Framework for Automatic SDL to C++ Translation

Dirk Trossen, Christian Cseh, and Roman Kogan
Department of Computer Science IV
Technical University of Aachen
Ahornstr. 55
D-52074 Aachen, Germany
{dirk,cseh,kogan}@i4.informatik.rwth-aachen.de

Key words: SDL, Automatic SDL Translation, Lazy Evaluation

Abstract: Due to the increasing complexity of distributed systems the need for a formal description of these systems arises. Specification description languages like SDL enable the formal specification of distributed systems for verification purposes of the system. For simplification of the development process the need for an automatic translation from the specification language to a programming language like C or C++ arises. High performance and a good readability of the translated code are crucial for the usability of an automatic translation. This paper introduces a framework for the automatic translation of an SDL specification to C++ providing high performance together with good readability of the resulting code. For that we present an object model together with an efficient implementation of the finite state machine with constant costs only using polymorphism. Additionally we present mechanisms to reduce copy and memory allocation operations. A performance evaluation for these mechanisms is also presented.

1. INTRODUCTION

Due to the increasing complexity of distributed systems the need for a formal description of these systems arises. Specification description languages like *SDL* [1][4] enable the formal specification of distributed systems with the means of finite state automatas which allow together with testing languages like *TTCN* [3] to verify the system on an abstract level

J. Wu et al. (eds.), *Formal Methods for Protocol Engineering and Distributed Systems*

without implementing it. For simplification of the development process the need for an automatic translation from the specification language to a programming language like C or C++ arises. Especially an object-oriented language like C++ with its features for component software like *polymorphism* and *templates* [8] forms an ideal platform for specification-based implementations of distributed systems. For the usability of an automatic translation there are mainly two criterias which have to be fulfiled. The translated system should provide high performance which mostly depends on the implementation of the finite state machine and on the number of copy operations which may occur when signaling between different processes due to missing pointers in languages like SDL. For code maintenance and perhaps manual improvements a good readability of the translated code should also be provided.

This paper introduces a framework for the automatic translation of an SDL specification to C++ program code providing high performance together with good readability of the resulting code. For that we present an object model to map the SDL instances to appropriate C++ objects. An efficient implementation of the finite state machine with constant costs only is outlined using polymorphism. Additionally we present a mechanism to reduce copy operations called lazy evaluation and a memory pool approach to avoid costly memory allocation and deallocation for fixed-sized objects like SDL signals.

The remainder of the paper is organized as follows. Chapter 2 gives an overview of related work in the area of automatic SDL to C/C++ translation. In the third chapter the framework is presented by outlining the object and activity model of our approach as well as mechanisms to reduce copy operations and memory allocations in the translated system. In chapter 4 a performance evaluation of the mechanisms is presented before concluding the paper.

2. RELATED WORK

In this chapter two existing techniques for the mapping of SDL entities to a programming environment, the server model and the activity thread model, are introduced. Their mode of operation and main features are presented, taking into account the problems of high performance and good readability.

Another aspect related to the automatic translation from SDL to C++ is the issue of memory management. The efficiency of the generated code depends on the number of copy operations that must be performed. Two mechanisms will be introduced: the *integrated packet framing* (IPF) and a linked list of buffers (*mbufs*).

The following discussion will prove that none of the existing approaches sufficiently fulfil the requirements of an efficient automatic SDL to C++ translation with respect to the translation model or the memory management.

2.1 Server Model

In the *server model* [5] active entities of the SDL specification, like a process, a timer or a channel, are implemented by threads or processes of a programming language or operating system. For this reason the operating system usually takes over the scheduling of active threads, for example in a round robin fashion. SDL signals are implemented as passive objects, not as threads. The code is generated with the help of a runtime library, where each SDL entity is mapped to a corresponding class. A communication channel is implemented as a thread with a FIFO queue, able to send and receive signals to and from other processes. If a process receives a signal, a virtual function transition() is called. Based on the incoming event and nested if-statements or a switch-case construct, a corresponding code segment is executed. A possible output is copied to the input queue of another process or of a communication channel. The *finite state machine* (FSM) of an SDL process is therefore realized by these conditional statements of the transition() function. The readability of the generated code is supported by the straight forward translation of SDL entities to object classes, which allows also to adopt the naming of the SDL specification. The readability of the FSM implementation within the process depends on the chosen technique. Mostly if-else or switch-case constructs are used to choose the appropriate transition which results in bad readability of the translated code (see chapter 4.1).

The performance of the server model is limited due to the direct coupling between active SDL entities and corresponding threads. A large number of active SDL objects, like processes, timers and channels, leads to a large task switching overhead and poor performance. The communication relationships between the active components may be analyzed at system start-up and are made available to all classes involved in the communication process. This information is accessed via a hash table which allows an efficient exchange of signals. Nevertheless for each output a signal instance is created by the new operator, which has a negative effect on the performance. Because the structure of the communication relations may be analyzed beforehand, it is possible to extend the server model in the way that a group of SDL processes that communicate with each other is mapped to a single thread. This optimization improves the performance of the server model to a certain degree, but still does not provide convincing results.

2.2 Activity Thread Model

In the *activity thread model* [2] SDL processes are implemented as procedures with their own private data space. The data part contains the variables and state information of the process instance. The FSM of the SDL process is realized through these procedures with signals being the active part. An incoming signal triggers the corresponding procedure, which handles the event. A possible output activates the next procedure, resulting in a chain of procedure calls, the so called activity thread. This thread is not related to a thread executed in an operating system, like in the server model described above. However the synchronous communication by procedure calls violates the semantics of SDL. In order to overcome this problem, the output signal should be buffered until the transaction is completed. For this reason two approaches have been proposed: the *basic activity model* (BAT) and the *enhanced activity model* (EAT).

The BAT implements a ring buffer, where the signals are stored. An incoming signal is removed from the head of the list and a possible output signal is enqueued at the tail. As a consequence this output signal is processed only after all preceding signals are consumed. The BAT is able to handle only a single external signal, if the ring buffer is empty and contains no more waiting signals. The processing of all other external signals is deferred, until the handling of the current external signal is completed. The EAT allows the processing of several external signals in parallel by creating an operating system thread for each of the external signals.

Due to the use of the *call/return* mechanism, the performance of the BAT model is considerably better than the one of the server model. The implementation of the call/return function is much more efficient than the task switching used in the server model. The EAT falls in between the server and the BAT model. The task switching overhead, associated with the threads of the EAT model, depends on the number of external signals and thwarts the advantages of the BAT model.

The readability of the generated code is poor, since all tasks and states of an SDL process are translated to separate procedures, which are mutually invoked. To the knowledge of the authors no automatic SDL translation tool based on the activity thread model is available up to now.

2.3 Integrated Packet Framing (IPF)

A common task inside a protocol layer n is the extension of a *service data unit* (SDU) received from the upper layer n+1 by the *protocol control information* (PCI), forming a *protocol data unit* (PDU). This PDU of layer n will then become the SDU of the lower layer n-1, and so forth. Usually copy

operations are performed in each layer appending the PCI to the SDU. The main idea of the *IPF approach* [9] is to allocate from the beginning a memory area large enough to accommodate all PCIs of all layers. This avoids the copying of SDUs to PDUs and allows all protocol layers to access the same memory area for changes of the data. A disadvantage of IPF is the requirement that no data manipulations must be performed on the data after an output operation, since this would violate the SDL semantics. Another drawback is the fact that IPF can not be used, if the protocol has to perform SDU fragmentation, due to a maximum message size. In this case the SDU has to be divided into several parts, where the PCI is appended to each of the parts, corrupting the idea of a continuous, linear built up of the final PDU. As many protocols provide a maximum transfer unit, fragmentation is very common in protocol design, making IPF an improper choice.

2.4 mbufs

In the implementation of the TCP/IP protocol another approach for efficient memory operations was used [10]. With this method a data instance is represented by a linked list of memory buffers (*mbufs*) of fixed size, called the mbuf chain. Each list element contains a header and a payload field. The header consists of a reference to its predecessor and successor, a length and type field as well as a pointer to the data. The data can be stored either in the payload section of the mbufs element or, if a larger area is needed, in a cluster allocated separately. The memory operations performed by the protocol, like adding a pre- or suffix or cutting out data, are realized by simple pointer manipulations inside the linked list, without any copying of data. Only fixed sized mbuf elements are handled, allowing the use of a memory pool for the efficient generation and deallocation of mbuf structures. Another advantage of the mbuf technique is the efficient transport of mbuf chains. If a data object, like a PDU, has to transferred to another process, a pointer reference to the beginning of the mbuf chain is sufficient. However, problems arise, when the semantics of SDL must be implemented with mbufs. Like with IPF, the access of data after an output operation will lead to an incorrect manipulation, since the reference to a single mbuf chain is shared by several processes. In order to comply to the SDL semantics, the sending process must provide a copy of the mbuf chain as an output. This copy operation is more costly for the linked list of an mbuf chain, than for a continuous memory area. Another drawback of the mbufs technique is the fact that the programmer has to provide functions for freeing the mbuf chain, which introduces another source for errors. Furthermore the automatic code generation process must be able to identify the proper locations for the

deallocation of mbuf chains inside the protocol specification, which is not a trivial task.

3. FRAMEWORK FOR SDL TO C++ TRANSLATION

This chapter introduces the framework for an automatic translation of an SDL specification to C++. For that, we present the object model of the framework followed by the activity model describing the realization of the finite state machine. Additionally we describe a mechanism to minimize the copy operations within the system to improve the performance of the translated code and a memory pool approach to improve memory allocation of fixed size elements like SDL signals.

3.1 Object Model

The object model of our framework is adapted from the hierarchy of an SDL system. The SDL components system, block, process, state, signalroute and channel are mapped to appropriate C++ objects. A generic *Block* is the central entity of the object model which is inherited by the *System* object for the representation of an SDL system.

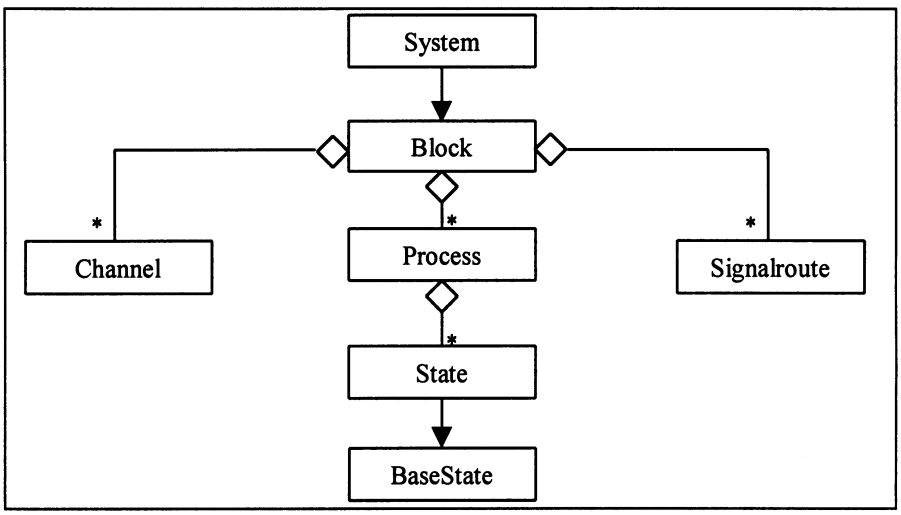

Figure 1: Framework Object Model

There may be different System objects in the code as well as several Block objects. Within each block there are several *Process* objects consisting

of several states represented by *State* objects. The states extend the *BaseState* object for an efficient implementation of the finite state machine (see next section). *Channel* and *Signalroute* objects complete the object model which is shown in figure 1 given in UML notation [7]. In the remainder of this paper we will concentrate on the realization of the finite state machine involving the Block, Process, State, and BaseState object. Due to space restricitions the Channel and Signalroute objects are not within the scope of this paper.

3.2 Activity Model

The activity model of the automatic translation is very important for the performance of the translated code. The presentation of the activity model is divided in two parts. In the first one the mapping of the SDL onto the operation system entities is shown before presenting the realization of the finite state machine in each block.

3.2.1 Mapping onto the Operation System

As presented in the related work there are several approaches to map SDL components like SDL system, SDL block and SDL process to the underlying operation system.

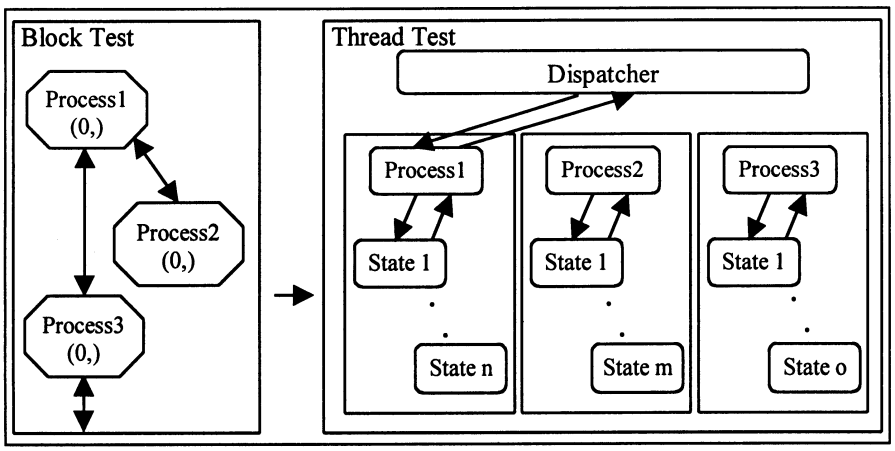

Figure 2: Framework Activity Model

In our framework the SDL system is created as a process while the SDL blocks are mapped to threads in the sense that each SDL block has its own thread. Within the thread the dispatcher of that SDL block is implemented using the call/return model. The dispatcher determines the addressed SDL process from the actual incoming signal and calls the appropriate state

transition method within that SDL process. With this model the number of threads is equal to the number of SDL blocks in the system, so it is constant which avoids costly thread creation like in the Activity Thread model [2].

3.2.2 Implementation of the Finite State Machine

Crucial for the performance of the translated system is the implementation of the finite state machine. In our framework we use the call/return method, which means that the appropriate state transition function is called and the dispatcher waits for completion of the function (return). Therefore there is a mapping between states, incoming signals, and appropriate functions to handle the state transitions. Common approaches use state functions in which all possible incoming signals are handled either by switch-case or if-constructs. Both approaches are poor in performance (linear dependence on the number of signals) as well as poor in readibility of the translated code.

In our approach we define a *BaseState* class for each process with a default transition for all possible signals which may be delivered to the process. Default transition means to consume the signal without any further action. Each incoming signal has its own virtual function with the default transition. For each possible state in the system a subclass is defined overloading the appropriate methods of class *BaseState* according to the possible incoming signals in this state. To demonstrate that, we assume a process with three possible incoming signals *e1*, *e2* and *e3* and three possible states *s1*, *s2* and *s3*. Signal *e2* is possible in all states while *e1* and *e3* may be incoming in state *s3* only. So the class definition looks like

```
class BaseState {
    virtual e1() {};
    virtual e2() {};
    virtual e3() {};
};
class s1 : public BaseState {
    virtual e2() {actions};
};
class s2 : public BaseState {
    virtual e2() {actions};
};
class s3 : public BaseState {
    virtual e1() {actions};
    virtual e2() {actions};
    virtual e3() {actions};
};
```

There are two advantages of that class structure. The first one is the readability of the code, because each state has its own class definition

consisting of methods for all incoming signals within this state. The second advantage is the efficiency of dispatching which is explained next.

During initialization of the dispatcher, for each process in the block, an array of method offsets is built using the offsets of the *BaseState* class of that process. These offsets are used in the dispatcher to call the appropriate method of the appropriate subclass. This is possible because the subclass consists of overloaded methods of the *BaseState* class only, so the offsets are the same. The offset array is indexed by the signal which is stored as an enumeration type. The actual state is stored as the pointer *ActualState* of type *BaseState* to the appropriate class and the method is called by jumping to the appropriate offset in the virtual table function of the actual state. If there is an illegal signal in a state, the dispatcher calls the default method of the *BaseState* class and consumes the signal (additionally some debug or exception code may be integrated in the default transition).

For our simple example the variable *OffsetArray* is filled during the process initialization with the offsets of the *BaseState* object by the instructions

```
OffsetArray[e1] = &BaseState::e1();
OffsetArray[e2] = &BaseState::e2();
OffsetArray[e3] = &BaseState::e3();
```

During runtime of the system, incoming signals are dispatched by using these offsets to call the method of the appropriate state. For instance, if there is an incoming signal *e2* and the current state is *s3*, the method is called by the instruction

```
(ActualState->*(OffsetArray[e2]))(Parameters)
```

with ActualState containing the pointer to the instance of s3 and signal a enumeration type of all signals. The dispatching is shown in figure 3. It can be seen that method calls of not implemented (illegal) signals fall back to the default transition of the *BaseState* class.

If there is a state transition in SDL, this is simply inplemented by changing the *ActualState* pointer after calling the transition method. This state transition approach is shown in figure 4. In the first step, the appropriate state method is determined and called. In the second step, at the end of the method, the *NextState* pointer is set to the new state. After returning from the state method, the dispatcher sets the *ActualState* to the *NextState* pointer and the state transition is completed.

To access process-wide variables we propose a *template-based* [6] implementation of the state objects. The parameter of the template is a pointer to the parent process to which the signal instance belongs. Using this pointer, process-wide variables like mapped SDL variables as well as pointers to the other state instances (needed for state transition) can be accessed within each state method.

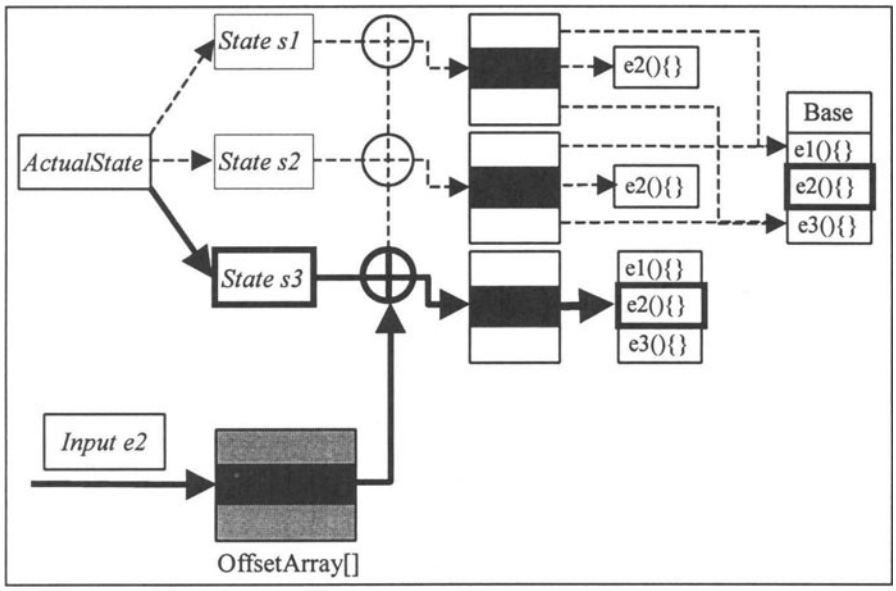

Figure 3: Dispatching Input Signal e2 in State s3

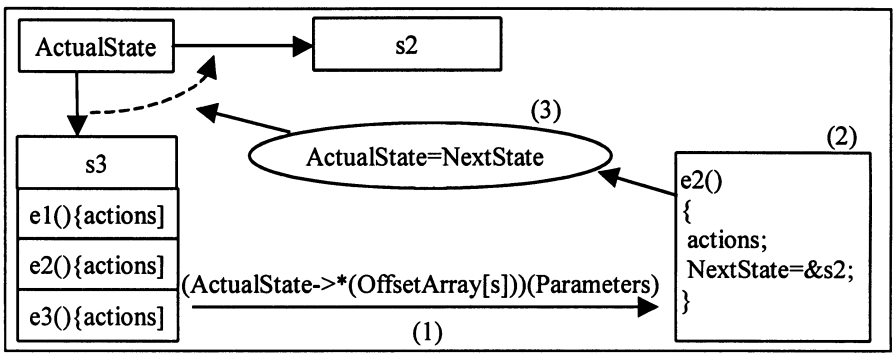

Figure 4: State Transition in FSM

With this dispatching method we are able to obtain constant instead of linear costs (see performance evaluation in chapter 4.1) for the dispatching which improves the overall performance of the state machine implementation. Additionally the readability of the translated code is improved by providing separate objects for each state with separate methods for each incoming signal for this state.

3.3 Lazy Evaluation

Beside the activity model and the finite state machine implementation of the translation, the functions for data manipulation are one of the main performance factors of the automatic translation from SDL to C++. Data manipulation in protocol specifications may occur either when sending SDL signals from one SDL process to another or by accessing the memory for operations like checksum calculation, adding prefixes/suffixes or cutting specific fields from the data. Especially the processing of protocol data units causes a high overhead during runtime when accessing and copying the memory [2].

For the optimization of those operations we define a specific abstract SDL data type, called *TransferBuffer*, which extends the mbufs principle [10] to any operation which may occur to buffers together with the *SmartPointer* approach [6] for reference counting of memory. This approach allows an optimization of copy actions within an SDL translation together with a high readability of the translated code.

The basic principle of the approach is very similar to the mbuf approach. Instead of determining the new buffer content after each operation (copy, cut or forwarding), a linked list of operations is built. The linked list is evaluated when building an output buffer (sending the buffer to the environment for instance) by calling the appropriate operations stored in the linked list. When signaling a TransferBuffer to another SDL process the pointer to the linked list is forwarded instead of copying the entire structure. Thus, operations are delayed until they are needed. This principle is also used in functional and imperative programming languages [6]. It is also possible to optimize the copy operations by deleting operations in the linked list when they are not necessary. When there are several instances of the memory (logical copies of instances) during runtime, each instance gets its own branch of operations which is unique for building the buffer at output of the data. Combined with the reference counting of the data, copying the memory for the operations (the data buffer) is avoided which is the main difference compared to the mbuf approach, where each logical copy of an instance gets its own memory copy.

As we said before, a new abstract data type is defined for this type of optimization. The abstract data type provides standard operations like assignments as well as special buffer operations like AddPrefix, AddSuffix or Cut. The translation is done by defining a C++ object called *TransferBuffer*. The object-oriented approach allows us to implement the operations of the abstract SDL data type directly as object methods either by overloading operations (like assignment) or defining appropriate methods like `AddPrefix()`. Thus the readability of the translated code is improved

which can be seen in the simple example in table 1. Two layers (implemented as SDL processes) are invoking certain operations on the data buffers which they get as an input signal. In the n-th layer some protocol control information (PCI) is added as a prefix to the service data unit (SDU) and the resulting protocol data unit (PDU) is signaled to the next ((n-1)-th) layer. In this layer, a specific area of the input SDU is cutted and a suffix is added before signaling this PDU to the next layer. Beside the FSM code (see chapter 3.2.2) the operations are directly translated in the appropriate C++ code which is very similar to the SDL code. The last `Out()` method of the specific signal instance (`Instance_c` in the example) to the environment is responsible for building the output buffer by evaluating the operation tree of the signal buffer and executing the operations.

	SDL	C++

Layer(n)	dcl pdu TransferBuffer;
	dcl pci TransferBuffer;
	...
	input a(sdu);
	...
	task pdu:=AddPrefix(sdu, pci);
	...
	output b(pdu);
		TransferBuffer pdu;
		TransferBuffer pci;
		...
		void State_X::Input_a(TransferBuffer sdu){
		...
		pdu=AddPrefix(sdu, pci);
		...
		Instance_b->Out(pdu);
		}

Layer(n-1)	dcl pdu TransferBuffer;
	dcl pci TransferBuffer;
	...
	input b(sdu);
	...
	task pdu:=Cut(sdu, from, size);
	task pdu:=AddSuffix(pdu,pci);
	...
	output c(pdu);
		TransferBuffer pdu;
		TransferBuffer pci;
		...
		void State_Y::Input_b(TransferBuffer sdu){
		...
		pdu=Cut(sdu, from, size);
		pdu=AddSuffix(pdu, pci);
		...
		Instance_c->Out(pdu);
		}

Table 1: Example for Lazy Evaluation

In figure 5 the resulting linked list of operations can be seen. The data of the n-th SDU is protected using the reference counting technique. The linked list consists of an operation pointer and a parameter structure for that operation. Implementing this as a union of possible parameters allows a linked list of fixed-size elements, so a *memory pool* approach (see chapter 3.4) can be used for further improvement by avoiding costly new/delete operations for memory allocation of the linked list elements.

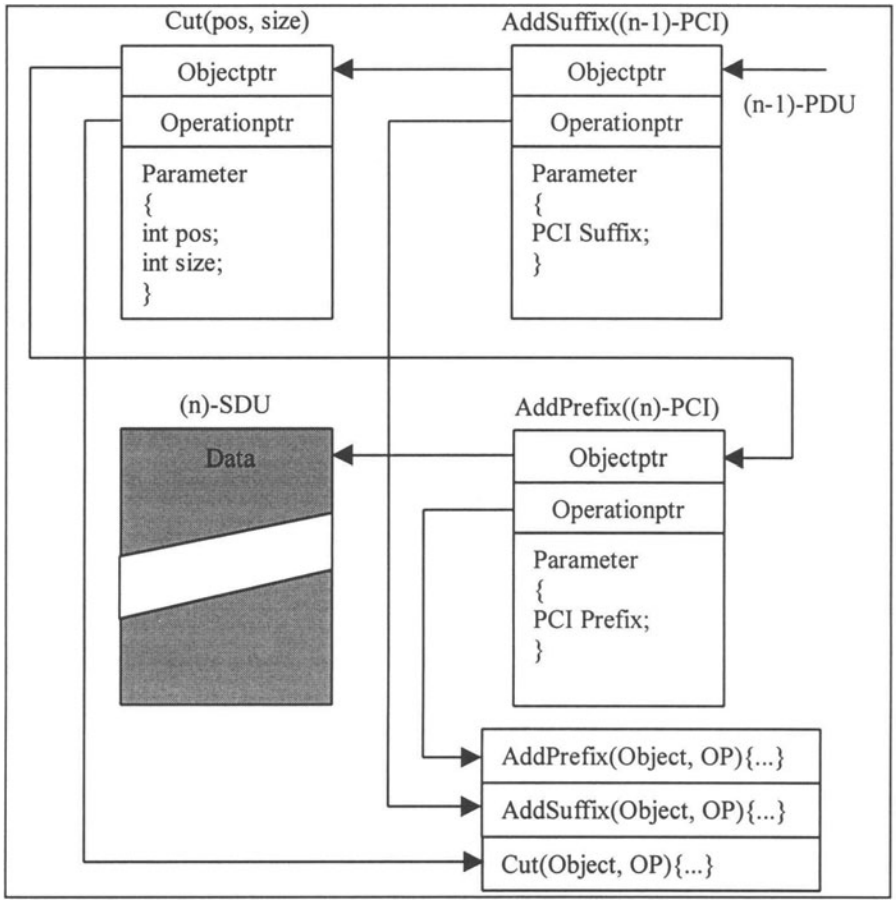

Figure 5: Example Linked List of Operations for Lazy Evaluation

The data part in figure 5 may also be implemented as a simple pointer or an mbuf structure as well instead of using the reference counting technique. But in that case, the programmer is responsible for the protection of the data when accessing several copies of it, e.g. by copying the data after each assignment.

3.4 Memory Pool

Another crucial performance issue for the translation of SDL to C++ is the memory allocation. Standard new/delete mappings lack in poor performance, because that sequence of allocation/deallocation has to be done dynamically during runtime at each creation and deletion of an SDL object like signals or data objects. Especially the creation of fixed-sized objects like

signals may be improved by using a memory pool which is also proposed in our framework.

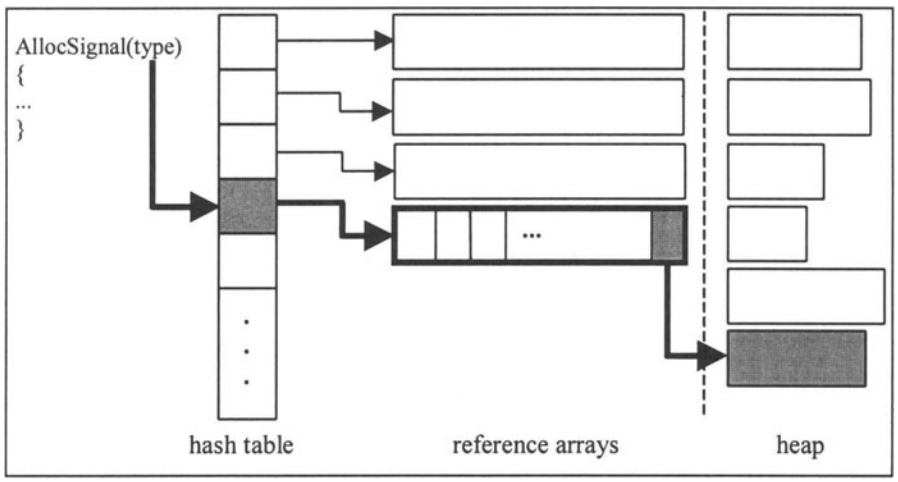

Figure 6: Memory Pool Realization

For the realization of the memory pool a hash table is built indexed by the type of the different objects we want to administrate with the pool (e.g. signals). Each entry in the hash table points to an array of references containing pointers to available heap-based memory of fixed size for the specific object. This references array is allocated according to a given default size at program startup.

When allocating memory for a specific SDL object, the index for the hash table is determined, the last entry of the appropriate reference array is returned, and the entry in the reference array is deleted. If the reference array is empty, a new object is allocated from the heap and returned. Alternatively, a certain amount of new objects may be created and added to the reference array to reduce heap allocation time.

When deallocating memory, the entry is inserted at the end of the appropriate reference array after determining the index from the hash table. The access to the reference arrays is coordinated using a simple mutex for protection.

The default size of the memory pool is very important for the performance. On the one hand, large default pools avoid the allocation of memory for resizing the pool using the heap during runtime. On the other hand, small default pools reduce the total amount of memory of the runtime environment at startup. So, there is a trade-off between performance and workset size when adjusting the default pool. Strategies like the expected or maximum pool size may be used to set the pool size at the startup of the program.

4. PERFORMANCE EVALUATION

The main focus of our framework for the automatic translation of SDL to C++ is the performance of the translated code. Nevertheless the readability of the translated code is another big issue for maintenance and manual adjustment of the resulting code. The latter issue is fulfilled by the object-oriented design approach we used for the translation. In the following we concentrate on the performance evaluation of our framework compared to related work analyzing the main parts of our approach, namely realization of the finite state machine, the lazy evaluation, and the memory pool. All results were obtained by measurements on a Pentium PC running with Windows NT4.0. In the diagrams the average values are shown.

4.1 Finite State Machine

Crucial for the performance of a finite state machine realization is the implementation of a state transition. While common approaches often use a simple if-else or switch-case model our framework uses a lookup in a virtual table indexed by the incoming signal. The goal of the performance evaluation is to look at the dependence of the state transition performance on the number of possible signals in a state. For that we compare our framework with the if-else and switch-case model.

The type of the signals received by the process, the number of states and the task to run in a transition does not influence the performance of the chosen dispatching method. Thus we use a very simple state machine for the evaluation consisting of one state only remaining in the same state after each transition. Additionally there are no tasks to be executed during a state transition and there are no parameters for the different signals. Thus the state machine for the evaluation is given by

```
STATE s1
    INPUT e1
        NEXTSTATE s1
...
    INPUT en
        NEXTSTATE s1
```

We generate a random sequence (of length 100000) of input signals for the process and run the same sequence for the if-else/switch-case model and our approach. Figure 7 shows the results for the comparision of if-else/switch-case and our approach with a varying number of possible signals.

As expected the values for the if-else method are increasing more than linearly with the number of possible signals. The values for our approach are constant due to the constant overhead for the table lookup of the appropriate signal method. Surprisingly the switch-case method also has constant costs

for the state transition. Those tests were performed with an implementation compiled with the Microsoft Visual C++ compiler which implements a switch-case sequence with linear numbers (the signals are enumerated as integers) as a table lookup as well, instead of executing consecutive if-else sequences.

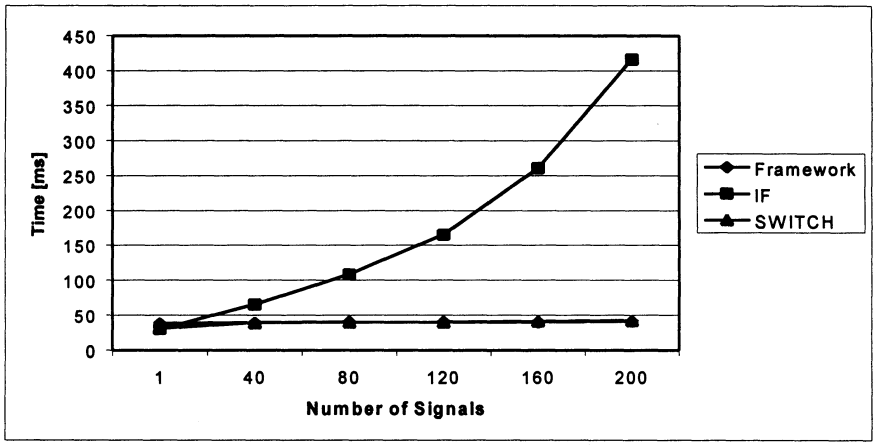

Figure 7: Performance of Finite State Machine

But this depends highly on the chosen compiler and it is not guaranteed that all compilers optimize the compiled code in that way. Using our framework leads to constant costs independent of the chosen compiler. It can also be seen from the diagram that the performance tradeoff between the if-else method and our framework is around six signals but the total performance difference (with one signal per process only) is lower than five percent. Considering the higher readability of the translated code we propose the usage of our framework independent of the number of signals.

4.2 Lazy Evaluation

The lazy evaluation approach, provided by the TransferBuffer data type together with the reference counting approach, minimizes copy operations of transfered data like SDL signals. Additionally a high readability of the translated code is provided by encapsulating the functionality in a C++ object.

In this section a comparison is presented between the lazy evaluation and the pure reference counting approach invoking operations directly instead of delaying them. The reference counting approach is used instead of a simple pointer implementation because SmartPointers (implemented with the reference counting technique) are common used in C++ for shared data [6].

Additionally the SmartPointer approach complies to the SDL semantics in contrast to the simple pointer approach. Two evaluation scenarios are performed. In the first one, the dependence on the number of processes traversed by a signal and the size of transfered data is shown while the second one illustrates a more realistic segmentation/reassembly example.

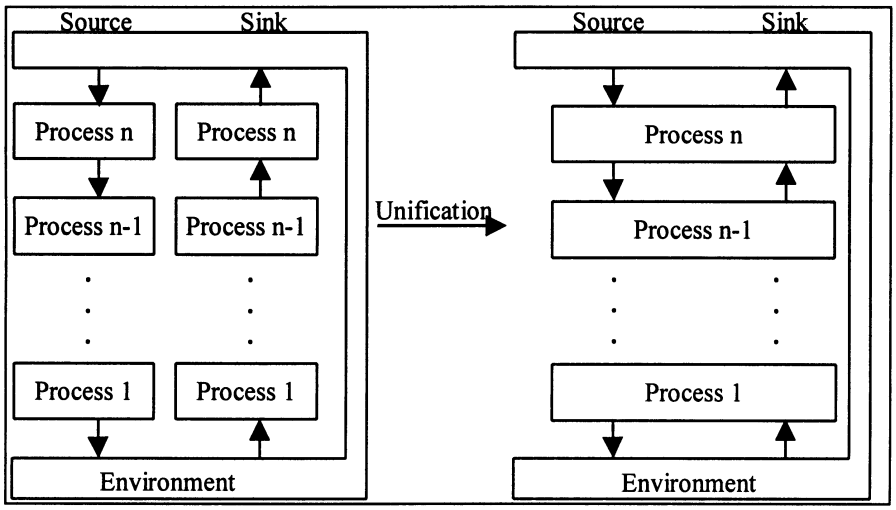

Figure 8: Environment of Scenario 1

Copy operations often occur when signaling data buffers from one SDL process to another. For the comparison of both approaches we use a layered process environment consisting of n processes which varies during the measurement. This results in building a 'vertical' chain of a certain number of processes transfering and processing signal data within the SDL specification before calculating an output buffer when sending the signal to the environment. The data is transfered from the top most process of the source to the environment that in turn sends the data to the sink which transfers the data up to the top most process again. At last the data is sent to the environment for output. For simplification, the source and sink are implemented in a unified environment as shown in figure 8. The total amount of time is measured for the transfer of 50 signals between starting the transfer for the first until the output of the last is finished at the sink. At each process of the source an `AddPrefix()` operation is invoked while the sink cuts the prefix from the data buffer using the `Cut()` operation. The amount of transfered data in each signal is set to 128 bytes, 64kBytes, and 128kBytes. The fixed-sized list elements of the lazy evaluation are allocated using the memory pool technique.

Figure 9 shows the results of the comparison. The term LE stands for lazy evaluation and denotes our approach. It can be seen that with increasing

buffer size and number of layers the total amount of time for the transfer can be dramatically reduced using the lazy evaluation technique. In the 128 byte case, the maximum performance gain is about 50 percent, while the transfer of 128kByte data is improved by a factor of more than 20 when using our approach. There is a small dependence on the number of processes only when using the lazy evaluation method. That can be explained with the avoidance of copy and memory allocation operations within the processes.

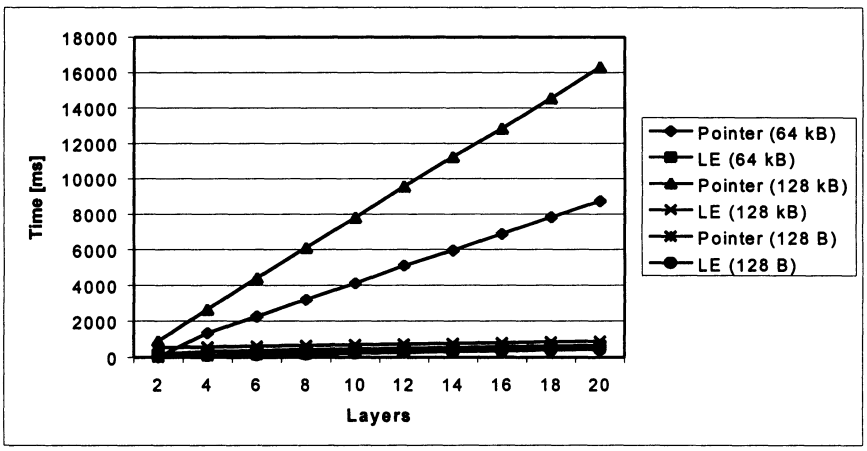

Figure 9: Performance of Scenario 1 for Lazy Evaluation

In the second evaluation scenario, a more realistic *fragmentation-reassembly* example is performed. There are four processes in the system performing operations like `AddPrefix()`, `Cut()` and `Union()` on the signal data (see figure 10). The data size is set to 64kByte and 128kByte while the segment size varies from 128 to 4096 bytes (the segments are built in the Split process and rebuilt in the Union process). The total amount of time for 30 data transfers is measured.

In figure 11 the comparison of both techniques can be seen. The performance gain is not as high as in the first scenario, because the length of the chain is much smaller (only five). The maximum improvement is about 80 percent in the 128kByte case and 45 percent for the smaller data units. With higher segment sizes the performance gain decreases due to the decreasing number of operations to be performed.

It can be seen from both scenarios that the lazy evaluation technique leads to a significant performance improvement depending on the chain length of processes until the data is sent to the environment and the size of data transfered with the signal.

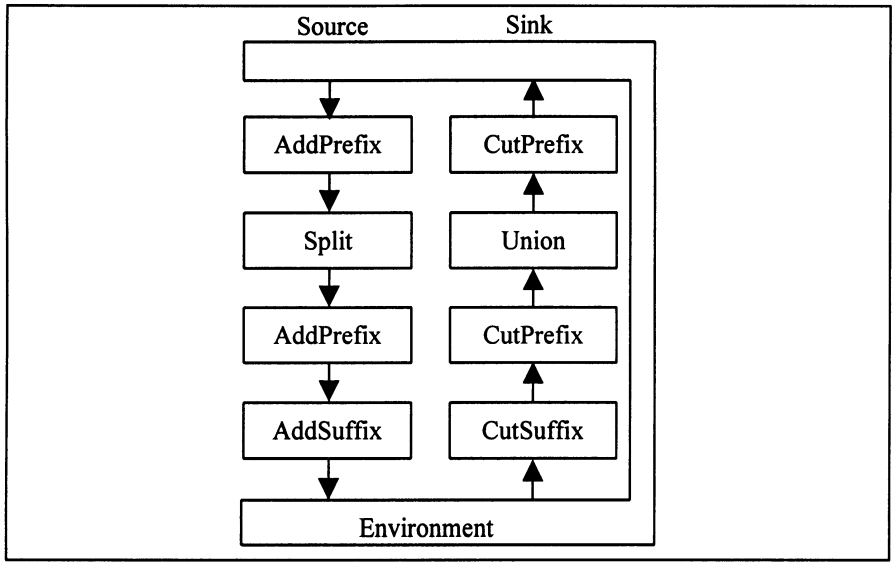

Figure 10: Fragmentation and Reassembly Example

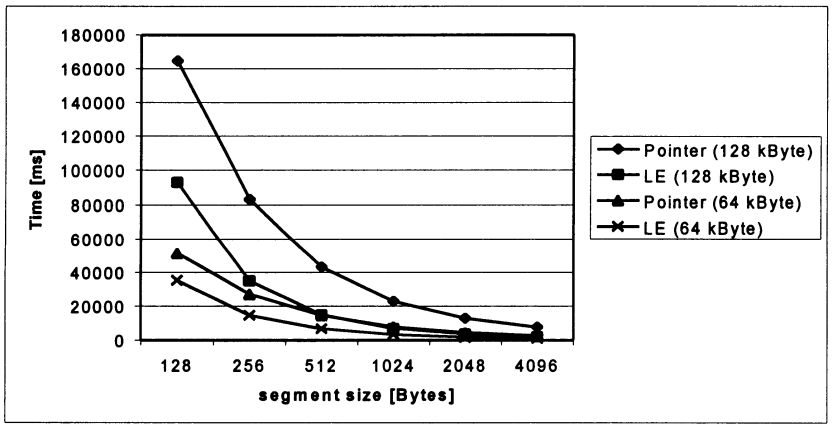

Figure 11: Performance of Scenario 2 for Lazy Evaluation

4.3 Memory Pool

Our framework proposes a memory pool for the optimization of memory allocation and deallocation for fixed-sized objects like SDL signals. Common approaches use the memory management functions of the operating or runtime system for allocation and deallocation. The performance evaluation of the memory pool procedure compares these approaches by generating a random sequence of object allocation and

deallocation requests of length m. The randomly generated number of objects n_i to allocate in request i is less than the memory pool size and the total number of objects N is defined as the sum over all n_i.

The objects are allocated and directly deallocated using the appropriate method (new/delete or pool approach). The time between the start of the first object allocation and the end of the last deallocation is measured. The size of the allocated objects in terms of user data is crucial for the overall performance. A user data size of zero bytes means that there is no specific data stored in the object except of a fixed size for administration of the object. In our measurements a user data part of zero and 104 bytes is used.

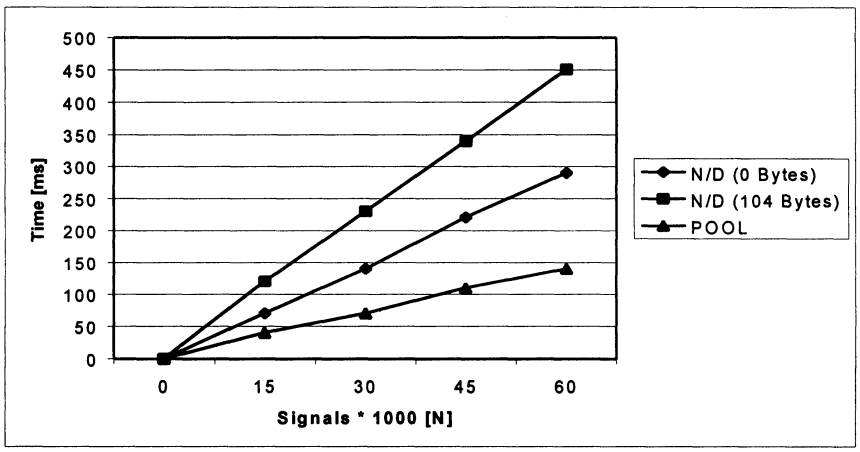

Figure 12: Performance of Memory Pool

Figure 12 shows the overall time used by the performance evaluation for the memory pool approach and the new/delete operation. It can be seen that for the zero byte case a performance gain of about 100 percent is obtained which is increased to 200 percent when allocating 104 bytes. The memory pool performance is independent of the size of the allocated object, because the hash table lookup does not depend on the size of the object.

5. CONCLUSION

Many distributed systems are specified using specification languages like SDL. For simplification of the development process the need for an automatic translation from the specification language to a programming language like C++ arises.

In this paper we presented a framework for the automatic translation of an SDL specification in C++ program code. For that, the main components

were outlined and described which are crucial for the performance and readability of the translated system. For the activity model of the framework we proposed a mapping of SDL blocks to operation system threads dispatching the incoming signals in each SDL process of the block. The dispatcher in each block was realized by using a hierarchy of subclasses derived from a default base class accessing the appropriate state transition method by a simple hash table. We also presented a mechanism to minimize buffer operations by delaying the evaluation of the operations until the buffer is sent to the output of the system. The framework was completed by a pool approach for fixed-sized objects to avoid costly memory allocation and deallocation operations. Our performance evaluation of the framework showed that the proposed mechanisms perform very well compared to other approaches.

It can be summarized that our framework for automatic SDL to C++ translation provides mechanisms with high performance and good readability of the translated code using the features of an object-oriented language like polymorphism.

In our future work we will concentrate on the comparison to other object-oriented languages like Java. Additionally we are going to implement the framework for the automatic translation from SDL to C++.

6. REFERENCES

[1] F. Belina, D. Hogrefe, A. Sarma: SDL with Applications from Specification, Prentice Hall, 1991

[2] R. Henke, P. Langendörfer, A. Mitschele-Thiel: Effiziente Implementierung formal spezifizierter Protokollarchitekturen (in german), Otto-von-Guericke-Universität, Fakultät Informatik, Technical Report IRB005/96, 1996

[3] ITU: Recommendation X.292 - OSI conformance testing methodology and framework for protocol recommendations for CCITT applications - The Tree and Tabular Combined Notation (TTCN), 1992

[4] ITU: Recommendation Z.100 - Specification and Description Language (SDL), March 1993

[5] N. Kikuchi,Y. Shigeta, K. Miyake, W. Tanaka, M. Nabeta: An Integrated System Development Method and Support System based on SDL and C++, SDL'89: The Language at Work, Elsevier Science Publishers, p.135-143, 1989

[6] S. Meyers: More Effective C++, 35 New Ways to Improve your Programs and Design, Addison-Wesley, 1997

[7] T. Quatrani: Visual Modeling with Rational Rose and UML, Object Technology Series, Addison-Wesley, 1997

[8] B. Straustrup: The C++ Programming Language, Addision-Wesley, 1998

[9] L. Svoboda: Implementing OSI Systems, IEEE Journal on Selected Areas in Communications 7(1989), p.1115-1130, 1989

[10] G. R. Wright, W. R. Stevens: TCP/IP Illustrated Volume2, The Implementation, Addison-Wesley, 1995