Embedding a Dialect of SDL in PROMELA

Heikki Tuominen

Nokia Telecommunications,
P.O. Box 372, FIN-00045 Nokia Group, Finland
heikki.j.tuominen@nokia.com

Abstract. We describe a translation from a dialect of SDL-88 to PROMELA, the input language of the SPIN model checker. The fairly straightforward translation covers data types as well as processes, procedures, and services. Together with SPIN the translation provides a simulation and verification environment for most SDL features.

1 Introduction

TNSDL, Nokia Telecommunications SDL, is a dialect of SDL-88 [6] used for developing switching system software. As one possibility to construct a simulation and verification environment for TNSDL an embedding in—or a translation to—PROMELA [1] has been investigated. The defined and mostly implemented translation provides an SDL front-end to the SPIN model checker [3] with properties similar to those of commercial SDL tools.

Using an existing tool, SPIN in this case, as the basis of an SDL verification environment might bring some compromises but certainly saves a lot of development effort when compared with implementing everything from scratch. Among the types of languages with advanced tool support, e.g. Petri nets and process algebras, PROMELA seems to be an ideal choice for SDL because of the close relationship between the languages. PROMELA being conceptually extremely close to SDL makes the translation fairly simple and allows preserving the structure of the system in the translation thus perhaps providing some advantage in the verification phase.

The close relation between SDL and PROMELA is no accident: the first version of PROMELA was designed as an extension to another language in order to support analysis of SDL descriptions [4]. Most SDL concepts have direct counterparts in PROMELA and the translation is very straightforward. The most notable examples are the processes, their dynamic creation, and communication through message passing. Some other features, like passing parameters to SDL procedures, require a slightly more complicated translation and for very few SDL aspects there seems to be no feasible representation in current PROMELA. The TNSDL pointer data type—not present in SDL-88—might serve as an example of this last category.

TNSDL differs from the SDL-88 recommendation in various details, including both simplifications and extensions. With respect to the translation to PROMELA the definition of data types and routing of signals are perhaps the two

most remarkable differences. Instead of the algebraic specifications in SDL-88 TNSDL has a C-like type system and instead of the channels and signal routes TNSDL includes a routing file which defines the destination processes for signals sent without an explicit process address. Translating the corresponding aspects of SDL-88 to PROMELA would probably be more complicated than the presented translation of TNSDL.

Essentially the same approach to SDL verification has earlier been applied at AT&T [2, 4]. The translation described in this paper attempts to cover more SDL features by making use of the recent extensions in the PROMELA language but unfortunately we cannot currently report on any practical applications of the method. There also exists an experimental verification system for TNSDL based on a translation to Petri nets [5] but for the above reasons we feel PROMELA to be a more natural target language.

The rest of the paper describes the translation from TNSDL to PROMELA organized as follows. Section 2 describes the translation of data types, the declarative part of SDL. The communication part, i.e. signal queues, and the operational part, i.e. processes and other similar entities, are covered in Sections 3 and 4, respectively. Finally, the implementation is touched on in Section 5 and the concluding Section 6 enumerates some limitations found in PROMELA and SPIN during the study. Overall, no formal definition of the translation is attempted, the description is based solely on examples and prose. Also, various technical details are ignored in the presentation, e.g. names of data types, processes, and procedures are assumed to be unique over the whole system to be translated.

2 Declarative SDL

The declarative part of SDL covers the definition of data types which are then used in variable declarations. In contrast to the algebraic specifications in the SDL-88 recommendation, TNSDL has a more implementation-oriented C-like type system with arrays, unions, pointers etc. As PROMELA has a similar, although more restricted, C-based type system the translation of data types is quite simple.

Like normal programming languages TNSDL provides a set of predefined data types and ways to construct new types on top of the existing ones. The PROMELA representation of these two faces of the type system is described in the two subsequent sections.

2.1 Predefined types

Predefined types are the basic ways to represent data. TNSDL has a set of normal integer types of various sizes and some types corresponding to fundamental SDL concepts. The representation of these types in PROMELA is defined in Table 1, the dashes indicate the currently missing counterparts of the marginal floating point number types. Constants of the predefined types are declared in

PROMELA using `define` directives as indicated in Table 2 for the `bool` and `condition` types.

TNSDL	PROMELA
bool	bool
byte	byte
word	unsigned : 16
dword	unsigned : 32
shortint	short
integer	short
longint	int
real	–
doublereal	–
character	byte
pid	chan
duration	unsigned : 32
condition	bool

Table 1. Translation of predefined types.

TNSDL	PROMELA
	`#define T true`
	`#define F false`
	`#define SUCCESS true`
	`#define FAILURE false`

Table 2. Translation of constants in predefined types.

2.2 Type expressions

Type expressions are ways to build new data types based on the existing ones. TNSDL includes seven kinds of them: type names, arrays, integer subranges, enumerations, structures, unions, and pointers. In this terminology PROMELA has only arrays and structures, even them in a restricted form, but they turn out to be sufficient for representing all TNSDL types except pointers.

PROMELA representations of some TNSDL type definitions are given in Table 3. Auxiliary types are used to overcome the limitations of PROMELA, mainly the lack of nested type expressions. Constants of the user-defined types,

enumeration constants among them, are directly replaced by their values in the translation and no `define` directives are thus needed for them.

3 Communication

Asynchronous communication through message passing is one of the characteristic features of SDL. An infinite queue for storing the received messages, or signals, is implicitly associated with each process. With some minor limitations the same mechanism is available also in PROMELA and the translation is again rather obvious.

The signal queues are not visible in SDL code but in PROMELA they have to be defined like variables: queues, or channels, have finite length and are typed. A typed channel can only carry signals whose parameters match its definition and since it is not known at translation time which signals will be sent to which processes all the channels are made "wide" enough to carry any signal present in the system. Each channel has thus appropriate fields for storing any signal with its parameters in a single slot. Actually the type of the channels is defined as a data type, a structure with proper fields for the parameters of the SDL signals. In addition, the structure contains fields for the signal name and the identifier of the sending process.

As an example, Table 4 contains the channel type for an SDL system with two signals, `sig1_s` and `sig2_s`. `mtype` is an enumeration type defining constants for all the signals in the system and `_sig_t` is the data type used to represent signals outside the channels. Unfortunately SPIN does not allow a channel with type `_sig_t` to be defined because certain data types are forbidden on channels. These include at least arrays whose base type is defined with a `typedef` and types containing the `unsigned` construct. For this reason the signals are on the channels represented using another type, `_csig_t`, which contains the same information as `_sig_t` but in a slightly different form. In `_csig_t` the critical arrays are represented as structures and the unsigned integers as signed ones. Signals are converted between these two representations using the generated PROMELA inline procedures, Table 5.

The mechanism used for routing signals which are sent without an explicit target process is simpler in TNSDL than in SDL-88. Instead of signal lists associated with signal routes and channels TNSDL contains specific syntax for defining the "default" target for a signal. The target is specified as one of the master processes of the system, which are created at system start-up and limited to have only one instance. The signal queue of such a master process is global in the PROMELA representation and it is directly used as the target in these cases.

4 Operational SDL

The behavior of an SDL system is produced by its processes, i.e. entities which receive and send signals as well as perform calculations. This section describes

TNSDL	PROMELA

`TYPE name_t`
`  REPRESENTATION byte;`
`ENDTYPE name_t;`
	`#define name_t byte`

`TYPE array_t`
`  REPRESENTATION`
`    ARRAY (2) OF struct_t;`
`ENDTYPE array_t;`
	`typedef array_t {`
	`  struct_t item[2];`
	`}`

`TYPE range_t`
`  REPRESENTATION (1:100);`
`ENDTYPE range_t;`
	`#define range_t byte`

`TYPE enum_t`
`  REPRESENTATION ENUM`
`    a,b,c`
`  ENDENUM;`
`ENDTYPE enum_t;`
	`#define enum_t byte`

`TYPE struct_t`
`  REPRESENTATION STRUCT`
`    f1 STRUCT`
`         f11 enum_t;`
`       ENDSTRUCT;`
`    f2 byte;`
`  ENDSTRUCT;`
`ENDTYPE struct_t;`
	`typedef struct_t_aux_1 {`
	`  enum_t f11;`
	`}`
	`typedef struct_t {`
	`  struct_t_aux_1 f1;`
	`  byte f2;`
	`}`

`TYPE union_t`
`  REPRESENTATION UNION`
`    f1 array_t;`
`    f2 struct_t;`
`  ENDUNION;`
`ENDTYPE union_t;`
	`typedef union_t {`
	`  array_t f1;`
	`  struct_t f2;`
	`}`

`TYPE pointer_t`
`  REPRESENTATION`
`    POINTER(union_t);`
`ENDTYPE pointer_t;`
	-

Table 3. Translation of type expressions.

TNSDL	PROMELA

```
SIGNAL
  sig1_s
  (
    par1 byte;
  );
  sig2_s
  (
    p1 array_t;
    p2 word;
  );
```

```
mtype = {sig1_s,sig2_s}
typedef _sig_t {
  mtype f1;
  chan f2; /* sender */
  byte f3;
  array_t f4;
  unsigned f5 : 32;
}
typedef _carray_t {
  struct_t item_0;
  struct_t item_1;
}
typedef _csig_t {
  mtype f1;
  chan f2;
  byte f3;
  _carray_t f4;
  int f5;
}
```

Table 4. Translation of signal declarations.

the representation of SDL processes in PROMELA. Also procedures and services, which can be considered as variations of the same theme, are discussed.

4.1 Processes

An SDL process is an extended finite state automaton which upon reading a signal from its queue carries out the associated computation and then enters the next state. The read, or received, signal is indicated in an INPUT statement which is followed by a sequence of action statements, called a transition. Each process also contains a START transition which is executed when a process instance is created. Furthermore, an instance can terminate itself using the STOP statement.

TNSDL processes are translated to PROMELA processes. The master processes correspond to `active` processes with global queues and the others to normal PROMELA processes with local queues. The translation is illustrated with an example containing both a master and a regular process in Table 6. The example also indicates the general structure of the generated processes, i.e. a sequence of variable declarations, state descriptions, and transitions. SDL transitions are sequences of statements which are translated to PROMELA statement by statement as illustrated in Table 7.

TNSDL	PROMELA

```
_csig_t _cs;
inline _receive_sig(ch,sig) {
  ch?_cs;
  sig.f1 = _cs.f1;
  sig.f2 = _cs.f2;
  sig.f3 = _cs.f3;
  sig.f4.item[0] = _cs.f4.item_0;
  sig.f4.item[1] = _cs.f4.item_1;
  sig.f5 = _cs.f5 + 32768;
}
inline _send_sig1_s(t,s,p1) {
  _cs.f1 = sig1_s;
  _cs.f2 = s;
  _cs.f3 = p1;
  t!_cs;
}
inline _send_sig2_s(t,s,p1,p2) {
  _cs.f1 = sig2_s;
  _cs.f2 = s;
  _cs.f4.item_0 = p1[0];
  _cs.f4.item_1 = p1[1];
  _cs.f5 = p2 - 32768;
  t!_cs;
}
inline _send_sig(ch,sig) {
  _cs.f1 = sig.f1;
  _cs.f2 = sig.f2;
  _cs.f3 = sig.f3;
  _cs.f4.item_0 = sig.f4.item[0];
  _cs.f4.item_1 = sig.f4.item[1];
  _cs.f5 = sig.f5 - 32768;
  ch!_cs;
}
```

Table 5. Translation of signal declarations, continued.

TNSDL	PROMELA

```
PROCESS tail;
  FPAR p1 byte;

  START;
  STOP;
ENDPROCESS tail;
```

```
proctype tail (chan PARENT;
               chan __offspring;
               byte p1) {
  chan SELF = [N] of {_csig_t}
  __offspring!SELF;
  goto _transition0;
  _transition0:
  goto end;
  end:
  skip;
}
```

```
MASTER PROCESS head;
  DCL x byte;

  START;
  CREATE tail(3);
  NEXTSTATE idle;

  STATE idle;
    INPUT sig1_s(x);
      TASK x := x + 1;
      NEXTSTATE idle;
  ENDSTATE idle;
ENDPROCESS head;
```

```
chan _head = [N] of {_csig_t}
active proctype head () {
  chan SELF= _head;
  chan SENDER, OFFPSRING;
  chan _offspring = [0] of {chan}
  byte x;

  goto _transition0;

  idle:
  do
  ::  _receive_sig(SELF,_sig) ->
      if
      :: sig1_s == _sig.f1 ->
         SENDER = _sig.f2;
         x = _sig.f3;
         goto _transition1;
      :: else ->
         skip;
      fi;
  od;

  _transition0:
  run tail(SELF,_offspring,3);
  _offspring?OFFSPRING;
  goto idle;

  _transition1:
  x = x + 1;
  goto idle;
}
```

Table 6. Translation of processes.

TNSDL	PROMELA
`OUTPUT sig1_s(100) TO SENDER;`	`_send_sig1(SENDER,SELF,100);`

`STOP;`	`goto end;`
	`...`
	`end:`
	`skip;`

`TASK a := b + 2;`	`a = b + 2;`

`CREATE client(1,2);`	`chan _offspring = [0] of {chan};`
	`run client(SELF,_offspring,1,2);`
	`_offspring?OFFSPRING;`

`DECISION x;`
`  (<3)  TASK y := y + 1;`
`  (4,5) TASK y := y + 2;`
`  ELSE  TASK y := y + 3;`
`ENDDECISION;`
	`if`
	`:: (x<3) -> y = y + 1;`
	`:: (x==4 || x==5) -> y = y + 2;`
	`:: else -> y = y + 3;`
	`fi`

`WHILE counter > 0;`
`  TASK counter := counter - 1;`
`ENDWHILE;`
	`do`
	`:: counter > 0 ->`
	`  counter = counter - 1;`
	`:: else -> break;`
	`od`

`NEXTSTATE idle;`	`goto idle;`
`JOIN spaghetti;`	`goto spaghetti;`

`SET(NOW+b,timer_a);`	`_rnd_rcv_timer_a(_timer,SELF);`
	`_rnd_rcv_timer_a(SELF,_timer);`
	`_send_timer_a(_timer,SELF);`

`RESET(timer_a);`	`_rnd_rcv_timer_a(_timer,SELF);` `_rnd_rcv_timer_a(SELF,_timer);`

Table 7. Translation of action statements.

Unexpected signals, i.e. ones without a matching INPUT statement in the current state, are normally "consumed implicitly" in SDL. This silent deletion can, however, be prevented using SAVE statements which preserve the indicated signals for later processing. SAVE statements are translated to PROMELA using additional local save queues to which the saved signals are sent. Always when the SDL state of the process changes the signal queue is prefixed with the contents of the save queue.

4.2 Timers

Timers are ways to incorporate real-time features in SDL descriptions. A process can set a timer to fire after a wanted period of time and the firing timer then sends an appropriate signal to the activating process.

As PROMELA has no real time related features the representation of timers is forced to be an approximation. Timers are in the translation represented by a process which receives all the requests corresponding to SET statements and simply sends the proper timer signal immediately back to the activating process. The translation of SET statements and their opposites, RESET statements, is sketched in Table 7; the `_rnd_rcv_timer_a` procedure is used to remove the possible earlier activation signal of the timer both from the signal queue of the timer process and the activating process.

4.3 Procedures

Procedures provide in SDL, like in normal programming languages, a mechanism for grouping certain behavior together. They are subroutines which can be called in processes, other procedures, and even in the procedures themselves. A procedure is defined very much like a process—it can contain states, inputs, transitions etc.—but is naturally less independent. For example, a procedure does not have a signal queue of its own, it relies on the queue of the calling process.

PROMELA inline procedures provide a useful mechanism for representing SDL procedures but they cannot be used alone because they are based on macro expansion and cannot represent cyclic recursive procedures. Thus, each SDL procedure is translated to a PROMELA inline procedure mostly following the rules defined for processes but in addition a simple PROMELA process is generated for each procedure invocation, i.e. a CALL statement. The additional process takes mainly care of calling the inline procedure with proper parameters.

SDL procedures can have two kinds of parameters: value and reference ones, also called IN and IN/OUT parameters. Value parameters represent the native mechanism in PROMELA and pose thus no problems in the translation but the reference ones implement a kind of remote referencing and require somewhat more effort. Roughly speaking, the values of the reference parameters are passed to the PROMELA process implementing the SDL procedure in the beginning and sent back to the calling process at the end of the execution. The same mechanism is used to make the process-level variables visible within the procedure when

required. As an example, Table 8 includes the translation of a simple TNSDL procedure.

4.4 Services

SDL services, or subautomata as they are called in TNSDL, are another way to create structure within a process. A process defined using services is internally like a set of processes but seen from the outside it behaves like a single process, i.e. it has only one signal queue and process identifier.

Each SDL service is translated to a PROMELA process and an additional main process is generated for controlling the aggregate. The main process takes care of first creating the service processes and then relaying the received signals to proper services as well as synchronizing the services. Each signal can be received only by one service and services within a process are not allowed to execute truly concurrently with each other. In addition to the normal communication mechanism services can use another one, called priority signals in SDL-88 and internal signals in TNSDL. These signals, always exchanged within an SDL process, have a higher priority than the normal ones and are in the PROMELA translation transferred through an additional internal signal queue.

Table 9 provides an example of the generated main process and Table 10 outlines the translation of a single service. In order to make the process-level variables visible to all the services they are in the beginning of the main process as well as after the transitions in the services sent to specific channels and received from them in the beginning of the transitions.

4.5 Modules

TNSDL modules, an extension to SDL-88, bring a possibility to share data among processes. Modules can contain procedures which can be called in any process of the system and variables which are visible to all procedures in the module. In the PROMELA representation the module variables are made global and the procedures are translated much like normal procedures.

5 Implementation

The described central parts of the translation have been implemented in an experimental tool called TNPRO. Actually TNPRO is an alternative back-end in a TNSDL to Petri net translator [5] which moreover uses the front-end of a TNSDL to C translator for parsing TNSDL. The functionality of TNPRO has so far been demonstrated only with small—a few hundred lines each—artificial TNSDL descriptions.

TNSDL

```
PROCESS dummy;
  DCL x byte;

  PROCEDURE proc;
  FPAR IN a byte,
       IN/OUT b byte;

    START;
    TASK b := a;
    RETURN;
  ENDPROCEDURE proc;

  START;
  CALL proc(3,x);
  STOP;
ENDPROCESS dummy;
```

PROMELA

```
inline proc (a,b) {
  goto _transition0;

  _transition0:
  b = a;
  goto _return;
}
proctype dummy () {
  chan _ret1 = [0] of {bool};
  byte x;
  chan _x = [1] of {byte}
  goto _transition0;

  _transition0:
  _x!x;
  run proc_1(_ret1,SELF,
             SENDER,3,_x);
  _ret1?true;
  _x?x;
  goto end;

  end:
  skip;
}
proctype proc_1(chan _ret2;
                chan SELF;
                chan SENDER;
                byte a;
                chan _b) {
  byte b;
  _b?b;
  proc(a,b);
  _return:
  _b!b;
  _ret2!true;
}
```

Table 8. Translation of procedures.

TNSDL

```
PROCESS auto;
  DCL x byte;

  SUBAUTOMATON auto1;
  ...
  ENDSUBAUTOMATON auto1;

  SUBAUTOMATON auto2;
  ...
  ENDSUBAUTOMATON auto2;
ENDPROCESS auto;
```

PROMELA

```
proctype auto () {
  byte x;
  chan _x = [1] of {byte};
  chan _int = [N] of {_csig_t};
  chan _sync = [1] of {byte};
  chan _auto1 = [0] of {_csig_t};
  chan _auto2 = [0] of {_csig_t};
  chan _iauto1 =[0] of {_csig_t};
  chan _iauto2 =[0] of {_csig_t};

  _sync!2;
  _x!x;
  run auto1(SELF,_auto1,_iauto1,
            _int,_sync,_x);
  run auto2(SELF,_auto2,_iauto2,
            _int,_sync,_x);
  do
  ::_sync?0 ->
    if
    ::_receive_sig(_int,_sig) ->
      if
      ::sig1_s == _sig.f1 ->
        _send_sig(_iauto1,_sig);
      :: sig2_s == _sig.f1 ->
         _send_sig(_iauto2,_sig);
      :: else -> _sync!0;
      fi;
    ::empty(_int) &&
      nempty(SELF) ->
      _receive_sig(SELF,_sig);
      if
      ::sig1_s == _sig.f1 ->
        _send_sig(_iauto1,_sig);
      :: sig2_s == _sig.f1 ->
         _send_sig(_iauto2,_sig);
      :: else -> _sync!0;
      fi;
    fi;
  od;
}
```

Table 9. Translation of services (subautomata).

TNSDL

```
SUBAUTOMATON auto1;
  START;
  NEXTSTATE idle;

  STATE idle;
    INPUT INTERNAL sig1_s(x);
      STOP;
  ENDSTATE idle;
ENDSUBAUTOMATON auto1;
```

PROMELA

```
proctype auto1 (chan SELF;
                chan _auto;
                chan _iauto;
                chan _internal;
                chan _sync;
                chan _x) {
  byte _cntr;
  _sync?_cntr;
  _cntr = _cntr - 1;
  byte x;
  _x?x;
  goto _transition0;

  idle:
  _x!x;
  _sync!_cntr;
  _cntr = 0;
  do
  :: _receive_sig(_auto,_sig) ->
     _x?x;
     if
     :: else -> _sync!_cntr;
     fi
  :: _receive_sig(_iauto,_sig) ->
     _x?x;
     if
     :: sig1_s == _sig.f1 ->
        x = _sig.f3;
        goto _transition1;
     :: else -> _sync!_cntr;
     fi
  od

  _transition0:
  goto idle;
  _transition1;
  goto end;

  end:
  skip;
}
```

Table 10. Translation of a single service (subautomaton).

6 Conclusions

A translation from TNSDL, a dialect of SDL-88, to PROMELA was outlined. PROMELA turned out to be an ideal target language for such a translation which in most places is very straightforward. PROMELA has, however, some restrictions and limitations whose elimination would further simplify the translation and make it feasible for industrial applications.

First, there are some TNSDL features which seem to have no proper counterpart in PROMELA and cannot be translated.

- PROMELA does not have a data type for real numbers. Fortunately these are very seldom used in the TNSDL world and when used could probably be approximated by integers.
- PROMELA has no pointers. Unfortunately they are widely used in switching software and in order to analyze real applications some representation for them should be found. Pointers could perhaps to some extent be simulated using channels in PROMELA: the channel variable could stand for the pointer and the contents of the channel for the pointed item.

Secondly, there are some constructs which are not allowed in PROMELA although they perhaps could be without any major redesign in SPIN.

- The type system could be more liberal, ideally the `typedef` construct could be as flexible as in C.
- The range of data types allowed on channels could be wider.
- The data type `chan` could be treated more consistently with other types. E.g. `chan` typed fields of structures cannot currently be assigned to variables in the following way.

```
typedef chan_t {
  chan f;
}
chan_t x;
chan y;
y = x.f;
```

  For this reason TNSDL process identifiers are actually represented as bytes in PROMELA. This works because bytes seem to be the C representation of `chan` types in the code generated by SPIN.
- The assignment statement could work directly also for types defined with the `typedef` construct. E.g. the following assignment is currently forbidden.

```
typedef assign_t {
  byte f;
}
assign_t x,y;
x = y;
```

– In the current form of the TNSDL to PROMELA translation remote referencing of variables would be very useful even if it would ruin some of the partial order reduction capabilities.

References

1. Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1991.
2. Gerard J. Holzmann. Practical methods for the formal validation of SDL specifications. *Computer Communications*, 15(2):129–134, March 1992.
3. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
4. Gerard J. Holzmann and Joanna Patti. Validating SDL specifications: an experiment. In Ed Brinksma, Giuseppe Scollo, and Chris A. Vissers, editors, *Protocol Specification, Testing and Verification, IX, Proceddings of the IFIP WG 6.1 Ninth International Symposium on Protocol Specification, Testing, and Verification, Enchede, The Netherlands, 6–9 June, 1989*, pages 317–326, Amsterdam, 1990. North-Holland.
5. Markus Malmqvist. Methodology of dynamical analysis of SDL programs using predicate/transition nets. Technical Report B16, Helsinki University of Technology, Digital Systems Laboratory, April 1997.
6. Roberto Saracco, J.R.W. Smith, and Rick Reed. *Telecommunications Systems Engineering using SDL.* North-Holland, Amsterdam, 1989.