetic.ALL;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY rrwdecode IS
    PORT (
        clock:          IN      std_logic;
        reset:          IN      std_logic;
        a:              IN      std_logic_vector(4 DOWNTO 0);
        csel:           IN      std_logic;      -- CS# chip select
        rw:             IN      std_logic;      -- R/W#
        rdb:            IN      std_logic;      -- RD#
        read:           OUT     std_logic;
        write:          OUT     std_logic_vector(7 DOWNTO 0)
    );
END rrwdecode;

ARCHITECTURE rrwdecodea OF rrwdecode IS
SIGNAL wen:     std_logic;

BEGIN
    read <= (NOT csel) AND rw AND (NOT rdb);
    wen  <= (NOT csel) AND (NOT rw) AND rdb AND (NOT a(4)) AND (NOT a(3));
    write(0) <= wen AND (NOT a(2)) AND (NOT a(1)) AND (NOT a(0));
    write(1) <= wen AND (NOT a(2)) AND (NOT a(1)) AND (    a(0));
    write(2) <= wen AND (NOT a(2)) AND (    a(1)) AND (NOT a(0));
    write(3) <= wen AND (NOT a(2)) AND (    a(1)) AND (    a(0));
    write(4) <= wen AND (    a(2)) AND (NOT a(1)) AND (NOT a(0));
    write(5) <= wen AND (    a(2)) AND (NOT a(1)) AND (    a(0));
    write(6) <= wen AND (    a(2)) AND (    a(1)) AND (NOT a(0));
    write(7) <= wen AND (    a(2)) AND (    a(1)) AND (    a(0));
END rrwdecodea;


------------------------------------------------------------------
-- the master control/status/revision id register
------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY mcsrr IS
    GENERIC (
        design:         INTEGER := 0;
        rev:            INTEGER := 0;
        nctrl:          INTEGER := 2
    );
    PORT (
        clock:          IN      std_logic;
        reset:          IN      std_logic;
        write:          IN      std_logic;
        din:            IN      std_logic_vector(31 DOWNTO 0);
        ctrl:           BUFFER  std_logic_vector(nctrl-1 DOWNTO 0);
        dout:           OUT     std_logic_vector(31 DOWNTO 0)
    );
END mcsrr;

ARCHITECTURE mcsrra OF mcsrr IS

BEGIN
    dout(31 DOWNTO nctrl+16) <= conv_std_logic_vector(0, 16-nctrl);
    dout(ctrl'LEFT+16 DOWNTO 16) <= ctrl;
    dout(15 DOWNTO  8) <= conv_std_logic_vector(design, 8);
    dout( 7 DOWNTO  0) <= conv_std_logic_vector(rev, 8);

    p1: PROCESS (reset, clock)
    BEGIN
        IF reset = '1' THEN
            ctrl <= conv_std_logic_vector(0, nctrl);
        ELSIF (clock'EVENT and clock = '1') THEN
            IF write = '1' THEN
                ctrl <= din(ctrl'LEFT+16 DOWNTO 16);
            ELSE
                ctrl <= ctrl;
            END IF;
        END IF;
    END process p1;
END mcsrra;
```

Figure C.4: FPGA bus interface

```
--
-- SDL Signal Interfaces
------------------------------------------------------------------------
LIBRARY IEEE;
LIBRARY UNISIM;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE UNISIM.ALL;

------------------------------------------------------------------------
--------------------------------------------------------------------
-- Galileo Signal Interface
--
-- on occurence of a rising edge on signal write,
-- data_in is captured, and sent as SDL signal
-- with a simple handshake on data_out
--------------------------------------------------------------------

LIBRARY IEEE;
LIBRARY CIOP;
LIBRARY UNISIM;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE CIOP.ALL;
USE UNISIM.ALL;


ENTITY write_to_sdl_signal IS
    GENERIC (data_width:        integer := 32);
    PORT (
        clock:          IN              std_logic;
        reset:          IN              std_logic;
        write:          IN              std_logic;
        data_in:        IN              std_logic_vector(data_width-1 downto 0);
        send:           OUT             std_logic;
        ack:            IN              std_logic;
        data_out:       BUFFER  std_logic_vector(data_width-1 downto 0)
    );
END write_to_sdl_signal;

ARCHITECTURE write_to_sdl_signal_a OF write_to_sdl_signal IS

COMPONENT rwreg_n
    GENERIC (width:     INTEGER := 32);
    PORT (
        clock, reset:   IN      std_logic;
        write:          IN      std_logic;
        din:            IN      std_logic_vector(width-1 DOWNTO 0);
        dout:           BUFFER  std_logic_vector(width-1 DOWNTO 0)
    );
END COMPONENT;

COMPONENT FDC -- Xilinx Library Component FDC
               -- D-FF without clock enable and asynchronous reset
   PORT(
      Q                             :  OUT   std_logic;
      D                             :  IN    std_logic;
      C                             :  IN    std_logic;
      CLR                           :  IN    std_logic);
end COMPONENT;

SIGNAL internal_send, internal_write, write_old:            std_logic;

BEGIN

   send <= internal_send;
```

```
   input_capture: rwreg_n
   GENERIC MAP (width => data_width)
   PORT MAP (
        clock=>CLOCK, reset=>reset,
        write=>write,
        din=>data_in,
        dout=>data_out);

   synchronize_write: FDC  -- asynchrones write-Signal auf clock synchronisieren
   PORT MAP (
    Q =>internal_write, D => write, C => clock, CLR => reset);


  write_edge: process (clock, reset)
  begin
    if reset = '1' then
                write_old <= '0';
    elsif clock = '1' and clock'event then

                if internal_write = '1' and write_old <= '0' then
                        internal_send <= '1';
                end if;

            if internal_send = '1' and ack = '1' then
                        internal_send <= '0';
        end if;

                write_old <= internal_write;

    end if;
  end process write_edge;

END write_to_sdl_signal_a;
------------------------------------------------------------------------



--------------------------------------------------------------------
-- on occurence of input signal edge send a sdl-signal
--------------------------------------------------------------------

LIBRARY IEEE;
LIBRARY CIOP;
LIBRARY UNISIM;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE CIOP.ALL;
USE UNISIM.ALL;


ENTITY edge_to_sdl_signal IS
    GENERIC (
        signal_id_width:        integer := 10;
        signal_id:              integer := 0;
        kind_of_edge:           integer := 0
    );
    PORT (
        clock:          IN      std_logic;
        reset:          IN      std_logic;
        signal_in:      IN      std_logic;
        send:           OUT     std_logic;
        ack:            IN      std_logic;
        data_out:       OUT     std_logic_vector(signal_id_width-1 downto 0)
    );
END edge_to_sdl_signal;

ARCHITECTURE edge_to_sdl_signal_a OF edge_to_sdl_signal IS
```

Figure C.5: Signal input components

```
COMPONENT FDC                             -- Xilinx Library Component FDC
                                          -- D-FF without clock enable and
                                          -- asynchronous reset
   PORT(
      Q:               OUT   std_logic;
      D:               IN    std_logic;
      C:               IN    std_logic;
      CLR:             IN    std_logic
   );
end COMPONENT;

SIGNAL internal_send, internal_write:     std_logic;
SIGNAL signal_in_neg, internal_write_neg: std_logic;
SIGNAL write_old:                         std_logic;
SIGNAL tmp:         std_logic_vector(32-1 downto 0);

BEGIN
  data_out <= tmp(signal_id_width-1 downto 0);
  tmp      <= conv_std_logic_vector(signal_id, 32);
  send     <= internal_send;
  signal_in_neg  <= NOT(signal_in);
  internal_write <= NOT(internal_write_neg);

  synchronize_write: FDC                 -- asynchrones write-Signal auf
  PORT MAP (                             -- clock synchronisieren
     Q   => internal_write_neg,
     D   => signal_in_neg,
     C   => clock,
     CLR => reset
  );

  write_edge: process (clock, reset)
  begin
    if reset = '1' then
        internal_send <= '0';
        write_old     <= '1';
    elsif clock = '1' and clock'event then
        if internal_write = '1' then
            if ((write_old = '0') and ((kind_of_edge = 2) or (kind_of_edge = 1))
) then
                internal_send <= '1';
            end if;
        else
            if ((write_old = '1') and ((kind_of_edge = 2) or (kind_of_edge = 0))
) then
                internal_send <= '1';
            end if;
        end if;
        if internal_send = '1' and ack = '1' then
            internal_send <= '0';
        end if;
        write_old <= internal_write;
    end if;
  end process write_edge;

END edge_to_sdl_signal_a;
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
--
-- SDL Signal Interfaces
------------------------------------------------------------------------
---------------------------------------------------------------------

LIBRARY ieee;
library DWARE,DW06;
LIBRARY UNISIM;
USE ieee.std_logic_1164.all;
use DW06.DW06_components.all;
use DWARE.DWpackages.all;
USE UNISIM.ALL;

--------------------------------------------------------------------
-- hw->sw message queue with sdl-signal input interface
--------------------------------------------------------------------

ENTITY hw_sw_queue_ln_dn_cn IS
   GENERIC (
      queue_width,queue_depth,num_channels: Integer);
   PORT (
      clk:                    IN      std_logic;
      reset:          IN      std_logic;
-- channel input
      send:                   IN      std_logic_vector(num_channels-1 downto 0
);
      ack:                    OUT     std_logic_vector(num_channels-1 downto 0
);
      data_in:        IN      std_logic_vector(queue_width*num_channels-1 down
to 0);
-- to HW/SW-interface
      not_empty:      OUT     std_logic;
      pop:                    IN      std_logic;
      data_out:       OUT     std_logic_vector(queue_width-1 DOWNTO 0)
      );
END hw_sw_queue_ln_dn_cn;

ARCHITECTURE queue_a OF hw_sw_queue_ln_dn_cn IS
   TYPE local_array is array (natural range num_channels-1 downto 0)
                       of std_logic_vector(queue_width-1 downto 0);
   SIGNAL din_int:      local_array;
   SIGNAL ack_internal: std_logic_vector(num_channels-1 downto 0);
   SIGNAL ff_in, ff_out: std_logic_vector(queue_width-1 DOWNTO 0);
   SIGNAL ff_write, ff_read, ff_full, ff_empty: std_logic;
   SIGNAL internal_pop, pop_old: std_logic;
   SIGNAL one, inv_reset: std_logic;

COMPONENT FDC -- Xilinx Library Component FDC
              -- D-FF without clock enable and asynchronous reset
   PORT(
      Q                            :  OUT   std_logic;
      D                            :  IN    std_logic;
      C                            :  IN    std_logic;
      CLR                          :  IN    std_logic);
end COMPONENT;

BEGIN

  fifo: DW_fifo_s1_sf
  generic map (depth => queue_depth, width => queue_width,
               ae_level => 1, af_level => queue_depth-1,
               err_mode => 0, rst_mode => 0)
  port map (data_in => ff_in,
            push_req_n => ff_write,
            pop_req_n => ff_read,
            diag_n => one,
            clk => clk,
```

```
            rst_n  => inv_reset,
            data_out => data_out,
            full => ff_full,
            empty => ff_empty);

   conv_data_in: FOR i in 0 TO num_channels-1 GENERATE
               din_int(i) <= data_in((i+1)*queue_width-1 downto i*queue_width);

   END GENERATE;

   not_empty <= NOT ff_empty;

   inv_reset <= NOT (reset);
   one <= '1';
   ack <= ack_internal;

   receive_process: PROCESS (clk, reset)
      TYPE state_type IS (ready, writing_fifo);
      VARIABLE channel: natural range num_channels-1 DOWNTO 0;
      VARIABLE state: state_type;
      VARIABLE next_channel: boolean;
   BEGIN

    IF reset = '1' THEN

      ff_write <= '1';
      ack_internal <= (OTHERS => '0');
      next_channel := true;
      channel := 0;
      ff_in  <= (OTHERS => '0');

   ELSIF clk'event and clk= '1' THEN

         IF next_channel = true THEN
            FOR i IN 0 TO num_channels-1 LOOP
               IF send(i) = '1' THEN
                   channel := i;
                   next_channel := false;
                   EXIT;
               END IF;
            END LOOP;
         END IF;

         IF next_channel = false THEN
            IF ff_full /= '1' THEN

               ff_in <= din_int(channel);
               ff_write <= '0';
               ack_internal(channel) <= '1';

            ELSE
               -- sendender Prozess bekommt kein ack -> blockiert,
               -- bis wieder Platz in queue
            END IF;
        END IF;

        IF ack_internal(channel) = '1' THEN
           ack_internal(channel) <= '0';
           next_channel := true;
        END IF;

        IF ff_write = '0' THEN
           ff_write <= '1';
        END IF;


   END IF; -- clk'event
```

Figure C.6: Signal output

```
   END PROCESS receive_process;

   synchronize_read: FDC  -- asynchrones read-Signal auf clock synchronisieren
   PORT MAP (
    Q =>internal_pop, D => pop, C => clk, CLR => reset);

  read_edge: process (clk, reset)
  begin
    if reset = '1' then
                pop_old <= '0';
                ff_read <= '1';
    elsif clk = '1' and clk'event then

                if internal_pop = '1' and pop_old <= '0' then
                        ff_read <= '0';
                end if;

            if ff_read = '0' then
                        ff_read <= '1';
        end if;

                pop_old <= internal_pop;

    end if;
  end process read_edge;

END queue_a;

--pragma translate_off
library DW06;
CONFIGURATION conf_queue OF queue_ln_dn_cn IS
   FOR queue_a
         FOR fifo:
          DW_fifo_s1_sf USE configuration DW06.DW_fifo_s1_sf_cfg_sim;
      END FOR;
   END FOR;
END conf_queue;
--pragma translate_on
```

```vhdl
--
------------------------------------------------------------------------
-- SDL Timer
--
--
-- start: '1' sets internal counter and resets alarm;
-- as soon as start is '0' the countdown starts
-- alarm: when the counter runs out the specified SDL signal is sent
-- to stop the timer start the timer with parameter 0 or assert reset
------------------------------------------------------------------------


library ieee;
LIBRARY UNISIM;
library synopsys;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
USE UNISIM.ALL;
use synopsys.attributes.all;

entity sdl_timer is
  generic(width :               integer := 8;
          signal_id_width:      integer := 3;
          signal_id:            integer := 2);
  port
    (
     clk,reset  : IN std_logic;
     start      : IN std_logic;
     ticks      : IN std_logic_vector(width - 1 downto 0);
     send       : OUT std_logic;
     ack        : IN std_logic;
     signal_out : OUT std_logic_vector(signal_id_width-1 downto 0)
     );
end sdl_timer;

architecture sdl_timer_a of sdl_timer is

  COMPONENT FDC -- Xilinx Library Component FDCE
               -- D-FF with clock enable and asynchronous reset
   PORT(
      Q                              :  OUT   std_logic;
      D                              :  IN    std_logic;
      C                              :  IN    std_logic;
      CLR                            :  IN    std_logic);
  end COMPONENT;

  signal alarm, ff_clear, one : std_logic;
  signal tmp: std_logic_vector (32-1 downto 0);

  attribute async_set_reset of reset: signal is "true";

begin

  one <= '1';

  signal_out <= tmp(signal_id_width-1 downto 0);
  tmp <= conv_std_logic_vector(signal_id, 32);
  -- (workaround because synopsys only accepts integer type generics)

   alarm_occured: FDC
   PORT MAP (
    Q =>send, D => one, C => alarm, CLR => ff_clear);

  sync_reset: process (clk, reset)
  begin
    if reset = '1' then
            ff_clear <= '1';
    elsif clk = '1' and clk'event then
            if ack = '1' then
                    ff_clear <= '1';
            else
                    ff_clear <= '0';
            end if;
    end if;
  end process sync_reset;


  main: process (clk, reset)
    variable ticker : unsigned(width-1 downto 0);
    variable run : boolean;
  begin
    if (reset = '1') then
      ticker := conv_unsigned(0, width);
      run := false;
      alarm <= '0';
    elsif clk = '1' and clk'event then
      if (start = '1') then
        ticker := unsigned(ext(ticks, width));
        alarm <= '0';
        if (ticker /= conv_unsigned(0, width)) then
          run := true;
        else
          run := false;
        end if;
      else
        if (run = true) then
          ticker := ticker - conv_unsigned(1, width);
          if (ticker = conv_unsigned(0, width)) then
            run := false;
            alarm <= '1';
          end if;
        end if;
      end if;
    end if;
  end process main;
end sdl_timer_a;
------------------------------------------------------------------------------
```

Figure C.7: SDL timer

```
--------------------------------------------------------------------------------
--

LIBRARY ieee;
LIBRARY SDL;
USE ieee.std_logic_1164.all;


ENTITY shared_var IS
   GENERIC (
      data_width,num_channels: Integer);
   PORT (
      clk:            IN      std_logic;
      reset:          IN      std_logic;
-- Kanaleingang
      request:        IN      std_logic_vector(num_channels-1 downto 0);
      grant:          OUT     std_logic_vector(num_channels-1 downto 0);
      write:          IN      std_logic_vector(num_channels-1 downto 0);
      data_in:        IN      std_logic_vector(data_width*num_channels-1 downt
o 0);
      data_out:       OUT     std_logic_vector(data_width-1 DOWNTO 0)
      );
END shared_var;

ARCHITECTURE shared_var_a OF shared_var IS
   SIGNAL   dout_int:   std_logic_vector(data_width-1 downto 0);
   type local_array is array (natural range num_channels-1 downto 0) of std_logi
c_vector(data_width-1 downto 0);
   SIGNAL din_int:      local_array;
BEGIN

   data_out <= dout_int;

   conv_data_in: FOR i in 0 TO num_channels-1 GENERATE
               din_int(i) <= data_in((i+1)*data_width-1 downto i*data_width);

   END GENERATE;

   access_control: PROCESS (clk, reset)
      VARIABLE channel: natural range num_channels-1 DOWNTO 0;
      VARIABLE next_channel: boolean;
   BEGIN

    IF reset = '1' THEN

      grant <= (OTHERS => '0');
      next_channel := true;
      channel := 0;
      dout_int <= (OTHERS => '0');

   ELSIF clk'event and clk= '1' THEN

           IF next_channel = false THEN
               IF request(channel) = '0' THEN
                  grant(channel) <= '0';
                  channel := 0;
                  next_channel := true;
               END IF;
           END IF;

           IF next_channel = true THEN
               FOR i IN 0 TO num_channels-1 LOOP
                  IF request(i) = '1' THEN
                      channel := i;
                      next_channel := false;
                      grant(channel) <= '1';
                      EXIT;
                  END IF;
               END LOOP;
           END IF;

           IF write(channel) = '1' THEN
                dout_int <= din_int(channel);
           ELSE
                dout_int <= dout_int;
           END IF;

   END IF; -- clk'event
   END PROCESS access_control;

END shared_var_a;
```

Figure C.8: Shared data component