ELSEVIER

Computer Communications 23 (2000) 1179–1195

computer communications

www.elsevier.com/locate/comcom

Improving the efficiency of automated protocol implementations using a configurable FDT compiler

H. Koenig[a,*], P. Langendoerfer[a], H. Krumm[b]

[a]*Brandenburg University of Technology at Cottbus, Department of Computer Science, P.O. Box 101344, D-03013 Cottbus, Germany*
[b]*University of Dortmund, Department of Computer Science, D-44221 Dortmund, Germany*

Abstract

The integration of efficient implementation techniques, which have been proven in manual coding, into FDT compilers is difficult because of the semantic constraints of the FDTs and the lack of language means to flexibly adapt to a given implementation context. In this paper, we discuss ways to improve the efficiency of automated protocol implementations to make them applicable to real-life implementations. For solution, we introduce the concept of a configurable FDT compiler that supports the application of different implementation techniques and the adjustment of the implementation to the given implementation context. The paper discusses the semantic conflicts to be solved when applying optimizing implementation techniques. It introduces a compile time reordering of transitions to cope with these problems. Finally we present measurements that prove a considerable efficiency gain of the generated code as well as a comparison with the *Cadvanced* compiler of the SDT tool set.

Keywords: Formal description techniques; Automated protocol implementation; Configurable FDT compiler; Activity threads; Variable implementation model; Implementation oriented specification; SDL

1. Motivation

Formal description techniques (FDTs) have successfully been applied to increase the quality of protocol developments and telecommunication systems [11,30]. However, their application is mainly focused on design, specification, verification, testing, and recently the performance analysis [7,20]. The implementation phase still represents a gap in this chain of protocol development steps. The main reason for this is that code automatically generated by an FDT compiler mostly does not fulfil the performance requirements of real-life protocol implementations.[1] Therefore, automatically derived implementations are rather used for prototyping than for final implementations of commercial products. Real-life protocol implementations are mainly manually coded. This process is lengthy and requires a lot of implementation design decisions that are not covered by formal verification. They can only be validated by a thorough test of the implementation. Automated protocol implementations, on the contrary, may bring some remarkable benefits: a considerable reduction of the duration of the implementation process, simplification of changes, better compliance between specification and implementation, and independence of the implementor. Formal description techniques will only be then successfully applied in practice if they support a continual application of FDTs in all phases of the protocol development process including the implementation phase. As long as protocol engineers are forced after finishing the design and verification phase to "rewrite" the protocol in C or another implementation language in order to obtain an efficient implementation, they will scarcely be willing to apply an FDT. The implementation process often takes almost the same time as the design and the elaboration of the formal description (usually several weeks or months). For a thorough application of the FDT in the whole protocol development process, techniques for deriving adequately efficient implementations from formal descriptions are indispensable.

Current FDT compilers are characterized by a straightforward implementation of the corresponding FDT semantics. They lack means to reduce the overhead caused by semantic constraints and to take the given implementation context into account [11,13,15,16]. In recent years several approaches have been proposed to improve the efficiency of automated implementations. These approaches cover a wide range of proposals from improved mapping strategies over

* Corresponding author.
E-mail address: koenig@informatik.tu-cottbus.de (H. Koenig).

[1] The notion real-life implementation denotes here protocol implementations which are aimed at a given target system to be applied in commercial products. They are optimally adjusted to the target environment (hardware, operating system, user requirements).

0140-3664/00/$ - see front matter
PII: S0140-3664(99)00245-5

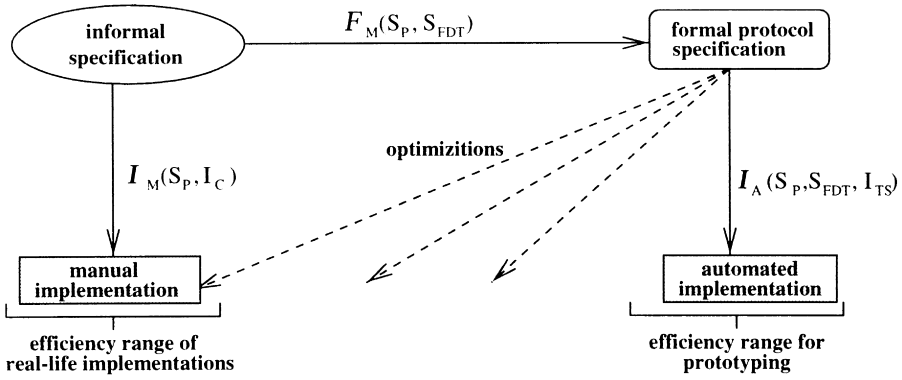


Fig. 1. Constraints of automated protocol implementation.

parallelism to code optimizations. All these approaches more or less focus on single aspects of the problem and therefore did not lead to an adequate increase of efficiency.

On the contrary, protocol implementation techniques meanwhile have become very sophisticated. They are optimized for the needs of the implementation context (environment) and take aspects into account which as, for instance, memory accesses are not covered by formal specifications at all. We believe that a remarkable increase in the efficiency of the automated protocol implementations can only be achieved if a more general approach is applied which combines the benefits of different proposed solutions. Such an approach especially requires means to integrate optimizing implementation techniques into FDT compilers and to enable FDT compilers to adapt to different implementation contexts.

In this paper, we discuss ways to improve the efficiency of the automated protocol implementations to make them applicable to real-life protocol implementations. For solution, we introduce the concept of a configurable compiler that supports the above mentioned requirements. We describe the main features of the concept and present a concrete solution for SDL, the COCOS compiler. Further, we present measurements that prove a considerably gain of efficiency using the approach.

The remainder of the paper is organized as follows. Section 2 discusses the inherent constraints of automated protocol implementation compared to handcoded implementations. In Section 3 we give a short overview of related work. Section 4 outlines the concept of configurable FDT compiler and discusses the required features. Section 5 introduces the protocol implementation techniques used in our approach and explains how they can be applied in automated protocol implementation. Section 6 discusses some specific mapping problems for a configurable compiler using the COCOS compiler as an example. Section 7 reports about measurements to evaluate the achieved efficiency gain. It also compares the COCOS compiler with the *Cadvanced* code generator of the SDT tool. The final remarks summarize the results and give an outlook on the future research.

2. Constraints of the automated protocol implementation

Automated protocol implementation considerably differs from the conventional nonautomated implementation process. It is characterized by several constraints that have to be taken into account for code generation. They are discussed in the sequel using Fig. 1. In the conventional implementation process a given (mostly informal) specification of the protocol is mapped into the structure of the given target system. The implementation I_M is designed by the implementor. It is subjectively shaped. The implementation constraints it has to take into account are the semantics of the protocol specification S_P and the implementation context I_C (operating system, implementation language). The implementor can optimally adapt the implementation to the given implementation context. By this, she/he has to ensure that the compliance between the specification and the implementation is preserved what has to be validated by conformance tests. The coding process is “manually” carried out by the implementor (to distinguish the conventional implementation techniques from automated ones bette, we call them manual or hand-coding techniques in the following).

The situation changes when automated implementation is applied. It requires formal protocol descriptions. In contrast to manual coding automated implementation has to take into account the formal semantics of the given FDT S_{FDT} to assure the correct interpretation of the formal specification, i.e. the transformation process I_A is determined by the semantics of the protocol specification S_P, the semantics of the FDT S_{FDT}, and the implementation environment I_{TS} which usually supplements the operating system by a specific runtime support. An obvious approach for automated protocol implementation is the straightforward implementation of the FDT semantics. It is often applied because the transformation is pretty simple. The correctness of transformation can be validated easily. The experience, however, has shown that such implementations usually do not fulfil the requirements of real-life protocol implementations [8,13–15,29]. For that reason, they

are mainly used for prototyping rather than for the development of final products.

The inefficiency of automatically derived protocol implementations arises because of the following reasons [15]:

- *Design of the implementation model during tool development phase*. The implementation model describes the logical structure of the implementation, its components and the interactions between them. It determines the (module) structure of the implementation, the interfaces (internal, external), the mapping on the process structure of the target operating system (process model), and the implementation environment. It automated protocol implementation, the implementation model defines the set of transformation rules for the code generator to map the formal specification into the implementation. Thereby, it determines the architecture of the generated implementations. In contrast to the manual implementation, however, the implementation model is designed during the tool development process without any considerations of specific implementation contexts.
- *Lack of capability to adapt to a given implementation environment*. Owing to the rigid implementation model the code generator is not able to adapt flexibly to the characteristics of the implementation context. FDT compilers do not possess means to select different implementation techniques or to introduce the parameters of the target system into the code generation.
- *Overhead because of the FDT semantics*. The semantics of the used FDT highly influence the design of the implementation model. The transformation of certain language features may require additional implementation efforts that cause a runtime overhead compared to hand-coded implementations. A well-known example for this is the scheduling algorithm for selecting the fireable transitions in Estelle [15,21].
- *Lack of integrating protocol layers*. Current FDT compilers mainly focus on single-layer implementations due to their orientation on prototyping. They do not provide means for an integrated implementation of several protocol layers.

In order to overcome these shortages of current FDT compilers optimizations have to be introduced into the code generation process that decrease the distance in efficiency between the hand-coded and the automatically generated implementations. Automated implementation techniques can only successfully be applied for real-life implementations of protocols if their efficiency comes close to the efficiency of the manual implementation techniques (see Fig. 1).

3. Related work

In recent years the derivation of efficient implementations from formal specifications has been investigated in several works. They have brought interesting experimental results, especially for FDTs based on finite state machines as Estelle and SDL, but the question whether FDT based implementations can compete with the hand-coded ones in performance has not yet been answered definitely. The approaches pursued in these works were different. The research on the development of efficient code generation concepts started at the beginnings of this decade. In Ref. [23] an intermediate level as a basis for the automated code generation was proposed. This intermediate level refines the given protocol specification, resolves nondeterminism and adds all the information needed for the implementation, e.g. about the target system. Hofmann [19] discusses the integrated implementation of several protocol layers by constructing the product automaton of the related protocol automata. In Ref. [15], experiments with the experimental Estelle-C compiler EECT are reported that applies optimized algorithms for implementing certain features of the Estelle semantics, e.g. for the selection of the fireable transitions and for the handling of the input queues of the modules. The results show that the performance of the generated implementations can be increased by optimizing algorithms but the improvements are not yet adequate for real-life implementations. Further, the concept of a variable implementation model was proposed in this work to better adjust the derived implementation to a given implementation environment. Several approaches were dedicated to the exploitation of the protocol inherent parallelism [6,12,31]. The obtained results are summarized in Ref. [13]. They show that the efficiency of the derived implementations can be improved by using parallelism. The potential of parallelism in protocols, however, is often too small and the additional overhead for synchronization and communication too high to achieve a considerable efficiency gain. More promising results are expected from an integrated handling of layers and data operations. Abbott and Peterson [1] present a compiler for a special language called Morpheus that uses the integrated layer processing (ILP) approach [10]. It achieves a performance gain of about 50%. Leue and Oechslin [28] describe an algorithm computing the *Common Path* from SDL specifications as a basis for ILP implementations. A compiler for deriving ILP implementations from Estelle is reported in Ref. [4]. The authors state that they achieve the same efficiency like manual implementations, but they do not describe how the semantic constraints are reflected in the mapping process. For standardized FDTs, such a compiler is not yet reported. Recent research [2,5], however, has pointed out that the application of ILP is not recommended in every case. Efficient implementations using the activity thread model [9,32] are reported in Ref. [16]. The approach can be applied to SDL without special assumptions. For Estelle, the application is limited as Estelle's parent/children priority principle introduces semantic constraints [17].

As long as there do not exist automated implementation techniques that provide an adequate efficiency, other

solutions are chosen in practice. Mansurov [29] and Hakanson et al. [14] propose to combine programming languages and SDL specification to increase the efficiency of automatically generated implementations. For the SDL, *Cmicro* code generator [33], the use of certain SDL constructs (object-oriented features, enabling conditions, continuous signals, import/export, view/reveal etc.) is entirely prohibited. Further, there is a recommendation to avoid the application of other constructs (`save`, `create`, `output to parent/sender`…). Thus, the implementation overhead caused by the semantics of these language elements is avoided.

Considering the different approaches, we state that in principle there are two ways to improve the performance of automatically generated implementations:

1. To support the implementation process by modifying or restricting the semantics of the FDT, i.e. to prohibit the use of certain language features, to introduce new language concepts or to employ specific specification styles for which an efficient implementation strategy exists.
2. To improve the mapping strategies and to make them more flexible.

In this paper, we follow the second way. To cope with the constraints of automated implementation a more general approach is required which combines the benefits of the different proposed solutions discussed above.

4. A configurable FDT compiler

The approach we persue to overcome the constraints of the automated protocol implementation is that of a configurable compiler. The following features characterize such a compiler:

- the support of optimizing implementation techniques;
- a flexible adaptation to different implementation contexts.

The compiler should give the implementor the possibility to select the most appropriate mapping for his/her problem. For example, the implementor should have the possibility to select between different implementation strategies when implementing signalling or data transfer protocols, or a single protocol or a whole protocol stack, respectively. It is obvious that this cannot be carried out by a simple straightforward implementation of the FDT semantics. Here the application of manual implementation techniques is indispensable. In addition to the selection of an appropriate implementation model, the implementor should be given the opportunity to introduce information on the used target system and the implementation context (number of processors, memory organization, timer values) as parameters for the code generation process to generate a tailored runtime system. Thus, it will be possible to reduce the semantic overhead of FDT based implementations, to integrate several protocol layers in the implementation process, and to make implementation models more flexible.

The development of a configurable FDT compiler requires the solution of several problems. These problems concern:

- the integration of optimizing implementation techniques;
- the mapping of the FDT elements into an appropriate internal representation;
- the design of the runtime system;
- language means for controlling the configuration and code generation process.

In the following we present with the SDL compiler COCOS a concrete solution for such a configurable compiler. Before describing the structure of the compiler we discuss the problems listed above. The solutions proposed in this discussion relate to SDL. Similar solutions can be developed for other FDTs if the specific of their semantics is taken into account [17]. The COCOS compiler currently supports three implementation techniques: the server model, the activity thread model and the integrated layer processing. Owing to lack of space we focus in this paper on the first two, implementation techniques. The generation of integrated layer processing implementations will be described in a separate paper.

5. Applied implementation techniques

In this section we discuss the semantic problems of the integrating server and the activity thread model implementations into FDT compilers. These techniques denote different process models. The process model describes the manner how the specification is mapped on the process structure of the implementation environment or operating system, respectively [32]. It represents the main component of the implementation model. We briefly introduce the principle of each process model and discuss its applicability. For the application of the activity thread approach, we present a new technique called *transition reordering* for mapping formal descriptions on activity threads during compilation. Beforehand, we begin with a short overview on the basic features of SDL.

5.1. SDL

SDL (*Specification and Description Language*) [22] is the specification language of the ITU-T. It is a widely accepted specification technique for the software design of telecommunication systems. SDL has also successfully been applied for the specification of communication protocols. The development of SDL started in the seventies. The language is redefined and extended in a 4-year cycle. Important versions are SDL'88 and SDL'92. SDL possesses two syntactical forms: the graphical representation SDL/GR

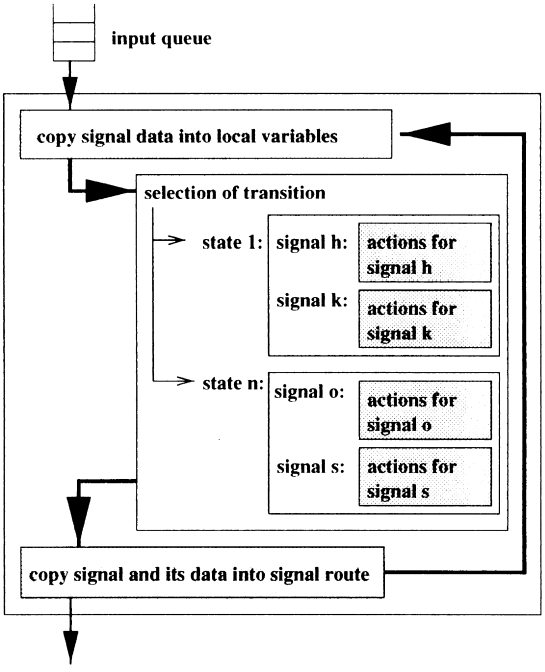


Fig. 2. Structure of a server model instance.

and the textual representation SDL/PR. The graphical representation is more widely used.

SDL distinguishes three description levels: system, block, and process. The system level forms the frame of the description. It represents an abstract machine that communicates with its environment. A system consists of several blocks that describe subsystems. The blocks in turn may contain subblocks thus forming a treelike specification structure. The blocks at the leave level consist of one or more processes. The process is the basic description element in SDL. It represents an extended finite state machine (EFSM). Processes are independent and run concurrently. They interact with other processes by exchanging messages, called signals in SDL. Each process has an unlimited input queue storing incoming signals and timer signals (timeouts). The input queue is a FIFO queue. As other formal description techniques SDL distinguishes between the definition of a process and the incarnation of process exemplars, called process instances. Every process instance gets a unique identification at runtime.

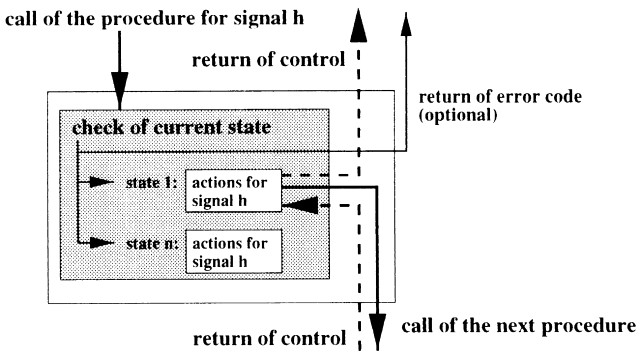


Fig. 3. Structure of an activity thread procedure.

The communication in SDL is asynchronous. Processes located in the same block exchange signals via signal routes. The interaction across block boundaries is carried out by means of channels.

For the description of the behaviour of the processes, SDL provides different symbols for indicating the states, the inputs and outputs, and local actions. The description is state oriented. It lists all states of the process. For each state, the possible input events and the transitions they trigger are specified. A transition contains local actions like assignments and other calculations. It may contain one or more outputs. At the end of each transition the successor state is indicated. In a current state, a process awaits and consumes the front signal of its input queue. If a transition is defined for this signal, the transition is executed. Otherwise, the process remains in its current state and the signal is removed. The `save` mechanism, however, can save a removed signal in order to match a subsequent transition. Timer signals are handled like any other signal and put in the input queue.

5.2. Implementation techniques

5.2.1. Server model

Basic principle: the server model [32] implements the protocol entities by a cyclic task (process or thread of the operating system or runtime system) (see Fig. 2). This task is similar to a device driver. It reads an incoming signal from an input queue, analyses it and switches to the code segment that handles the signal. The code segment represents a transition of an extended finite state machine. After handling the signal a possible output is written in the input queue of the protocol entity of the next layer and the server returns to the beginning to read the next signal. The protocol entities are usually executed in a round robin fashion so that every entity can proceed. This technique also supports the handling of spontaneous transitions.

Applicability: the automatic derivation of code according to the server model is straight-forward, because the above explained basic principle corresponds to the description paradigm of most FDTs (whereby the decision, which code segment the server switches to, is determined by semantic rules of the FDT). Therefore, automated implementation techniques usually apply this process model. The server model, however, exhibits heavy overhead for storing the events between the protocol entities and for process management so that it limits the generation of efficient code [16].

5.2.2. Activity thread model

Basic principle: the activity thread technique [17,32] implements a protocol entity as a set of procedures. For each input signal, a procedure is provided. It usually implements a transition of the protocol automaton. The active

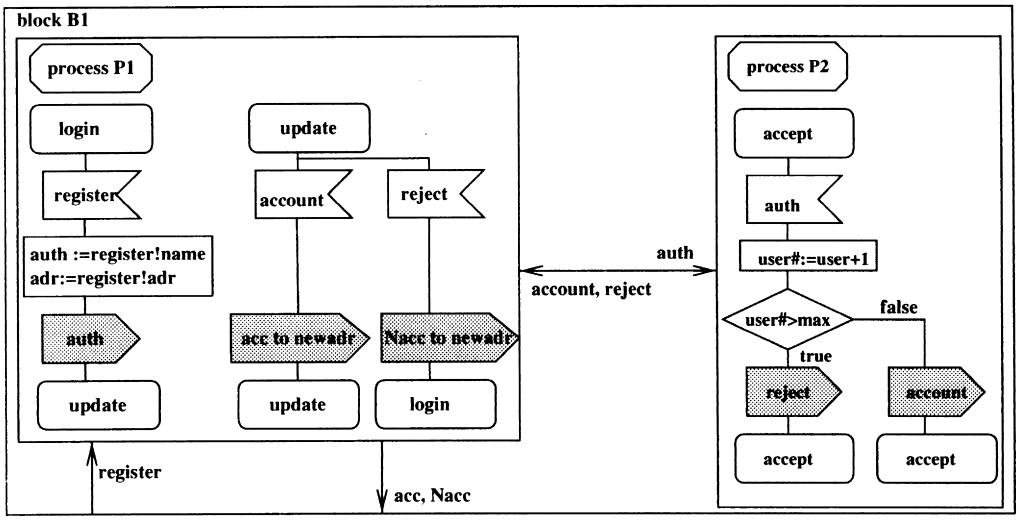


Fig. 4. Example for discarding of a signal.

elements in this model are the signals. An incoming signal activates the corresponding procedure, which immediately handles the signal, and when producing an output calls the respective procedure of the next entity. The sequence of inputs and outputs (*input* → *output* → *input*…*input* → *output*) results in a sequence of procedure calls called activity thread. Note that the term activity thread does not refer to an operating system thread. It denotes the execution path a signal triggers in the protocol stack. Activity threads can run in both directions, upwards and downwards. The respective procedure calls are also denoted as upcalls and downcalls. Fig. 3 depicts the structure of an activity thread procedure. After calling the procedure first the state of the protocol automaton has to be determined. If the entity is not in the state for handling the signal the control is returned to the calling entity, optionally with an error code. Otherwise the respective transition is executed. If the transition contains an `output` the corresponding procedure is called. An activity thread terminates if a transition contains no `output` statements or if the incoming signal cannot be handled in the current state (see above). In these cases the control is returned to the initiator of the call sequence.

Applicability: the activity thread technique provides very efficient implementations because it avoids the storing of signals when calling the next protocol entity [9]. Its semantic model (synchronous computation and communication), however, considerably differs from the semantic model of SDL. The application of synchronous communication in SDL implementations causes several semantic conflicts. Therefore, SDL compilers usually apply the server model that supports a straight mapping of the SDL semantics instead of the more complicated activity thread approach.

5.3. Handling of semantic conflicts

The semantic conflicts which may appear when directly mapping SDL specifications into activity threads can be divided in two groups: interferences of transitions, and overtaking of signals. In both cases they are caused by the appearance of cyclic process call sequences at runtime. In real-life implementations server and activity thread model are usually combined. The server model is used to implement the asynchronous interface to the environment whereas the activity thread technique is applied within the protocol stack. In such combined implementations an overtaking of signals may also appear [27].

5.3.1. Interference of transitions

In the communication between two or more processes the activity thread implementation principle may cause situations in which a signal is consumed by the receiving process before the sender has finished its transition. This may lead to the following errors:

- discarded signals;
- inverted order of state changes and variable assignments.

The discarding of signals may occur when a procedure call implementing an `output` statement is executed before the assignment of the next state. In this case the process instance cannot change its state until the called procedure has returned control. Fig. 4 gives an example for such a situation.[2]

A conforming implementation has to guarantee that process *P1* changes to state *update* after sending the signal *auth*. Then it can accept the signals *account* or *reject* by means of which process *P2* responds to *auth*. In contrast to the expected correct behaviour the signals *account* and *reject*, respectively, are always discarded by process *P1*, because *P1* remains in state *login* until the procedure call

[2] In the sequel we mark the relevant output statements by grey shadows to point out the discussed semantic effects.

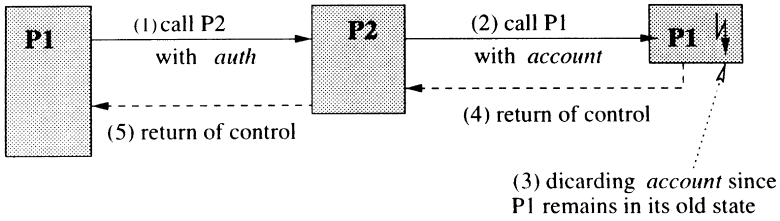


Fig. 5. Process call sequence for example of Fig. 4.

for the output of *auth* is finished. Fig. 5 shows the resulting cyclic process call sequence.

5.3.2. Overtaking of signals in activity thread implementations

SDL guarantees that the order of signals sent via a signal route or a channel is preserved at destination. If, for instance, a signal *s1* is sent at time *t* and a signal *s2* at time $t + x$, *s1* has to be delivered before *s2*. When straightway applying the activity thread approach signals may overtake each other.

This situation may happen if a transition contains several `output` statements. The mapping of `output` statements onto procedure calls causes that the second, third and any further procedure call is delayed until the first one and all successor calls have been finished. If one of these successors uses a signal route that is also used by one of the other *output* statements of the transition with multiple outputs, the signals overtake. Fig. 6 shows an SDL specification in which the preconditions for signal overtaking are fulfilled.

In a straightway-generated activity thread implementation of the specification given in Fig. 6 the signal *register* would trigger the following execution. Process *P1* sends signal *auth* to process *P2* that replies with *notok* or *ok*. *P1* immediately forwards the signal *reject* respectively account to process *P3*, which responds with *acc*, or *Nacc* to the address given in *newadr*. After that the control returns to *P1* which sends the signal *adr* to *P3*. In contrast to the specified behaviour, one of the signals *account* or *reject* now has overtaken *adr*. As consequence, *P3* sends the signal *acc* or *Nacc* to a wrong address. The resulting process call sequence is depicted in Fig. 7.

5.3.3. Overtaking of signals in combined implementations

An implementation, which uses both process models, combines rather different implementation approaches. The server model processes are usually executed by a runtime system scheduler whereas the activity thread procedures are executed whenever a signal arrives. In such a combined implementation overtaking of signals may appear if the following preconditions are fulfilled:

- at least one server, model process communicates via the global input queue with two activity thread process;
- these activity thread process are communicating via procedure calls with each other.

Fig. 8 shows a specification that fulfils these preconditions. We suppose that Process *P1* is mapped on a server model process, while processes *P2* and *P3* are implemented using the activity thread model. In a straightforward implementation of the specification the signal *register* would trigger the following execution. The process call sequence belonging to this example is represented in Fig. 9. Process *P1* appends the signals *auth* and *adr* to the global input

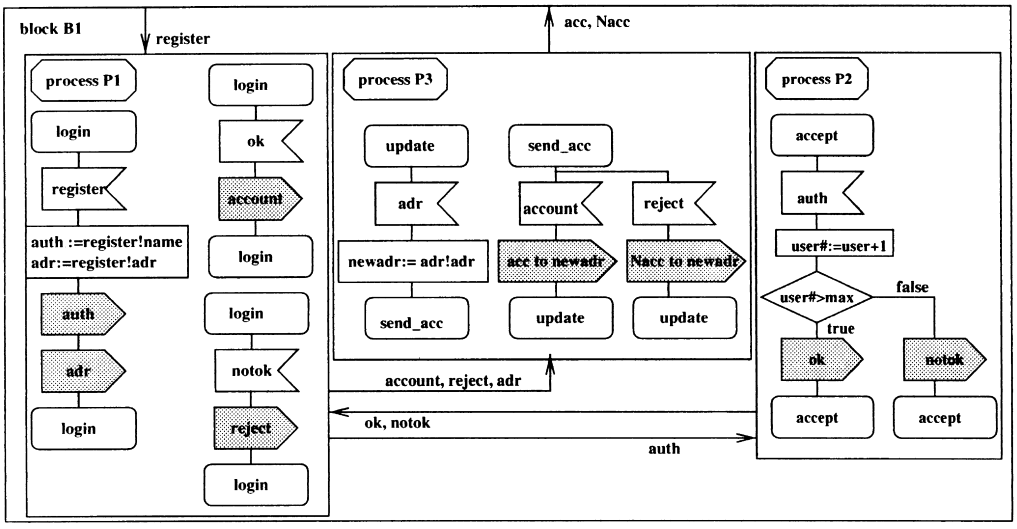


Fig. 6. SDL system which allows signal overtaking.

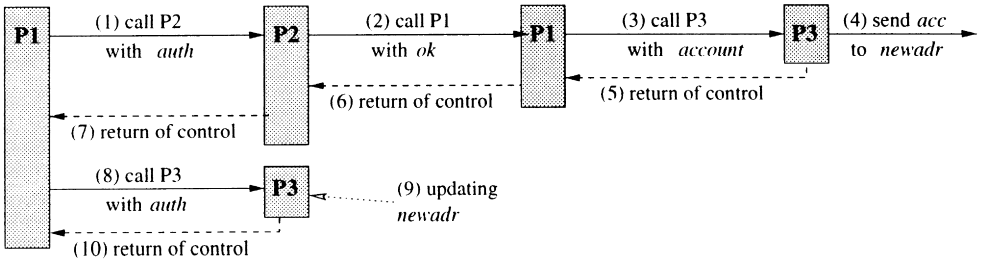


Fig. 7. Process call sequence for example for Fig. 6.

queue (see Fig. 9(b)). Then the scheduler executes *P2* with signal *auth*. *P2* immediately sends one of the signals account or *reject* to *P3* (see Fig. 9(c)). *P3* forwards *acc* or *Nacc* to the address given in *newadr*. After that the control returns to the scheduler which executes now *P3* with *adr* (see Fig. 9(d)). In contrast to the specified behaviour, one of the signals *account* or *reject* now has overtaken *adr*. As consequence, *P3* will send the signal *acc* or *Nacc* to a wrong address.

5.4. Transition reordering

5.4.1. Principle

So far the semantic conflicts described above could only be resolved at runtime. In Ref. [16] this was carried out by extending the activity thread implementation with an activity thread scheduler and a global signal list. Fig. 10(a) depicts the structure of such an implementation. When executing an `output` statement the signal and its receiver are stored in a signal list. The activity thread scheduler processes the signal list in a FIFO manner. It always activates the receiver processes of the first entry. The receiver process consumes the signal and executes the respective transition. By this, further signals can be added by means of an append function to the signal list if the transition contains `output` statements. After finishing the procedure the control is returned to the scheduler. The shortage of this mechanism is that the execution of the `output` statement is delayed, i.e. it may be processed after `output` statements of other transitions. This leads to an asynchronous communication and considerably increases the overhead of the implementation compared to a pure activity thread implementation.

In the following we present an approach called transition reordering, which enables the handling of during compilation. The approach adapts the principle of code restructuring which is applied for loop optimization in modern programming language compilers [3]. The transition reordering designates that the SDL statements of a transition are not implemented in the order as they are specified. The reordering only concerns the `output` statements. They are reordered at compile time in such a way that the semantic conflicts described above cannot occur. The `output` statements are now implemented in two steps.

- Replace the `output` statement by an operation that stores the signal.
- Append the `output` statement with the stored signal to the end of the transition. If a `decision` statement

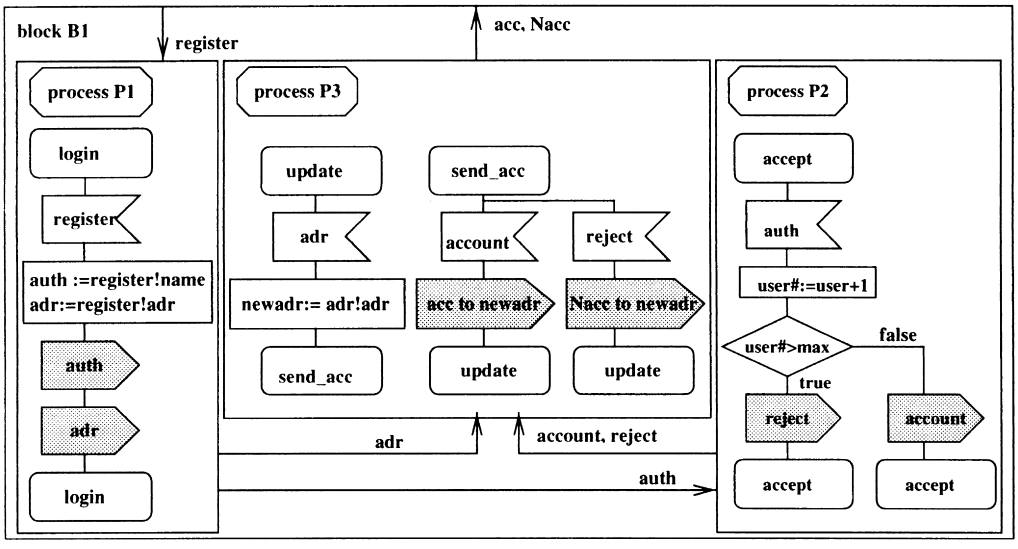


Fig. 8. SDL system with possible signal overtaking in case of a combined implementation.

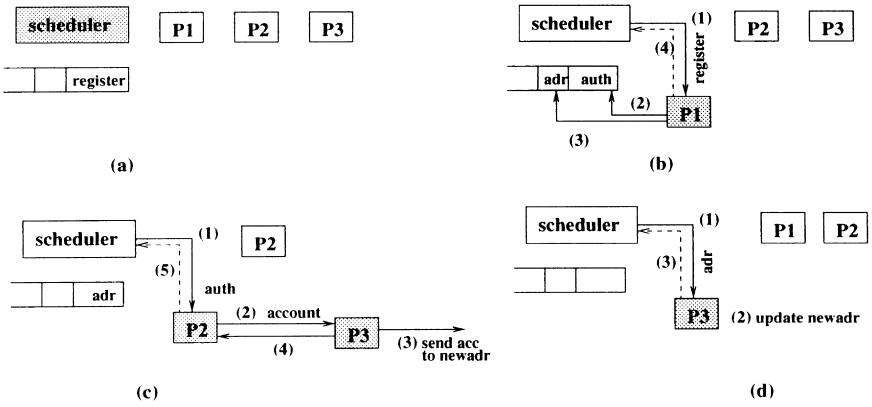


Fig. 9. Process call sequence for example of Fig. 8.

follows the `output` statement the modified `output` statements has to be appended to each branch of the `decision` statement.

All other SDL statements are implemented in the order they are specified, i.e. only the sending of the signals is delayed. This approach does not require any additional runtime support. The `output` statements are executed after the transition has reached the successor state. Thus, according to the activity thread principle the receiver process is triggered by the sender process. The control only returns to the receiver process if no further `output` statements have to be processed or if in connection with a server model implementation a signal is sent to an asynchronous (buffering) interface (see Fig. 10(b)). Using transition reordering the indirect realization of the activity thread proposed in Ref. [16] can be avoided. The transitions can now straightway be mapped on procedures.

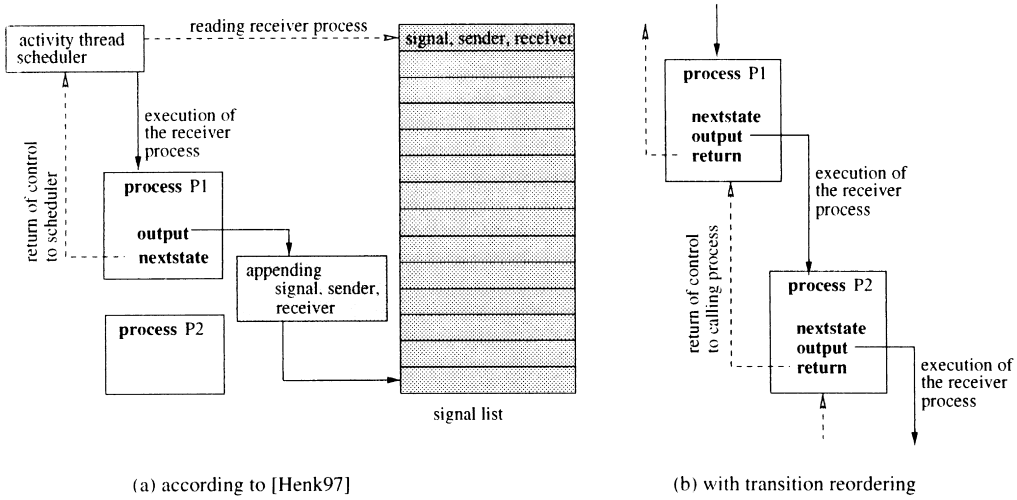
The reordering of actions inside a transition is allowed, because of the following:

1. SDL processes are nonpreemptive. A transition is finished before a new transition can be executed.
2. Signals cannot be modified after the `output` statement.

These conditions guarantee that the shift of the `output` statements to the end of the transition does not influence the compliance between specification and implementation. Condition (1) expresses that the `output` statements are also executed if they are implemented at the end of the transition, because there is no statement in SDL, which can interrupt the execution of the transition. Condition (2) ensures that a statement following the `output` statement cannot modify the data transferred by a signal. Thus, the delay in the sending of the signals cannot lead to a transmission of falsified data.

The transition reordering cannot be used to avoid semantic conflicts if:

- `output` statements are used inside a loop for which the number of executions cannot be computed during compile time;
- two or more `output` statements of the same transition



(a) according to [Henk97] (b) with transition reordering

Fig. 10. Mapping on activity threads.

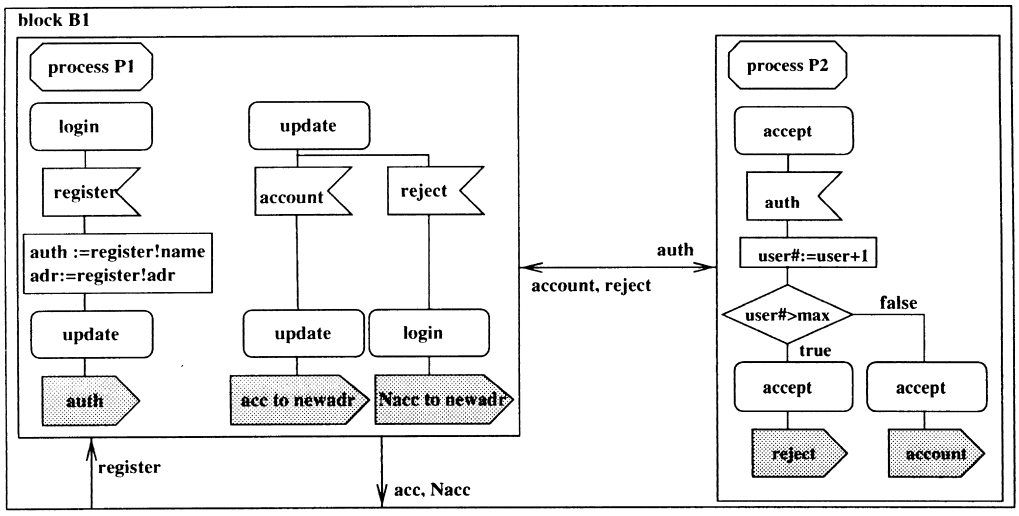


Fig. 11. Example of Fig. 4 after transition reordering.

trigger cyclic process call sequences that terminate with this transition;
- several `output` statements send signals to the same receiver and the `output` statement that triggers the cyclic process call sequences is not specified as last.

In these exceptional cases which can be detected during compilation a buffered signal exchange, similar as in Ref. [16], has to be generated.

In the next sections we show how the semantic conflicts discussed above are avoided by applying *transition reordering*.

5.4.2. Avoiding transition interferences

The reordering of transitions ensures that assignments to variables and state changes are carried out in the correct order even if cyclic process call sequences appear. It prevents the discarding of signals and the inversion of variable settings. The conformance between specification and implementation is preserved. Fig. 11 shows the specification of the example from Fig. 4 after transition reordering. Fig. 12 depicts the respective process call sequence and indicates that the implementation is executed correctly.

5.4.3. Avoiding overtaking of signals in activity thread implementations

Since at compile time the logical structure of the implementation is known the compiler can determine whether there are transitions with multiple outputs which lead to an overtaking of signals. This is carried out by analysing the execution paths and the state changes of the transitions as described next. Based on this information it can determine in which order the procedure calls implementing `output` statements have to be inserted into the final code to prevent the overtaking of signals.

In the first step a data flow analysis is applied to detect possible cyclic process call sequences. For this purpose, for each signal sent in a transition with several `output` statements the set of successor signals and of receiving processes is determined. The algorithm is recursively applied to the successor signals and terminates when the signals are either sent to the environment or to a process instance implemented according to the server model. For this analysis, it is assumed that each receiving process is in a state where it can accept the incoming signal and executes the corresponding transition. Thus, we obtain the execution path triggered by each `output` statement. If possible cyclic behaviour is detected it is proved whether the first and the last transition of the call sequence use the same signal route. If this happens the `output` statement triggering the cyclic process call sequence has to be executed last so that the signal which may be overtaken is sent before the other signals.

Fig. 13 shows the order in which the example given in Fig. 6 is implemented after reordering of the transitions. Note that the outputs of process *P1* are not only delayed but that their order has also been changed. Fig. 14 shows again the resulting process call sequence. In this implementation the signals *adr* and *account* arrive in the correct order at *P3*.

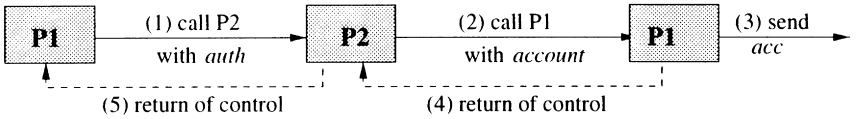


Fig. 12. Process call sequence for example of Fig. 4 (correct behaviour).

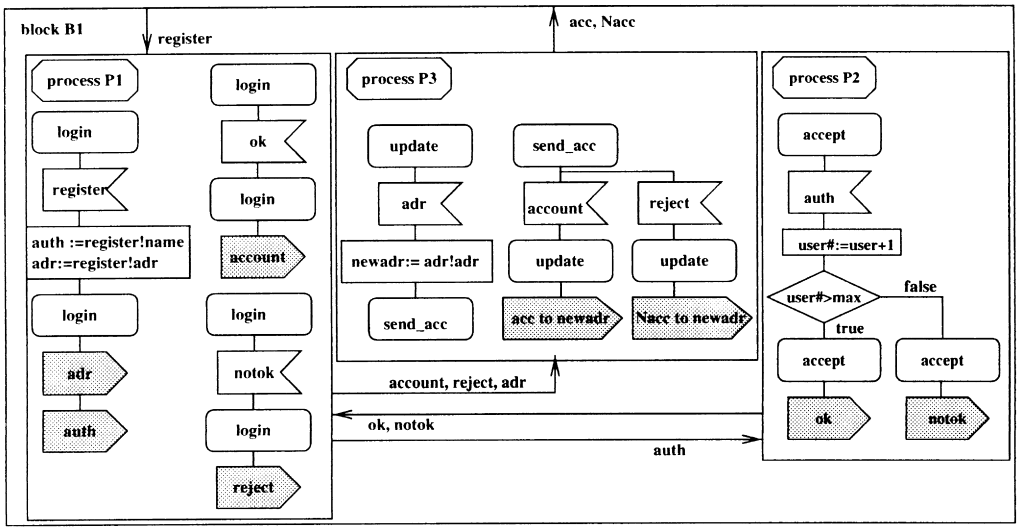


Fig. 13. Example of Fig. 6 after transition reordering.

5.4.4. Avoiding overtaking of signals in combined implementations

To avoid signal overtaking in combined implementations the execution path triggered by each `output` statement have to be determined. This is carried out by applying the algorithm described in Section 5.4.3 to each signal that a server model process sends to an activity thread process. Then it is checked whether there are process that receives signals via their input queue as well as via procedure calls. If no such process exists no additional measures have to be taken. Otherwise it has to be assured that the signals sent via the input queues are consumed always first. To achieve this the `output` statements that do not trigger an execution path are implemented first. Then the statements are implemented whose execution paths do not contain `outputs` to processes that are also addressed by the server model process under implementation. Finally the remaining `output` statements are implemented.

Fig. 15 shows the execution order of the example given in Fig. 8 after the reordering of transitions. Note that the order of the `outputs` of process *P1* has been changed. Fig. 16 shows again the resulting process call sequence. In this implementation the signals *adr* and *account* arrive in the correct order at *P3*.

6. Further mapping issues

The development of a configurable compiler further requires solutions for the mapping of the basic structures of the given FDT into an internal representation, for the runtime support of the compiler, and for the controlling of the compiling process. These issues are considered in this section. The discussion is again oriented to SDL.

6.1. Implementation of SDL processes

To handle SDL processes for different implementation techniques during the code generation process a generic internal structure for the processes is required. An obvious solution for this is the use of re-entrant procedures, which embody the behaviour part of the process [16] (see Fig. 17). For mapping SDL processes on re-entrant procedures, the data parts have to be separated. They are stored in a so-called instance control block (ICB) which is created for each process instance. Each instance control block contains the following data:

- all variables of the process;
- all internal information as `offspring, parent, self` and `sender`;

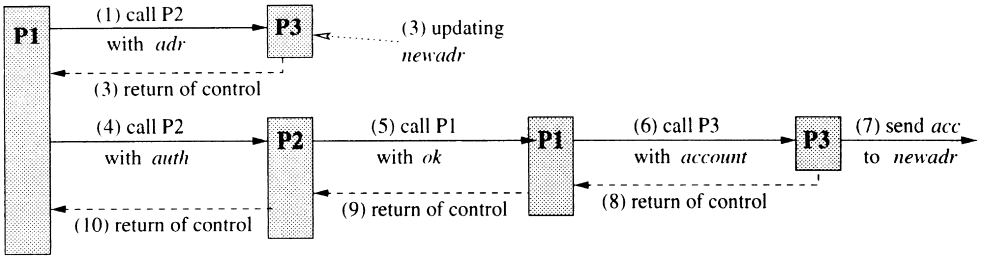


Fig. 14. Process call sequence for example of Fig. 13 (correct behaviour).

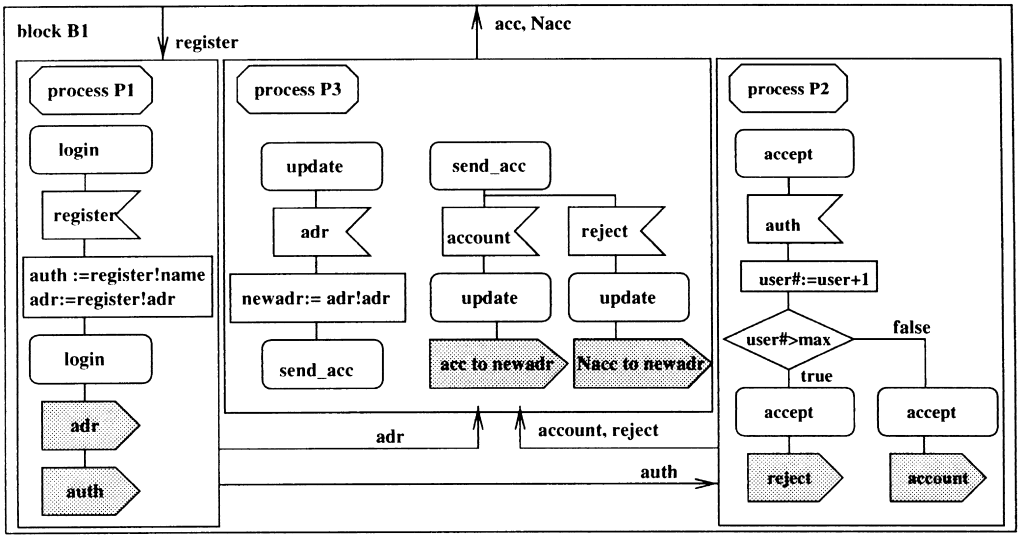


Fig. 15. Example of Fig. 8 after transition reordering.

- a list of all saved signals (*save list*);
- the state of the process.

The implementation of the re-entrant procedures differs for the chosen implementation technique. When the server model is applied one procedure (*SM procedure*) is implemented for each SDL process type. In activity thread implementations a procedure for each input signal (*At procedure*) is generated. All re-entrant procedures are extended by a function that checks whether the *save list* in the instance control block is empty or not. In the latter case, the elements of the *save list* are executed first.

6.2. Runtime system

For the runtime system of a configurable compiler, similar decisions have to be taken. It must be capable of simultaneously supporting different implementation techniques. This requires means to support the communication between SDL processes that are implemented according to different implementation strategies as well as a scheduler that is able to manage all kinds of runtime system tasks.

In server model implementations an individual input queue is used to buffer signals. But it has proved that it is more efficient to use a common input queue for all processes [15] in the implementation. In order to improve the efficiency of the server model implementations we use a common input queue. The activity thread concept dissolves the queue principle, since the signals are immediately processed. However, a buffered interface is needed at the interface to the environment (application, network) to accept the incoming signals and for signal exchange between server model processes and activity thread model processes. This interface is provided by using the common input queue.

Normally, server model processes are scheduled in round robin fashion. This introduces a considerable scheduling overhead. In order to decrease this overhead it is useful to introduce a ready queue for the scheduler. This ready queue can be used to schedule server model processes as well as

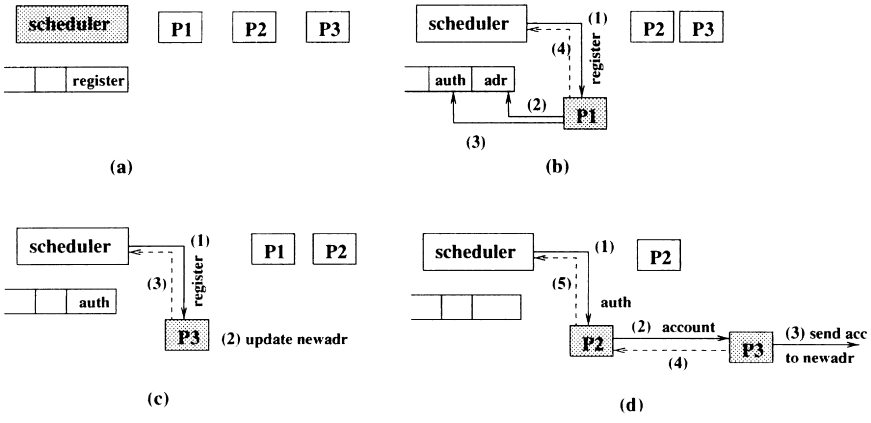


Fig. 16. Process call sequence for example of Fig. 8 (correct behaviour).

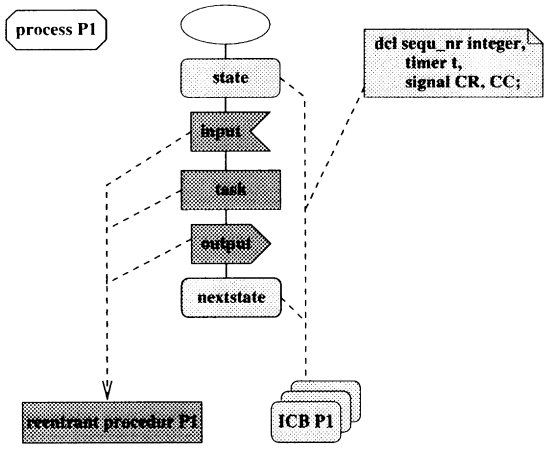


Fig. 17. Splitting of SDL processes into data and behaviour part.

activity thread model processes, since both are implemented as procedures. Server model processes are identified by their names whereas the activity thread procedures are identified by the signal name. This ready queue has to provide the following information:

- the identity of the receiving process instance;
- the identity of the sending instance;
- the name of the signal.

The distinction between a common input queue and a ready queue introduces additional overhead as the identity of processes that received a signal has to be appended to the ready queue. This overhead can be avoided if the global input queue is used as ready queue. We call this combined queue *procedure call list* (PCL). It provides all information needed for scheduling and contains additionally a *pointer to the data of the signal*.

Further, the runtime system has to provide a timer management and an interface to the environment. There are no specific demands on these components by the configurable compiler. Fig. 18 shows the structure of such a runtime support system. It can be realized using operating system threads. The description of a concrete solution of such a runtime system is given in Ref. [26].

6.3. Configuration and control language

A configurable compiler requires language means to support the configuration and to control the code generation process. For this purpose, a specific control language is required which allows the implementor to indicate implementation decisions as the selected implementation technique or to introduce parameters of the target system. This information refines the protocol specification to an implementation-oriented specification that is finally the basis for code generation. These refinements can be either given in a separate configuration file or can be embedded into the formal specification. The main drawback of configuration files is that it is difficult to specify implementation decisions concerning single transitions, as it is for instance needed for integrated layer processing implementations. When embedding implementation decisions into specifications this shortage can be avoided. This can be carried out by extending the FDT or by applying annotations. The extension of the FDT would lead to interferences with already available tools. Therefore, we decided to define a control and configuration language for SDL called iSDL [25].

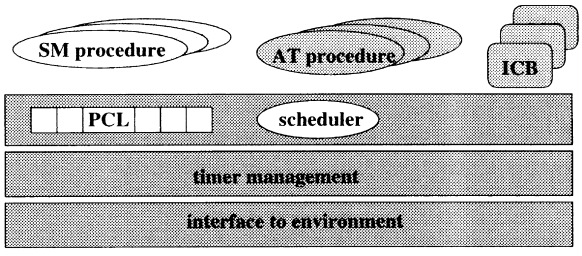


Fig. 18. Runtime support system for a configurable compiler.

The iSDL statements are syntactically included as comments in the SDL specification (SDL/GR: comment and text symbols, SDL/PR: comments). The use of comments does not interfere the processing of the specification with existing SDL tools such as SDT [17] and Geode [34] and simplifies the reuse of the specification. The iSDL annotations are marked by the keywords iSDL and iSDLend, or for short by ${ and }. The iSDL statements are separated by semicolon.

iSDL annotations may be introduced at any level of an SDL specification. Annotations at system or block level are applied to all substructures. iSDL descriptions given at system level may be refined or changed at block level. Note that for elaborating the implementation-oriented specification the implementor does not need a detailed knowledge of the structure of the compiler. She/he must be familiar with the selected implementation technique and the memory organization of the target system. Whether the chosen implementation strategy may be applied in combination with the given implementation context is detected automatically during compilation. Below we give some examples of elements of the annotation. A complete description of iSDL is given in Ref. [25].

- *Mapping on the target hardware*. iSDL assists to control the mapping of processes of the implementation on processor clusters, e.g.
  `define cluster` cluster1: processor1, processor2 `shared memory`; `mapping` process_1, process_2 `on` cluster1;
  The implementation of the signal exchange depends on the underlying hardware. If SDL processes are mapped on processors with distributed memory message passing has to be applied. For shared memory references are exchanged via the procedure call list. Shared memory allows more efficient implementations.
- *Selection of the process model*. iSDL allows the implementor to select between the supported process

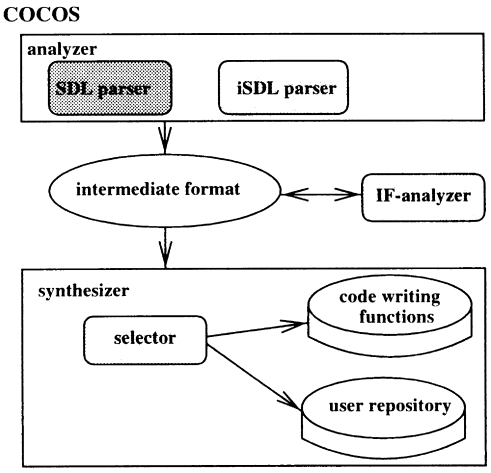


Fig. 19. Structure of the COCOS compiler.

models, e.g.

`process model` server;

The server model can be used independently of the memory organization of the target hardware. The activity thread model can only be applied for target systems with shared memory.

- *Definition of time units*. SDL does not allow defining timer ranges required for the final implementation. This information can be added by means of iSDL. The predefined time units are s, ms, μs.
- *Definition of the length of input and save queues*. In SDL, the length of an input queue is unlimited. For the generation of efficient implementations, it is necessary to know the length of the input queues to initialize the PCL when setting up the SDL system. Thus, no time is lost for allocating entries to the PCL during runtime.

  `input queue` 30;

  For the same reason, iSDL also supports the definition of the length of the save list. The distinction between input queue and save list is necessary, because activity thread implementations do not use input queues at all.

Further, iSDL provides means to support integrated layer processing implementations and to evaluate the performance of the selected implementation solution.

6.4. The SDL compiler COCOS

In this section we give an example for a configurable compiler which has been implemented at the Brandenburg University of Technology for the FDT SDL. It is called COCOS (*COnfigurable COmpiler for SDL*) [26]. Owing to its experimental character the compiler does not support the full scope of SDL. The COCOS parser accepts the full SDL'92 language. The code generation supports SDL'92 without object oriented features and the axiomatic definition of data types. COCOS implement the principles the principles, which were introduced in the previous sections.

A configurable compiler requires special means for the parsing process. In addition to the syntax and semantics check of the formal description, it has to prove the correctness of the implementation-oriented settings. As these statements are not handled as pure FDT extensions a second parser is needed for their compilation. To prove whether the implementation oriented settings assure a conform implementation of the formal description it is useful to transfer both specifications into a common intermediate representation which can be analysed by the same tool component. The intermediate representation is also required to determine, process, and document changes in the implementation order as it is applied, for instance, for the transition reordering. Moreover, a selector component is required which controls the generation of different code sequences for an FDT statement from the intermediate representation depending on the selected implementation model.

The COCOS compiler implements these design principles. It consists of three main components: the analyser, the intermediate format analyser (IF-analyser), and the synthesizer. The structure of the compiler is depicted in Fig. 19.

The basis for the code generation is an SDL protocol specification which was refined with implementation related information presented in iSDL. The implementation-oriented specification is input to the analyser for syntax checking. The analyser consists of two parsers, one for the SDL text and one for the iSDL annotation. It outputs code in an intermediate format in which the SDL constructs are denoted by an implementation oriented representation or a corresponding default value. The intermediate representation is used for computations needed to detect semantics violations and to optimize the performance. These computations are performed by the IF-analyser. They comprise:

- the detection of cyclic execution which prevents the application of the activity thread model;
- the identification of the receiving process instances as far as possible at compile time.

If semantic violations are detected the code generation stops and indicates the respective errors. Otherwise, the intermediate code including the results of the analysis is input into the synthesizer for final code generation.

The synthesizer consists of three parts: the *selector*; a set of code *writing functions*; and a *user repository*. For certain SDL statements, e.g. output, there exist several mapping rules. Each mapping rule implements a separate code writing function. The selection of the concrete rules is determined by the iSDL specification. The selector identifies the mapping rule and calls the respective code writing function. The code segments provided in the user repository are inlined by the selector at the places where the corresponding actions are specified. This is also controlled by means of iSDL.

The synthezier generates various C files, which comprise the implemented SDL processes and the required configurable components of the runtime support system (see Fig. 20). All these files plus the static part of the runtime system

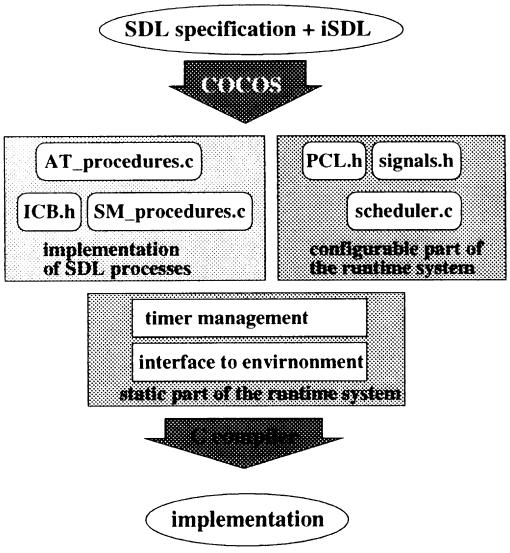


Fig. 20. Code generation process.

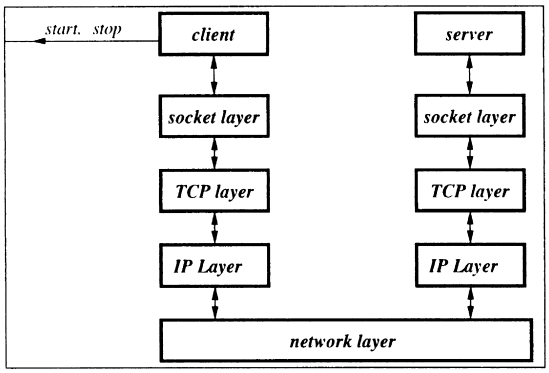


Fig. 21. Structure of the applied SDL specification.

(timer management, interface to the environment) have to be translated by a C compiler to obtain executable code.

7. Performance measurements

In this section we describe measurements which demonstrate the effect of the approach. For the measurements, we used the SDL specification of a client/server application based on a TCP/IP protocol stack described in Ref. [18]. The structure of the specification is depicted in Fig. 21.

The client process generates data that have to be transmitted to the server process and to be confirmed. Before the client can open a TCP connection it has to ask for a socket. The server process initiates a passive open to the socket layer. Then it listens. The socket process forwards the application data to the TCP process and vice versa. The TCP process contains the known functionality of the protocol: division of the application data into TCP segments, timer control of the transmission, discarding of duplicate IP packets, flow control, congestion control and error handling. The IP process has a simplified functionality in this specification. It only supplements the IP headers to TCP packets and assigns the incoming IP packets to TCP connections. Fragmentation is not included. The network process stimulates the network and transfers the packets from the client to the server site.

We generated two implementations with COCOS. In the first one all SDL processes were implemented as server model processes. In the second we combined server model processes and activity thread model processes. The SDL process *SOCKET*, *TCP* and *IP* are implemented as activity thread processes using *transition reordering*. We also present measurements which compare the efficiency of these implementations with code generated by a commercially available tool—the *Cadvanced* code generator of the SDT tool version 3.4 [33].

COCOS as well as *Cadvanced* implement the whole specification by a single operating system process. Thus, the interfaces between the generated code and the operating system do not influence the measurement results. Note that both compilers use the same technique to reduce the operating system overhead. For each signal that is sent memory has to be allocated to store its data. This memory is usually allocated by the operating system at runtime. *Cadvanced* [33] as well as COCOS allocate this memory during system initialization.

The generated C code of all implementations was compiled with the gcc compiler without any optimizing options in order to assure that the performance gain is achieved by the applied implementation techniques and not due to compiler optimizations.

We measured the time for the transmission of a sequence of packets starting from the transmission of the first packet until the reception of the last acknowledgement at client site. The process client sends the signal start to the environment when it forwards the first packet. The end of the transmission is indicated by the stop signal.

The measurements were made on a Sun Sparc 20 workstation with four processors and Solaris 2.5 as the operating system. We measured the time for the transmission of 1000, 10 000 and 20 000 packets with a packet size of 1024 byte. Each measurement was repeated 50 times. To exclude exceptional behaviour we determined the frequency of the measured values. Then we defined an interval with a length of 100 ms around the most frequent values. The values shown in Fig. 22 represent the arithmetic middle of the values of this interval.

The results show that the implementations generated by COCOS achieve a 50–110% better performance than those of Cadvanced. For the server model implementation, the performance gain is achieved by using a predefined length of the individual input queues and due to the optimized scheduling (see Section. 6.2). The activity thread based

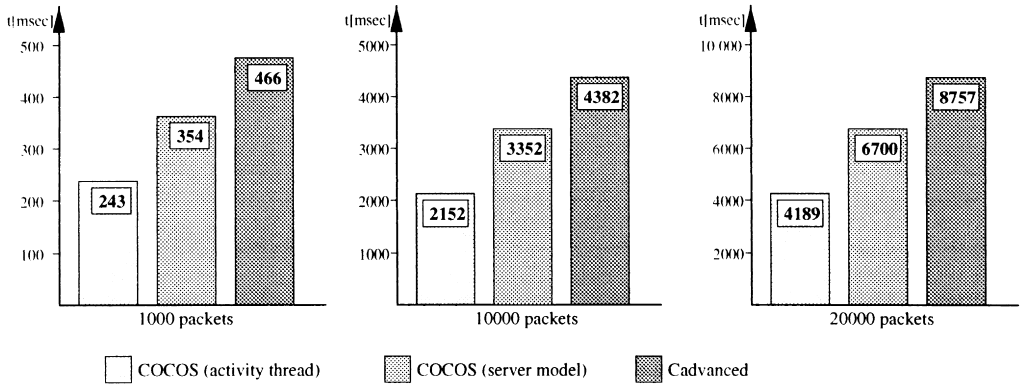


Fig. 22. Measurements of different implementation strategies.

processes do not use the procedure call list for communication. Thereby they do not have to be scheduled by the runtime support system. This brings another performance gain of 35%.

Further performance improvements will result from the avoidance of data copy operations [26] and an optimized timer management.

8. Concluding remarks

In this paper, we have discussed steps to improve the quality of automated implementations. The objective of our work has been to increase the performance of automatically derived code so that this technique can be applied to the implementation of real-life protocols. This can only be achieved if the efficiency of the generated code comes close to that of manually developed implementations. A successful solution of this problem will considerably increase the acceptance of formal description techniques in practice because it permits a continual application of a FDT based technology in all phases of the protocol development process from design to test.

Our approach combines two concepts: the integration of proved implementation techniques applied for manual coding and the integration of features to make the transformation process more flexible. This helps to overcome the rigid mapping rules applied so far in FDT compilers. It gives the implementor the opportunity to take the characteristics of the given implementation context into account. She/he can select the most appropriate implementation model and generate a tailored runtime support system.

For the code generation process, we use an implementation oriented specification level that refines the protocol specification by adding all needed information about the implementation and the target system. The introduction of an implementation oriented specification level brings several benefits for the implementation process: It allows introducing implementation-oriented information into the protocol specification. It documents implementation design decisions at specification level. This facilitates the reuse of the implementation-oriented specification and the implementation as well. It can be further used to add features to evaluate the performance of different implementation options.

By integrating efficient implementation techniques which are proved in manual coding we have shown that, on the one hand, techniques such as the activity thread model can be applied for automated protocol implementation and that, on the other hand, there is still a large potential for optimizations in this area. The presented transition reordering approach allows it to dissolve most semantic conflicts during compilation.

The described concepts have been implemented in the configurable SDL compiler COCOS. The measurements we have presented indicate a considerable increase in the efficiency of the generated code compared to existing SDL tools as the *Cadvanced* compiler. Efficiency improvements from 50 up to 110% depending on the applied technique show that a considerable progress has been achieved.

In a next step, we will further evaluate the applicability of the different implementation techniques of COCOS including the integrated layer processing component. So we plan to compare our tool with hand-coded implementations of appropriate protocols. The latter seems to be a simple task, but there scarcely exist real-life protocol implementations of protocols that are derived from SDL specifications, e.g. for the TCP/IP protocol stack, that could be taken as basis for such comparisons. For TCP/IP, there is the additional problem that most real-life implementations are kernel-integrated implementations. Such transformations from a formal description are not possible at the current state of the research. They require further investigations. Though TCP/IP implementations will scarcely be the preferred application area of automated protocol implementation techniques. We see the application of these techniques mainly for application-near protocols and for newly designed protocols to fast derive an efficient implementation from the formal description of the design, which can be further, optimized if required.

Acknowledgements

The research described is supported by the Deutsche Forschungsgemeinschaft under grant No. 1273/11-1.

References

[1] M. Abbott, L. Peterson, Increasing network throughput by integrating protocol layers, IEEE/ACM Transaction On Neworking 1 (1993) 600–610.
[2] B. Ahlgren, M. Bjorkman, P. Gunningberg, Integrated layer processing can be hazardous to your performance, in: W. Dabbous, C. Diot (Eds.), Protocols for High-Speed Networks V, Chapman and Hall, London, 1996, pp. 167–191.
[3] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, Reading, MA, 1996.
[4] T. Braun, I. Chrisment, C. Diot, F. Gagnon, L. Gautier, P. Hoschka, ALFred, an ALF/ILP Protocol Compiler for Distributed Application Automated Design, Rapport de Recherche Nr. 2786, INRIA, 1996.
[5] T. Braun, C. Diot, Protocol implementation using integrated layer processing, In ACM SIGCOMM, 1995.
[6] J. Bredereke, R. Gotzhein, Increasing the concurrency in Estelle, in: R. Tenney (Ed.), Formal Description Techniques VI, North-Holland, Amsterdam, 1994.
[7] S. Budkowski, Estelle development toolset, Computer Networks and ISDN Systems 25 (1992) 63–82.
[8] O. Catrina, E. Lallet, S. Budkowski, Automatic protocols implementation using Estelle Development Toolset (EDT), Institut National des Télécommunications (INT) Evry, Rapport de Recherche, 1997.
[9] D.D. Clark, The structuring of systems using upcalls, Proceedings of the 10th ACM SIGOPS Symposium on Operating Systems and Principles (1985) 171–180.
[10] D.D. Clark, D.L. Tennenhouse, Architectural considerations for a new generation of protocols, ACM SIGCOMM (1990) 200–208.
[11] Y. Dong, Y. Lu, Q. Gao, Specification, validation and implementation of ATM UNI signaling protocols in SDL, in: R. Dssouli, G. Bochmann, Y. Lahav (Eds.), SDL'99 The Next Millenium, Elsevier, Amsterdam, 1999, pp. 341–354.
[12] S. Fischer, W. Effelsberg, Efficient configuration of protocol software for multiprocessors, in: R. Puigjaner (Ed.), High Performance Networking VI, Chapman and Hall, London, 1995, pp. 195–210.
[13] R. Gotzhein, J. Bredereke, W. Effelsberg, S. Fischer, T. Held, H. Koenig, Improving the efficiency of automated protocol implementation using Estelle, Computer Communications 19 (1996) 1226–1235.
[14] P.-O. Hakansson, J. Karlsson, Verhaard.: combining SDL and C, in: A. Cavalli, A. Sarma (Eds.), SDL'97 Time for Testing, Elsevier, Amsterdam, 1997, pp. 383–396.
[15] T. Held, H. König, Increasing the efficiency of computer-aided protocol implementations, in: S. Vuong, S. Chanson (Eds.), Protocol Specification, Testing and Verification XIV, Chapman and Hall, London, 1995, pp. 387–394.
[16] R. Henke, H. König, A. Mitschele-Thiel, Derivation of efficient implementations from SDL specifications employing data referencing, integrated packet framing and activity threads, in: A. Cavalli, A. Sarma (Eds.), SDL'97 Time for Testing, Elsevier, Amsterdam, 1997, pp. 397–414.
[17] R. Henke, A. Mitschele-Thiel, H. Koenig, On the influence of semantic constraints on the code generation from Estelle specifications, in: T. Mizuno (Ed.), Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/PSTV XVII'97), Chapman and Hall, Amsterdam, 1997, pp. 399–414.
[18] J. Hintelmann, R. Westerfeld, Performance analysis of TCP's flow control mechanisms using queueing SDL, in: A. Cavalli, A. Sarma (Eds.), SDL'97 Time for Testing, Elsevier, Amsterdam, 1997, pp. 69–84.
[19] B. Hofmann, Generation of efficient protocol implementations based on Estelle specifications, PhD Thesis, University of Mannheim, 1993 (in German).
[20] G.J. Holzmann, Design and validation of protocols: a tutorial, Computer Networks and ISDN Systems 26 (1993) 981–1017.
[21] ISO IS 9074: Estelle—A Formal Description Technique Based on an Extended State Model, 1989.
[22] ITU-T. Z.100: Specification and Description Language (SDL), ITU, 1993.
[23] H. Koenig, Experience in computer-aided protocol implementation, in: L. Csaba, K. Tarnay, T. Szentivanyi (Eds.), Computer Networking, North-Holland, Amsterdam, 1990, pp. 385–394.
[24] E. Lallet, S. Fischer, J.-F. Verdier, A new approach for distributing Estelle specifications, in: G. Bochmann, R. Dssouli, O. Rafiq (Eds.), Proceedings of the Eighth International Conference on Formal Description Techniques, Montreal, 1995, pp. 439–448.
[25] P. Langendörfer, Definition of the implementation-oriented annotation iSDL, Preprint. I-04/1998, BTU Cottbus.
[26] P. Langendörfer, H. König, COCOS—a configurable SDL compiler for generating efficient protocol implementations, in: R. Dssouli, G.V. Bochmann, Y. Lahav (Eds.), SDL'99 The Next Millennium, Elsevier, Amsterdam, 1999, pp. 259–274.
[27] P. Langendörfer, H. König, Automated protocol implementations based on activity threads accepted, Seventh International Conference on Network Protocols, IEEE Society Press, New York, 1999 (in Press).
[28] S. Leue, P. Oechslin, On parallelizing and optimizing the implementation of communication protocols, IEEE/ACM Transactions on Networking 4(1) (1996).
[29] N. Mansurov, A. Chernov, A. Ragozin, Industrial strength code generation from SDL, in: A. Cavalli, A. Sarma (Eds.), SDL'97 Time for Testing, Elsevier, Amsterdam, 1997, pp. 415–430.
[30] O. Monkewich, SDL-based specification and testing strategy for communication network protocols, in: R. Dssouli, G.V. Bochmann, Y. Lahav (Eds.), SDL'99 The Next Millennium, Elsevier, Amsterdam, 1999, pp. 123–134.
[31] R. Plato, T. Held, H. König, PARES—a portable Estelle translator, in: P. Dembinski, M. Sredniawa (Eds.), Protocol Specification, Testing and Verification XV, Chapman and Hall, London, 1996, pp. 383–399.
[32] L. Svobodova, Implementing OSI Systems, IEEE Journal on Selected Areas in Communications 7 (7) (1989) 1115–1130.
[33] Telelogic Malmö AB, SDT 3.4 User's Guide. SDT 3.4 Reference Manual, 1999.
[34] Verilog Toulouse: Geode User's Guide, Reference Manual, 1995.