Protocol Analysis using a timed version of SDL

Falko Bause, Peter Buchholz
Informatik IV, Universität Dortmund
Postfach 50 05 00, FRG-4600 Dortmund 50

Abstract
A modified version of SDL (Timed SDL; TSDL) suitable for performance evaluation and validation is presented. The modifications are done in a way, that TSDL is very close to SDL. A prototypic version of a program package is described, which takes a TSDL model as input and creates an internal representation of an equivalent Finite State Machine. Furthermore efficient analysis algorithms for partial and exhaustive examination of the state space, described by the TSDL model, are integrated into the program package, so that validation and performance evaluation of TSDL models can be done automatically.

1. Introduction

In the recent years Formal Description Techniques (FDTs) have been developed for a standardized specification of communication protocols. Examples for FDTs are Estelle [ISO87a], LOTOS [ISO87b] and SDL [CCIT88]. The main goal of FDTs is to ensure a correct specification and implementation. The use of FDTs allows the partial automation of the validation and implementation of a given specification.

Another important aspect of communication protocol development is the performance analysis of the protocol. Typical problems of protocol performance analysis are the estimation of throughputs or the optimization of parameter settings (e.g. timeout length or buffer places). A lot of work has been done in the area of protocol performance analysis, most of the techniques are based on Markovian modelling of the protocol [DICH88,KRIT84,87]. The state space of a Markov chain describing a realistic protocol can become very large, therefore approximative techniques based on partially generation of the state space are very important, although those techniques do not give exact results. However, the description of several techniques is usually rather low level starting with the Markov chain description of the protocol. Although for general performance analysis problems tools have been developed [POTI85], which allow a high level specification of a model and an automatic mapping from the high level model description to the low level analysis algorithms, tools including specialized algorithms and description languages for protocol performance analysis exist only very rarely.

The specification, validation and performance analysis of a protocol using FDTs and performance modelling tools requires two models. One model given in an FDT for the specification and validation and another model for performance analysis. Amongst the effort of writing two models there are several other problems, especially it is not clear if the two models are really equivalent and the low level description of a performance model requires a lot of knowledge about the analysis algorithms. Clearly it would be preferable to specify one model,

which can be used as a specification of the protocol and for validation and performance analysis purposes. In order to reach this goal and to be consistent with current developments in protocol specification, tools have to be developed which allow the automatic or partially automatic implementation, validation and performance analysis from a given FDT description.

In this paper we describe a prototype version of a tool for the validation and performance analysis of SDL specifications. Although we have not adressed the automation of the implementation process, we think that these aspects can be integrated or the SDL description can be used in tools supporting implementations. The SDL language has been chosen as a representative for FDTs, because SDL seems to become the most widespread FDT and allows the graphical and textual specification of a protocol. Unfortunately performance aspects are not adressed in SDL (and the most other FDTs). The aspects of time and probability have to be introduced into SDL to allow the performance analysis of SDL models. However this causes the extension of the SDL syntax which has to be done with care to be as consistent as possible with the given standard. A related approach adding performance aspects to FDTs is given in [BOVA88].

The paper is structured as follows. In section 2 we introduce a timed version of SDL to allow the mapping of SDL descriptions to Markov chains. Section 3 describes briefly the algorithms used for performance analysis and validation of protocol specifications. Main attention has been addressed to non-exhaustive techniques. The structure of a program package for analysing the timed version of SDL is given in section 4. In section 5 an example is presented.

This paper is based on the work done by the members of a students group directed by the authors. We wish to express our appreciation to all members of this group.

2. Timed SDL (TSDL)

Timed SDL (TSDL) is an extension of SDL [CCIT88], which is suitable to describe timing aspects important for performance analysis. The extension is performed in a way that the SDL version of CCITT is a real subset of TSDL. So protocol specifications in SDL are also valid TSDL descriptions. Furthermore one of our goals was to introduce only slight modifications, so that TSDL is very close to SDL. Using SDL descriptions, one can easily enrice his/her model with timing aspects for performance evaluation.

Because SDL processes are Extended Finite State Machines (EFSM) [BEHO89, HOGR89], it is natural to put more information on the transitions of such a process. Typically some states are vanishing states and the timeless transitions to other states are described by a discrete probability function (cf. section 3). The other class of states are tangible states and are left with a certain rate or after a specified time interval. Introducing probability and time is achieved by the following modification to the SDL syntax.

```
<transition> ::= {<transitionstring> [<terminator statement>]}|
                 <terminator statement>
<terminator statement> ::= [<label>] [<valuation>] <terminator>
                    <end>
<valuation>  ::=    TRATE  <expression>  <end>  |
                    TTIME  <expression>  <end>  |
                    TPROB  <expression>  <end>
```

where the modified part of SDL’s syntax is written in bold. <expression> is a valid SDL expression, which should be of type real. "TPROB <expression>" is the probability for this transition. "TTIME <expression>" and "TRATE <expression>" are specifying the duration of the transition. These constructs can be used equivalently, because "TTIME a" is equivalent to "TRATE 1/a". The semantic difference between these syntactic constructs is based on the analysis method. The following TSDL code illustrates the use of the <valuation>-expression.

```
STATE get_from_host;
      PROVIDED TRUE;
            TRATE λ;
            NEXTSTATE send;
ENDSTATE get_from_host;

STATE get;
      INPUT message;
            TPROB prob_damaged_message;
            NEXTSTATE -;
      INPUT message;
            TPROB 1.0-prob_damaged_message;
            NEXTSTATE send_to_host;
      INPUT ack;
            TPROB prob_damaged_ack;
            NEXTSTATE -;
      INPUT ack;
            TPROB 1.0-prob_damaged_ack;
            NEXTSTATE get_from_host;
ENDSTATE get;
```

One major difference between SDL and TSDL is the use and scope of identifiers. In contrast to SDL, identifiers of TSDL can be declared at every hierarchical level, e.g. on block level or on substructure level. The scope rules are similar to high level programming languages. Variables declared at system level are global variables and can be used for the description of experiment series, e.g.

```
SYSTEM example;

   SIGNAL buffer(BOOLEAN), ack(BOOLEAN);

   DCL timeout   REAL :=  20,
       prob_err  REAL :=  0.001,
                         /* probabilty of damaged message */
       rate_hs,          /* rate host => sender */
       rate_rh   REAL := 32.0,  /* rate receiver => host */
       rate_sr   REAL :=  8.0,  /* rate sender => receiver */
       rate_ack  REAL := 64.0;  /* rate for acknowledge */

   CHANNEL chan
      FROM host          TO wireblock     WITH buffer;
      FROM wireblock     TO host          WITH ack;
```

timeout, prob_err, rate_hs, rate_rh, rate_sr, rate_ack are global variables, which can be used to specify experiment series to solve e.g. tuning problems (see the Control File in section 4 and the example in section 5).

We also introduced a few constructs for inspecting a processes input queue and the state of a process. This was done due to modelling convenience, because SDL offers no other possibility of inspecting the messages in a queue than reading them. Using such constructs the description of collisions is straight forward and easier to understand, which is demonstrated by an example in section 5. In expressions the following extensions can be used:

- QUEUELEN gives the number of elements in a processes queue
- INSTATE(<state>) is true if the specified process is in state <state>.
- INQUEUE(<signal name>) is true if the queue contains the signal <signal name>.

Example:

```
v > 5*j AND BLOCK b4/PROCESS p1 A=4
AND PROCESS p3 INSTATE(waiting)
AND ( (BLOCK b1/PROCESS p2 QUEUELEN = 0)
      OR BLOCK b1/PROCESS p1 INQUEUE(ack) )
```

Modelling some examples using these extensions showed us that TSDL is suitable for the description of timing aspects and that existing SDL descriptions can be easily transformed into the timed version by specifying the missing rates and probabilities. So TSDL is very close to the standard.

3. Algorithms for protocol analysis

Before we introduce the algorithms used for protocol analysis we give a brief overview of results to be calculated and a more general framework for the analysis which will be further investigated in the following section. The analysis consists of three main parts, the construction of the state space, the validation and the performance analysis. Of course the three parts are not performed sequentially, validation has to be done partially during state space generation and when using a special non-exhaustive technique described below also performance analysis is done during state space generation.

Every analysis starts with a fixed state of the protocol given by the states of all SDL processes. The state space generation generates step by step all states of the protocol building the successors of all already inspected states. Since the model class specified by TSDL includes timeless and timed transitions both classes of transitions can be enabled in given state. In such a case only the timeless transitions are allowed to fire to generate the set of successor states. This interpretation is consistent with the concepts used in GSPNs [MABC84] and allows the deletion of "timeless" states as described below.

The goal of validation is the detection of errors in the protocol specification. We can classify two classes of errors, hard errors that can be detected by a tool and which do not allow the performance analysis of the protocol and warnings which might announce an error in the specification but do not terminate the performance analysis. Typical hard errors are deadlocks and livelocks. In case of a deadlock all or some of the SDL processes are blocked in a fixed state. A livelock indicates a situation where the state changes without making real progress, in the sense that the system will never return to the starting state. Of course, the occurrence of deadlocks or livelocks depends on the starting state which has to be chosen manually by the user in an appropriate way.

The existence of warnings might indicate an error, but this can not be decided automatically by a tool. A tool can only specify the system state where a strange event has been detected. The user

has to inspect his specification afterwards to decide wether an error has occured or not. The following warnings can be detected:

- unrecognized_transition:
  Describing the transitions of SDL processes that have never fired during analysis.
- unrecognized_timer_overflow, unrecognized_signal:
  A signal (timer) has been deleted from a queue without treating it in a process.
- duplicated_timer_set:
  A timer is set once more before it has been ended.

The result of performance analysis is the time between two consecutive entries to states from a given set of marked states. This set of states might but need not include the initial state of the protocol. We will call this time turnaround time. Depending on the set of states used for performance analysis the turnaround time can describe the mean duration of a packet transmisson, the recovery time after a failure, etc.

For protocol analysis two different approaches can be used. The first one is the simulation of the protocol [WEST86]. Simulation has the advantage of avoiding the construction of large state spaces, but is computational expensive. In the area of protocol analysis there are several events that occure very seldom (e.g. the loss of a message, channel failures, etc.) but nevertheless these events are important for the protocol behaviour and performance. Using simulation for the analysis implies the well known problems when dealing with models including frequent and rare events. Although some advances have been made in handling rare events [WALR87], there are still a lot of open problems.

The other class of approaches, which will be used here, works on the state space of the protocol, using Markovian assumptions for the time dependent behaviour of the protocol. Since the state space of the most realistic protocols becomes very large, the use of exhaustive analysis techniques is often impossible. Nevertheless we start with the description of the **exhaustive analysis**, because it provides a base for the non exhaustive techniques.

During the generation of the state space a first step of validation can be performed. If a state without any successor is detected, then this indicates a static deadlock. After the state space has been generated as a whole, livelocks can be detected by running iteratively through the states and marking a state whenever it is connected through a single transition with a marked state. The algorithm starts with the initial state as the only marked state and terminates if all states are marked or if there are still some states not marked but none of them has a connection to a marked state. The latter situation indicates a livelock. The different warnings can also be noticed during or after state space generation.

If no error has occured during validation, performance analysis can be executed. The state space S of the overall protocol contains two different classes of states, T the set of tangible states and V the set of vanishing states. The terms tangible and vanishing have been adopted from the analysis of generalized stochastic Petri nets [MABC84], the former describes states which will be left by timed transitions, the latter states are left by instantaneous transitions. If we sort the states starting with the tangible states, followed by the vanishing states, the matrix of state transitions $\underline{U}$ looks like follows.

$$\underline{U} = \begin{pmatrix} \underline{D} & \underline{E} \\ F & G \end{pmatrix} \tag{3.1}$$

where $\underline{D}\underline{e}^T + \underline{E}\underline{e}^T = \underline{0}\, \underline{F}\underline{e}^T + \underline{G}\underline{e}^T = \underline{e}^T \qquad \underline{e} = (1.0, \dots 1.0)$

For calculating the steady state distribution $\underline{p}$, which is necessary for performance evaluation, all vanishing states have to be discarded and the generator matrix of a Markov chain has to be constructed as described in (3.2).

$$\underline{p}\underline{Q} = \underline{0} \quad \underline{p}\underline{e}^T = 1.0 \tag{3.2}$$

where $\underline{Q} = \underline{D} + \underline{E}\,(\underline{I} - \underline{G})^{-1}\,\underline{F} \quad \underline{Q}\underline{e}^T = \underline{0}$

The inverse of the matrix ($\underline{I}$-$\underline{G}$) and the stationary solution exist since the protocol includes no deadlocks or livelocks after validation [LABH87]. An overview of efficient algorithms for the solution of the stationary distribution of a Markov chain is given in [KRSM91]. From the steady state distribution, the turnaround time T can be estimated as given in (3.3).

$$T = \Big(\sum_{s \in M} (p(s)q(s,s)) \Big)^{-1} \qquad q(s,s) \in \underline{Q} \tag{3.3}$$

where M is the set of marked states.

As mentioned, the exhaustive analysis can not be used for most of the realistic protocols since the state spaces become too large. But the following **non-exhaustive techniques** are based on exhaustive analysis.

The first **non-exhaustive** technique is exclusively devoted to the **validation** of a protocol without estimating the performance of the protocol. The main idea is to restrict the number of states which are generated and explored. Of course, like other non-exhaustive techniques, the non-exhaustive validation can only detect errors and is not able to show that the specification is error free (in the sense that no errors of the type handled here are present). The idea of the non-exhaustive validation is rather simple, the parameter number_of_steps specifies the depth of the reachability tree built up during validation. The states are explored in a breadth first ordering up to the given limit of steps. Errors can be detected as described for the exhaustive case. Unfortunately the detection of livelocks has to be modified since the uncomplete reachability tree contains states without connection to the initial state. An algorithm for detection of livelocks has to find cycles in the reachability tree by running through the different subtrees.

The **non-exhaustive performance analysis** algorithm is based on [RUDI84]. Every transition is valued either with a probability or with a rate. As described previously out of a fixed TSDL state all possible transitions are valued either with probabilities or rates. Probabilities out of a fixed state are normalized to 1.0 during state generation. Since we assume Markovian behaviour of the model, probabilities can be attached to transitions with rates by dividing the rate of the transition by the sum of all rates out of the state. The algorithm is divided in the following two steps:

1. Searching for marked states
   Since the states for performance analysis are marked at the level of single processes, these states first have to be identified at the level of the overall state. Therefore the reachability tree is generated in a depth first order building all possible paths starting with the initial state. The generation of a path is terminated if the initial state or a state belonging to the class of marked states is reached or if the probability of a path calculated as the product over all

transition probabilities along this path falls below a predefined minimum $p1_{lim}$. At the end of step 1 a list of marked states has been generated. If the initial state is a marked state then step 1 is skipped, since no successors of this state have to be generated. If the list of marked states is empty then step 2 is skipped and an error message is printed, since the turnaround time can not be calculated.

2. Calculation of the turnaround time
   For every state in the list of marked states the reachability graph is generated in a depth first ordering. The generation terminates if the probability along the path falls below a predefined minimum $p2_{lim}$ or if the path reaches a marked state. If the marked state is not in the list of marked states, then the state is added to the list. Paths starting and ending in a marked state are called a scenario. Every scenario i has a probability p_i and a mean duration d_i given by the sum of the mean transition times in the scenario. After scanning all states from the list of marked states, the mean turnaround time is given by $\sum p_i d_i$. The sum $1-\sum p_i$ gives a hint on the degree of approximation, since it describes the probabilties of paths that have not been taken into account.

During the execution the model is checked for deadlocks or livelocks, if one of these errors is detected, the algorithm terminates. The warnings specified previously are also given after successful termination of the algorithm.

The last algorithm implemented is based on the combined **probabilistic validation and performance evaluation** published in [DICH88]. Since the algorithm has to be extended and modified for the use in our model class, we introduce the new version in more detail concerning the modified parts. Let us start with a brief description of the algorithm. For the use of the algorithm the state s_0 has to be tangible, since vanishing states are implicitly deleted. For the calculation of the turnaround time only tangible marked states are considered for the same reasons.

1) Initialize the set of unexplored states $S_u=\{s_0\}$ (s_0 is the initial state of the protocol) and the set of explored states $S_e= \emptyset$.
2) Find a state s_i in S_u with the highest weight w_i. (the calculation of w_i is described below)
3) Explore s_i by finding all its tangible successor states and transform the transition rates into transition probabilities.
4) Update the weights of the discovered states for every timed transition originated in s_i. Add newly discovered successor states to S_u and remove s_i from S_u and add it to the set of explored states S_e.
5) If a stopping criteria is fulfilled then stop else goto step 2.

In step 3 all vanishing states have to be deleted allowing the analysis of a model containing only tangible states. Unfortunately we can not use (3.2) since vanishing states have to be deleted during state space generation. For a state s_i all tangible successors have to be estimated. Since this step deletes all vanishing states and the starting state s_0 is tangible, also s_i is a tangible state. Exploring the transitions and successor states, the successor state can be either tangible or vanishing. In case of a tangible successor the algorithm continues as described below. Vanishing successor states have to be deleted. Let s_j be a vanishing successor state of s_i reached by a rate λ. Let S_{vj} be the set of vanishing states, including s_j as the first state, that are reachable from s_j using only timeless transitions and S_{tj} the set of tangible states reachable from s_j by timeless transitions. Let further $\underline{P}_{vj}$ be the transition probability matrix of the states from

the set S_{vj} and $\underline{P}_{tj}$ the transition probability matrix from state of S_{vj} to states of S_{vj}. Using these matrices, the rates from s_i to all states $s_k \in S_{tj}$ can be calculated as shown in (3.4).

$$\lambda_{ik} = p_{ij}\, \underline{e}_1\, (\underline{I} - \underline{P}_{vj})^{-1}\, P_{tj}\, \underline{e}_k^T \qquad (3.4)$$

where $\underline{e}_z$ = is a vector with 1.0 at position z and 0.0 elsewhere

λ_{ik} is the rate from s_i to the k-th tangible successor state

p_{ij} is the transition probability from s_i to s_j

If the inverse of the matrix $(\underline{I} - \underline{P}_{vj})$ does not exist, then this indicates a livelock. After using (3.4) for every vanishing successor state of s_i, the set of successor states includes only tangible states. Since the algorithm has been developed for models where all transitions are valued with probabilities, the rates of the transitions have to be transformed to probabilities. Such a transformation can be simply performed by dividing all transition rates by a factor α, which is an upper bound for all transition rates. α can be calculated as the sum of the maximum transition rates of the different SDL processes. Furthermore a new transition starting and ending in state s_i has to be introduced, the probability of this transition is $1.0-\lambda_i/\alpha$, where λ_i is the rate out state s_i (the sum of all transition rates λ_{xz}). The above transformation is well known in Markov chain theory and called uniformization or randomization [cf. GRMI84]. The new process describes a discrete time Markov chain and the time between the two transitions is exponential distributed with mean duration $1/\alpha$. Of course, the algorithm can also be used with the rate matrix after introducing some slight modifications.

During the runtime of the algorithm, the knowledge about the states and transitions is restricted to all transitions between explored states and all transitions from explored to unexplored states. Using a state transition diagram the situation is described for an example model in figure 1.

Of course, from partial knowledge of the state space and the transitions we can not expect to get exact results for the protocol performance. But we are able to estimate the best and the worst case behaviour. In the worst case, everytime the protocol enters an unexplored state, a deadlock or livelock occurs. Figure 1 describes this situation, since all unexplored states are deadlocks. The mean turnaround time of a protocol with a deadlock or livelock is infinite because the initial state will never be reached after entering a state belonging to the set of deadlock or livelock states. Therefore an upper bound for the turnaround time is infinite until all states have been explored. This result, however, is known without any analysis. But using the information gained so far, also a lower bound for the mean time to failure can be estimated by calculating the mean time between the start in s_0 and the first entry into the set of unexplored states.

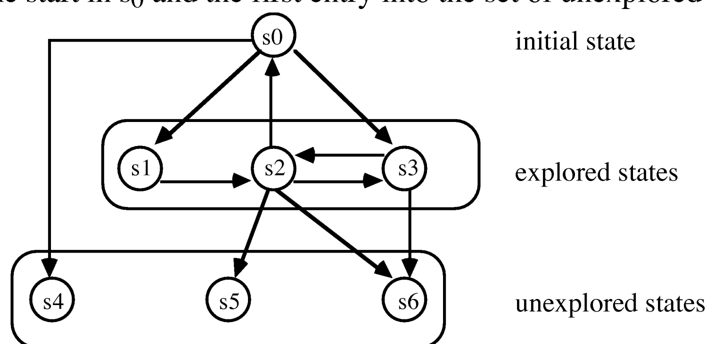

Figure 1: A state transition diagram of the known states and transitions during runtime

A lower bound for the turnaround time can be estimated by the introduction of artifical transitions with probability 1.0 from all unexplored states to s_0 (see figure 2) and the estimation of the mean time between two consecutive visits to s_0.

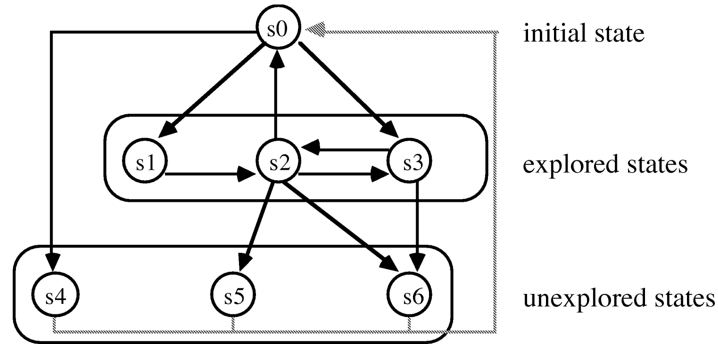

Figure 2: State transition diagram with new transitions to s_0

Let $x^m(i,j)$ be the number of visits of the process in state j before returning to s_0 the next time, required the actual state is i (i and j are discovered states) and m transitions have been explored yet. Let $\underline{X}^m$ be a matrix containing the values $x^m(i,j)$, clearly the dimension of $\underline{X}^m$ equals the number of discovered states. Let the m+1-th transition start in state s_i and end in state s_j. If s_j is not discovered yet, then the dimension of $\underline{X}^m$ is increased by one and the value $x^{m+1}(j,j)$ is initialized with 1.0, otherwise $\underline{X}^m$ and $\underline{X}^{m+1}$ have the same dimension. Supposing that $\underline{X}^m$ is known, the elements of $\underline{X}^{m+1}$ can be calculated as shown in (3.6) [cf. DICH88].

$$x^{m+1}(k,l) = x^m(k,l) + x^m(k,i) * x^m(j,l) * p_{ij} / (1.0 - p_{ij}x^m(j,i)) \qquad (3.6)$$

where p_{ij} is the probability of the newly discovered transition from s_i to s_j.

For newly discovered states only one row of the matrix has to be estimated, a transition to an explored state requires the calculation of all matrix elements. If $p_{ij}x^m(j,i)<1.0$ is not valid, this indicates a deadlock or livelock and the algorithm stops printing an error message. From the values $x^m(k,i)$ the lower bound of the turnaround time T and the mean time to failure T_f can be calculated as described in (3.7).

$$T \geq 1/\alpha \sum_{j\in d} x^m(1,j) \qquad T_f \geq 1/\alpha \sum_{j\in d} x^m(1,j) \Big/ \sum_{l\in u} x^m(1,l) \qquad (3.7)$$

where d is set of discovered states and u is the set of unexplored states.

The actual weight of an unexplored state s_j is given by $x^m(1,j)$. If, after the exploration of a state, the weights of all unexplored states fall below a given minimum, the algorithm terminates.

In conclusion we can state that the algorithm is usefull in cases where a lot of states are not discovered, because their weights fall below the boundary. Although nothing can be said about the behaviour in the set of not discovered states, a lower bound for the mean time to failure has been calculated. Choosing an error tolerance small enough, we can assume that errors if they occur are very seldom and that the calculated performance quantities reflect the normal behaviour of the protocol. On the other hand, choosing the error tolerance too small most of the protocol states are inspected in a very inefficient way, because the algorithm needs much space

and time. One way of improving the algorithm might be the aggregation of explored states with a "similar" behaviour. The last topic is a subject for further research.

4. A Tool for analysing TSDL-Models

The processes of a TSDL-model are described by EFSMs. On the other hand, the algorithms for protocol analysis, presented in the former section, are designed for Markov processes, which can be easily transformed into a FSM-representation. A natural idea for automatic validation and performance evaluation of TSDL-models is therefore to transform the TSDL-description of a system into an equivalent FSM-description. Figure 3 presents the main modules of the TSDL-Tool we have developed. The most important module is the "**TSDL-Parser and State Generator**". Given a TSDL description in textual form (like SDL/PR) and a Control File (described below), the TSDL-Parser creates an internal representation of an equivalent FSM. This module supports the analysis modules by exporting some functions especially for calculating

- the initial state and
- the successor states of a given state.

Starting with the analysis of a protocol one has to choose an appropriate initial state. Choosing the cross product of the processes start states as initial state is a bad choice. The transitions leading from the start state to its successor state often indicate an initiation phase to get the protocol running, e.g.

```
PROCESS sender;
      DCL nr,
          sended BOOLEAN;
          frame_nr INTEGER;

      TIMER t;

      START;
            TASK frame_nr := 0,
                 sended := FALSE;
            NEXTSTATE wait_for_host;

      STATE wait_for_host;
     ....
```

The start state is therefore never reached again, resulting in the detection of a livelock using the described algorithms. So a better choice is to take the cross product of the starts successor states, which is done by the exported function firststate() of the module "TSDL-Parser and State Generator". The function nextstate(Z) of this module gives a list of all successor states of state Z. If the list is empty a static deadlock is detected. Furthermore for every successor state the following information is exported:

- *timeflag* is true, if the corresponding transition is timed (see keyword TRATE and TTIME)
- *transnr* is the internal number of the transition generated by the TSDL-Parser
- *prob* specifies the probability of the corresponding transition (if timeflag is false)
- *rate* specifies the time (rate) for the corresponding transition (if timeflag is true)
- *vec* is an encoded TSDL-state.

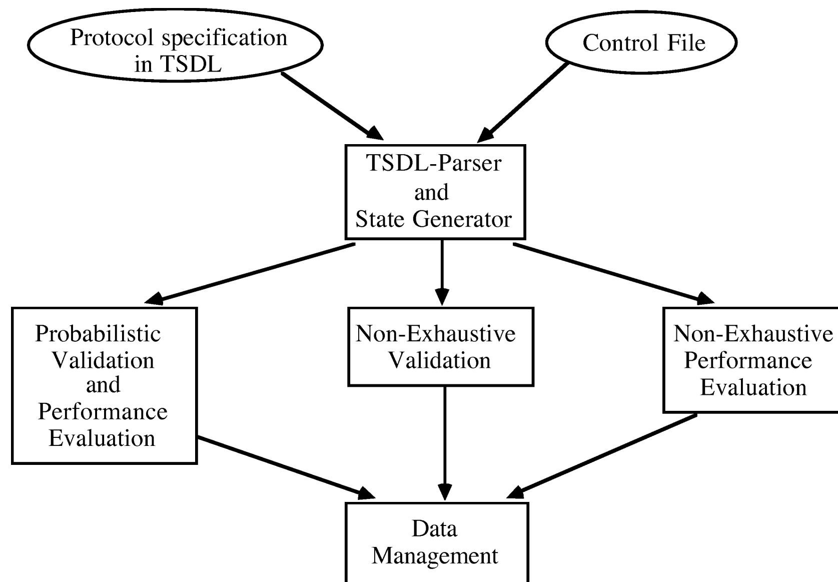

Figure 3: The tools main modules

Another set of functions deals with the translation of warnings and error messages into a TSDL-like form. The TSDL-Parser encodes all generated TSDL states into an integer vector and assigns an integer number to all transitions and exports this information to the analysis modules. In case of an error or a warning the analysis modules can inform the protocol analyst in a form according to the TSDL description using these funtions.

The analyser modules (Probabilistic Validation and Performance Evaluation, Non-Exhaustive Validation, Non-Exhaustive Performance Evaluation) are also using the functions of a Data Management Module for performing essential operations on the reachability tree, e.g.

- insertions of a generated state into the reachability tree, if it is not already in the tree
- searching, if a state is in the reachability tree build up to now.

So the efficiency of the analysis algorithms is mainly influenced by the efficiency of these operations. This was the main reason for implementing them in a different module, so that tuning is very easily possible.

As shown in Figure 3 the "TSDL-Parser and State Generator"-module takes a second input, the Control File. This file contains information about the initial values of all global variables and lists the parameters for different analysis methods. An example of a Control File is given below.

```
SDL-FILE              system.sdl;
RESULTFILE            system.res;
QUEUE-LENGTH  5;

EXPERIMENT            Probval;  /* name of the experiment */
VARLIST;
REAL                  rate_timeout 0.05;  /* parameters of */
REAL                  prob_err 0.0001;    /* the experiment */
MASK                  process  sender  instate(get_ack);
          ...
```

```
/* specification of analysis dependent parameters */
```

There are two important entries, we want to draw the readers attention to by printing them in bold. The first entry specifies an upper bound for the queue length of all modelled TSDL-processes. Todays available algorithms for analysing protocols and those described in section 3 can only deal with a finite number of states. Introducing rates and probabilities (yielding markovian models) often leads to an infinite state space. E.g. if the timer delay of a sender's protocol is specified by an exponentially distributed random variable, there is a trajectory of the Markov process creating an infinite number of messages in the receiver's input queue. Therefore an upper bound for the queue's capacity was introduced. If a message is sended to a saturated queue, it is discarded, meaning totally lost and a warning is printed. The second bold printed entry of the Control File describes the mask for specifying the class of marked states (cf. section 3). The expression for describing such a mask must be an extended SDL-expression (cf. section 2), which evaluates to a boolean value.

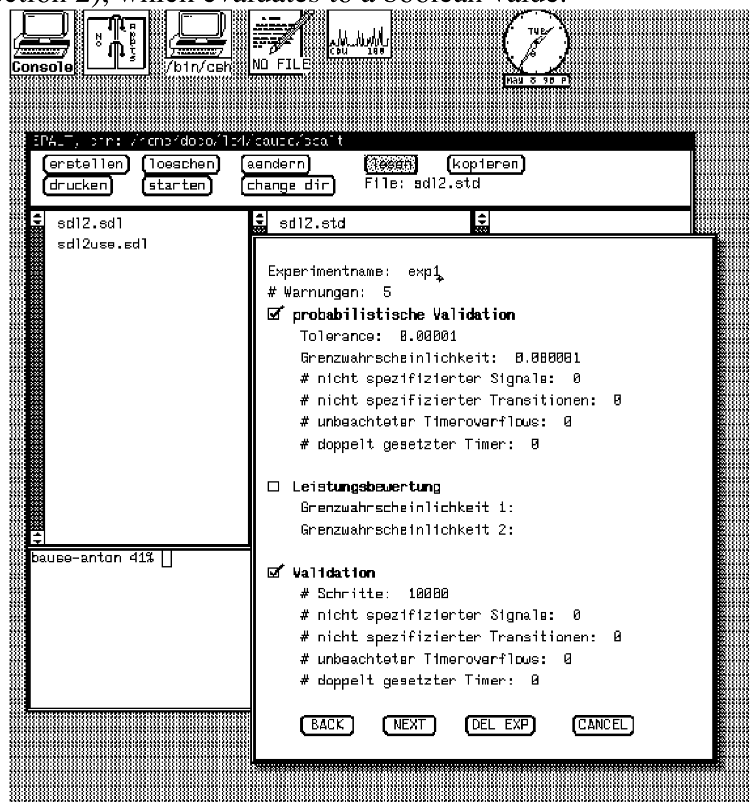

Figure 4: The graphical user interface

The main emphasis of the prototypic implementation was in testing the extensions of SDL and their suitability to performance evaluation. So only a subset of the SDL constructs is supported. The most important restrictions are

- dynamic creation of processes,
- refinement of signals,
- block substructuring and
- the SDLs data type concept (based on abstract data types).

The tool was implemented in the programming language C on a Sun3/60-system. The TSDL-Parser and State Generator uses LEX and YACC for parsing (and compiling) the TSDL-input. The tool is portable to every machine supporting C, because all inputs can be done in plain ASCII. An additional program is available under the SunTools environment, which guides the user in creating the TSDL-model and the Control File (see Figure 4).

The user can choose between solving the TSDL-model in interactive or in batch mode, which is comfortable for experiment series. Due to the tools modularity further analysis algorithms can be easily integrated, especially because of the interface's simplicity offered by the "TSDL-Parser and State Generator". This is a subject for further development.

5. Example

The TSDL-model given below is a description of a simple Positive Acknowledgement/ Retransmission (PAR) protocol [TANE88]. Data is transmitted in one direction over a noisy communication channel. For simplicity it is assumed that the sender's host has always packets to send. In TSDL this protocol can be described as follows.

```
SYSTEM PAR;

   SIGNAL message(BOOLEAN), ack(BOOLEAN);
   DCL timeout   REAL :=  20,
       prob_err  REAL :=  0.001, /* probability of damaged message */
       rate_hs,                  /* rate host1 => sender */
       rate_rh   REAL := 32.0,   /* rate receiver => host2 */
       rate_sr   REAL :=  8.0,   /* rate for as packet
                                    sender => receiver */
       rate_rs   REAL := 64.0;   /* rate for an ack
                                    receiver => sender */
   CHANNEL channel1
      FROM host_imp1 TO wireblock WITH message;
      FROM wireblock TO host_imp1 WITH ack;
   CHANNEL channel2
      FROM host_imp2 TO wireblock WITH ack;
      FROM wireblock TO host_imp2 WITH message;

   BLOCK host_imp1;
      SIGNAL request;
      SIGNALROUTE k1 FROM ENV TO sender WITH ack;
      SIGNALROUTE k2 FROM sender TO ENV WITH message;
      SIGNALROUTE k3 FROM host_1 TO sender WITH message;
      SIGNALROUTE k4 FROM sender TO host_1 WITH request;

      CONNECT channel1 AND k1, k2;

      PROCESS sender;
         DCL nr,
             frame_nr,
             request_sended BOOLEAN;
         TIMER t;

         START;
               TASK frame_nr := FALSE,
                    request_sended := FALSE;
               NEXTSTATE wait_for_host1;
```

```
      STATE wait_for_host1;
         PROVIDED NOT request_sended;  /* send request */
            TASK request_sended := TRUE;
            OUTPUT request VIA k4;
            NEXTSTATE -;

         INPUT message;   /* fetch packet */
            TASK frame_nr := NOT frame_nr,
                 request_sended := FALSE;
            NEXTSTATE timedelay;
       ENDSTATE wait_for_host1;

       STATE timedelay;
          PROVIDED TRUE;
             TRATE rate_hs;
             NEXTSTATE send;
       ENDSTATE timedelay;

      STATE send;
         PROVIDED TRUE;  /* set timer */
            SET (NOW+timeout,t);
            OUTPUT message(frame_nr) VIA k2;
            NEXTSTATE wait_ack;
      ENDSTATE send;

      STATE wait_ack;
         INPUT ack(nr);  /* ack received*/
            RESET(t);
            DECISION nr = frame_nr; /* correct ack? */
               (TRUE): NEXTSTATE wait_for_host1;
                (FALSE):NEXTSTATE  send;
       ENDDECISION;
         INPUT t;         /* timer times out before
                                 ack is received */
            NEXTSTATE send;
      ENDSTATE wait_ack;
   ENDPROCESS sender;

   PROCESS host_1;
      START;
            NEXTSTATE send_message;

      STATE send_message;
         INPUT request;
            OUTPUT message VIA k3;
            NEXTSTATE -;
      ENDSTATE send_message;
   ENDPROCESS host_1;
ENDBLOCK host_imp1;

BLOCK host_imp2;
   SIGNALROUTE k5 FROM ENV TO receiver WITH message;
   SIGNALROUTE k6 FROM receiver TO ENV WITH ack;
   SIGNALROUTE k7 FROM receiver TO host_2 WITH message;

   CONNECT channel2 AND k5, k6;

   PROCESS receiver;
```

```
      DCL nr,
          expected BOOLEAN;
      START;
            TASK expected := TRUE;
            NEXTSTATE get_message;
      STATE get_message;
         INPUT message(nr);
            OUTPUT ack(nr) VIA k6;
            DECISION nr = expected; /* correct packet? */
               (TRUE): OUTPUT message VIA k7;
                       TASK expected := NOT expected;
                       NEXTSTATE timedelay;
               (FALSE):NEXTSTATE -;
               ENDDECISION;
      ENDSTATE get_message;
      STATE timedelay;
         PROVIDED TRUE;
            TRATE rate_rh;
            NEXTSTATE get_message;
      ENDSTATE timedelay;
   ENDPROCESS receiver;
   PROCESS host_2;
      START;
            NEXTSTATE get_message;
      STATE get_message;
         INPUT message;
            NEXTSTATE -;
      ENDSTATE get_message;
   ENDPROCESS host_2;
ENDBLOCK host_imp2;
BLOCK wireblock;
   SIGNALROUTE k8 FROM ENV TO wire WITH message;
   SIGNALROUTE k9 FROM wire TO ENV WITH message;
   SIGNALROUTE k10 FROM ENV TO wire WITH ack;
   SIGNALROUTE k11 FROM wire TO ENV WITH ack;
   CONNECT channel1 AND k8, k11;
   CONNECT channel2 AND k9, k10;
   PROCESS wire;
      DCL nr BOOLEAN;
      START;
            NEXTSTATE get;
      STATE get;
         INPUT message(nr);
             TPROB prob_err;
             NEXTSTATE -;       /* damaged message */
         INPUT message(nr);
             TPROB 1.0-prob_err;
             NEXTSTATE modmessagetime;
```

```
            INPUT ack(nr);
                TPROB prob_err;
                NEXTSTATE -;       /* damaged message */
            INPUT ack(nr);
                TPROB 1.0-prob_err;
                NEXTSTATE modacktime;
         ENDSTATE get;

         STATE modmessagetime;
            PROVIDED TRUE;
               TRATE rate_sr;
               NEXTSTATE messagetimeover;
         ENDSTATE modmessagetime;

         STATE modacktime;
            PROVIDED TRUE;
               TRATE rate_rs;
               NEXTSTATE acktimeover;
         ENDSTATE modacktime;

         STATE messagetimeover;
            PROVIDED queuelen = 0;
               OUTPUT message(nr) VIA k9;
               NEXTSTATE get;
            PROVIDED queuelen > 0;
               NEXTSTATE collision;
         ENDSTATE messagetimeover;

         STATE acktimeover;
            PROVIDED queuelen = 0;
               OUTPUT ack(nr) VIA k11;
               NEXTSTATE get;
            PROVIDED queuelen > 0;
               NEXTSTATE collision;
         ENDSTATE acktimeover;

         STATE collision;
            PROVIDED queuelen = 0;
                NEXTSTATE get;
            INPUT *;
                NEXTSTATE -;
         ENDSTATE collision;
      ENDPROCESS wire;
   ENDBLOCK wireblock;
ENDSYSTEM PAR;
```

timer: (sec)	1.0	2.0	2.5	2.75	**3.0**	**3.25**	3.5	4.0	5.0	6.0
throughput: (packets/sec)	3.845	3.945	3.960	3.978	**3.979**	**3.979**	3.978	3.973	3.956	3.911

Figure 5: Results of an experiment series for determining the optimal timer rate

Using the "Probabilistic Validation and Performance Evaluation"-module no deadlocks and livelocks are detected. The warning "unspecified reception of signals" is printed in the following two situations, where a collision has occurred:

- The sender waits for an acknowledgement and the receiver has transmitted a packet to its host, sending an acknowledgement to the wire process at the same time. If the wire process is in state "modmessagetime", there is no transition specified for reading the acknowledgement. Analysing the description of the wire process shows the reader that this situation isn't a model error, because the wire process can proceed to the state "collision" (via state "messagetimeover"), where the input signal is read.
- The sender's timer has timed out too fast and two messages collide on the wire. If the wire process is in state "modmessagetime" and a message is in its input queue, a similiar situation is present.

The validation phase also detects a transition that was never used (see the bold printed line in the TSDL description). The transition described in the decision statement of state "wait_ack" in the sender process indicating an acknowledgement with an incorrect frame number is never enabled. The reason for this is based on the wire processes behaviour in loosing a defect message, but never modifying it. This shows the protocol analyst that the protocol description above is a simplification of a noisy communication channel and that the change of some bits and the detection by checksum algorithms is not considered in this model. On the other hand the reader might expect a similiar behaviour on the receiver's side, i.e. that the transition in the false-branch of the decision statement in state "get_message" of process receiver also never fires. It is easy to see that this can't hold, e.g. consider the following situation, where the receiver has send an acknowledgement, which has collided with a message (because the timer timed out too early). The next transmitted message will have a frame number, which is not expected by the receiver.

For performance evaluation an interesting result is the number of packets that are transmitted per time unit. This throughput can be calculated using the probability of the steady state distribution for being in the state described by the mask "block host_imp1/process sender instate(timedelay)". The throughput is automatically calculated by the tool as
"P[block host_imp1/process sender instate(timedelay)] * the total rate out of this state".

Given the parameters shown in the TSDL description, one might be interested in the timeout delay getting the maximum throughput. This optimal timeout delay can be found by starting a series of experiments giving the following results, where the region of the optimal timeout duration is written in bold.

6. Conclusion

We have presented an integrated approach for the validation and performance evaluation of communication protocols specified in an extended version of SDL. The use of SDL specifications for performance analysis purposes seems to be a natural way of including perfomance analysis in the design process of a protocol. But one has to be aware, that caused by the high level description and the complex nature of the protocol the number of states can become very large. Therefore the use of non-exhaustive technqiues are neccessary, although these techniques provide only approximations and it is not clear how good the approximations are. Another crucial point is the interpretation of the results, of course the detection of deadlock and livelocks can be provided by the tool, but warnings have to be interpreted by the user, which can be quite hard if the protocol is fairly complex. Our experience show that the use of the algorithms in common tool environment is very important, since it is the only way to allow

an efficient analysis of a protocol which requires experiment series, batch runs and an appropriate representation of the results.

References

[BEHO89] F. Belina, D. Hogrefe (September 1989) The CCITT-Specification and Description Language SDL, Computer Networks and ISDN Systems..

[BOVA88] G.V. Bochmann, J. Vaucher (1988) Adding performance aspects to specification languages; Protocol Specification, Testing and Verification VIII, S. Aggarwal, K. Sabnanj, North Holland

[CCIT88] CCITT (March 1988) Z.100: CCITT SPECIFICATION AND DESCRIPTION LANGUAGE SDL, COM X - R 15 - E, Genf.

[DICH88] D. D. Dimitrijevic, M.-S. Chen (August 1988) An Integrated Algorithm for Probabilistic Protocol Verification and Evaluation, IBM Research Report RC 13901,

[GRMI84] D. Gross, D.R. Miller (1984) Randomization Technique as a Modeling Tool and Solution Procedure for Transient Markov Chains; Operations Research, vol. 32, no. 2, March-April 1984

[HHMC90] E. Heck, D. Hogrefe, B. Müller-Clostermann, Hierarchical Performance Evaluation Based on Formally Specified Communication Protocols, to be published in IEEE Trans. on Computers, Special Issue on Protocol Engineering, 1991.

[HOGR89] D. Hogrefe (1989) ESTELLE, LOTOS and SDL, Springer-Verlag.

[ISO87a] ISO DIS9074 (1987); Estelle: A formal description technique based on an extended state transition model

[ISO87b] ISO DIS8807 (1987); Lotos: A formal description technique

[KRIT84] P. S. Kritzinger (1984) A Performance Model of the OSI Communication Architecture, IBM Research Report RZ 1346.

[KRIT87] P. S. Kritzinger (May 1987) Protocol Performance Using Image Protocols (in: H. Rudin, C. H. West, Proc. Seventh International Workshop on Protocol Specification, Testing and Verification, Zürich, pp. 321-335)

[KRSM91] U. Krieger,M. Sczittnick, B. Müller-Clostermann, (1990) Modelling and Analysis of Moden Telecommunication Networks by Markovian Techniques: Foundations, Algorithms and an Example, accepted for IEEE Trans. on Comm., special issue on Computer -aided Modeling

[LABH87] R. Lal, U.N. Bhat (1987) Reduced Systems in Markov Chains and their Applications in Queueing Systems 2, pp 147-172

[MABC84] M.A. Ajmane-Marsan, G. Balbo, G. Conte (1984) A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems; ACM trans. on Comp., vol. 2, no. 5, May 1984, pp93-122

[POTI85] D. Potier (ed.) (1985) Modelling Techniques and Tools for Performance Evaluation, North Holland 1985

[RUDI84] H. Rudin (April 1984) An Improved Algorithm for Estimating Protocol Performance, IBM Research Report, RZ 1314.

[RUDI86] H. Rudin (September 1986) Tools for Protocols Driven by Formal Specifications, IBM Research Reports, RZ 1525.

[TANE88] A. S. Tanenbaum (1988) Computer Networks, Second Ed., Prentice-Hall Inc., Englewood Cliffs, New Jersey.

[WALR87] J. Walrand (1987) Quick Simulation of Queueing Networks: An Introduction, 2nd International Workshop on Applied Mathematics and Performance Reliability of Comp./Comm. Systems, University of Rome 1987

[WEST86] C. H. West (1986) Protocol Validation by Random State Exploration, IBM Research Reports, RZ 1482, Zürich (also in: Proceedings of the 6th Int. Workshop on Protocol Specification, Testing and Verification, B. Sarikaya, G. V. Bochmann).