HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **17**; April 1997
ISSN 0783–540X
ISBN 951–22–4008–4

DYNAMICAL ANALYSIS OF SDL PROGRAMS WITH PREDICATE/TRANSITION NETS

TERO JYRINKI

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
P.O.Box 1100, FIN–02015 HUT, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports — ISSN 0783–540X
No. **17**; April 1997 — ISBN 951–22–4008–4

Dynamical analysis of SDL programs with Predicate/Transition nets

Tero Jyrinki

Abstract: In this report an analyzer based on formal methods for concurrent programs is presented. Since the seventies the Specification and Description Language SDL has been used for developing distributed and concurrent systems. These systems, which include e.g. telephone exchanges, have become extremely complicated during the last decades. Due to the complexity resulting from the concurrent transactions in systems, the development of these systems is expensive and design errors are hard to find.

TNSDL is a dialect of SDL created at Nokia Telecommunications. In this report a TNSDL analyzer created in the EMMA project at the Helsinki University of Technology is presented. The analyzer first translates a TNSDL description to a Predicate/Transition net which is then analyzed with the Predicate/Transition net reachability analysis tool, PROD. In the end the analysis results are translated back to TNSDL.

The EMMA analyzer supports developing and maintenance of the concurrent programs. With it various properties of the system can be proved, errors can be searched and detected and different execution paths can be traced.

Keywords: SDL, TNSDL translator, Predicate/Transition nets, reachability analysis

Printing: Libella Painopalvelut Oy; Otaniemi 1998

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
P.O.Box 1100, FIN–02015 HUT, FINLAND

Phone: $\frac{90}{+358\text{–}0}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–0–451 3369
E-mail: lab@saturn.hut.fi

Contents

1 Introduction **1**

2 Predicate/Transition Nets **3**

2.1 The Basic Form of Predicate/Transition Nets 3

2.2 Analysis of Predicate/Transition Nets 9

2.3 PROD Reachability Analysis Tool 10

2.3.1 Using PROD 10

2.3.2 The Query Program Probe 12

3 SDL **15**

3.1 The Basic Concepts of the SDL Language 15

3.2 The Structure of the SDL Language 16

3.3 Validation of SDL Systems 17

3.4 Differences Between TNSDL and SDL88 18

4 Implementation of EMMA **20**

4.1 From TNSDL to the Predicate/Transition Net Model 20

4.1.1 Handling the Queue 22

4.1.2 Processes and Procedures 25

4.1.3 Variables and Expressions 31

4.1.4 Conditional Branching and Loops 33

4.1.5 Reducing the Concurrency 35

4.2 The Structure of the Compiler 36

4.2.1 User Definable Options 37

5 Implementation Problems **39**

5.1 General Problems 39

5.2 Problems with Variables and Expressions 40

5.3 Problems with Queues 41

5.4 Problems with the Execution Resource Handling 41

5.5 Problems with the Time Concept 42

5.6 Problems with the Programming Environment 42

6 Using EMMA **43**

6.1 The Efficiency of EMMA 43

6.2 A Small Example 44

7 The Future of EMMA **46**

8 Conclusions **47**

A An Example of Using EMMA **50**

A.1 TNSDL program 50

A.2 The First Node of the Probe Output 51

A.3 Ouput from the Filter 52

1 Introduction

The Specification and Description Language SDL is a standard language developed and standardized by CCITT in the seventies. It is used around the world to specify and describe different kinds of systems. Systems are developed using SDL descriptions which can be translated automatically into computer code. These systems, which often are distributed and highly concurrent (e.g. telephone exchanges), have grown continuously all the time and it is very important to find design errors as early as possible in the system development phase. This is due to the fact that if an error is made in one design phase and it is not found, the correction of that error is ten times more expensive in the next design phase. The complexity and in particular the concurrency of systems leads also to the situation where it is almost impossible to find the chain of events which caused the system to get into an error state by using conventional programming tools like debuggers.

The reachability analysis offers a precise way to verify and observe the behaviour of distributed and concurrent systems. With reachability analysis, an abstract model of the system is used to generate a reachability graph. The complete reachability graph consists of all reachable states which the system can be in and all transition instances fired in the system. With reachability analysis it is possible to verify some desired properties of the system and also find out all the chains of events which have caused error states within the system.

TNSDL is a dialect of the SDL-88 standard ([SDL88]) and it is used at Nokia Telecommunications not only to specify but also to implement systems. The differences between TNSDL and SDL-88 are not very important for this work, but some are mentioned in the text. This report is about the *EMMA* (Extendible Multi Method Analyzer) project which has been financed by Nokia Telecommunications in Finland. The project began in 1995 and the writer has been in it since the begin of the main implementation phase march 1996. The purpose of the EMMA project was to create a TNSDL system analyzer which would eventually be in industrial use and which has the following functions:

1. Takes any description written in TNSDL language.
2. Constructs a formal model from it.
3. Performs the reachability analysis for that model.
4. Returns the analysis results in TNSDL form.

The user also has the possibility to query about various properties of the system and EMMA gives answers based on the reachability analysis. It is also notable that EMMA has to translate the *complete* TNSDL, all of its data types etc.

The formalism used in EMMA to construct the formal model is *Predicate/ Transition nets* because they are capable of modelling concurrency very well and there exists the *PROD* reachability analyzer [VHHP95] for Predicate/ Transition nets. PROD is used as one part of EMMA and with it the net constructed by EMMA's net model generator can be analyzed.

The requirements for EMMA's analysis are tough because already with small SDL programs it is a fact that reachability graphs tend to grow so fast that the creation of the complete graph is not possible. This is called the state space explosion. Several decisions and solutions had to be made in order to achieve a formal model i.e. Predicate/Transition net whose reachability graph would not explode.

EMMA makes it possible to dynamically analyze TNSDL programs. It is notable that EMMA is a complete analyzer, it accepts all TNSDL statements. With EMMA concurrency errors inside the system can be detected. Also finding reasons for these errors becomes a lot easier. Because PROD deals with Predicate/Transition nets the results (e.g. a chain of events resulting in an error state) have to be translated back to the TNSDL form to be easily understandable for TNSDL programmers at Nokia. EMMA does also this translation with a program called `filter`.

In Section 2 the Predicate/Transition-nets are presented as well as the PROD analyzer. Predicate/Transition-nets are described and the basic concepts are defined. It is shown how to use PROD to make an analysis out of a Predicate/Transition-net with an example. In Section 3 there is a description of the structure of the SDL language.

In Section 4 it is explained how EMMA translates SDL to a Predicate/Transition-net. It is explained statement by statement how the net is created. The overall structure of the analyzer and the user definable options are also discussed. The Section 5 presents all kinds of problems encountered in the EMMA project and discusses how they were solved.

In Section 6 there is some information gained during the test phase of EMMA and a small example is presented. The appendix A shows some output data of the example. The Section 7 describes how EMMA should be developed in the future.

2 Predicate/Transition Nets

Predicate/Transition nets (Pr/T-nets) are a formal way to describe various dynamical systems. The basic net theoretical system model was proposed by C.A Petri in 1962. In [Gen87] H.J. Genrich presents a way to merge and abstract the basic net theoretical system in order to get a new theoretical model, Pr/T-nets, which belong to high-level Petri nets. In Pr/T-nets a dynamical system is modelled with a set of individuals ordered by functions and relations. The structure is partially static and partially dynamic. Pr/T-nets were first formulated by Genrich&Lautenbach in 1981. The formal presentation of Pr/T-nets and their analysis given in the following subsections is based on definitions given in [Gen87] and [Rau90].

There were many reasons to select the Pr/T-nets to the EMMA project for modelling systems created with SDL programs. The first is of course their excellent ability to model concurrency. They stand on a solid theoretical ground and the graphical presentation of the Pr/T-nets is very easy to understand. Pr/T-nets also are high level Petri nets which means that their expressive power is good enough to model a high level language like SDL. In addition to these facts it was important that a reachability analysis tool PROD (discussed in Section 2.3) is available for analyzing the Pr/T-nets.

2.1 The Basic Form of Predicate/Transition Nets

A relational structure is used for modelling a dynamic system. It is a tuple of objects, R = (D; f_1,...,f_k; R_1,...,R_n), where D is a non-empty set of individuals and f_i is a function and R_j is a relation in D. The static part of the relational structure is an ordinary relational structure called *support* of the dynamic system (see definition 6). The structure is also dynamical because some relations are variable. Their extensions may vary from state to state. The dynamics is represented as an annotated net called Pr/T-net. In Pr/T-nets the variable relations appear as places. The names of the functions (operators) and relations (predicates) in the net appear in annotations.

Definition 1 (The vocabulary of the first-order language) *Let for each index n≥ 0 there be two sets: n-ary operators and n-ary predicates. These operators and predicates form the vocabulary of the first-order language* ***L*** *which consists of expressions. Expressions are either terms or formulae. A set of symbols, V, denotes individual variables. The set V is disjoint from operators and predicates.*

The construction of terms and formulae is as follows.

1. Terms:

(a) A variable is a term.

(b) If $f^{(n)}$ is a n-ary operator and v_1,...,v_n are terms then f(v_1,...,v_n) is a term.

(c) No other expression is a term.

2. Formulae:

   (a) If v_1 and v_2 are terms then v_1=v_2 is an atomic formula.

   (b) If $P^{(n)}$ is a n-ary predicate and v_1,...,v_n are terms then Pv_1,...,v_n is an atomic formula.

   (c) If p_1 and p_2 are formulae then $\neg p_1$ and $(p_1 \vee p_2)$ are formulae.

   (d) If x is a variable and p a formula the (∃ x)p is a formula.

   (e) No other expression is a formula.

An occurrence of a variable x is called a *free occurrence* if it is not in the range of a (∃x) or (∀x). The set of variables that occur freely in an expression (term or formula) is called the index of that expression. If the index set is empty the expression is called *closed*.

Definition 2 (Substitution) *Let E be an expression, $x_1,\ldots,x_n$ be variables and $v_1,\ldots,v_n$ terms. A* substitution *α is defined as $\alpha = \{x_1 \leftarrow v_1,\ldots,x_n \leftarrow v_n\}$. When every free occurence of x_i is substitued with v_i for $1 \leq x \leq n$ the result is called an α* -instance *of E and defined E:α = E:$\{x_1 \leftarrow v_1,\ldots,x_n \leftarrow v_n\}$.*

Definition 3 (L-structure) *Given a language L, a* structure for *L (also: L-structure) is a structure where every operator $f^{(m)}$ of L denotes a m-ary function of R (marked: f_R) and every predicate $P^{(n)}$ of L denotes a n-ary relation of R (marked: P_R).*

The structure R assigns to each term which has no free variables an individual of R. To each formula which has no free variables (proposition) a truthvalue (*true* or *false*) is assigned.

This first-order structure R is dynamic which means that some of its functions and relations are static and some are variable. The set of predicates can be divided into two subsets. One of the subsets includes the static predicates and the other one the variable predicates of that set.

Definition 4 *Let L be a first-order language,* ***A*** *a finite set of formulae of L, p, q formulae of L, R a L-structure and r a formula of L_R. Then*

1. *r is said to be valid in R iff each closed instance of r is true in R. This is expressed as $R \models r$.*

2. *p is said to be valid iff it is valid in every L-structure. This is expressed as* $\models p$.

3. *p implies q iff* $(p \rightarrow q)$ *is valid. This is expressed as* $p \Rightarrow q$.

4. *p is called a* logical consequence *of* $\boldsymbol{A}$ *iff p is valid in every model of* $\boldsymbol{A}$. *This is expressed as* $\boldsymbol{A} \models p$.

Definition 5 (Symbolic sums) *Given a first-order language L, for each n* ≥ 0 *a class* $LC^{(n)}$ *of* symbolic sums *of tuples of length n is defined:*

1. *The constant 0 is in* $LC^{(n)}$.

2. *If* $v_1,\ldots,v_n$ *are terms, the n-tuple* $(v_1,\ldots,v_n)$ *is in* $LC^{(n)}$.

3. *If* l_1 *and* l_2 *are in* $LC^{(n)}$, *then* $(l_1 + l_2)$ *is in* $LC^{(n)}$.

4. *If l is in* $LC^{(n)}$ *and x is a non-negative integer, then xl is in* $LC^{(n)}$.

5. *No other expression is in* $LC^{(n)}$.

Definition 6 (Pr/T-net) *Let L be a first-order language and let* L_s *designate the sublanguage which consists of the predicates denoting static relations. A Pr/T-net is a 3-tuple* $PRT_L=(N,A,M^0)$ *where N is the underlying directed net, A is its* annotation in *L and* M^0 *is its* initial marking.

1. *N is a directed net N=(S,T;F) where S is a set of places, T is a set of transitions and F is a flow relation from places to transitions and transitions to places.* $S \cap T = \emptyset$ *and* $F \subseteq (S \times T) \cup (T \times S)$.

2. *A is the annotation of N,* $A=(A_N,A_S,A_T,A_F)$ *where*

   (a) $A_N = R$ *is a first-order structure for* L_s *called the support of a marked net.*

   (b) A_S *is a bijection between S and the set of variable predicates. It tells the size of the tuples in each place.*

   (c) A_T *is a mapping of T into the set of formulae that use only operators and static predicates.*

   (d) A_F *is a mapping of the set of arcs, F, into the set of symbolic sums of tuples of terms of L such that for every arc* $(x,y) \in F$, *whether it is coming out of a place or going into,* A_F *for that arc i.e.* $A_F(x,y)$ *must be in this set of symbolic sums.*

3. M^0 *is a initial marking of the places. It is a mapping that assigns to each place s in S a symbolic sum of tuples whose length has to be the same as the size of the tuples in place s. If n is the index of the predicate annotating s, then* $M^0(s)$ *is in* $LC^{(n)}$.

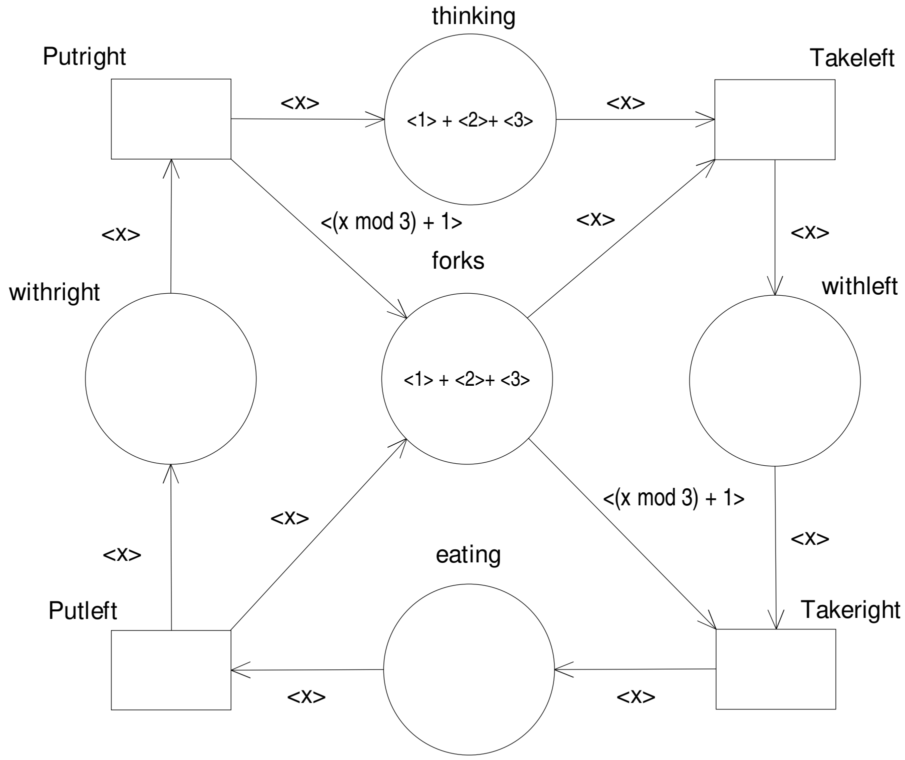


Figure 1: A Pr/T-net model of the dining philosophers problem.

In Figure 1 there is an example of a Pr/T-net. The places are modelled as circles and the transitions as boxes in the figure. Arrows connect places and transitions. Arc(x,y) is modelled as a directed arrow between elements x and y. The length of a tuple (of tuples in a symbolic sum) in a certain place is the *arity* of the place. In Figure 1 the arity of the place `thinking` (found from A_S) is 1.

Figure 1 describes the Pr/T-net solution to the problem of thinking philosophers (note that the solution is actually incorrect, which will be explained later). Here the case with 3 philosophers is considered. They are sitting around a round table and they are thinking. If anyone gets hungry he first takes one fork with the left hand and then one fork with the right hand. Then he begins to eat. After eating he first puts the fork in the left hand back to the table, then the right one and finally starts thinking again. There is one fork on each side of every philosopher but because the table is round there are only three forks for three philosophers.

The net in the Figure 1 is in the initial state so that the initial marking can be seen. There is a symbolic sum of 1-tuples (tuples with the arity 1) in places `thinking` and `forks` which show that each philosopher is thinking and every fork is on the table. If philosopher 3 gets hungry he first takes the fork in his left hand. This is modelled with the transition Takeleft. The variable `x` in the annotation of the arrow between the place `thinking` and the transition Takeleft is substituted by number 3 and the two incoming arrows to the transition show that the fork number 3 is taken from the table and philosopher 3 is not thinking anymore. The tuple $\langle 3 \rangle$ disappears from the places `thinking` and `forks` and goes to the place `withleft` indicating that philosopher 3 has the left fork in his left hand. Next philosopher 3 takes the right fork. This is modelled with the transition Takeright. The number of the right hand fork is (`x` modulo `n`) + 1. Here the `n` is 3 and `x` is 3 so the fork number 2 is taken and the tuple $\langle 2 \rangle$ disappears from the place `forks`. Now the philosopher 3 is eating and there is $\langle 3 \rangle$ in the place `eating`. After eating the forks are put back and finally after the transition Putright the system is back in the initial state.

A_T (from definition 6), which is a mapping of the set T to the set of formulae called transition selectors, contains the *gate* of each transition. In Figure 1 there are no gates for transitions, because usually the simple selectors are not marked in the pictures. For example the selector for the transition Takeleft would be `x`=`x` but there is no need to explicitly mark it. The selectors come from the arc annotations of the Pr/T-net (A_F in the definition 6). Note that the same selector `x`=`x` is the selector for all four transitions in this net.

Definition 7 (Feasibility) *Let $t \in T$ be a transition and $s \in S$ be a place of the Pr/T-net, $p = A_T(t)$ the formula annotating t and α be a substitution that replaces all variables occurring in t by constants. The substitution α is called* feasible *iff*

1. *α satisfies p, i.e. there is a substitution β such that $R((p{:}\alpha){:}\beta) = true$.*

2. *α creates a set on every incident arc. For every place s and arc annotation $l = A_F(s,t)$ or $l = A_F(t,s)$, $R(l{:}\alpha)$ is a linear combination with coefficients being either 0 or 1.*

3. *α does not generate an impurity. For no s with both $(s,t) \in F$ and $(t,s) \in F$ there is a resulting tuple that occurs at both arcs:*
   $R(A_F(s,t){:}\alpha) \cap R(A_F(t,s){:}\alpha) = 0$.

The formalism of the Pr/T-nets is further generalised in [Gen87] from places representing sets to places representing multi-sets. The items 2 and 3 of the above definition of feasibility are then dropped. Thus many copies of a tuple are allowed at the same place. Genrich however states that Pr/T-nets marked by multi-sets need not be a formal generalisation but a special case of strict Pr/T-nets where places represent sets.

Definition 8 (α-instance) *Let $t \in T$ be a transition and $s \in S$ be a place of the Pr/T-net, $p = A_T(t)$ the formula annotating t and α be a feasible substitution for t. Then the α-instance of t, designated by $t{:}\alpha$ is a pair of sets $({}^\bullet(t{:}\alpha),(t{:}\alpha)^\bullet)$ of closed atomic formulae.*

A formula $Pd_1...d_n$ is in ${}^\bullet(t{:}\alpha)$ iff there is a place s such that $(s,t) \in F$ and

1. *s is annotated by P, $P{=}A_S(s)$.*
2. *The coefficient of $R(d_1...d_n)$ in $R(A_F(s,t){:}\alpha)$ equals 1.*

A formula $Pd_1...d_n$ is in $(t{:}\alpha)^\bullet$ iff there is a place s such that $(t,s) \in F$ and

1. *s is annotated by P, $P{=}A_S(s)$.*
2. *The coefficient of $R(d_1...d_n)$ in $R(A_F(t,s){:}\alpha)$ equals 1.*

Pr/T-net is dynamic because transitions can be *fired*. A transition has to be *enabled* before it can be fired. The following definitions define these two concepts.

Definition 9 (Enabledness) *Let M be a marking of a Pr/T-net, t a transition and α a feasible substitution of t. Transition $t{:}\alpha$ (the α-instance of t) is* enabled *at the marking M, if*
$\forall s \in S \mid [R(A_F(s,t){:}\alpha) \subseteq R(M(s)) \land R(A_F(t,s){:}\alpha) \cap R(M(s)) = \emptyset]$.

This is denoted $M[t{:}\alpha\rangle$.

Definition 10 (Firing rule) *A feasible α-instance of t, $t{:}\alpha$, enabled at a marking M can* fire *to a marking M′, if*
$\forall s \in S\ R(M'(s)) = R(M(s)) \setminus R(A_F(s,t){:}\alpha) \cup R(A_F(t,s){:}\alpha)$.

This is denoted $M[t{:}\alpha\rangle M'$.

In Figure 1 it can be seen that the transition Takeleft is enabled for example in the case where the feasible α-instance of the transition is when `x` is substituted by 3 and the gate of the transition is `x`=`x`. Then it can be seen that for every input place of the transition (`thinking` and `forks`) $\{\langle 3\rangle\} \subseteq \{\langle 1\rangle,\langle 2\rangle,\langle 3\rangle\}$ and for every output place $\{\langle 3\rangle\} \cap \emptyset = \emptyset$. And if the enabled transition Takeleft α-instance is fired in Figure 1 in the initial marking M_0 the tuple $\langle 3\rangle$ is removed from the input places `thinking` and `forks` and the same tuple is added to the output place `withleft` of the transition as was explained earlier.

2.2 Analysis of Predicate/Transition Nets

The reachability analysis is an efficient method to analyze the Pr/T-nets. The aim of the reachability analysis is to compute the graph of all system states that can be reached from its initial state. The analysis is equivalent to an analysis of all sequences that the system can execute, but it is considerably more efficient. The analysis creates a *reachability graph* which consists of a set of markings V (vertices) and a set of edges E in the following way.

Definition 11 (Reachability graph) *The* reachability graph *of a system is a pair* $\langle V,E \rangle$*, where* V *is a set of markings and* $E \subseteq V \times T \times V$ *and the following holds:*

1. $M_0 \in V$.
2. *If* $M \in V$ *and* $M[t\rangle M'$*, then* $M' \in V$ *and* $\langle M,t,M' \rangle \in E$.
3. *V and E have no other elements.*

The set of vertices V contains all reachable states of the system and the set of edges E contains all the actions happened (fired transition instances) in the system. When a reachability graph has been created it is possible to find *terminal nodes* of the system and also check if certain properties hold in the system.

Definition 12 (Dead marking) *A marking M is* dead*, if for all t in T none of the feasible* α*-instance* $t{:}\alpha$ *is enabled i.e.* $\forall t \in T \neg M[t{:}\alpha\rangle$*. The vertex which represents a dead marking in the reachability graph is called a* terminal node *or a* deadlock *of the graph.*

As was mentioned earlier, the Pr/T-net of Figure 1 introduced to solve the dining philosophers problem is not correct. When the reachability graph is created a deadlock node and a path (transition sequence) leading to that node can be found (see also Section 2.3). It is the situation where each philosopher has taken the left fork. Thus the marking of the net would be that there is a sum of tuples, $\{\langle 1\rangle + \langle 2\rangle + \langle 3\rangle\}$ in the place `withleft` (all other places are empty). This marking is a deadlock because there is no transition enabled in the net.

The main problem with the reachability analysis is the state space explosion. The size of the reachability graph grows very fast (exponentially) with the size of the system and its net description. If it is sufficient to find out if the system has any terminal nodes then it is not necessary to create the whole reachability graph. There are also various methods which reduce the size of the reachability graph (see [Rau90]). Unfortunately the two methods used in the PROD analyzer: the symmetry method and the stubborn set method, both require the unfolding of the Pr/T-net. In the EMMA project the unfolding of the created net was not possible because the net was so complex that the creation of the unfolded net would have taken too much time. Instead of these two methods *on-the-fly verification* methods had to be used. They are also supported in PROD. The desired property can be verified also during the net creation thus it is not always necessary to create the whole net. In order to get the best possible results the use of on-the-fly methods was taken into consideration already in the construction of the EMMA's net model generator (see Section 4). It is also possible to do queries in the query language of `probe` also for a reachability graph which is not complete, which is important when the graphs are very large.

2.3 PROD Reachability Analysis Tool

PROD, a Pr/T-net reachability analysis tool, has been developed by a group of researchers of Digital Systems Laboratory and students of Helsinki University of Technology. PROD creates a reachability graph of a system modelled as a Pr/T-net. From that graph the user can search for terminal nodes, paths (transition sequences) leading to those terminal nodes and also paths which satisfy some given property. The complete manual of PROD can be found in [VHHP95].

In the EMMA project the PROD analyzer is included inside EMMA. The Pr/T-net created by the net model generator is fed to PROD which returns the results to be analyzed further. For the EMMA project PROD was improved with some additional features which included a proper command line editor and the possibility to redirect the results of the query from the command line to a file. Later when the user interface of EMMA will be complete the user of EMMA does not have to be familiar with PROD.

Because it was impossible to unfold the nets created with the EMMA's net model generator it was important to put some effort in making the net model generator to produce as small net as possible from the SDL program (see Section 4). After that it is possible to use the query language of PROD in the analyzing phase to verify different kinds of properties of the system and to search for deadlocks and paths leading to deadlocks in the system. As was mentioned earlier it is also possible to use on-the-fly verification during the creation of the reachability graph. This is supported by *testers* and *verify formulae* in PROD.

2.3.1 Using PROD

The following example presents the Pr/T-net of Figure 1 converted into the PROD's net description language. PROD compiles the description into an executable program which creates the reachability graph of the net. This compilation is done with the `prpp` program which is a part of PROD. The other parts are `strong`, which computes the strongly connected components of the graph and `probe`, which is the query program. The PROD compilation is started with the `prod` batch program.

From the net description the syntax for presenting places and transitions of the net can be seen. In places the name of the place, upper and lower limits for the marking and the initial marking are given in the description. Transitions have name, input and output place names (and arc annotations) and the transition selector (gate). The transition selector states a condition for a transition to fire. It is an integer-valued expression containing for example input variables (the tuples coming from the input places). The expression at the gate must have a value different from 0 before the transition can be fired. A transition can also have a comp block which is written in the C language and which for example can be used to define values for input and output variables.

As earlier also here it is not necessary to put trivial transition selectors to the net description. PROD knows that if for example x has been used in both input and output places the tuple substituting it has to be the same in both places. The net description language is actually the C preprosessor language extended with net description primitives.

```
#define n 3
#define LEFT(x)  (x)
#define RIGHT(x) (1 + ((x) % n))
#place thinking  lo(<.1.>) hi(<.n.>) mk(<.1..n.>)
#place forks     mk(<.1..n.>)
#place withleft  lo(<.1.>) hi(<.n.>)
#place eating    lo(<.1.>) hi(<.n.>)
#place withright lo(<.1.>) hi(<.n.>)
#trans Takeleft
   in  { thinking: <.x.>; forks: <.LEFT(x).>; }
   out { withleft: <.x.>; }
#endtr
#trans Takeright
   in  { forks: <.RIGHT(x).>; withleft: <.x.>; }
   out { eating: <.x.>; }
#endtr
#trans Putleft
   in  { eating: <.x.>; }
   out { withright: <.x.>; forks: <.LEFT(x).>; }
#endtr
#trans Putright
   in  { withright: <.x.>; }
```

```
   out { thinking: <.x.>; forks: <.RIGHT(x).>; }
#endtr
```

On-the-fly verification in PROD is done either with *testers* or *verify formulae*. Testers are special places added to the net description and connected to the transitions of the net. At any reachable marking, a tester contains exactly one tuple which is the state of the tester. The tester can be used for finding deadlock and livelock states, as well as some undesirable reject states. PROD stops the generation of the reachability graph and displays a report when the state of the tester is undesirable. Verify formulae can also be included in the net description in the form of "#verify formula". The PROD automatically constructs a tester place corresponding to this formula. The formula is a linear time temporal formula stating statements like "A until B", "eventually A" and so on. For more information about testers and verify formulas look at [VHHP95].

2.3.2 The Query Program Probe

The main analyzing results in the EMMA project are obtained with the use of the query program `probe`. Its query language can be used for searching certain nodes with certain properties or it can be used for searching certain paths with certain properties. The user interface of `probe` can be seen in the next example. `Probe` prompts a number and # after it. The number is a node in the reachability graph. So after starting `probe` there is 0# indicating that the first node of the graph is under examination. Now a `look` command prints the marking of the net i.e. the places and their markings at the current node. At the initial state there is an initial marking in the net which can also be seen in Figure 1.

```
0#look
Node 0
  thinking: <.1.> + <.2.> + <.3.>
  forks: <.1.> + <.2.> + <.3.>
------------------------------------------------
```

Next a `succ` command is given, which shows the successor arrows of the current node (i.e. enabled transitions and their possible instances). For example in the initial node of the reachability graph of the dining philosophers problem there are 3 possible successors. In each of them the transition Takeleft is fired and in the arrow 0 the x in the arc annotation is substituted by a constant 1.

```
0#succ
Arrow 0: transition Takeleft, precedence class 0
  x = 1
```

```
to node 1
------------------------------------------------
Arrow 1: transition Takeleft, precedence class 0
  x = 2
to node 2
------------------------------------------------
Arrow 2: transition Takeleft, precedence class 0
  x = 3
to node 3
------------------------------------------------
Node 0 has 3 successor arrows
------------------------------------------------
```

Next the arrow 2 is chosen and it is seen that the transition fires with x substituted by 3 which leads to the node 3 of the reachability graph. The command `goto` is used to enter this node. Next the marking in the node 3 is checked. It shows, as expected, a situation where the philosopher number 3 has taken a fork in his left hand and the other philosophers are thinking and the other forks are on the table.

```
0#goto 3
3#look
Node 3
  thinking: <.1.> + <.2.>
  forks: <.1.> + <.2.>
  withleft: <.3.>
------------------------------------------------
```

A command `strong` has been used before starting `probe` to generate the strongly connected components of the graph. A strongly connected component of the graph is a part of the graph where it is possible to find a path from each node of the component to all other nodes in that component of the graph. If the whole graph is strongly connected, there is a path from every node to all other nodes in the graph. In `probe` the command `statistics` shows all kinds of information about the reachability graph. The number of nodes shows the reachable nodes of the graph. The number of arrows shows the number of fired transition instances in the graph. The number of terminal nodes in this case shows the terminal node mentioned in 2.2. Note that it is possible to stop generating the graph and do queries with `probe` also in a partially generated graph. From the number of nodes that have been completely processed it can be seen if the graph is fully generated or not. There are in this case 2 strongly connected components in the graph. That is because the terminal node itself is the other one. If it was not, then the number of nontrivial terminal strongly connected components would not be 0.

```
3#statistics
Number of nodes: 20
Number of arrows: 27
Number of terminal nodes: 1
Number of nodes that have been completely processed: 20
Number of strongly connected components: 2
Number of nontrivial terminal strongly connected components: 0
3#showends %0
Node 10, belongs to strongly connected component %%0
  withleft: <.1.> + <.2.> + <.3.>
------------------------------------------------
1 path end nodes
------------------------------------------------
```

In `probe` there is a special set, named %0, reserved for paths leading to the terminal node. It is possible to use the `showends` command to search that node. The `showends` command shows the end nodes of the paths, in this example it shows the node 10 (see above) and the marking in it which is the situation explained in 2.2 where each philosopher has taken a fork in his left hand.

With the `query` command all kinds of queries can be made about different properties of the net. Nodes in the graph with certain places containing certain markings (tuples in the markings can be handled one field at a time) can be searched for as well as paths leading to those nodes. The verbosity level of the query can be chosen among 4 alternatives. The user can query one or all the nodes or paths satisfying the query formula. For example user can formulate a query and ask for nodes of the graph where no one of the philosophers is eating. Cycles in the graph can also be detected. In the next example there is a query for finding *one* path (transition sequence) which leads to a deadlock state. As can be seen, `probe` presents one path leading to the deadlock node 10 starting from the node 0 where an arrow 0 transition instance is fired. `Probe` also does a new set, %1 of the result of the query. This set can be studied separately.

```
0#query dspan(true,false) %0
   0  [0>    1  [0>    4  [0>   10
1 paths
Built set %1
```

3 SDL

Telephone manufacturers discovered already in the late sixties and early seventies that neither programming languages nor hardware description languages provided good enough tools to develop and maintain highly complex switching systems. Telephone exchanges are still among the most complex real-time systems existing. The specification and description language, SDL, was originally developed to meet requirements of the designers of switching systems. This presentation of SDL is based on articles [Bra95] and [Hog95] which were presented at the SDL Forum 1995. SDL88 is the version presented in this report because Nokia's TNSDL used in the EMMA project is a dialect of SDL88.

SDL was designed for systems that are reactive, concurrent, real-time and distributed. The behaviour of this kind of systems is dynamic, invisible and of an infinite nature although the behaviour is critical to the purpose and quality of these systems. SDL had to be able to describe the behaviour of a system better than anything else had been able to do before. Even today there is a lot of effort put in finding better ways to describe behaviour. It was also important to make SDL easy to understand and implementation-independent. Because SDL was also to be used in system verification and validation it had to have a solid formal representation.

Nowadays SDL is an international standard in the telecommunications area and it is used in system development to specify the required system behaviour as well as to design the system and make an optimal implementation of a system. SDL is also used for verification and validation. In 1992 a new object oriented version of SDL was presented and SDL is being developed further all the time.

3.1 The Basic Concepts of the SDL Language

Because SDL is implementation-independent, it tries to say as little as possible about the internal construction of the systems and all about how external stimuli and responses to these are related. The basic concepts of SDL were chosen on the basis of those goals. In SDL there are no built in assumptions about physical co-localisation of processes. Concurrency is modelled in SDL with *processes* which behave concurrently with each other as well as with the environment. Communication between processes and environment is handled with concurrent signal routes and *channels*. Processes interact mainly by asynchronous *signals*. Processes have a sequential internal behaviour because they are *sequential finite state machines* with *timers*. Processes behave in discrete state transition steps and have the ability to store and manipulate data, and to supervise time.

SDL also has concepts for the composition of subsystems which is important because of the size of the systems. *Systems* and *blocks* can be composed of blocks and channels. Blocks can be composed of processes and signal routes and processes in turn can be composed of *procedures*. There are also type concepts for signals, procedures, data, services, processes, blocks and systems and it is possible to define new types from other types.

3.2 The Structure of the SDL Language

SDL has two alternatives for syntax, the graphic representation SDL/GR and the textual SDL/PR. The graphic representation is preferred by most people because it shows the relationships more clearly than the textual representation. The textual representation, on the other hand, does not require expensive tools because an ordinary text editor can be used for creating programs. And when different graphical editors may not be compatible, a textual representation can be exchanged between vendors quite easily. In SDL a process is not a piece of behaviour, instead it is an object with a fully defined behaviour. It can only be composed in parallel with other objects to form a structure of an object with a concurrent behaviour. Process (and any object) instances can be identified.

If the structure of the SDL language is examined in a top down manner, the uppermost level of the structure is the system level. In SDL/GR there is a system diagram and SDL/PR there are keywords "SYSTEM" and "ENDSYSTEM". The system level description shows the blocks of which the system is composed. In a system diagram channels between blocks (and between blocks and the environment) and signals inside channels are also shown. The system diagram tells little about the behaviour so lower level structures are needed.

Next comes the block level. Inside blocks there can again be blocks, channel, process or signal definitions and also type references. Note that this decomposition of system to blocks and blocks to processes is only a logical decomposition of a system. The behaviour of system is also not known yet. For that the process level is needed.

SDL processes are described as finite automata. These finite automata are extended Finite State Machines, FSM, which suit excellently the purpose of SDL and allow a description of the behaviour by stimuli-response. FSM is widely used in the hardware and protocol design as well as in software engineering. SDL extends FSMs in a couple of ways and each process in SDL consists of the *input port*, the *FSM*, *timers* and *variables*. A signal arriving at the process goes to the input port which (theoretically) contains an unbounded queue. A FSM consumes signals according to the first-in-first-out principle unless the order is affected by a SAVE statement (see Section 4 for details of SDL statements). For each consumed signal the FSM makes a transition from one *state* to another. After consumption (input) there can be an SDL *transition* which consists of a sequence of statements (do not confuse the concepts of SDL transition and Pr/T-net transition). Communication in SDL is asynchronous. In synchronous communication both participants are simultaneously engaged into the connection (like in function calls). In the case of asynchronous communication the participants are not forced to work simultaneously. Both of them can execute other activities and, if programmed explicitly, receive later an acknowledgement indicating that the communication has been performed (like when messages are sent between blocks). In SDL there can be *services* inside processes. They are always a part of a set of automata and they can be used to group certain behaviour together. In EMMA services are called *subautomata* (see Section 3.4).

Each signal has a name which the FSM uses to select the transition. In each transition FSM may alter variables, generate output signals and perform timer operations. All this behaviour is found in a process diagram. For each state a set of transitions is defined. For each transition the input signals that may trigger the transition and a possible sequence of statements are specified. The transition ends with the *next state* which shows what happens next. Processes can be created at start-up or as a result of a *create* request from another process. A process can also call a *procedure* which help to achieve top down decomposition as well as bottom up composition of behaviours. Create requests and procedure calls can have arguments. If some signal unspecified in the state is next in the queue it is simply discarded and the process remains in the same state. This mechanism is referred to as *implicit consumption* (non-persistent input). See Section 4 for further information on this important feature. If the user does not want to consume some signal a SAVE statement can be used.

3.3 Validation of SDL Systems

Specifying a system with SDL resembles programming in any programming language. SDL program needs to be tested because it surely can contain errors. This testing is called validation. In addition to SDL descriptions there may be a number of other specifications refining each other which may be written using some other specification technique. The validation should also make sure that each of them are consistent with the other. The term verification is often used in place of the validation but this does not matter because the distinction between these two terms is not generally possible. The important thing in doing the validation is that before it is possible, the system has to be specified in some way, preferably in a formal way.

A validation model is executable so it is possible that the SDL program to be validated itself is a validation model. Because the programs are often very large (state space explosion) it may be better to make several separate validation models for separate aspects of the validation. In building the validation model it is important to try to avoid the state space explosion. This can be achieved by limiting the number of states and limiting the size of an individual state. In the EMMA project SDL programs created in industry were analyzed so there were not very big possibilities to change the SDL program, instead the emphasis in optimization had to be put on the generation of the Pr/T-net. For information about optimization techniques in EMMA see Section 4.

Validation can be either static or dynamic. Static analysis deals with evaluation of the system without executing any functions. It can be divided into syntax and semantic analysis. Correct syntax analysis is equivalent to a program which has been successfully compiled. Semantic analysis checks that the system definition is well defined, complete and self consistent. Static analysis alone is not enough to show that the model expresses correctly the intended functions.

Dynamical analysis requires the model to be executed. The behaviour of the model is evaluated with respect to assertions and invariants that define the desired system behaviour. These assertions and invariants are created by studying the system in detail. Also such properties as deadlock freeness, the use of variables before their definition and the possible implicit consumption can be studied. A suite of different test cases can be prepared to test particular functions of the model. In dynamical analysis two classes of validation methodologies can be found, exhaustive and non-exhaustive. The non-exhaustive analysis cannot be used to make precise statements about properties of systems but it can be used for searching errors. Also when the exhaustive analysis is not completely possible because of the state space explosion the non-exhaustive analysis can be used. The exhaustive analysis performs a complete analysis of a system to determine its properties. The most significant exhaustive validation methodology is the reachability analysis because it is powerful and easy to automate.

3.4 Differences Between TNSDL and SDL88

TNSDL is based on SDL88 but Nokia has done several modifications to the language. Some simplifications and some additions have been done and some things as channels and signal routes have been replaced by other mechanisms. The concepts of module and service are added in TNSDL (the service in SDL88 is close to the subautomaton in TNSDL). Some things have been renamed (for example Priority Inputs are called Input Internals in TNSDL). In TNSDL the MACRO feature has been omitted and a WHILE construct has been added to allow loops in a structured way without using JOINs.

All differences between TNSDL and SDL88 are, however, quite minimal from the implementation point of view. In EMMA SDL88 could have been used as well and almost all of the code (and problems) would be the same as with using TNSDL.

4 Implementation of EMMA

The EMMA project started in June 1995 with a preliminary investigation of possibilities of this kind of project was done. Some earlier research results in this area can be found in [Ket89]. The main purpose of EMMA was to build a TNSDL to Pr/T-net translator and a program for interpreting the results of the analysis in terms of TNSDL. The final user interface for this system will be made by Nokia. EMMA has to be able to translate the complete TNSDL so that it will be possible to use EMMA as a part of the debugging and verifying process in industrial switching systems development. EMMA is written in C++ using the GNU compiler in a PC/Linux environment. The structure of EMMA can be seen in Figure 2. The part of EMMA mainly relating to this report is marked with a dashed line.

The TNSDL translator was made by Nokia. It reads the program written in textual TNSDL and produces an executable C program from it. It also produces two symbol files, TESTP and FCF which are used by EMMA. The TESTP file contains the external symbol table of the TNSDL program and the FCF file contains the flow control information. The TESTP and FCF parsers fill the symbol table of EMMA with information about the complete detailed structure of the TNSDL program found in the TESTP and FCF files. The net model generator starts the creation of the PRODL (PROD net description language) net by going through the symbol table hierarchically. The generator also reads and parses the user definable options which are taken into consideration during the generation of the net. When the net is ready it is output to PROD. Analysis are made in PROD and finally the results are sent to the `filter` program which converts them back to the TNSDL form which is more readable for verification. In the following sections these phases are discussed in detail and an example of the process is given.

This section first explains in which way the TNSDL program can be translated to a Pr/T-net by going through all important statements of the SDL. Then the actual structure of the translator, which generates the net model, is discussed (Section 4.2).

4.1 From TNSDL to the Predicate/Transition Net Model

During the project it was noticed that many modifications needed to be made to the original plan presented during in the preliminary investigation. The requirement to be able to translate the complete TNSDL caused a lot of problems although the basic statements were quite easy to translate efficiently. The main thing which had to be kept in mind all the time was that it is important to try to prevent the state space explosion. Only a small increase

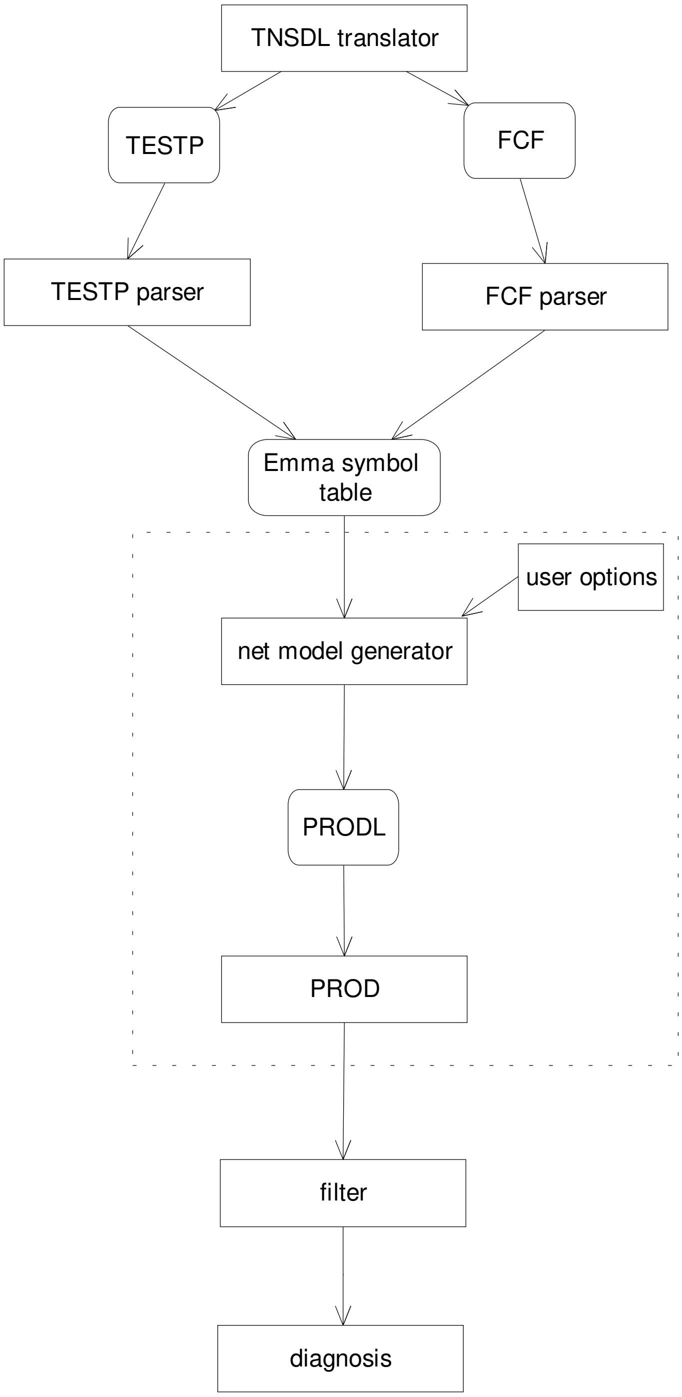


Figure 2: The structure of EMMA.

in the number of transitions in the net can double the size of the reachability graph!

The first notion to be explained is the *control flow*. A control flow is a concept used to describe the progress of the program i.e. which processes are executed and in which order. In EMMA the control flow token is a tuple of the form $\langle pid, state \rangle$ for every process where `pid` is the process identifier and `state` can contain some state specific information. The control place which contains this token shows the net transition which can be fired in a particular process. Every net transition corresponding to an SDL statement has an input arc from a control place and an output arc to the control place of the next net transition. In EMMA, the control flow usually means the same thing as the control flow token.

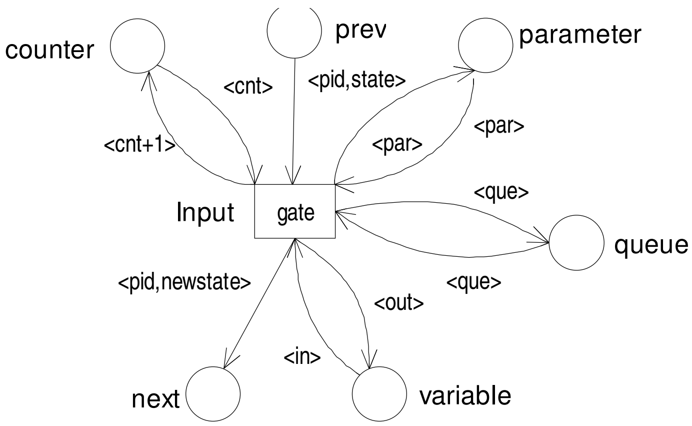
4.1.1 Handling the Queue

An important concept in SDL is the queue, which the processes use to communicate concurrently with each other. Each process has its own queue which contains the received signals. In EMMA it is modelled by a `queue` place in a Pr/T-net. The queue is of course not infinite as in the SDL standard but of very limited length. In EMMA the length of the queue is extremely important because a long queue will cause a state space explosion. In EMMA the queue length is a compile time constant (currently 3) but it can be changed if necessary. The queue place contains one token (a tuple of arity 5) for each slot in the queue. The tuple contains the following fields: $\langle n, pid, spid, sig, par \rangle$ where

1. `n` is the queue place number.
2. `pid` is the process identifier.
3. `spid` is the process identifier of the sender process.
4. `sig` is the signal number.
5. `par` is the signal parameter information.

If a queue is empty there is still a token for each slot in the queue but the tokens contain dummy values in their `spid`, `sig` and `par` fields. The number of tokens at a `queue` place is always the same, only the contents of the tuples is changed.

The complete Pr/T-net model of the queue needs two other places in addition to the `queue` place: `free` and `cursor`. The value of the `cursor` place shows the signal to be handled next. The value of the `free` place shows the first empty place in the queue.



gate: sig = 12
in: <pid,1,FromValue1> + <pid,2,FromValue2>
out: <pid,1,NewValue1> + <pid,2,NewValue2>
par: <MsgId,1,1,NewValue1> + <MsgId,1,2,NewValue2>
que: <n,pid,spid,sig,MsgId>

Figure 3: A part of a net model of the INPUT statement.

The process identifier is an identification number given to each process. Master processes are processes which exist from the startup of the SDL program and they have constant `pids`, hand processes (processes created later by the CREATE statement) receive their `pids` during creation (see Section 4.1.2). The signal number is an index to the *Signaltable* array which contains all signals of the system. Because PROD handles only numbers in markings the parameters of signals have to be encoded some way. The parameter identifier `par` is a number indexing tuples in a `parameter` place which contains the actual parameters of the signal. The OUTPUT statement puts the parameters to the `parameter` place and INPUT reads them. The right parameters for each INPUT or OUTPUT are found by using the parameter identifier field `par`.

In many cases the target of the OUTPUT statement is defined dynamically. This complicates the net model of that statement because there must be a possibility to output a signal to every possible queue. Because the Pr/T-net can not be changed dynamically, the only way to do this is to make one transition for each queue in the system. The `gate` condition in the output transitions then selects dynamically to which queue the signal actually is sent. Although many net transitions have to be created for one OUTPUT statement, this does not cause the reachability graph to grow because there can only be one enabled transition at a time.

The INPUT statement `INPUT off_hook_s (line)` is translated into a Pr/T-net in the way described in Figure 3. The INPUT statement reads a signal `off_hook_s` with code 12 which has one parameter `line`. The control flow comes from some state (the place `prev`) and goes next to the beginning of some net transition (the place `next`). The `line` variable (the place `variable`) is a field variable with two fields of integers so the parameter of the signal has also two values (see also Section 4.1.3). Notice how the right value for it is found from the `parameter` place by using the `MsgId` (parameter identifier) field. The `MsgId` coming from the queue place must be the same as the `MsgId` in the token from the `parameter` place. The value of the variable (containing in this case the values of the both fields of the field variable: `NewValue1` and `NewValue2`) is moved from the `parameter` place to the `line` variable. The old value is taken away by the `in` token and the `out` token is put back. The gate of the transition makes sure that the transition does not fire if the current signal is not the right one.

The Input transition increments the cursor of the queue in the `cursor` place so that it points to the next message. This is not shown in the picture. Note also that the token taken from the queue is returned unchanged. The place `counter` comes from the user definable options of EMMA (see Section 4.2.1). Its value is increased by one. The places dealing with the execution resource handling mechanism (see Section 4.1.5) are not shown in the figure. The predefined SENDER variable (see Section 4.1.2) of the process is set to the `pid` of the process which had sent the message.

Intuitively the INPUT statement should take the signal away from the queue after consuming it. This is, however, not done in the model because a SAVELATEST statement might need it later in the SDL program. In the net model the SAVELATEST statement is handled by creating a `savelatest` place for each process. This place only contains the information whether there is a SAVELATEST statement or not in the SDL transition. After the INPUT there may be various other statements in the SDL transition which are explained in the following sections. The SAVE statement is easy to implement. It causes a signal to be saved so that it can be consumed later. In the Pr/T-net it only increments the value in the `cursor` place. The gate of the SAVE transition makes sure that the current signal in the queue is the right signal to be saved.

Finally at the end of an SDL transition there is often a NEXTSTATE statement. It removes the signals consumed by INPUT if there has been no SAVELATEST statement. This information can be found from the `savelatest` place of the process. The token in the `cursor` place representing the cursor is reset to zero so that the signals saved by SAVE statements are available again. The `queue` place and the `free` place are also set according to the information from the `savelatest` place. The contents of the `savelatest` place is reset. The NEXTSTATE statement moves the control into the next SDL state by passing the control flow token to the beginning of the next `state` place.

If there has been no SAVELATEST statement in the SDL transition the contents of the `cursor` place is needed to define which signal should be removed from the queue in the NEXTSTATE statement. For different cursor values there has to be different net transitions. Also the case, where there has been a SAVELATEST statement in the program, needs its own transition. Again, this does not enlarge the reachability graph because only one Nextstate transition is enabled at a time.

The purpose of the *implicit consumption* (sometimes referred to as non-persistent input) is to remove those signals from the queue which have no corresponding INPUT or SAVE statement. The process does not change its state and the signal just vanishes from the queue. The net transition handling implicit consumption is of the same type as the net transition created in the INPUT statement. The difference is that in implicit consumption the state of the process does not change and the net transition puts the control flow token back to the same place it got it from.

There must be an Implicit Consumption transition for each slot in the queue and these transitions must also rearrange the queue correctly. For example, if there is a signal to be implicitly consumed in the second slot in the queue (cursor points at the second slot in the queue) and the length of the queue is 3, the Implicit Consumption transition for the second place removes the signal from the queue, moves the token of the third slot to the second slot and adds an empty slot to the third place. In the Pr/T-net there is an `implicit consumption` place for each process which contains information about signals not handled by any INPUT or SAVE statements. The Implicit Consumption transitions use that place to check whether consumption is needed or not. It must be noted that although there is a transition for each queue slot only one of them is needed at a time so the reachability graph does not grow. There is one set of Implicit Consumption transitions for each process.

4.1.2 Processes and Procedures

There are two types of processes in TNSDL. Master processes are created at startup time. Hand processes are created by master processes with a CREATE statement. Each process instance has a unique process identifier `pid`. Process instances can be killed with a STOP statement. In the Pr/T-net model master processes have an initial marking in their `queue` places and predefined variables, while hand processes receive marking via the Create transition. The predefined variables are SENDER, PARENT, OFFSPRING and SELF. SELF contains the `pid` of the process. OFFSPRING of the creating process holds the `pid` of the created process. PARENT contains the `pid` of the creator of the process (NULL for a master process) and SENDER contains the `pid` of the process from whom the last received message came from. These values are

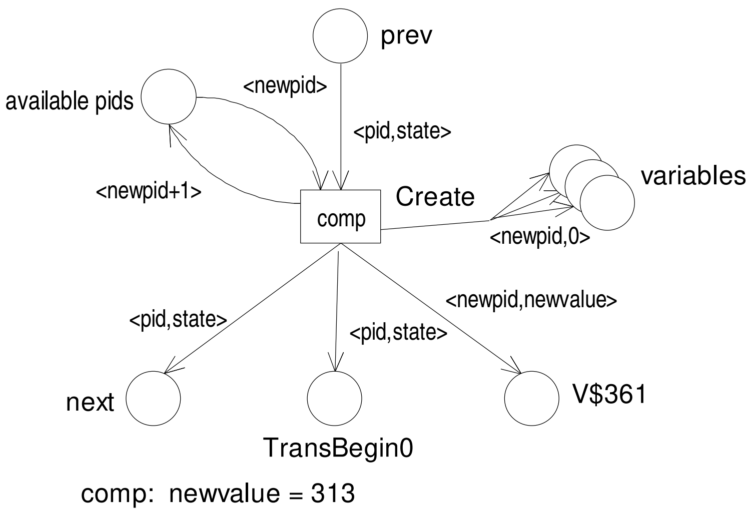


Figure 4: A net model of the CREATE statement.

initially NULL which is marked by 0.

In Figure 4 there is a picture of a net created by the statement `CREATE process_name (313)`. This statement creates a new instance of the process `process_name`. The Create transition initializes the variables of the new process. This is shown as `variables` places in the picture. If the variable is a simple variable with an integer value, the token $\langle newpid, 0\rangle$ is put to the place, where `newpid` is the `pid` of the created process taken from the `available pids` place. The value of the `available pids` place is incremented by one and put back to the place so that the next Create transition gets a different `newpid` for the created process. If a variable is of a more complex type a simple pair is not enough. See Section 4.1.3 for details. Note also that the notations of tokens in the pictures in this report lacks dots surrounding the tuple compared to the notations used in PROD.

It is possible to pass arguments to the CREATE statement. In Figure 4 the value '313' is given as an example. This value is of a simple integer type. The corresponding variable in the created process has a name which is the number of the symbol table entry, in this case 361. EMMA names the place with V$361. In the Create transition it is sufficient to put the tuple $\langle newpid, newvalue\rangle$ at that variable place. Because there may be several instances of the same process which all have variables with the same name, it is necessary to include the `pid` of the process in this tuple. And because a new process is created, there is no previous value with that `pid` in the place. Notice that the `newpid` is the same as before and the `newvalue` comes from the comp block of the transition which states that `newvalue` has a value of 313.

The control flow comes from the place `prev` and continues to the place `next`. The place `TransBegin0` marks the start of the new created process. The control flow token is transferred to both `next` and `TransBegin0`. Thus the old process continues working *concurrently* with the new created process. There are also some places concerning the implicit consumption, subautomata and execution resource handling (see Section 4.1.5) which are not shown in the picture. The queue of the process is also created in the Create transition by putting initial markings at the `queue` places as well as to the `savelatest` place. The Create transition must also initialize the predefined variables of the new process. These places have been omitted from Figure 4.

The STOP statement only can terminate a process. Thus only the process can terminate itself. In a Pr/T-net the Stop transition has to remove all the markings the CREATE statement has added. It has to remove the markings of all variables of the process and empty all `queue` places of the process including the internal queue places if there are subautomata in the system. STOP uses the same tables (see Section 4.2) as the CREATE statement to find the variables of the process. The Stop transition consumes tokens from all the above mentioned places but it does not put anything back to those places. It only puts the control flow token at the `stop` place. After that the control token does not move anymore and it is possible to see which process has stopped by looking at the `pid` field of that token.

The net model of the CALL statement needs more than one net transition. This is because there must be a place in the net where the control can wait for the execution of the procedure. After the Call transition the control flow token waits in the `check` place and when the procedure has finished the Callcheck transition is fired. The CALL statement can have parameters which are passed to the procedure in the same way as in the CREATE statement. A CALL parameter can, however, also be of the type IN/OUT which means that the value of the variable in that parameter can be changed by the procedure.

A procedure uses the queue of the process which called it. In generating the Pr/T-net it has to be noted that if there is a statement involving the use of the queue (e.g. INPUT) then it is possible that there is a signal in the queue sent to the process which the procedure can't handle. This kind of signal must be saved (implicit save) in the procedure so that the process can consume it after the return from the procedure. The actual implicit save transitions are built in the generation phase of the automata (see Section 4.2).

The RETURN statement is closely related to the CALL because it returns the control from the called procedure. The RETURN can also have a parameter which is the return value of the procedure. The TASK statement is also closely

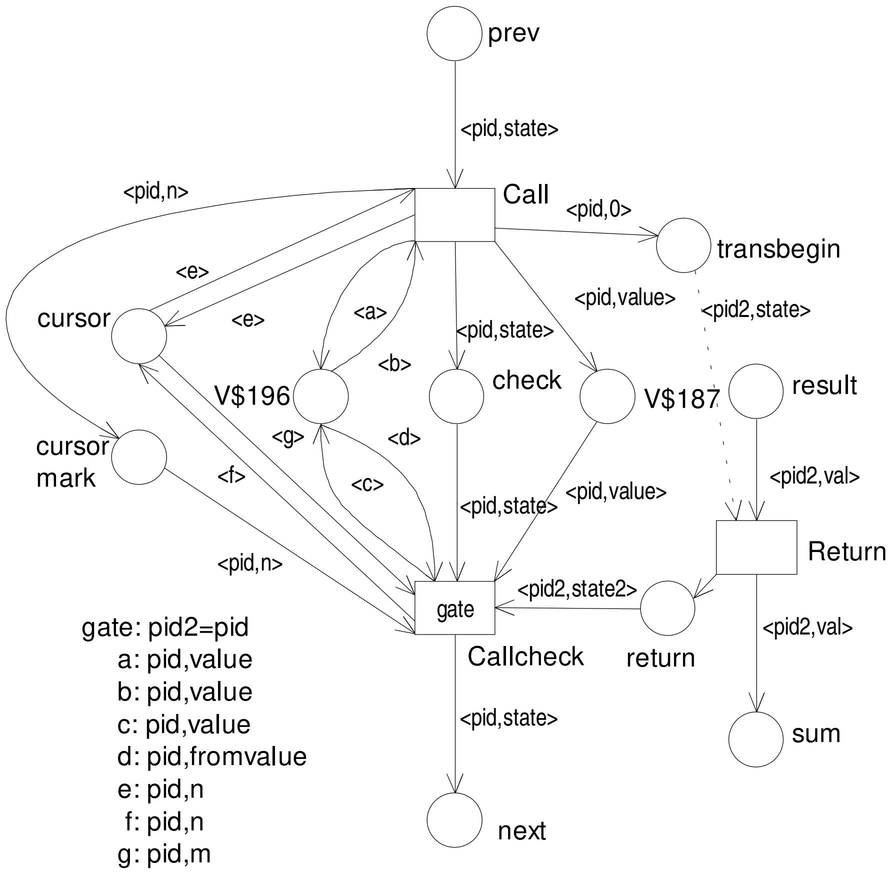


Figure 5: A net model for the CALL and RETURN statements.

related to CALL. There are three forms of TASK statements. The first form is an *assignment* where a TASK statement is used to give a value to a variable. The second is a *synchronous call* where TASK is used to call a routine written in some other programming language. The third form is an *informal remark* containing a textual description of what the TASK statement will do. The third form can be used as a placeholder while developing a TNSDL program. In the net model generator the TASK statement is almost the same as the CALL statement because it usually calls some action which is described as a procedure. In the case of an assignment the net model generator should simply perform the assignment (see Section 4.1.3 for assignments).

In Figure 5 there is a picture of a net created by EMMA from CALL and RETURN statements. The net is a part of a larger TNSDL net. The Call and the Callcheck transitions come from the TASK statement, the Return transition comes from the RETURN statement. These statements are shown below as well as a part of the procedure definition showing the type of the `parameter`.

```
TASK sum =  counter(parameter);
...
PROCEDURE counter;
  FPAR IN/OUT seed byte;
...
  RETURN result;
```

The parameter `seed` is of the type `byte`, i.e. it is an integer. The IN/OUT means that when the procedure returns to the calling process, the value of `seed` is transferred back to the parameter variable of the process. In Figure 5 the control flow token is read by the Call transition from the `prev` place. The Call transition saves the value of the cursor of the calling process by copying the token in the `cursor` place to the `cursormark` place. If any Implicit Save transitions are fired then after the procedure has ended the implicitly saved signals can be put back to the queue by using the information in the `cursormark`. The Call transition also starts the called process by moving a control flow token to the start place of the procedure (the `transbegin` place). The control is divided into two paths as in the CREATE statement and the other control flow token is put to the `checkplace` where it waits for the procedure to end. Let the symbol table number of the actual parameter in the TASK statement with name `parameter` be 196 and let the corresponding parameter variable `seed` in the procedure (a formal parameter) have the number 187. The codes matching the names can be found from the TESTP symbol table produced by TNSDL translator (see Figure 2). The Call transition moves the value from the place `V$196` to the place `V$187`.

At the end of the procedure there is the Return transition. The control flow token of the procedure is read from the last place of the procedure (dashed line in the picture) and continues to the `return` place which is connected to the Callcheck transition. If the procedure has states (INPUT statements) then the Return transition must remove the received signals. In a stateless procedure (a procedure with only one SDL transition which is the start transition) this is not necessary. The removal of the signals needs one net transition for each queue slot and one for the case when there has been a SAVELATEST statement. Again it must be noted that in the case of a procedure with states, although there are net transitions for each queue slot only one of them is fired at a time and the reachability graph size is unaffected. The `savelatest` place tells if there has been a SAVELATEST statement or not and the value at the `cursor` place selects the correct net transition. The Return transition arranges the queue signals in the `queue` place and when needed removes the parameters of the signal from the parameter place. The queue handling places are not shown in the figure.

The RETURN statement can return a `result` value which is put into the corresponding `return` place. It is possible to use expressions in the RETURN statement (see Section 4.1.3 for expressions). If there is an expression in the statement or if the return value has many fields, indexes etc. the transition will be more complex than shown in the picture. Now the case is simple and for example the `comp` block of the transition is empty. The basic idea is, however, that the value of the `result` variable of the procedure is moved back to the corresponding variable in the calling process which in this case is the variable `sum`. As before the places involving the execution resource handling mechanism (Section 4.1.5) are not included in the picture.

Note that when there are many instances of the same procedure at the same time running in the system, there may be many control flow tokens arriving at the `prev` place trying to fire the Call transition and it is possible to have many instances of the called procedure existing at the same time. They are, however, separated by the `pid` in the control flow token. The control flow of the calling process waits in the `check` place for the procedure to return control, after which the Callcheck transition is ready to fire. The gate of the transition tests that the `pid` coming from the `check` place is the same as the `pid` coming from the `return` place. The value of the IN/OUT parameter is returned from the formal parameter to the actual parameter. Note that the token from the formal parameter place `V$187` is not returned there.

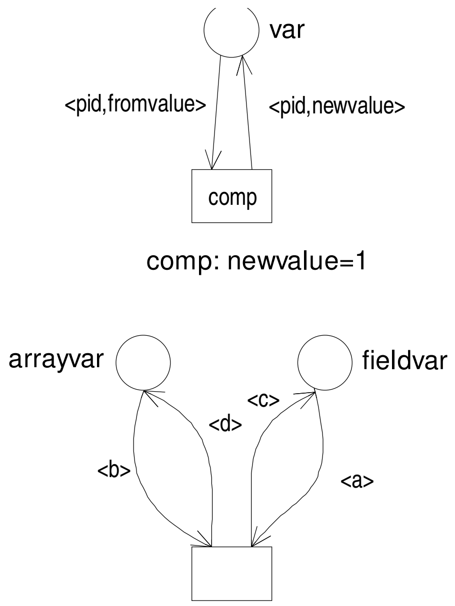
The implicit save is handled in the Callcheck transition. There is a special implicit save queue for each process and if there are signals in that queue they must be moved back to the actual queue after the return from the procedure. Again it is noticed that there must be several Callcheck transitions for each possible case, also one transition for the case that no implicit save has happened and the implicit save queue is empty. These implicit save queue places are not shown in the picture.

There are also two special cases of the CALL statement that have to be mentioned. It is possible that the service which is called has no TNSDL description. In this case EMMA either picks the return value of such a service randomly or it asks the user to enter the value (see Section 4.2 for user options). It is also possible that the procedure calls itself. In this case a control flow token of the form $\langle pid, state \rangle$ cannot be used because then there would be two similar control flow tokens in the system. To distinguish the tokens, there has to be one additional field in the control flow tuple, `callnumber`, which would guarantee that the control flow tokens do not interfere with each other. The gate of the Callcheck transition must then include also the expression which tests that the `callnumber` fields are the same in addition to the `pid` field. In the current implementation recursive calls are not yet implemented because the additional field would have to be added also to all variables to handle recursive values.

Timers are conceptually difficult to model in the Pr/T-nets because there is no concept of "time" in Pr/T-nets. A timer is initialized in a SET statement with some value to last maybe a couple of seconds. After the specified time has elapsed the timer expires and sends some timeout signal. In EMMA timers were implemented by using *timer windows* which either allow or do not allow the firing of the timer. The user can control the timer window, i.e. the user can decide the limiting net transitions in TNSDL between which the timer can expire (see Section 4.2.1 for user options). This implementation allows precise enough modelling of the timers and at the same time limits the size of the reachability graph because it is not possible for a timer to expire outside its window. A RESET statement stops the timer. At the Pr/T-net level there is a special place for timers. There is always a token, a *frame*, at that place for every possible instance of the timer. If a SET statement occurs, it changes that token to indicate the new timer instance. The token contains some number, a parameter, which identifies the timer. The expiration time for the timer is ignored in EMMA because the firing is controlled with the expiration window. Information on the expiration windows can be found also in [Mal97].

4.1.3 Variables and Expressions

Expressions can be used almost anywhere in the TNSDL statements. The parameters in CREATE, CALL, OUTPUT, INPUT, DECISION, TASK and RETURN statements can contain expressions. The handling of the expressions is not easy because variables appearing in them can have complex types. It is easy to translate simple expressions involving integers or variables of the integer type to a Pr/T-net but more complex types cause an enormous increase in the complexity of the net model generator and a considerable increase of the



a: <.pid,9,0,4,FromVal1.>+<.pid,9,0,5,FromVal2.>+<.pid,9,0,6,FromVal3.>+
<.pid,9,1,4,FromVal4.>+<.pid,9,1,5,FromVal5.>+<.pid,9,1,6,FromVal6.>;
b: <.pid,0,4,AssignVal1.>+<.pid,0,5,AssignVal2.>+<.pid,0,6,AssignVal3.> +
<.pid,1,4,AssignVal4.>+<.pid,1,5,AssignVal5.>+<.pid,1,6,AssignVal6.>;
c: <.pid,9,0,4,AssignVal1.>+<.pid,9,0,5,AssignVal2.>+<.pid,9,0,6,AssignVal3.>+
<.pid,9,1,4,AssignVal4.>+<.pid,9,1,5,AssignVal5.>+<.pid,9,1,6,AssignVal6.>;
d: <.pid,0,4,AssignVal1.>+<.pid,0,5,AssignVal2.>+<.pid,0,6,AssignVal3.>+
<.pid,1,4,AssignVal4.>+<.pid,1,5,AssignVal5.>+<.pid,1,6,AssignVal6.>;

Figure 6: Handling assignment of expressions.

complexity of the net model itself. The main reason for this is that a Pr/T-net place can only hold integers and more complex data types must be converted to these simple building blocks which are then united in the net transition to form a variable. The basic principle in storing a complex datatype is to put several tokens with several fields at the variable place. For example if the variable in question is an array variable, then in the variable place there is one token for each index number and each token contains one value of the array. The tokens are tuples of the form $\langle pid, index, value\rangle$. The name of the variable place indicates the name of the variable. Many instances of the same process may exist at the same time and they all have variables of the same name so there has to be a process identifier (`pid`) in the token indicating to which process the value belongs. `Index` tells the array index and `value` is the value of that array element.

Figure 6 shows two transitions. The transitions could for example be created from a simple TASK statement but only arcs related to the expression handling are shown. In the top one there is a simple expression coming from the `TASK var=1` statement where the variable `var` of a simple type (integer, boolean, byte etc) is given a value 1. An old value is taken from the variable place and a new value is returned. The value for the `newvalue` field of the token is formed in the `comp` block of the transition.

In the second transition there is a more complex TASK statement of the type `TASK fieldvar.one := arrayvar` where the variable `arrayvar` is a 2x1 array of values which are of the struct type with three fields. The `fieldvar` variable is of the struct type with two fields. The first field is of the same type as the variable `arrayvar`. The statement means that the contents of the first field of the `fieldvar` variable is replaced with the contents of the `arrayvar`. The structure of for example the tuple $\langle a \rangle$ shows that the internal representation for the first field of the `fieldvar` is the number "9" and the both items of the array of that field are changed. The array indexes are "0" and "1" and both of them are of the struct type containing three fields. The symbol table numbers for these three fields are "4", "5" and "6". Thus the first field of the `fieldvar` contains an array with 3 values in both of the two indexes of that array. The total of six values have to be replaced.

In the first case it is sufficient to move one value i.e. one tuple but in the second case over 20 tuples are included in the arc expressions although only one field has to be moved! The assignment statement causes a lot of work if the variables in it have complex types. In EMMA there are recursive functions which handle the expressions one variable at a time. These functions are used when generating net models for all kind of statements for handling the expressions.

The module concept is in SDL used for routines which can be called by every process. The module variables are global because they are common to all processes. In EMMA there normally is always a `pid` entry in the token at the variable place and a process can access only tokens which have the same `pid` as the process. The module variables do not have any `pid` entry because they must be accessible by all processes.

4.1.4 Conditional Branching and Loops

The program control flow branches and loops is handled by the DECISION, JOIN and WHILE statements. The JOIN statement can be used to jump to some point in the code. That point is indicated with the LABEL statement and the JOIN acts like a GOTO statement in some programming languages. The JOIN statements are analyzed in the init phase (see Section 4.2) and the jumping points are put into a table. Thus the Join transition just passes the

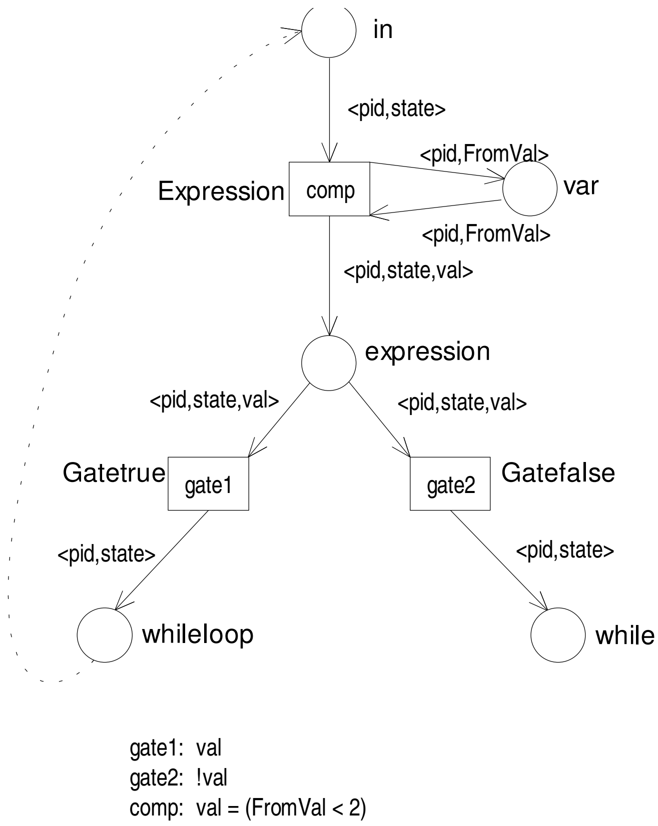


Figure 7: A part of a net model of the WHILE transition.

control flow token from the previous place to the next place found in that table.

The JOIN statement is not needed very often in TNSDL because the WHILE statement is preferred in the most cases. It can be quite complicated and it needs three transitions in the Pr/T-net. The WHILE statement consists of the conditional expression and a list of statements. If the conditional expression is evaluated to `true` (its value is not zero) the statements in the list are executed. After that the conditional expression is evaluated again and as long as it is true the statements are executed over and over again.

The Pr/T-net generated from the WHILE statement is seen in Figure 7. The Expression transition evaluates in its comp block the conditional expression and moves the result `val` in the control flow token to the `expression` place. Depending of the boolean value in the `val` field either the Gatefalse transition or the Gatetrue transition is fired. If the value is `true` the Gatetrue transition fires and the control flow token goes to the `whileloop` place. After that place there is a chain of net transitions (the dashed line) and places in the net which have been generated from the statement list of the WHILE. The last place of the chain is the `in` place after which the Expression transition can be executed again. The loop ends when the Gatefalse transition fires.

The DECISION statement corresponds to the SWITCH statement in the C programming language. There is a decision expression and a number of branches which each have a branch expression and a list of statements which may be empty. The decision expression is evaluated first and the result is compared to the each branch expression. The branch whose corresponding expression equals the value of the decision expression is selected and the statements in that branch are executed. After that the execution of the program continues at the statement following the DECISION statement except in the case the selected branch contains NEXTSTATE, JOIN or some other terminal statement when the program continues at the transition specified by those statements.

If none of the branch expressions equals the value of the decision expression then nothing is done and the program control flow goes to the first statement following the DECISION. There is one special branch called ELSE which does not have any corresponding expression but if there is a ELSE branch in the DECISION statement it is executed if no other expression matches the value of the decision expression.

In EMMA there is a special variable in the Decision class which holds the value of the decision expression. Branches belong to the ConditionAction class. Each condition action branch consists of its own net transition sequence whose gate consists of an expression of the form: branch expression equals the value of the decision expression. An ELSE branch of course has to have a gate which is the negation of the values of all other branches. After the Decision transition each condition action branch goes through every net transition in the branch and makes sure that the last net transition of the branch is connected to the `return` place which is the same for all branches of the Decision net model. In case there is no ELSE statement in the DECISION statement then a dummy Else transition has to be created. This is because if none of the branch expressions matches the decision expression, then an additional net transition is needed to pass on the control flow token (or else the decision would end up in a deadlock!).

4.1.5 Reducing the Concurrency

As mentioned earlier it is very important to keep the size of the reachability graph as small as possible. The more concurrent transitions there are in the system the bigger the reachability graph will be. In EMMA it is possible to radically reduce unnecessary concurrency in the system. The unnecessary concurrency comes from the fact that every SDL statement in EMMA is concurrent. For example let there be two concurrent processes in the system and let both of them have SDL transitions which consist of many SDL statements. If the SDL transitions do not have anything to do with each other (they do not use the same global variables, they do not exchange any messsages etc.) then it is possible to execute one SDL transition completely one at a time without losing any reachable state. In this way the SDL transitions would be atomic. Using default options EMMA, however, interlaces every statement within the SDL transitions. The produced net model allows every possible execution order of the statements which results to a big reachability graph.

To prevent this behaviour the execution resource handling mechanism is provided in EMMA. This mechanism creates two `resource distributor` places. The idea is that when a SDL transition begins it *takes* the resource by replacing a token at the `resource distributor` place. With a locked token resource no other SDL transition can proceed. When the SDL transition ends it *opens* the lock by changing the token in the `resource distributor` place. After that some other SDL transition can take the resource. The execution resource handling mechanism must be attached to every Pr/T-net transition in the net. This kind of locking would make every SDL transition in EMMA atomic. This is, however, not a correct way to model SDL. For example when the SDL transition changes the data in a global module variable which is used by some other process too, it certainly matters in which order the statements in SDL transitions are executed. The same applies also to the sending of signals with OUTPUT statements. That is why EMMA provides the `AtomicTransitions` option (see Section 4.2.1 for user options) for opening the lock in the middle of an SDL transition in case an OUTPUT or an module variable dependent statement appears. Without this opening of the lock much of the behaviour of the system would be prohibited and the execution resource handling mechanism would be almost useless. It is also possible to tell EMMA to release the resource at some specific Pr/T-net transition (actually this can not be done for every transition). Information on the execution resource handling mechanism can also be found from [Mal97].

4.2 The Structure of the Compiler

The object oriented C++ language is used in the programs of EMMA. The object class structure of EMMA is based on the structure of the SDL program. The highest level is the FcfFlow class. If some variable is needed almost everywhere in the program it is defined in the FcfFlow. The classes are hierarchical as is SDL and the process class includes automata and automata include states and SDL transitions. SDL transitions include transition elements and a transition element is usually a SDL statement. Another big group of classes are the expression classes. Expressions are used in parameters of the statements and a large variety is available starting with a boolean literal. A binary expression can contain two expressions which are joined by some operator. Expressions allow recursion and can be very complex.

EMMA starts by checking the user definable options. Then it reads the TESTP and FCF symbol tables. Then the `init` function for FcfFlow is called in the first pass and in the second pass the `generate` function for FcfFlow is called. The `init` function of the FcfFlow will call the `init` functions of all top level classes which in turn call the `init` functions of the lower level classes. The `init` and `generate` functions are called recursively for all classes starting with the top level and going down to the bottom. The reason for using two phases is that we need to go through the whole SDL program structure two times. In the first time a lot of variables are initialized and some analysis are made. For example the tables containing the variables of each process are created in the initialization phase. The analysis allows some optimization of the net in the generation phase. The generation phase actually generates net transitions and places.

4.2.1 User Definable Options

The user definable options may have a considerable influence on the generated net model and on the analysis of the model. By using wisely the user options it is possible to reduce the size of the reachability graph significantly. In many cases the analysis may not be possible without using carefully tuned user options. Options are put in the *option file* which is a normal text file. All options have some value which is given in the form *optionname* = value. A binary option is "on" if the value is not zero. The option file can contain comments.

`RandomSync` option: This option specifies whether sync procedures return a random value. If not the user is asked to enter the return value. Normally the value of this option is not zero.

`InitVariables` option: This option indicates that the variables of the system are initialized with dummy values. This option is always needed and the value of the option should not be zero except for some special tests.

`MonitorDeadLocks` option: This option uses the feature of the PROD analyzer. If this option is given a value which is not zero, it will add a deadlock monitor place to the net (see Section 2.3.1 for on-the-fly verification methods in PROD).

`Verify` option: This option also uses the on-the-fly verification method of the PROD. It allows the user to enter a verify formula which can be used for example for monitoring the implicit consumption.

`AtomicTransitions` option: This option reduces the concurrency in the system by using resource distributor places and thus significantly reduces the size of the reachability graph. The value zero means that no resource distributor places are used. The value 1 means that the SDL transitions are atomic. Further the resource lock is opened for OUTPUT signals (value 2) and module variable references (3). The value 1 gives the smallest reachability graph but the value three is usually to be preferred because it will give the most accurate net model. The value zero will most likely produce very big reachability graph unless the system being modelled is very small.

`Constraint` option: This option determines the maximal number of times a net transition can be fired. The constraint is given by telling the name of the SDL transition, the number of the corresponding SDL statement within it and the number of times a transition can fire. This option can reduce the size of the reachability graph considerably, but it will also decrease the accuracy of the generated net model.

`ReleaseLock` option: It is possible to force a Pr/T-net transition to release the execution resource if needed. This can be useful if the user knows that a net transition has to release the resource in order to get all states of the system in the reachability graph. This option is used together with the `AtomicTransitions` option.

`MonitorTimerConsumption` option: The expiration of any timer can be monitored by using this option. This option uses the on-the-fly verification of PROD so the generation of the reachability graph is stopped if any timer expires.

`TimerWindowString` option: There are special locking places in the net for timers. If there is not a certain token in the locking place the timer can not expire. It is possible to control the possible expiration of timers by defining precisely the start transition and the end transition. The timer can expire only between instances of these two transitions. This option also reduces the size of the reachability graph.

Those are the most important options of EMMA. The `filter` program has options of its own and these options are also given in the same option file. More information about `filter` can be found in [Mal97].

5 Implementation Problems

The preliminary investigation report [HOPV95] made in the summer of 1995 described quite accurately how the SDL program should be translated into a Pr/T-net. However the actual implementation in EMMA differs a lot from it. It is true that the most of the TNSDL statements are easy to translate into a Pr/T-net so that only one transition is created to the net for one SDL statement. But it was the versatility of TNSDL described in the following sections that caused many problems. Thus the created net model had to be changed many times.

5.1 General Problems

Pr/T-nets describe systems in much lower level than SDL language. That is why some SDL concepts like complex data types and timers were difficult to model with these nets. A fundamental problem with Petri-nets is the difficulty to check whether there is a marking at a certain place or not. The generated Pr/T-net in EMMA is based on the assumption that there is *always* a marking at the place, only the contents of the token is changed. A certain token can be declared a NULL token meaning that the place is empty when that token is at the place. This arrangement may make the net model generator more complex but it will solve the problem without increasing the size of the net more than marginally.

The other fundamental problem is the fact that the structure of the Pr/T-net is static. It is not possible to dynamically change it when it has been generated. Thus there must be net transitions connected to all possible places. Transition gates can then be used for selection. Although the size of the net grows because of this the reachability graph will not grow because the net transitions with dynamical target are all mutually exclusive. The static structure of the Pr/T-net causes some problems with synchronic services. If a service has no TNSDL definition EMMA will give a random value for its return value. Because the Pr/T-net is static the value will be the same whenever the service is called. To fix this a place for the random values should be created. When a return value is needed, it could be taken from this place which will contain several different values. These values could also be given by the user.

The PROD analyzer offers in addition to the on-the-fly method two methods for searching the reachability graph. These methods use depth-first or breadth-first search. In the testing phase it was noticed that these two methods are not efficient enough. They will go through every node of the graph which makes them unacceptably slow for very large graphs. The user options for EMMA were designed to reduce this problem but clearly there is a need for new algorithms for searching paths in the reachability graph.

The obvious problem is always the size of modern industrial SDL programs. It is often impossible to generate a reachability graph for the whole program, but if the user can isolate the problematic part of the program and knows how to use EMMA efficiently this analyzer can be used for example for studying the interface between two processes to find properties which otherwise could not be found.

5.2 Problems with Variables and Expressions

As mentioned complex expressions caused a lot of work and an expression tree concept had to be introduced. This means that variables are divided into the smallest possible pieces so that the leaves of the tree are annotated with the value of one atomic element of the variable. The structure of the tree is determined by the type of the variable which can be found by looking at the TESTP file. For example an array type variable then consists of several tokens because every one of the values in the array generates one token.

The versatility of TNSDL allows for example all kinds of assignment statements. To deal with them a lot of code was needed because the net generator had to decide which tokens of variable with a complex type had to be moved in each case. Still there are some cases of assignments which are a problem for the generator. For example if there is a binary expression `a = b*4 + b*5` where `b` refers to some field in the struct type variable then the generator has a problem. This is because we need to take a token (or several tokens) representing the value of `b` out of the variable place so that it could be multiplied by 4. But after this the next part `b*5` also needs these same tokens in the same TNSDL transition but there are no tokens at the place `b` anymore! Of course it is easy to analyze this expression and to notice that the same tokens are needed but the systems are dynamical. For example let there exist a binary expression `a = c[d*4-e] + c[5]` where `c` is an array type variable representing an array with 10 elements. Now, if `c` equals 2 and d equals 3 we have the same problem as in the previous example. When the net is being constructed the values of the `d` and `e` are not known. The analysis can not be done at compile time and it is necessary to add a complex `comp` block which will handle all situations and value combinations.

The PROD analyzer usually delimits the integer values in the tokens to less than approximately 2000. This limitation can be avoided by using a command line parameter in PROD which will prevent PROD from unfolding the net but as was noticed earlier it is often impossible to do the unfolding anyway. The other way to handle this limitation is to change the SDL program so that there are no big constants in the code because usually this does not affect the analysis.

The number of dimensions of arrays is limited as well as the number of existing processes. This is entirely a computational problem. Large multidimensional arrays would have hundreds or thousands of leaf values so there would be hundreds or thousands of tokens at the array variable place which would cause the analysis to work very slowly. And if there are hundreds of processes working simultaneously in the system the reachability graph would be far too big for any computer to handle no matter what optimization techniques were used.

5.3 Problems with Queues

The SDL standard requires the length of the queue to be infinite. In the real world this can of course not be implemented. In EMMA the length of the queue is usually only three or four. Longer queues tend to exapand the reachability graph very fast. Fortunately the queue length three is sufficient for complete analysis in many cases. It is, however, possible that the system model deadlocks if the queue is filled up. Then the default length of the queue in EMMA has to be changed.

Internal inputs, INPUT, OUTPUT, SAVE and SAVELATEST statements were also a bit problematic because the handling of the message parameters is difficult (see the previous section). Problems were solved by coding the parameters to a different place and by creating multiple transitions. Fortunately the gates for the net transitions could be constructed so that only one transition is enabled at a time so the reachability graph is not affected much. It was discovered later that the NEXTSTATE statement can not remove parameters of the INPUT statement from the parameter place if there has been a JOIN statement which jumps to another SDL transition. This is not a big problem because these parameters are indexed with a unique number so the transition may leave the parameters in the `parameter` place.

5.4 Problems with the Execution Resource Handling

The execution resource handling mechanism works quite well in reducing the size of the reachability graph but it is not perfect. It is possible that EMMA will not show all states of the system because the mechanism may prevent some behaviour. For example if a hand process accesses the data of a master process or a program makes some operating system call, then the mechanism may prevent some behaviour which would be possible in the real system and would be seen without locking. This is not a big problem because the benefits of the execution resource handling are greater than the drawbacks. However, when some error in a program is searched with EMMA, the user has to know that it is possible that in some situation EMMA may not show all states of the system.

5.5 Problems with the Time Concept

There is no time concept in the Pr/T-nets and timers can not be modelled very well. A simple solution is to allow the timer to expire in all possible phases of the program. In this case, however, the timers will create a lot of useless states to the reachability graph. An improvement was obtained by introducing a *timer window* which means that the user can control the earliest and latest point in the program where the timer is allowed expire. The problem with this approach is of course that if a timer related error in the program occurs outside the timer window, it is not found. If the error is not found the user can, however, move the timer window. This solution is quite sophisticated because it bypasses the fact that there is no time in the Pr/T-nets and still makes possible the complete analysis of a timer within the timer window.

It is possible to give a maximal time of existence for a process in the CREATE statement. After this maximal time a TIME_QUOTA signal would have to be sent. Various solutions for implementing this in EMMA has been proposed but right now EMMA just bypasses this parameter. At the time being this is not an important issue.

5.6 Problems with the Programming Environment

The C++ language was used in the EMMA project. This caused two kinds of problems. The first problem was the memory allocation errors which were caused by programming errors. The problem is that it is relatively hard to find this kind of errors but there exist tools for finding this kind of errors. A program named *Checker* was found to be very useful.

The second problem was the class structure. In an object oriented language a class structure of the program is designed first. In EMMA the net model generator was a completely new kind of a program and there was no earlier experience to rely on. Thus in some cases it was noticed that a different class structure would have been better. For example some linked lists should have been linked in both directions in order to make the traversing of the list easier. It was also noticed, that there was actually no need for separate index, field and binding variable classes because a variable type can be of arbitrary complexity and the type of the variable has to be checked by a recursive function anyway. Also some modifications to the class hierarchy of the program would have prevented the use of some global variables.

6 Using EMMA

This section presents some information gathered from various tests made with EMMA and contains a simple example of the analysis of a TNSDL program. The tests show that the generation of the Pr/T-net with EMMA is very fast and the time for that phase is negligible compared with the time consumed in the reachability graph generation of that net (with PROD). Also the tests show clearly that with EMMA whole systems can seldom be analyzed. Instead they must be divided into smaller parts.

6.1 The Efficiency of EMMA

The `cell` example models one mobile phone communications protocol used in mobile phone networks. It took only a few seconds to produce a Pr/T-net out of this example which contains 430 lines of TNSDL code, but the building of the executable from that net took half an hour. After that the first version of the example created a very large reachability graph (the generation of the graph was stopped when the size of the graph was over 400MB). By analyzing this incomplete graph various inefficiencies were found. After some optimizations and the reduction of the number of processes from nine to three the building of the executable with PROD still took half an hour, but the creation of the reachability graph took only a couple of minutes and the complete cycle for the creation of the graph took less than 40 minutes.

The computer used in this test was a 90MHz Intel Pentium machine with 64MB of memory. The produced Pr/T-net contained 516 transitions and 284 places. The resulting reachability graph contained approximately 3400 nodes and its size was less than 6MB which is relatively small. It is notable that the size of the Pr/T-net was almost the same after the optimization but the change in the size of the reachability graph was dramatic. Still all essential behaviours of the program could be verified after optimization. The tests verified two things. The first was that the *optimization* of the program is essential for the success of the analysis. The second was that in general the analysis of thousands of lines of TNSDL programs has to be divided into smaller pieces for EMMA and the most important complexity factor is usually the *number of concurrent processes*. The size of the reachability graph which can be analyzed with a Pentium class computer is about 50MB. This kind of graph would contain about 30000 nodes and a complex query made with the `probe` program would take a couple of hours. Of course it is possible to analyze also bigger graphs if a partial analysis is sufficient.

6.2 A Small Example

In appendix A there is a very simple example of a TNSDL program which contains one process. The process contains one procedure which counts a sum of integers from 1 to the given parameter. The result is returned via the IN/OUT parameter `result`. The block structure of the TNSDL program is presented first and after all block descriptions the actual TNSDL description of the only process can be seen.

Because the program is simple and there are no concurrent processes the analysis results are small enough to be shown in the appendix. The program starts from the `START` transition which calls the procedure. The procedure counts the required sum by using the WHILE statement. After the procedure has ended the return value is moved from the `res` variable in the process to the variable `r`. Then the `START` transition ends and the next state is `serverp_idle` which starts to wait for the signal `never_comes`. No process will ever output this signal so the system goes into a deadlock state. Because of the very simple program there is only one path of events ending in the deadlock state. Now EMMA is used to verify this predicted behaviour.

First the program is input to the TNSDL translator (see Figure 2) which produces the required tables. Then EMMA is started. It generates the Pr/T-net in a few seconds. The net consists of 29 transitions and 38 places. Then PROD is used to generate the executable which is in its turn executed. The reachability graph has 22 nodes and one terminal node. Now all paths leading to the terminal node are searched. With the query language of PROD the following query is done in the first node of the reachability graph:

```
0#query volatile verbose bpath(true,true) %0
```

The result of the query is put into a separate text file. Only one path is found which is the expected one. In appendix A.2 there is the first node of the output from `probe`. It shows the initial marking of the net and the name of the transition which is fired first in the path. As can be seen, the output contains a lot of data even though it is only the data of the first node in the path and it is a bit hard to see whether the initial state of the system matches the state defined in the TNSDL program. When the large amount of data coming from `probe` as a result of the previous query is fed through the `filter` program it produces the output seen in the appendix A.3.

If the output of `filter` is examined it is seen that the first statement executed in the path is the CALL statement as expected. Notice how easily all statements executed in the system along the path can be read from the output. The user of EMMA quickly sees what has happened in the system and notes that the program has completed (the last transition which has fired is `NEXTSTATE serverp_idle`).

In order to show the possibilities of the `filter` program an additional output from `filter` has been added to the end of the listing in the Section A.3. The output contains the NEXTSTATE transition and the state of the system after the transition. The output is created by changing the options of the program to produce more information. `Filter` shows all values of the variables in the process in the last node of the reachability graph. Note the result of the procedure call in the `res` variable and the same value in the `r` variable. Also the values of the special variables are shown as well the state of the queue (which is empty) and the `pid` of the process.

7 The Future of EMMA

Despite the problems with EMMA the project succeeded in all of its goals so well that a continuation project has been started at Digital Systems Laboratory and a pilot project will be done at Nokia. The analysis methods will be studied and PROD will probably be completely rewritten. It will be extended with new features concerning the generation and search of the reachability graph. The idea is to give the user more control of how PROD will generate the graph. The user will in the future be able to tell PROD what is the interesting part of the graph (which can be generated first). Also the path searching algorithms will be developed and the user will get new possibilities to interact with the search. Reduction of the size of an individual state in PROD will also be studied. By optimizing the storage of the state information in the computer memory this may be possible.

The structure of the net model may also be developed. The goal of the optimization is to reduce the number of transitions and places in the net. However, it must be realized that the simple folding (e.g. using only one `queue` place for all processes) of the net does reduce the number of transitions and places in the net but it *does not* necessarily reduce the size of the reachability graph at all! Folding only complicates the gate expressions in the transitions. In some cases the net created from the individual SDL statements, however, can be optimized so that the size of the reachability graph decreases. Another way to optimize the net is to develop the analysis made in the initialization phase, it may even be necessary to add a special analysis pass. Some additional features could be added to EMMA. For example an additional place which would tell the number of running processes would be useful in the analysis.

There are still some programming errors in EMMA and some TNSDL structures may not work. An extensive testing of EMMA with large TNSDL systems is necessary to find all errors. The EMMA tool should eventually be integrated with the existing testing environment at Nokia. The purpose is to create a graphical user interface for EMMA to be used in a Windows NT environment. The user interface and other improvements should make utilizing EMMA so easy that eventually there will be no need to be verifying specialist in order to get useful results. Then an ordinary TNSDL programmer will be able to use EMMA as a kind of a debugger and prototype tester.

More information about EMMA can be found in various papers including [HMJed], [Mal97], [Hus96a] and [Hus96b]. In addition to two reports and other papers several new papers on this subject will be produced by the new projects in the near future.

8 Conclusions

In the EMMA project an analyzer for TNSDL programs using Pr/T-nets was created. The usefulness of Predicate/Transition nets was put to a hard test with a large-scale industrial problem. The project was notable also because the main goal of the project was to be able to analyze a *complete* programming language (TNSDL) extensively used in production. The project was a success and it proved that Predicate/Transition nets can be used in this kind of analysis. Moreover the project financier was satisfied with the project results.

EMMA can be used both for analyzing processes and for supporting the development of new process families. In order to use EMMA efficiently the user has to be specialized in verifying SDL programs with Pr/T-nets. This so called verification engineer has to know the limitations of EMMA and the possible optimization techniques using both the user options of EMMA and optimizing the SDL program. With EMMA it is already in the design phase possible to find errors which are difficult to detect by other analysis methods. EMMA can also be helpful in finding errors related to concurrent behaviour of existing distributed processes.

Although large SDL systems can not be analyzed as a whole, they can be analyzed in smaller parts by studying the interaction between individual processes inside the system. The ability of EMMA to verify certain properties and to find certain paths will help in creating SDL programs which contain less errors as well as maintaining existing programs created by SDL in the future. The need for this kind of analyzers will grow all the time and the new projects started in Helsinki University of Technology and the pilot project in Nokia are examples of the importance of the research and development in this area.

References

[Bra95] Rolv Braek. SDL Basics. In Øystein Haugen, editor, *SDL'95 Forum Tutorial*, pp. 11–29, 1995.

[Gen87] H.J. Genrich. Predicate / Transition Nets. W. Brauer, W. Reisig and G.Rozenberg(eds.): Petri Nets: Central Models and Their Properties. Advances in Petri Nets 1986, part I. Lecture Notes in Computer Science vol 254. Berlin, Springer-Verlag. pp. 207–247, 1987.

[HMJed] Nisse Husberg, Markus Malmqvist, and Tero Jyrinki. The Emma Analyzer. Technical report, Digital Systems Laboratory, Helsinki University of Technology, ESPOO, Finland, (to be published).

[Hog95] D. Hogrefe. Validation of SDL Systems. In Øystein Haugen, editor, *SDL'95 Forum Tutorial*, pp. 95–105, 1995.

[HOPV95] Nisse Husberg, Leo Ojala, Olli-Matti Penttinen, and Antti Vainonen. Report of a preliminary investigation: EMMA - an Extendible Multi Method Analyzer. Unpublished, 1995.

[Hus96a] Nisse Husberg. Formal Models for Real Programs. presented at the 8th Nordic Workshop on Programming Theory, December 1996.

[Hus96b] Nisse Husberg. SDL Modelling with High Level Petri Nets. In *Workshop on Concurrency, Specification & Programming*, volume Informatik-Bericht Nr. 69. Humboldt-Universität zu Berlin, September 1996.

[Ket89] Esa Kettunen. Petri Net Based Reachability Analysis of SDL Descriptions. Master's thesis, Digital Systems Laboratory, Helsinki University of Technology, (in Finnish), 1989.

[Mal97] M. Malmqvist. Methodology of Dynamical Analysis of SDL Programs with Predicate/Transition Nets. Technical Report ??, Digital Systems Laboratory, Helsinki University of Technology, ESPOO, Finland, December 1997.

[Rau90] M. Rauhamaa. A Comparative Study of Methods for Efficient Reachability Analysis. Research Report 14, Digital Systems Laboratory, Helsinki University of Technology, ESPOO, Finland, September 1990.

[SDL88] CCITT Specification and Description Language (SDL). Technical Report Z.100 (1988), ITU-T, 1988.

[VHHP95] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. PROD Reference Manual. Technical Report 13, Digital Systems Laboratory, Helsinki University of Technology, ESPOO, Finland, August 1995.

A An Example of Using EMMA

A.1 TNSDL program

```
SYSTEM test;
/*procedure, in, in/out, while*/
ENDSYSTEM test;

IMPLEMENTATION testing OF SYSTEM test;
  SYSTEMBLOCK wl;
ENDIMPLEMENTATION testing;

IMPLEMENTATION wl_this OF SYSTEMBLOCK wl;
  SIGNAL
    never_comes = 0x200e;

  SERVICEBLOCK server;
ENDIMPLEMENTATION wl_this;

SERVICEBLOCK server;

ENDSERVICEBLOCK server;

  IMPLEMENTATION server OF SERVICEBLOCK server;
    PROGRAMBLOCK serverp;
  ENDIMPLEMENTATION server;

    PROGRAMBLOCK serverp;

    ENDPROGRAMBLOCK serverp;

      IMPLEMENTATION serverp OF PROGRAMBLOCK serverp;
        MASTER PROCESS serverp;
      ENDIMPLEMENTATION serverp;



MASTER PROCESS serverp;

  DCL res dword,
      r dword;

  PROCEDURE sum_of_integers;
```

```
  /* counts sum of integers from 1 to i */
    FPAR IN i dword,
         IN/OUT result dword;

    DCL counter dword;

    START;
    TASK result := 0;
    TASK counter := 1;
    WHILE counter <= i;
      TASK result := result + counter,
           counter := dword (counter + 1);
    ENDWHILE;
    RETURN;
  ENDPROCEDURE sum_of_integers;

  START;
    CALL sum_of_integers(3, res);
    TASK r := res;
    NEXTSTATE serverp_idle;

  STATE serverp_idle;
    INPUT never_comes;
      NEXTSTATE -;
  ENDSTATE serverp_idle;

ENDPROCESS serverp;
```

A.2 The First Node of the Probe Output

```
PATH
Node 0
  AvailablePids: <.1.>
  Process$1$V$Offspring: <.0,0.>
  Process$109$SaveLatest: <.0,0.>
  Process$109$AutomataStates: <.0,127,0.>
  Process$109$Queue: <.0,0,0,0,0.> + <.0,0,1,0,0.> +
<.0,0,2,0,0.> + <.0,0,3,0,0.>
  Process$109$Cursor: <.0,0.>
  Process$109$Free: <.0,0.>
  Process$109$ImpSaveQueue: <.0,0,0,0,0.> + <.0,0,1,0,0.> +
<.0,0,2,0,0.> + <.0,0,3,0,0.>
```

```
  Process$109$ImpSaveFree: <.0,0.>
  Process$109$V$Self: <.0,0.>
  Process$109$V$Sender: <.0,0.>
  Automaton$127TransBegin0: <.0,0.>
  V$120: <.0,0.>
  V$121: <.0,0.>
  V$126: <.0,0.>
  GlobalOutputId: <.0.>
Arrow 0: transition Automaton$127$Trans0$1$Call$122$,
         precedence class 0
  pid = 0
  state = 0
  n = 0
  FromValue3 = 0
  NewValue1 = 3
  NewValue2 = 0
```

A.3 Ouput from the Filter

```
############################################################
 PATH
############################################################

Master process: serverp   (pid = 0)
CALL sum_of_integers( 3 , res);

Procedure     : sum_of_integers   (pid = 0)
TASK result :=  0

Procedure     : sum_of_integers   (pid = 0)
TASK counter :=  1

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Evaluating condition expr.)

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Condition is true.)

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i
TASK result := result + counter
```

```
Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i
TASK counter := dword (conversion) counter +  1

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Evaluating condition expr.)

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Condition is true.)

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i
TASK result := result + counter

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i
TASK counter := dword (conversion) counter +  1

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Evaluating condition expr.)

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Condition is true.)

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i
TASK result := result + counter

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i
TASK counter := dword (conversion) counter +  1

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Evaluating condition expr.)

Procedure     : sum_of_integers   (pid = 0)
WHILE counter <= i   (Condition is false.)

Procedure     : sum_of_integers   (Returning from procedure )

Master process: serverp   (Returning from procedure )

Master process: serverp   (pid = 0)
```

```
TASK r := res

Master process: serverp   (pid = 0)
NEXTSTATE serverp_idle;


/***************************************
 Filter output with maximal verbosity
***************************************/
-----------------------------------------------------------

TRANSITION:     (Arrow 0)
Master process: serverp   (pid = 0)
Automaton     : GOG  (in START transition)
Element Nro   : 3
 -----
NEXTSTATE serverp_idle;

-----------------------------------------------------------

NODE 21:
 ---
Available Pids: 1
 ---
 ---
Process serverp:  (pid = 0)
State: serverp_idle
Queue : - - - -
Cursor: 0
Free  : 0
*Specials:
Offspring = 0     (at top level)
Self = 0
Sender = 0
*Variables:
res = 6
r = 6
counter = 4     (in the procedure)
```