Automated Protocol Implementations Based on Activity Threads*

Peter Langendörfer and Hartmut König
Department of Computer Sience
Brandenburg University of Technology at Cottbus
PF 101344, D-03013 Cottbus
Germany
{pl,koenig}@informatik.tu-cottbus.de

Abstract

In this paper we present a new approach for the automated mapping of formal descriptions into activity thread implementations. Our approach resolves semantic conflicts by reordering of statements at compile time. This simplifies the mapping process and considerably improves the efficiency of the generated code. The approach is implemented in the SDL compiler *COCOS*. We describe the approach as well as its implementation and prove how semantic conflicts are resolved. Finally we present measurements which show the achieved performance gain.

1 Motivation

The development of communication software is known to be an expensive and tedious process. Formal description techniques (FDTs) can help to considerably increase the quality of the protocols and telecommunication systems, and to shorten the time for their development. However, the benefits of FDTs are mainly used in the design, specification, performance prediction, verification, and testing phase. Automated implementation still represents a gap in this chain of development steps. Protocol implementations are mostly carried out in traditional manner, i.e. manually. The main reason for this is that code automatically generated by an FDT compiler is mostly inadequate for applications in a real-life environment [Held95, Mans97]. This shortage of automated protocol implementation cannot be compensated by other benefits such as a considerable reduction of the duration of the implementation process, independence of subjective implementation decisions, better conformance to the specification, and simplification of changes.

There have been many attempts to improve the efficiency of automatically derived code [Held95, Gotz96]. These approaches are mostly based on the use of improved algorithms for implementing features of the FDT semantics. The achieved performance increase of these approaches is limited. The use of advanced implementation techniques such as activity threads (upcalls) [Henk97] or integrated layer processing [BrDi95] appears more promising.

This paper deals with activity thread implementations. We present an approach for the automated mapping of formal descriptions into activity thread implementations. It is based on a new technique, called *transition reordering* to resolve semantic conflicts at compile time. We focus on SDL [ITU93] but the transition reordering can also be applied for other FDTs if their specific semantic constraints are taken into account.

The is paper structured as follows. After giving a short introduction into SDL we outline the activity thread mapping principle and the transition reordering approach. Next we describe their implementation in the SDL compiler *COCOS*. Finally we present measurements which prove the achieved performance increase including a comparison with the *Cadvanced* code generator of the SDT tool.

2 SDL

SDL (Specification and Description Language) [ITU93] is the specification language of the ITU-T (former CCITT). 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 and the textual representation SDL/PR. The graphical representation is more widely used.

SDL distinguishes 3 description levels: system, block, and process. The system level forms the frame of the description. It represents an abstract machine which com-

*The research described is supported by the *Deutsche Forschungsgemeinschaft* under grant Ko 1273/11-1

municates with its environment. A system consists of several blocks which 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 an unique identification at runtime.

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 queue.

3 Mapping of SDL specifications onto activity threads

In this section we describe the principle of the transformation of SDL specifications onto activity threads as well as the possible semantic conflicts.

3.1 Mapping principle

The activity thread technique [Clar85, Svob89] implements a protocol entity as a set of procedures where for each input signal a corresponding procedure exists. The active 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.

The activity thread technique provides very efficient implementations because it avoids the storing of signals when calling the next protocol entity [Clar85]. 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. They are discussed in the next subsection. Another constraint of this transformation is that spontaneous transitions cannot be handled [Henk97]. They must be removed when preparing the implementation. Therefore, SDL compilers usually apply the server model which supports a straight mapping of the SDL semantics instead of the more complicated activity thread approach.

3.2 Semantic conflicts

The semantic conflicts which may appear when directly mapping SDL specifications onto 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. Due to lack of space we cannot discuss all possible semantic conflicts here. We focus on transition interferences and overtaking of signals in combined implementations[1].

3.2.1 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, and
- 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

[1] The technique to resolve overtaking of signals in pure activity thread implementations is described in detail in [Lang99a]

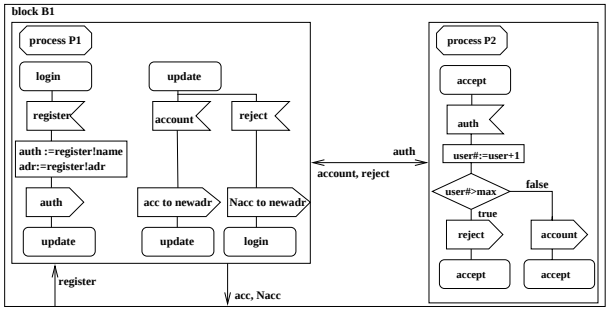


Figure 1: Example for discarding of a signal

the assignment of next state. In this case, the process instance cannot change its state until the called procedure has returned control. Figure 1 gives an example for such a situation. An implementation that is conform to the specification 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 for the output of *auth* is finished. Figure 2 shows the resulting cyclic process call sequence.

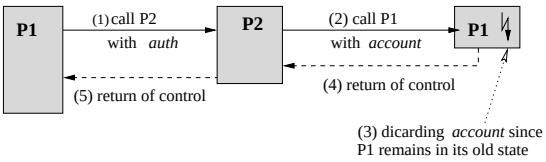


Figure 2: Process call sequence for example of Figure 1

3.2.2 Overtaking of signals

An implementation which uses both process models combines rather different implementation approaches. Unlike the activity thread model the server model implements each SDL process by a task which are usually executed in round robin manner by a runtime system scheduler. Each process posseses an input queue. For increasing the efficiency of server model implementations it is useful to replace the individual input queues by a global input queue [Held95, Lang99]. In such a combined implementation overtaking of signals may appear if the following prerequisites are fullfilled:

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

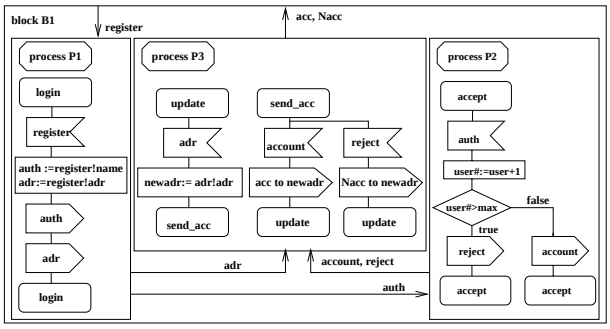


Figure 3: SDL system with possibile signal overtaking in case of combined implementations

Figure 3 shows a specification which fulfills these preconditions. Process $P1$ is mapped on a server model process and processes $P2$ and $P3$ are implemented using the activity thread model. In a straight-forward implementation of the specification the signal *register* would trigger the following execution. Process *P1* appends the signals *auth* and *adr* to the global input queue (Figure 4(b)). Then the scheduler executes $P2$ with signal $auth$. $P2$ immediately sends one of the signals *account* or *reject* to $P3$ ((Figure 4(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. 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.

4 Transition reordering

4.1 Principle

So far semantic conflicts described above could only be resolved at runtime. In [Henk97] this is done by extending the activity thread implementation with an activity thread scheduler and a global signal list. Figure 5a represents 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 process 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

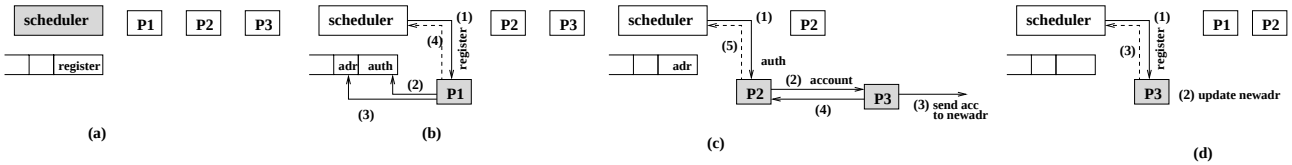
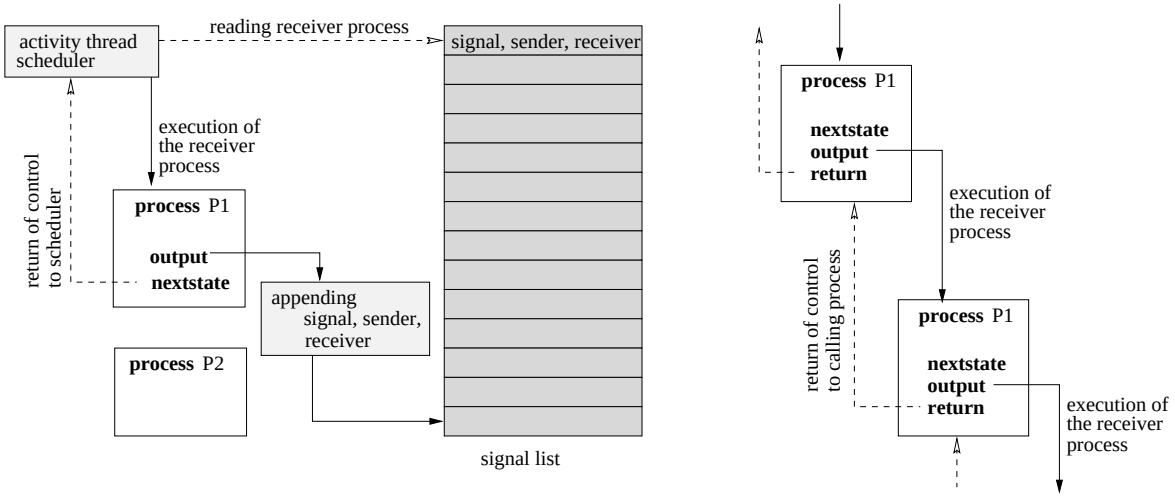


Figure 4: Process call sequence for example of Figure 3



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

Figure 5: Mapping on activity threads

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. Such an activity thread implementation corresponds to a server model implementation with a global input queue as described above.

In the following we present an approach, called *transition reordering*, which allows it to handle the semantic conflicts during compilation. The *transition reordering* designates that the SDL statements of a transition are not implemented in the order as they are specified. They are reordered at compiler time in such a way that the semantic conflicts described above cannot occur. The approach adapts the principle of code restructuring which is applied for loop optimization in modern programming language compilers [Aho96]. The reordering only concerns the `output` statements. They are implemented in two steps:

- Replace the `output` statement by an operation which stores the signal.
- Append the `output` statement with the stored signal to the end of the transition. If a `decision` statement follows the `output` statement the modified `output` statement has to be appended to each branch of the `decision` statement.

All other SDL statements are implemented in the order as 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 when 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 Figure 5b). Using transition reordering the indirect realization of the activity thread proposed in [Henk97] can be avoided. The transitions can straightly be mapped on procedures, i.e. a direct derivation of activity threads is possible by means of transition reordering.

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

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 the data transferred by a signal cannot be modified by a statement following the `output` statement. 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
- a cyclic process call sequence triggered by a multiple output terminats with the same transition that started the cyclic execution.

In these execptional cases which can be recognized during compilation a buffered signal exchange, combarable to [Henk97], is generated.

In the next subsections we show how the semantic conflicts discussed in section 3 can be avoided by applying *transition reordering*.

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. Figure 6 shows the specification of the example from Figure 1 after transition reordering.

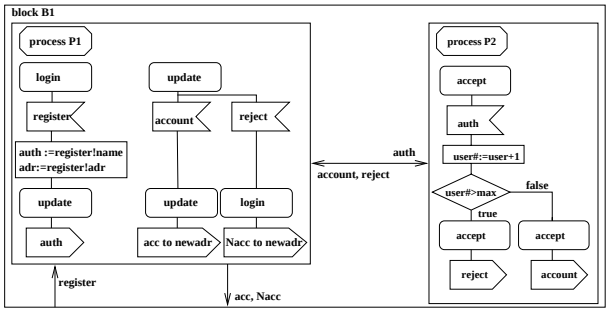


Figure 6: Example of Figure 1 after transition reordering

Figure 7 depicts the respective process call sequence and indicates that the implementation is executed correctly.

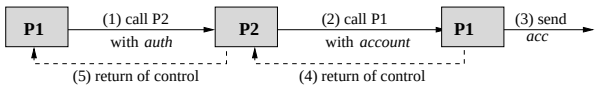


Figure 7: Process call sequence for example of Figure 6

4.3 Avoiding overtaking of signals

Since at compile time the structure of the SDL system as well as the logical structure of the implementation are known the compiler can determine whether the prerequisites for signal overtaking are fulfilled. This is done by analysing the execution pathes and the state changes of transitions as described below. First for each signal that a server model process sends to an activity thread process the set of successor signals and the set of their 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 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 execute the corresponding transition. Thus, we get the execution path triggered by each `output` statement. Then it is checked whether there are processes that get signals via the global signal 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 global input list reach their receivers always first. This is achieved by implementing first the output statements that do not trigger the sending of signals to a process that is also receiver of signals send in the transition under implementation. Figure 8 shows the execution order of the example given in Figure 3 after reordering of the

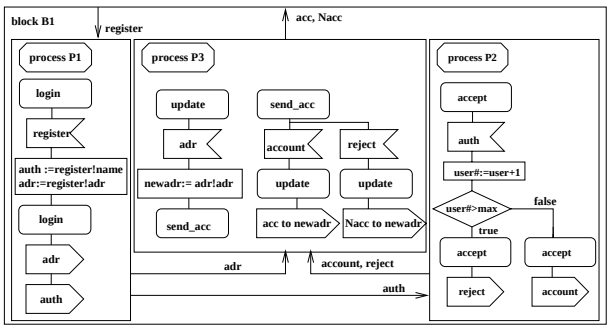


Figure 8: Figure 3 after transition reordering

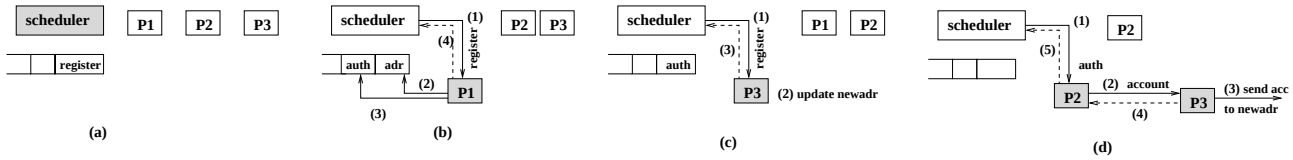


Figure 9: Process call sequence for example of figure 8

transitions. Note that the outputs of process *P1* are not only delayed but that their order has also been changed. Figure 9 shows again the resulting process call sequence. In this implementation the signals *adr* and *account* arrive in the correct order at *P3*.

5 Tool support

The transition reordering approach as described in Section 4 has been implemented in the SDL compiler *COCOS (Configurable Compiler for SDL)* [Lang99]. *COCOS* is a configurable compiler that supports different implementation strategies. Currently these are the server model and the activity thread model. Another feature of *COCOS* is its ability to adjust to the given implementation context. This feature overcomes one of the main shortages of automated implementation: the rigid implementation model. Depending of the selected implementation strategy a tailored runtime system is generated which optimally adapts to the implementation environment. The configuration process is controlled by implementation oriented annotation called iSDL which beside the selection of the mapping strategy integrates implementation oriented information such as number of processors and kind of memory organization into the code generation process. A detailed description of *COCOS* and iSDL is contained in [Lang99].

The *COCOS* compiler consists of three main components: the analyser, the intermediate format analyser (IF-analyser), and the synthesizer. The basis for the code generation is an SDL protocol specification which was refined with implementation related information presented in iSDL. This 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 where 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 execution pathes which prevent the application of the activity thread model,
- the avoidance of data copy operations, and
- the identification of the receiving process instances if they are not explicitly given.

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

The synthesizer consists of three parts: the selector, a set of code writing functions and a user repository. The selector is controlled by a set of mapping rules which determine the transformation from SDL to C. For each mapping rule, a separate code writing function is implemented. 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. For certain SDL statements like the `output` statement, several mapping rules exist. In activity thread implementations the sending of signals is mapped on a procedure call, whereas it has to be implemented by a buffered handing over of references in server model implementations. The code segments provided in the user repository are inlined by the selector at the places where the corresponding actions are specified.

6 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 [Hint97]. The client process generates data which have to be transmitted to the server process and 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

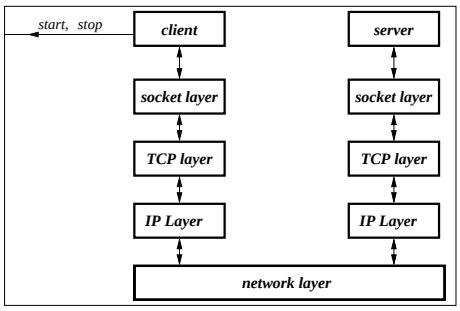


Figure 10: Structure of the applied SDL specification

incoming IP packets to TCP connections. Fragmentation is not included. The network process simulates 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 mentioned as *COCOS SM* in Figure 11. In the second we combined server model processes and activity thread model processes. The SDL processes $SOCKET$, TCP and IP are implemented as activity thread processes using the *transition reordering*. This implementation is mentioned as *COCOS COMB* in Figure 11. Additionally, we applied a commercially available tool - the *Cadvanced* code generator of the SDT tool version 3.4 [Tele98] for code generation. In all three implementations the whole specification is implemented as a single operating system process. Thus, the interfaces between the generated code and the operating system do not influence the measurement results. The generated C code of all implementations was compiled with the gcc compiler without any optimizing options in order to asure that the performance gain is achieved by using the *transition reordering* and not due to compiler optimizations.

We measured the transmission-time for 5 Mbyte, 10 Mbyte, 20 Mbyte and 50Mbyte files. The process client sends the signal *start* to the environment to start the measurement when it begins the transmission. The end of the transmission is indicated by the signal *stop* that the process client sends when it received the acknowledgement for the last segment of the transmitted file.

The measurements were made on a Sun Sparc 20 workstation with four processors and Solaris 2.5 as operating system. Each measurement was repeated 50 times. First we computed the arithmetic middle of all measured values and then we eliminated all measured values that differed more than 5 per cent. After that the arithmetic middle was computed again. These results are given in Figure 11.

The results show that activity thread implementations with transition reordering achieve a remarkable better performance (between 25 - 30 per cent) than those which apply the approach of [Henk97]. This increase can be explained by the optimized mapping strategy which resolves semantic conflicts at compile time.

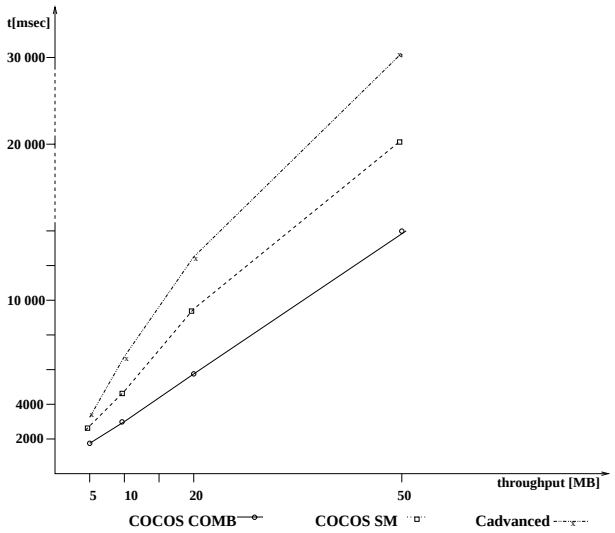


Figure 11: Measurements of different implementation strategies

The comparison with the *Cadvanced* code generator shows that both activity thread implementations achieved an about 60 and 120 per cent, respectively, better performance. This proves that by using the activity thread technique a considerably better performance can be obtained. 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* [Tele98] as well as *COCOS* allocate this memory during system initialization.

Further performance improvements are expected the avoidance of data copy operations [Lang99] and an optimized timer management.

7 Concluding remarks

In this paper, we have presented with the *transition reordering* an approach which allows it to efficiently map SDL specifications onto activity thread implementations. The transition reordering resolves almost entirely semantic conflicts at compile time. Additional runtime support is only needed for some exceptional cases. The approach also supports combined server and activity thread model implementations.

The approach has been integrated in the configurable SDL compiler *COCOS*. The measurements we have presented indicate a considerable increase in the efficiency of the generated code. For the application of the transition reordering, an increase of about 25 to 30 per cent was measured. Compared with the code generated by a commercially available tool like the *Cadvanced* compiler an increase up to 120 per cent was observed. Further improvements are expected by optimizations such as the avoidance of data copy operations and a more efficient timer management which are step by step introduced into the *COCOS* compiler. The considerable progress achieved and the mentioned further optimizations show that there is still a large potential for improving the efficiency of automated code generation.

Currently we are introducing the integrated layer processing as third implementation strategy in the configurable compiler. Thereafter, we plan to evaluate our tool and the different implementation techniques by comparing it with hand-coded implementations of the TCP/IP protocol stack or other appropriate protocols. The latter task seems simple, but there scarcely exist real-life protocol implementations of the TCP/IP protocol stack or other protocols which are derived from a SDL specification and thus could be used as a basis for such a comparison. For TCP/IP, there is the additional problem that most real-life implementations are kernel-integrated implementations. The derivation of such kind of implementations is not the objective of our current research. TCP/IP implementations though will not be the main application area of automated protocol implementation techniques. We see the potential application of automated implementation techniques more in the area of application protocols and especially for newly designed protocols to fast derive an implementation from the formal description which can be optimized if needed. This will help to close the ”implementation gap” in the protocol development process mentioned in the introduction and allow a continuous protocol development based on formal descriptions from the design via automated code generation to testing.

References

[Aho96] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers: principles, techniques, and tools*. Addison Wesley, 1996.

[BrDi95] T. Braun; C. Diot.: *Protocol implementation using integrated layer processing*. In ACM SIGCOMM, 1995.

[Clar85] Clark, D. D.: *The structuring of systems using upcalls*. Proc. 10 th ACM SIGOPS Symp. Oper. Syst. Principles, 1985, pp. 171-180.

[Gotz96] Gotzhein, R.; Bredereke, J.; Effelsberg, W.; Fischer, S.; Held, T.; Koenig, H.: Improving the Efficiency of Automated Protocol Implementation Using Estelle. Computer Communications 19 (1996), pp. 1226-1235.

[Held95] Held T.; König, H.: *Increasing the Efficiency of Computer-aided Protocol Implementations*. In Vuong, S., Chanson, S. (eds.): Protocol Specification, Testing and Verification XIV, Chapman & Hall, 1995, pp. 387-394.

[Henk97] Henke, R.; König, H.; Mitschele-Thiel, A.: *Derivationof Efficient Implementations from SDL Specifications Employing Data Referencing, Integrated Packet Framing and Activity Threads*. In Cavalli, A.; Sarma,A. (eds.): SDL'97 Time for Testing. Elsevier, 1997, pp. 397-414.

[Hint97] Hintelmann, J.; Westerfeld, R.: *Performance Analysis of TCP's Flow Control Mechanisms Using Queueing SDL*. In Cavalli, A.; Sarma, A. (eds.): SDL'97 Time for Testing. Elsevier, 1997, pp. 69-84.

[ITU93] ITU-T. Z.100: *Specification and Description Language (SDL)*. ITU, 1993.

[Lang99] Langendörfer, P.; König, H.: *COCOS - A Configurable SDL Compiler for Generating Efficient Protocol Implementations*. In Dssouli, R.; Bochmann, G.V.; Lahav, Y. (eds.): SDL'99 The next Millennium, Elsevier, 1999, pp. 259-274.

[Lang99a] Langendörfer, P.; König, H.: *Deriving Activity Thread Implementations from Formal Descriptions Using Transition Reordering*. accepted for the 12th International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'99), to appear with Kluver, 1999.

[Mans97] Mansurov, N.; Chernov, A.; Ragozin A.: *Industrial Strength Code Generation from SDL*. In Cavalli, A.; Sarma, A. (eds.): SDL'97 Time for Testing. Elsevier, 1997, pp. 415-430.

[Svob89] Svobodova, L.: *Implementing OSI Systems*. IEEE Journal on Selected Areas in Communications, 7(7),1989, pp. 1115-1130.

[Tele98] Telelogic Malmö AB: *SDT 3.4 User's Guide*. SDT 3.4 Reference Manual. 1998.