Generating readable programs from SDL

Nikolai Mansurov, Alexei Ragozin

Institute for System Programming, Russian Academy of Sciences
25 B.Kommunisticheskaya, Moscow, 109004, Russia
tel: (7095) 912-5317 fax: (7095) 912-1524
email: {nick,ar}@ispras.ru

Abstract

Several techniques for generating readable programs from SDL specifications are presented. These techniques can be used to produce programs in a standard object-oriented programming language which preserve the appearance of the source SDL specification. The main technique is to postpone most semantic transformations until the run-time of the generated system. We show how the so-called start-up time of the generated program can be used to restructure it from a readable form (which is used for inspections) into efficient run-time structures (which are used to execute it within the corresponding run-time support system). We present some details of the new approach and compare it with a few other code generators. We show that the application of the new techniques does not imply any loss of efficiency of the generated programs as compared to more conventional approaches.

Keywords

SDL, code generation, readable programs, semantic transformations

1. INTRODUCTION

In this paper we discuss the possibilities of generating readable programs from SDL. “Readability” is of course a subjective notion. In order to position ourselves onto a more solid foundation we decided to define **“readability”** *as preserving the appearance of the original SDL specification in the generated program*.

According to this definition the most readable document is the SDL specification itself. We treat both the original SDL specification and the generated program strictly as texts. Under SDL specification we would always understand “SDL/PR”. There are two reasons for restricting ourselves to text. First, we only consider code generation into traditional imperative languages, which do not have a standard graphical representation (such as SDL/GR). Second, the questions of readability when applied to graphical representations are much more complex, and our attempt to define “readability” as structural equivalence will be less advantageous. According to our definition of readability, generated program which on average uses e.g. 3 lines of code per SDL line, will be more readable than the program which uses e.g. 4 lines of target code per SDL line. Another aspect of our definition of readability is the fragmentation of the generated program. For example, a generated program with 10 consecutive lines of target code for some SDL construct will be more readable than a program with total of only 8 lines for the same SDL construct which are generated as two consecutive fragments. In the above example the loss of readability is due to the distortion of the structure of the original SDL specification.

For designers familiar only with SDL and not with the target language our definition of readability will probably be as good as any subjective one. Designer can easily inspect such generated program simply because he can use SDL knowledge to understand it.

This paper is dealing with the following research question: Is it possible to generate a program in some standard imperative (procedural or object-oriented) language while preserving almost one-to-one (even optical) correspondence with the original SDL specification? In other words, what are the limits of molding the appearance of generated programs and what techniques are available for that purpose? These questions are part of a more complex research problem: What are the properties of semantic-preserving mappings between two formal languages (source and target language) and how these properties can be used to design a transformation system (e.g. a code generator) to produce target language programs with given qualities (e.g. readable, modular, efficient, etc.). We believe that it is important to investigate this area because automatic code generation is one of the attractions of using formal description techniques, such as SDL [1]. As we mentioned in our earlier paper [3], several code generators from SDL are available [6], however their wide acceptance in software industry is yet to come [4]. Lack of the readability of the generated programs might be one of the reasons, which still hinders the use of automatic code generators.

We revisited various mapping decisions in existing code generators from SDL to imperative programming languages. The main result of our research is that it is possible to significantly increase the readability of the generated programs. However the new approach requires new code generation techniques. The main technique is to postpone most semantic transformations until the run-time of the generated system. We suggest to use the so-called start-up time of the generated system to restructure it from a form that is used for inspections into efficient internal run-time structures which are used during the execution of the system within the corresponding run-time support system.

Restructuring “inspectable” representations of the generated program into “executable” representations is the key of the new approach. On the other hand, significant language support is required in order to preserve most of the appearance of the original SDL specification in the generated program. We used C++ as the target language for our readable mapping. Important technique is to use macrodefinitions in order to “camouflage” certain details of the generated program and thus mold its appearance.

We have developed prototype code generator from SDL to C++ in which suggested techniques were used. Development was based on our previous SDL translator [2]. The prototype implements mapping decisions for most of SDL-92 constructs. The readable mapping presented in this paper heavily uses most of C++ features: classes and objects are used to represent SDL scope units; nested class definitions are used to represent nesting of SDL scope units; most SDL operations are represented as C++ methods; SDL inheritance and virtual redefinition are represented as C++ inheritance and virtual methods redefinition; structural types with formal context parameters are in most cases represented as C++ classes with templates.

The rest of the paper has the following structure. In section 2 we present a sample output from our code generator. This example demonstrates that indeed the generated program can preserve an almost optical correspondence with the original SDL specification. In section 3 we discuss the most important techniques of our approach – the use of start-up time of the generated system and the use of macrodefinitions in the generated program. Section 4 demonstrates internal details of the mapping. In section 5 we describe results of comparison between several code generators. Section 6 concludes the paper.

2. SAMPLE GENERATED PROGRAM

Figure 1 demonstrates an example of our readable mapping from SDL to C++.

SDL	C++
PROCESS Ping;	PROCESS_DECL(PING);
TIMER Tmr;	TIMER_DECL0(TMR);
DCL	DCL(COUNT,int);
Count INTEGER;	STATE_LIST(
	STATE_DECL(WAIT));
	ENDPROCESS_DECL(PING);
START;	PROCESS_DEF(PING)
TASK Count := 0;	TIMER_DEF(TMR);
Lab :	START;
OUTPUT Ping;	TASK COUNT=0;
SET(NOW + 10, Tmr);	LAB:
NEXTSTATE Wait;	OUTPUT(PING());
	SET(NOW+10,TMR());
	NEXTSTATE(WAIT);
	ENDSTART;
STATE Wait;	STATE(WAIT);
INPUT Pong;	INPUT(PONG,);
RESET(Tmr);	RESET(TMR());
TASK Count := Count + 1;	TASK COUNT = COUNT +1;

DECISION Count = 100000;	DECISION(COUNT==100000);
(true) :	CASE(IS_EQUAL(COUNT==100000,
OUTPUT Finish;	TRUE));
STOP;	OUTPUT(FINISH());
else :	STOP;
JOIN Lab;	ELSE
ENDDECISION;	JOIN(LAB);
	ENDDECISION;
INPUT Tmr;	INPUT(TMR,);
STOP;	STOP;
ENDSTATE;	ENDSTATE;
ENDPROCESS Ping;	ENDPROCESS_DEF(PING);

Figure 1. SDL process representation in C++

3. TECHNIQUES FOR GENERATING READABLE PROGRAMS

In this section we suggest techniques which can be used to control the appearance of the generated programs and can thus be used to generate quite readable code.

Let's introduce some terminology for discussing *appearance* of generated programs and source specifications. We assume an abstract syntax view of a program as being built from *constructs*. Pure abstract syntax view of a program considers only semantics of each construct. In our approach some syntactical considerations are also required. Generated program consists of two groups of constructs: *domain constructs* and *glue constructs*. **Domain constructs** correspond to the definitions and "operations" (i.e. any usage of a definition) in the original SDL specification. **Glue constructs** are additional operations (and sometime even definitions) which coordinate the work of domain constructs both semantically and syntactically. Glue constructs are mostly used to bridge the syntax and semantic gap between the source and target languages. Both domain constructs and glue constructs affect the appearance of the generated program. We believe that it is the glue constructs that can make generated programs difficult to inspect.

Informally, the strategy of generating readable code should aim at representing source constructs with as few domain constructs as possible while also keeping the amount of glue constructs to minimum. Usually, there exist several alternatives for mapping each source construct into domain constructs, but not all of the resulting combinations result in correct programs. The amount of domain constructs per each source construct is roughly determined by the semantic gap between the source and target languages.

The usual techniques for bridging this gap is to use run-time support system which encapsulates some (or even most) of the domain constructs [5]. Run-time support system is especially useful for mapping data of the source language. However the use of the run-time support libraries alone does not lead to a one-to-one correspondence between the source program and the target program because of the differences in control structures of the two languages.

We believe that limiting our design space to available *direct* mappings as shown above is too restrictive for our purposes. Instead we suggest to use *indirect* mappings. In an indirect mapping domain constructs are used as instructions to build semantically equivalent representation of a source construct rather than as such representation itself. Indirect mappings imply that the execution of the generated system starts with an additional phase at which run-time structures are built and initialized. We suggest to use the name *start-up time of the generated system* for this phase. Essentially, the start-up phase of the generated system is used to restructure it from one *form* (instructions to build run-time structures) into another *form* (the run-time structures themselves). Indirect mappings dramatically increase design space of available mappings and allow considerable flexibility in the appearance of the generated program because they are not any more restricted to preserving semantic equivalence using domain constructs.

SDL mappings are affected by another fundamental problem. Extensions of the language in SDL have no explicit semantics. Instead, the standard [1] suggests transformations of SDL extensions to basic SDL. Most of SDL translators perform specified transformations before generating code. This means that the source constructs which are the input to the code generator are not the same as the source constructs in the original SDL specification, and the structural equivalence is lost even before the code generator is started. Transformations affect readability since they cause considerable duplication (as well as fragmentation) of code.

We suggest that *any* readable mapping should **postpone** most of SDL transformations to a later stage and use the original SDL specification as its input. We suggest that these transformations should be done at the start-up time of the generated program.
To summarize, start-up time is used for the following purposes.

- Some actions that are usually performed in SDL analyzer and code generator can be done at start-up time. For example, all numbering of generated entities (e.g. *signals*, *blocks*, *processes*) can be done dynamically so that program text looks less like object code and is more appropriate for human inspection.
- The use of start-up initialization can considerably decrease the size of the generated program. Size of the generated program is proportional to a number of generated classes and class methods. Start-up time allows to perform instantiation of object of universal classes defined in run-time support system instead of generating specific classes for each SDL object. C++ methods with varying number and types of parameters are used for this purpose.
- More efficient run-time structures of the executing system can be build at start-up time than during code generation. The delay of creating efficient run-time structure:
  - keeps generated program readable (since effective things look more like object code);
  - allows to form extremely effective execution-oriented system representation at start-up (e.g. routing tables, graph tables).
- Run-time configuration of executing system is also established at start-up time. The same generated program potentially can work in different configurations. E.g. mapping of SDL blocks and processes to operating system tasks does not impact on the representation of generated program. The concrete physical configuration of the system is expressed by specific makefile.
- Some standard SDL transformations can be effectively performed at start-up time instead of doing that in SDL analyzer (e.g. multiplication of *asterisk states*, *inputs*, *saves*, transformation of *continuous signals*). Such late transformations allow
  - to preserve SDL system structure in generated program;
  - to minimize duplication of code in generated program;
  - to implement effective support algorithms for most SDL shortcuts.

4. DETAILS OF THE MAPPING

This section describes the internals of our mapping for selected SDL constructs. First part describes the mapping of SDL graph. Second part outlines mapping for SDL syntype.

4.1 Mapping for SDL graph

Descriptions of graphs constitute the major part of an SDL specification. We believe that descriptions of graphs in the generated program are the most likely candidates for human inspections. This is why readable mapping for SDL graphs is very important.

Basic SDL graph

SDL graphs for process, service and procedure are mapped to separate C++ classes inheriting appropriate base classes from support system. Generated class has the following properties.

- General functionality of SDL graph (e.g. *nextstate*, *stop*, *output*, *set*, *reset*) is encapsulated in public methods of base classes.
- Definitions inside graph (e.g. *sorts*, *signals*, *timers*) are represented as nested C++ class definitions.
- SDL variable definitions correspond to public class data members with appropriate sorts.
- Graph variables are mostly used in graph body (one of graph methods) but can be used from outside the graph class (e.g. from graph derivative, child procedure, with *view* from another graph).
- Graph body is encapsulated in a single method – constructor of class. All states and all graph transitions are joined together.

Let’s consider the structure of the graph in more details. Basically there are two approaches to representation of FSM in procedural programming language.

- Two nested switch statements (first factored by *state*, second - by *stimuli*) containing transitions on branches of the nested switch. It is easy to implement an FSM in this way but there are few disadvantages: access to transition is not always direct and effective (sometimes switch-statement is translated to sequence of nested if statements); transition can not be redefined dynamically.
- Table-driven approach is based on two-dimensional array indexed by state and stimuli. Cells of the table contain references to transition bodies. This approach is very efficient and flexible.

Table-driven mapping of SDL graph is suitable for our purposes because it supports our technique of restructuring the system at start-up time because FSM tables can be constructed dynamically.

Figure 2 demonstrates the fragment of generated program for an SDL graph before and after C++ preprocessing.

SDL	C++	Preprocessed C++

STATE st1;
SAVE sig2;
INPUT
sig1(i,c);
…
	STATE(ST1);
	SAVE(SIG2);
	INPUT(SIG1,
	PAR1(I),
	PAR2(C));
	…
		CurrentState=staST1 = TransitionTable->RegisterState("ST1");{
		InitSaveInTransitionTable(CurrentState,sigSIG2::Id) ;
		CurrentSignal=sigSIG1::Id;
		if(setjmp(TmpJumpAddr) == 0)
		InitTransitionTable(TmpJumpAddr, CurrentState, CurrentSignal);
		else {
		(SDLGraph*)this=CurrentGraph;
		sigSIG1* Stimuli=CurrentSignal;
		I = Stimuli->Par1;
		C = Stimuli->Par2;
		SDLScheduler::SetSenderAndDeleteSignal();
		…
		}

PROVIDED
p = sender
…
ENDSTATE;
	INPUT_CONTINUOUS
	(P == SENDER);
	…
	ENDSTATE;
		CurrentSignal=sigContinuous::Id;
		if(setjmp(TmpJumpAddr) == 0)
		InitTransitionTable(TmpJumpAddr, CurrentState, CurrentSignal);
		else {
		(SDLGraph*)this=CurrentGraph;
		if (P == (*(CurrentGraph->Instance()->Sender())) {
		sigContinuous* Stimuli=CurrentSignal;
		SDLScheduler::SetSenderAndDeleteSignal();
		}
		else
		longjmp(EnablingConditionFail,1);
		…
		}

Figure 2. Mapping for SDL graph

Constructor of the C++ class representing SDL graph can be logically divided into two parts.
1) Start-up code (typed in *italic* font). During the start-up time of the generated system the graph is created and its constructor is called. Start-up part of constructor is traversed and addresses of all transitions in all states are stored in the corresponding transition table. Standard C technique of non-local control passing (functions *setjmp* and *longjmp*) is used to store transition address. SDL state identifiers and signal identifiers are enumerated dynamically. Special value of transition address is used to specify *save* reaction on SDL signal. Special signal identifier is used to store reaction on continuous signal.
2) Run-time code is executed when graph transitions are performed. At the beginning of transition the following actions are performed.

- Value of graph class variable *this* is restored. It should be done because execution program stack can be crashed after non-local control passing. Values of graph variables are stored outside the stack (in heap) and don’t need to be updated. Unfortunately not all C++ compilers support assignment to variable *this.* In our prototype we decided to use the extension available in GNU C++ compiler, however portable techniques of restoring stack are available as well [7].

- If enabling condition for transition is available, it is calculated. If *false* then control is passed back to graph dispatcher.
- Graph variables are initialized by signal parameters.
- Special actions are performed: value of *sender* is reset; signal object is deleted.

Each SDL transition has a terminator (e.g. *nextstate, stop, join, return*). Terminators use non-local control passing to call system scheduler (part of the run-time support system) to start a new transition. Non-preemptive event-driven scheduling scheme is used.

Some SDL graph extensions

Our mapping directly supports most of SDL-92 extensions (shortcuts). Below we consider mapping details for some of SDL graph extensions.

Asterisk state, *inputs* and *saves* are easily integrated into table-oriented FSM implementation. Usually duplication of graph transitions occurs when *asterisk* constructs are transformed. We perform the same transformations at start-up time. We only duplicate transition addresses across the cells of graph tables. All these addresses refer to a single piece of code corresponding to *asterisk* transition. Figure 3 demonstrates internals of graph code for *asterisks*. For all *asterisks* appropriate registration methods are called at start-up time. Real duplication of transitions is performed at the last moment of graph initialization after all normal transitions were registrated.

Our approach to representation of FSM allows direct mapping of SDL graph inheritance hierarchy (which can take place for process and service types) to C++ class hierarchy. In general there are the same concepts of inheritance in SDL and C++. Derived type has the same functionality as base type but adds something new. C++ class for derived graph inherits class for base graph. Derived class inherits all definitions of base graph: variables, gates, sorts, other nested definitions,states and transitions. Derived graph inherits base transition table and extends it with own transitions and states. C++ constructor of the base graph is executed before constructors for derived graphs. As in case of asterisks there is no duplication of code: derived class contains only definitions from appropriate derived SDL type. Both languages allow to redefine some constructs from the base type in the derived class. Actually, C++ provides a more powerful mechanism for redefinition: the choice of method to be called can be done dynamically (for virtual methods). While in SDL all solutions are done statically. Virtual redefinition of *start*, *input* and *save* is easily represented as reinitializing of appropriate cell in transition table of derived graph. Old transition reference becomes useless and is rewritten with address of other transition. Here we use the flexibility of start-up system customization.

SDL	C++	Preprocessed C++

STATE *;
  SAVE *;
  …
ENDSTATE;
	ASTERISK_STATE;
	  ASTERISK_SAVE;
	  …
	ENDASTERISK_STATE;
		CurrentState=TransitionTable->RegisterAsteriskState();
		{
		  InitAsteriskSaveInTransitionTable(CurrentState);
		  …
		}

STATE *;
  INPUT *;
  …
ENDSTATE;
	ASTERISK_STATE;
	  ASTERISK_INPUT;
	  …
	ENDASTERISK_STATE;
		CurrentState=TransitionTable->RegisterAsteriskState();{
		if(setjmp(TmpJumpAddr) == 0)
		  InitAsteriskInpuInTransitionTable(TmpJumpAddr ,CurrentState);
		else {
		  (SDLGraph*)this=CurrentGraph;
		  SDLScheduler::SetSenderAndDeleteSignal();
		  …
		  }
		}

Figure 3. Mapping for asterisks

4.2 Mapping for SDL *syntypes*

Most of SDL sorts are mapped to separate C++ classes. We consider example of SDL *syntype* mapping in order to illustrate ideas of run-time system initialization. Generated C++ class for *syntype Natural* is demonstrated at Figure 4. Class *soNATURAL* has own constructor and operators but it doesn't have any data members: data is stored in base class *soINTEGER.* Constraints of *syntype* are stored in special container object *NATURALConstraints* of parameterized *class SyntypeConstraints <class Syntype, class BaseClass>*. To perform range condition check method *soNATURAL::check()* is called which in turn calls method *check* of object-container. The class for container of constraints is defined in the run-time support system. Each *syntype* has a single container object. This object is instantiated at start-up time using the universal constructor of its class.

SDL	C++	Preprocessed C++

SYNTYPE
Natural =
  Integer
  CONSTANTS
>= 0
ENDSYNTYPE;
	SYNTYPE_DECL(Natural,/*=*/Integer);
	
	SYNTYPE_DEF(Natural,/*=*/Integer)
	CONSTANT(
	  GREATER_OR_EQ(Integer,0))
	END_SYNTYPE_DEF(Natural);
		class soNATURAL: public soINTEGER {
		  SoNATURAL();
		  SoNATURAL(soINTEGER&);
		  soNATURAL& operator=(soINTEGER&);
		  void check();};
		SyntypeConstraints<soNATURAL,soINTEGER>*
		  NATURAL_Constraints;
		*SyntypeConstraints<soNATURAL,soINTEGER>**
		NATURALConstraints = new SyntypeConstraints
		<soNATURAL,soINTEGER>
		(ckGreatOrEqual,0,ckNothingMore);
		void soNATURAL::check()
		  {NATURAL_Constraints->check();}

Figure 4. Mapping for SDL syntype *Natural*

Constructor of container object for *syntype* constraints uses varying length parameter list (*SyntypeConstraints(...)*). The following agreement is used to parse parameters.

- Each range condition starts with a tag (e.g. *ckGreaterOrEqual*) which specifies kind of constraint.
- The tag is followed by one or two boundary values.
- List of constraints is terminated with special value *ckNothingMore*.

Constructor of constraints creates the internal list of constraints. This list is later used in method *check*.

The mapping of *syntypes* illustrates the idea of restructuring the system at start-up time. Instead of explicitly generated **code** that performs check of constraints we use **data** values for representing constraints that are passed to constraint constructor. The initialization of *syntypes* is performed only once at start-up time.

5. COMPARISON

In this section we introduce some metrics to analyze static characteristics of generated programs, in particular those which formalize our definition of readability. We use metrics to compare our code generator with several other generators. At the end of the section we will also consider some overall performance characteristics of the generated programs.

5.1 Readability metrics

1. Coefficient $E = V_{target}/V_{src}$ (expansion),
   where V_{target} is volume of generated program, V_{src} is volume of source fragment. Volume is measured in non-empty lines of code (LOC). The coefficient estimates compactness of the mapping. Compact target program requires less time for inspection. According to our informal definition of readability, the goal is to reduce E.

2. Coefficient $S = R_{nosrc}/R_{src}$(structure transformation),

where R_{nosrc} is a quantity of objects and relations in generated program that differ from any object or relation in source specification plus quantity of SDL objects and relations that are omitted in generated program, R_{src} is a total number of objects and relations in source fragment. S is a coefficient of transformation of source structure. According to our definition of readability, S should be close to zero.

3. Coefficient $F= 1/n\sum^{n}_{i=1} F_i$ (fragmentation)
   Where F_i is a fragmentation of given (one of n) source SDL construct. It represents the number of consequent fragments in generated program to which the given SDL construct is mapped. According to our definition of readability, coefficient F should be as close to 1 as possible. This coefficient provides some measure of structural distortion, introduced by a given mapping. The structure of the original SDL specification can be distorted in two ways: 1) some source construct is split into two or more non-consecutive domain constructs; 2) domain constructs are duplicated for some source construct. Both situations decrease readability according to our definition. However we believe that the second distortion is more serious. Value F = 1 is interpreted as structural equivalence of two systems. According to our definition of readability, structural equivalence is the key issue, however complete preservation of appearance of the original SDL specification in the generated program involves the balance of all metrics.

4. Coefficient $U = U_{target}/U_{src}$(usage of names),
   frequency of source name usage in generated program (U_{target}) in comparison to frequency of this name's usage in the original specification (U_{src}). The U metric is somewhere similar to F but its growth does not mean duplication of source code. It rather means frequency of SDL object referencing in target representation. The best result is U=1 when the source name is mentioned exactly in the same way in source and target representations.

5. Coefficient $P = P_{orig}/P_{src}$(preserving of names),
   where $Porig$ is a quantity of source names that are represented unchanged in the target code, $Psrc$ is a total number of names in source fragment. Keeping source names unchanged in the generated program simplifies code inspections. According to our definition of readability, value P=1 is the best result.

5.2 Performance metrics

To compare performance characteristics of target systems from different generators and run-time support systems we use traditional metric T_{bench}. T_{bench} is a time value necessary for executing of benchmark test. Time is measured with use of Quantify [8] profiling tool that allows to get run-time performance of the software with great precision.

5.3 Result of measurements

We use well known SDL example Ping-Pong to compare our approach to other available SDL code generators. Our C++ programs are compared with

- application C code generated by SDT code generator [9];
- C code produced by ObjectGEODE code generator [10];
- C++ code generated by RASTA translator [2].

	Total (LOC)	E overall	E for structural Components	E for communication Components	E for SDL graph
SDL	67	-	-	-	-
WE	221	3.3	7.6	1.3	2
SDT	649	9.7	18.6	5.2	7.8
ObjectGeode	815	12.2	17.6	1	12.6
RASTA	441	6.6	8.3	2.3	7.1

Table 1. Expansion coefficient E for Ping-Pong

Table 1 compares volume of SDL specification for Ping-Pong with volumes of automatically generated programs. We compare coefficient *E* for several groups of SDL constructs: structural components (*system*, *block*, other infrastructure), communication components (*channels*, *signalroutes*, *signals*, *signallists*), SDL graph body (internals of *process*, *service*, *procedure*). Our mapping is especially advantageous for SDL graph body that takes the most part of specification.

Column P in Table 2 shows how original SDL specification names and original specification structure are kept in generated program. Usually code generators use prefixes and suffixes to make names globally unique. We pay special care to keeping source names in generated programs. Prefixing of SDL names is done inside macrodefinitions so that their original appearance is preserved.

The coefficient *S* illustrates how much the structure of generated program differs from source. Value of *S* is affected by two factors: 1) transformation of source SDL structure (standard SDL transformations, bringing of SDL specification into executable form) and 2) introduction of non-SDL-related services directly into generated program (e.g. trace information, mapping of system into target OS tasks).

	S	For structural constructs		For communication constructs		For SDL graph		P
		F	U	F	U	F	U	
WE	0.1	5.6	5.4	5.2	1.2	1.2	1.2	1
SDT	0.4	7.6	19.3	6	4.7	2.1	5.0	0.6
ObjectGeode	0.5	4.3	12	1.7	0.5	1.5	2.6	0.5
RASTA	0.5	7.3	7.8	2.3	0.4	1.5	1.5	0.6

Table 2. Coefficients S, F, U and P

Coefficients *F* and *U* show how single SDL construct is distributed over generated program. It is obvious that structural components are fragmented more than graph components that are usually localized in single function (method).

Generator	$T_{pingpong}$	$T_{typebasedpingpong}$
WE	0.59	1.02
SDT	0.75	1.53
RASTA	1.62	-

Table 3. Execution time on Ping-Pong test

Table 3 reflects performance speed of generated code on benchmark tests. The measurement was done on SUN ULTRASPARC-1 (147MHz) platform. Values in the second column corresponds to Ping-Pong specification with 100 000 repeats. Value in the third column corresponds to the typebased version of Ping-Pong which uses *inheritance*, *virtual inputs* and additional *continuous signals*. Readable C++ programs generated using the new techniques does not show any decrease in performance as compared to programs produced by industrial-strength commercial code generators. We attribute good performance of our generated programs to the following factors:

- start-up time allows to build efficient routing structure for sending signals;
- non-local control passing allows very efficient context switching between SDL process instances and graph dispatching;
- direct support for some SDL shortcuts (e.g. *continuous signal*) is more efficient than the corresponding transformation in the analyzer;
- customized memory allocation functions reduce overhead for object creation and deletion.

6. CONCLUSIONS

In this paper we investigated some possibilities of generating readable programs from SDL specifications. Readability was defined as preserving the appearance of the original SDL specification in the generated program. Several metrics were suggested for a more formal definition of readability. We have developed a prototype code generator which implements new techniques suitable for producing very readable generated programs. Our code generator uses C++ as the target language and is based on our previous SDL translator RASTA [2].

New techniques were investigated for the purposes of our project. The main technique is to postpone most semantic transformations until the run-time of the generated system. We show how the so-called start-up time of the generated program can be used to restructure it from a readable form (which is used for inspections) into efficient run-time structures (which are used to execute it within the corresponding run-time support system). Another important technique is to use macrodefinitions in order to "camouflage" certain details of the generated code (the so-called glue constructs).

Comparisons to several other SDL code generators according to suggested metrics show that our approach produces more readable code. It is interesting to note, that the application of the new techniques does not imply any loss of efficiency of the generated programs as compared to more conventional approaches.

6. REFERENCES

[1] Z.100 (1993), CCITT Specification and Description Language (SDL), ITU-T, June 1994.

[2] N. Mansurov: A. Kalinov; A. Ragozin; A. Chernov: "Design Issues of RASTA SDL-92 Translator", in R. Braek, A. Sarma (Eds.) SDL'95 with MSC in CASE, Proc. of the 7-th SDL Forum, Oslo, Norway, 6-29 September, 1995, Elsevier Science Publishers B. V. (North-Holland), pp. 165-174.

[3] N. Mansurov; A. Ragozin; A. Chernov; I. Mansurov: "Industrial strength code generation from SDL", in A. Cavalli, A. Sarma (Eds.) SDL'97: TIME FOR TESTING – SDL, MSC and Trends, Proc. Of the 8-th SDL Forum, Evry, France, 23-26 September, 1997, , Elsevier Science Publishers B. V. (North-Holland), pp. 415—430.

[4] A. Mitschele-Thiel; P. Langendorfer; R. Henke: "Design and optimization of High-Performance Protocols with the DO-IT Toolbox", in FDT IX: Theory, application and tools (eds. R. Gotzhein, J. Bredeke), Proc. of the FORTE/PSTV'96 symposium, Germany, Kaiserslautern, 8-11 October 1996, Chapman & Hall, pp. 45-60.

[5] R. Braek; O. Haugen:"Engeneering Real-Time Systems", Prentice Hall BCS Practitioner Series 1993.

[6] Demonstrations, in SDL'95 with MSC in CASE (eds. R. Braek, A. Sarma), Proc. of the 7-th SDLForum, Oslo, Norway, 26-29 September, 1995, Elsevier Science Publishers B. V. (North-Holland), pp. 373-388.

[7] K Ahrens; J. Fischer; D.Witaszek: "A process library for simulation in C++", available at ftp: *ftp://ftp.informatik.hu-berlin.de/pub/local/simulant/odem/*.

[8] Quantify 2.0, Pure Software Inc. 1309 South Mary Avenue Sunnyvale, CA 94087, U.S.A.

[9] "Telelogic Tau 3.2 Reference Manual": Telelogic AB, P.O. Box 4128, S-203 12 MALMO, Sweden.

[10] VERILOG: "ObjectGEODE Toolset Documentation", 1996.

ACKNOWLEDGES

We would like to thank Tord Andreasson, Jan Karlsson from Telelogic AB (Sweden) for discussing various aspects of industrial-strength code generation and for providing feedback on this research.