HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **16**; April 1997
ISSN 0783–540X
ISBN 951–22–4007–6

METHODOLOGY OF DYNAMICAL ANALYSIS OF SDL PROGRAMS USING PREDICATE/TRANSITION NETS

Markus Malmqvist

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
No. **16**; April 1997

ISSN 0783–540X
ISBN 951–22–4007–6

Methodology of Dynamical Analysis of SDL Programs Using Predicate/Transition Nets

Markus Malmqvist

Abstract: The rapid increase of parallel and distributed systems has brought new problems related to the correctness of the systems. In this work the automatic verification tool EMMA is presented, which uses Predicate/Transition nets to model TNSDL programs. The verification is based on reachability analysis with the PROD analyzer. Several methods to avoid state space explosion are discussed, e.g. model optimization, advanced state space generation algorithms and direct TNSDL program manipulation. The emphasis in this work will be on model optimizations for industrial TNSDL programs, but non-exhaustive methods are also considered. Key principles used in the modeling of TNSDL programs are also explained. In the EMMA project the *complete* TNSDL language has been modeled. The difference between the model and the implementation is small, because both are generated automatically from the same TNSDL specification. The results of the reachability analysis are translated back to TNSDL making the tool easier to use for specialists not acquainted with net theory.

Keywords: reachability analysis, SDL, formal methods, Predicate/Transition nets, verification, PROD

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

The rapid increase of parallel and distributed systems, for example in telecommunications, has brought new problems related to the correct behaviour of the systems. Unfortunately, concurrent systems often have unwanted execution paths which can have serious consequences. These paths can be extremely hard to detect in the systems and therefore new methods based on formal models are needed to ensure the correctness. The complexity of concurrent systems also makes automatic verification tools necessary.

One way to do automatic verification is to perform an exhaustive search of the state space of the system model to find the unwanted paths and states. State space generation must be a well-defined procedure for the formalism used to construct the system model. In the EMMA project systems written in a dialect of SDL called TNSDL are modeled by Predicate/Transition nets.

The space and time requirements of state space generation are very high, because there often is a very large number of states in the state space of a concurrent system. Therefore it is necessary to reduce the number of generated states. To this end, more sophisticated state space generation algorithms can be developed. With these algorithms only part of the state space is inspected, but verification of properties is still possible. One can also optimize the system model, which can make the state space of the model smaller mainly by restricting the concurrency. Direct manipulation of TNSDL programs before modeling can also greatly reduce the state space.

This work is mainly about modeling principles in EMMA and methods directed towards use in the analysis of real industrial systems. Some can be implemented in the net model generator of EMMA while others must be implemented in the reachability analyzer. A few of the methods already exist in EMMA, but a great deal will be implemented in the new EMMA project starting at the Digital Systems Laboratory in 1997.

Predicate/Transition nets are shortly presented in Section 2. In Section 3 the SDL language, the differences between SDL and TNSDL and the basic concepts of verification of SDL programs are discussed. The structure, basic ideas and some essential modeling principles of EMMA are explained in Section 4. More details about the modeling of TNSDL statements with Pr/T nets can be found in [18]. In Section 5 the state space explosion problem is discussed. The emphasis is on the optimization of the Pr/T net models of TNSDL programs. The ideas and motivation behind the already implemented optimizations are explained and some improvements are also suggested. Finally Section 6 describes some problems and possible enhancements of EMMA.

2 Petri Nets

Net theory is a systems theory which can be used to describe systems whose structure and behavior can be seen as resulting from the distributedness and combinatorial nature of their states and state changes. There are many net formalisms, some are simple but have relatively limited expressive power while others are very complex and have great expressive power. For example *Condition/Event systems (CE-systems)* [8] and *Place/Transition nets (P/T nets)* are simple and limited while *Predicate/Transition nets (Pr/T nets)* are more complex. In the EMMA project it was necessary to represent many kinds of data in net models. Therefore a more expressive net formalism was needed and Pr/T nets were selected partly also because a Pr/T net analysis tool already was available.

The following presentation of Pr/T nets is based on [8], [21] and [22].

First we define a *net* as a base for *Place/Transition nets (P/T nets)* and *Predicate/Transition nets (Pr/T nets)*. Then *Place/Transition nets (P/T nets)* and *Predicate/Transition nets (Pr/T nets)* are defined. A Pr/T net can be *unfolded* into a P/T net and a P/T net can be *folded* into a Pr/T net.

Definition 1 (net) *A* net *is a triple $N = (S, T; F)$, where S is the set of* places, *T is the set of* transitions *and F is the* flow relation. *The sets satisfy the following properties:*

1. $S \cap T = \emptyset$.
2. $F \subseteq (S \times T) \cup (T \times S)$.

Usually places are drawn as circles or ellipses, transitions as boxes or lines and flow relations as directed arcs.

Definition 2 (Place/Transition net (system)) *A* P/T net *is a 2-tuple $PT = (N, M_0)$, where N is the underlying net $N = (S, T; F)$ and $M_0 : S \to \boldsymbol{N}$ is the* initial marking.

Note that we are here using P/T nets without weights and capacities. Thus in a P/T net all arcs carry only one *token* and at any place there can be any number of tokens.

Predicate/Transition nets allow the modeling of *individuals with dynamic properties and relations*. Symbols that denote individuals are called *constants* and symbols denoting relations are called *predicates*. Another class of symbols represents *variables*. If in a proposition a constant is replaced with a variable, a propositional scheme called a *formula* is acquired. From a formula, many predicates can be derived by *substituting* the variables with individuals.

A dynamic system is modeled in terms of a set of individuals that is structured by functions and relations. Because the *structure* will have dynamic properties in addition to static ones, the notion of relational structure must be made dynamic. A structure is a tuple, $\mathcal{R} = (D; f_1, ..., f_k; R_1, ..., R_n)$, where D is a non-empty set of *individuals* called the *domain* of $\mathcal{R}$, the f_i are *functions* in D and the R_j are *relations* in D. In a *dynamic structure* some relations will be *variable* in the sense that their extensions may vary from state to state.

The static part of the relational structure will remain an ordinary relational structure. It is called the *support* of the dynamic system. The dynamic part will be represented as an annotated net, called *Predicate/Transition net (Pr/T net)*. Variable relations appear as places of the net. The names of the functions and relations of the static part will appear in the annotations.

In Pr/T nets annotations of places are variable predicates. Thus a predicate P is true for some set of individuals and implicitly false for other individuals. The set on individuals for which the predicate is true can change when transitions are fired.

In the following definitions, the concept of *first-order language* is used. A first order language has a set of *operators* and a set of *predicates*. These together form the *vocabulary* of the language. The expressions of a language are either *terms* or *formulae*. Terms are constructed from variables, individuals and operators. Formulae are constructed from terms, predicates, logical connectives and the existence quantifier applied to variables. A formal definition of first-order languages can be found in [8].

Definition 3 (symbolic sum) *Given a first-order language L, for each $n \geq 0$ a class $LC^{(n)}$ of* symbolic sums *of tuples of length n is defined in the following way.*

1. *The constant 0 is in $LC^{(n)}$.*
2. *If $v_1, ..., v_n$ are terms, then the n-tuple $\langle v_1, ..., v_n \rangle$ is in $LC^{(n)}$.*
3. *If $l1, l2$ are in $LC^{(n)}$, then $(l1 + l2)$ is in $LC^{(n)}$.*
4. *If l is in $LC^{(n)}$ and $z \in \mathbf{N}$, then zl is in $LC^{(n)}$.*
5. *No other expression is in $LC^{(n)}$.*

Now the class of *strict* Pr/T nets for a given first-order language L will be defined. Strict means that a tuple can occur in a place only once.

Definition 4 (class of strict Predicate/Transition nets) *L is a first-order language, L_s is the sublanguage using only the predicates Π_s denoting static relations. The class PRT_L of* strict Predicate/Transition nets *for language L consists of marked annotated nets, $MN = (N, A, M_0)$, where N is the underlying net, A is its* annotation *in L, and M_0 is its* initial marking.

1. *N is a net, $N = (S, T; F)$.*
2. *A is the annotation of N, $A = (A_N, A_S, A_T, A_F)$, where*
   - *$A_N = \mathcal{R}$ is a first-order structure for L_s, called the support of MN;*
   - *A_S is a bijection between set of places, S, and the set of variable predicates, Π_s;*
   - *A_T is a mapping of the set of transitions, T, into the set of formulae that use only operators and static predicates. Thus the formulae is in L_s.*
   - *A_F is a mapping of the set of arcs, F, into the set of symbolic sums of tuples of terms of L, LC, such that for an arc $(x, y) \in F$ leading into or out of a place s, $A_F(x, y)$ is in $LC^{(n)}$, where n is the index of the predicate annotating s*
3. *M_0 is a marking of the places. It assigns for each place $s \in S$ a symbolic sum of tuples of constants. If n is the index of the predicate annotating s, then $M_0(s)$ is in $LC^{(n)}$.*

Let α be a *substitution* that assigns values to every variable in an expression e. Then $e : \alpha$ is the corresponding *closed* instance of e.

Definition 5 (transition instance) *Let t be a transition and α be a substitution. If α assigns values to all variables found in $A_F(x, y)|x = t \vee y = t$ and $A_T(t)$, then $t : \alpha$ is called a* transition instance. *Transition instance is* feasible *if $\mathcal{R}(A_T(t) : \alpha) = true$ and the closed arc expressions $A_F(x, y)|x = t \vee y = t$ all represent sets.*

Definition 6 (enabled transition instance) *A feasible transition instance $t : \alpha$ is* enabled *at marking M, denoted by $M[t : \alpha\rangle$, if $\forall s \in S[\mathcal{R}(A_F(s, t) : \alpha) \subseteq \mathcal{R}(M(s)) \wedge \mathcal{R}(A_F(t, s) : \alpha) \cap \mathcal{R}(M(s)) = \emptyset]$*

In other words, a feasible transition instance $t : \alpha$ is enabled, if the needed tuples are available in places $s \in \bullet t$ and there is room for the generated tuples in places $s \in t\bullet$.

An transition instance $t : \alpha$ enabled at marking M can *fire* to marking M', denoted by $M[t : \alpha\rangle M'$, if $\forall s \in S \mathcal{R}(M'(s)) = \mathcal{R}(M(s)) \setminus \mathcal{R}(A_F(s, t) : \alpha) \cup \mathcal{R}(A_F(t, s) : \alpha)$.

Thus when an enabled transition instance $t : \alpha$ is fired at marking M, the resulting marking is M' if the arcs of the transition t change the marking accordingly when the substitution α is applied to the annotations of the arcs of t.

Definition 7 (reachability graph) *A* reachability graph *is a tuple* $G = (V, E)$*, where* V *is the set of vertices and* E *is the set of edges. The initial marking* $M_0 \in V$*. If* $M \in V$ *and* $M[t : \alpha\rangle M'$*, then* $M' \in V$ *and* $\langle M, t : \alpha, M'\rangle \in E$*. There are no other members in* V *and* E*.*

A very simple example follows. Suppose we must solve a problem in modulo 22 arithmetic. We must visit all possible values by performing 22 additions in a suitable sequence. The numbers that can be added are predefined: Minus four can be added three times, minus five can be added five times, minus eight can be added four times, five can be added four times, six can be added three times and eight can be added three times. Evidently each addition must result in a number that has not yet been visited, because there are exactly 22 additions available. We can of course start from anywhere, so let's start from 0.

This is a very simple problem, one possible Pr/T net providing the solution can be seen in Figure 1. Place `NotAdded` holds all the numbers that are still available for addition. Negative numbers are transformed into their positive equivalents in modulo 22 arithmetic. Place `NotVisited` contains all values that are not yet acquired as a result of an addition. Place `Value` holds the current value. Place `Counter` is used to identify those addition sequences that use all 22 numbers. These sequences are solutions to the problem. Because any visited value is removed from the `NotVisited` place, any state that has no such addition available that would produce an unvisited number is a terminal state and the terminal state where all numbers are used for addition is the solution state. The net has only one transition that performs addition. It non-deterministically chooses a number available for addition and updates the `NotVisited`, `Value` and `Counter` places accordingly. The number in the `Counter` place is always incremented by one.

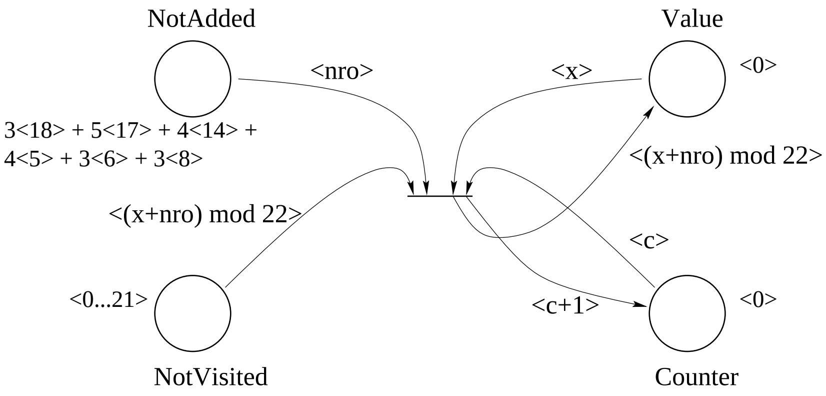

Figure 1: A solution to the modulo 22 arithmetic problem.

Now a formal version of the net in Figure 1 is given. The net is $MN = (N, A, M_0)$, where N is the underlying directed net $N = (S, T; F)$. The set of places is $S = \{s_1, s_2, s_3, s_4\}$. The annotations of places, $A_S(s_1) = \texttt{NotAdded}$, $A_S(s_2) = \texttt{NotVisited}$, $A_S(s_3) = \texttt{Value}$ and $A_S(s_4) = \texttt{Counter}$. The set of transition contains the only transition which will be called t. The annotation of t is, $A_T(t) = \emptyset$. The set of arcs will contain all the arcs, (s_1, t), (s_2, t), (s_3, t), (s_4, t), (t, s_3) and (t, s_4). The annotations of the arcs are, $A_F(s_1, t) = \langle nro \rangle$, $A_F(s_2, t) = \langle (x + nro) \ mod\ 22 \rangle$, $A_F(s_3, t) = \langle x \rangle$, $A_F(s_4, t) = \langle c \rangle$, $A_F(t, s_3) = \langle (x + nro) \ mod\ 22 \rangle$ and $A_F(t, s_4) = \langle c + 1 \rangle$.

3 SDL

The following presentation of basic SDL features is based on [2]. The need for a well-defined language for specification and description of concurrent systems first arouse in switching systems development in the late 1960s - early 1970s. Telephone manufacturers realized that communication in concurrent systems had to be supported better on the language level. This is not surprising, because telephone switching systems are among the most complex real time systems. The methods used to meet these needs turned out to be useful in other similar contexts also.

The created language, SDL, is intended for both implementation-independent specification and description of actual implementations. Hence the name Specification and Description Language – SDL. The main concern was to provide better ways to describe system behavior. In addition, SDL should be intelligible to humans and formal enough to support analysis of behaviors. The first recommendation appearing in 1976 has been updated four times, in years 1984, 1988, 1992 and 1996. The version of year 1984 added architectural composition to the basic sequential behavior approach of the first and second ones. The version of year 1992 made SDL an object oriented language.

For a language to be useful it must be possible to map the concepts of the application domain to the concepts of the language. In the case of SDL, the problem domain is reactive behavior. The main concepts of the reactive behavior domain are *concurrency*, *communication between external users and between system parts* and *sequential behavior of individual system parts*. Communication is *asynchronous signaling*, not direct manipulation. In SDL concurrency in a system is modeled by independent processes and concurrency in the environment is assumed to behave in a similar manner. The processes communicate via signal routes and channels. The signals carry data, but can't directly manipulate the state of a process. The sequential behavior of these SDL processes is well-defined, because they are *extended finite deterministic automata* with timers and the theory of automata is well established. The concepts are not bound to any particular technology which is essential for the independence of the implementation. The fact that internal and external interfaces of a system are assumed to be similar is important. It makes it possible to freely determine the boundary of a system, and removes all need for built-in assumptions about physical co-localization of processes.

SDL has two different syntactic forms, one graphical and the other textual. However, text is used also in the graphical form where it is more appropriate. The graphical form is more intuitive while the textual form doesn't require any expensive tools and is therefore implementation-independent and used as an exchange format between tools.

3.1 Finite Automata and SDL Processes

A *finite deterministic automaton* is an automaton that reads input symbols. Reading a symbol causes a *state transition* in the automaton. When an automaton has read all input symbols, it can either accept or reject the input symbol string. The following definition is taken from [19].

Definition 8 (finite deterministic automaton) *A* finite deterministic automaton *is a quintuple* $M = (K, \Sigma, \delta, s, F)$*, where*
K *is a finite set of* states*,*
Σ *is an* alphabet*,*
$s \in K$ *is the* initial state*,*
$F \subseteq K$ *is the set of* final states*,*

and δ*, the* transition function*, is a function from* $K \times \Sigma$ *to* K*.*

Finite means that an automaton has a finite number of states. *Deterministic* means that in every state there is at exactly one state transition defined for each symbol in the alphabet. The automaton accepts an input symbol string if is in a final state after reading the string.

SDL processes can be described as *extended finite deterministic automata.* From this point on, these are simply called *automata.* The automaton allows behaviors to be described completely and unambiguously in terms of external stimuli and responses. However, it differs from the formal definition of finite deterministic automata. It may be that in a state there is no state transition defined for some symbols of the alphabet. If this kind of input symbol is received when the automaton is in the state, the symbol is simply *discarded.* In SDL, the symbols of the alphabet are *messages.* In addition, in SDL there are no formal final SDL states and thus the concept of accepting an input symbol string is undefined at the language level. Each SDL process consists of four main parts: the input port, the automaton, timers and variables (Figure 2).

An *SDL state* accepts a number of messages and it defines an SDL transition for every message it accepts. The automaton can consume messages only when

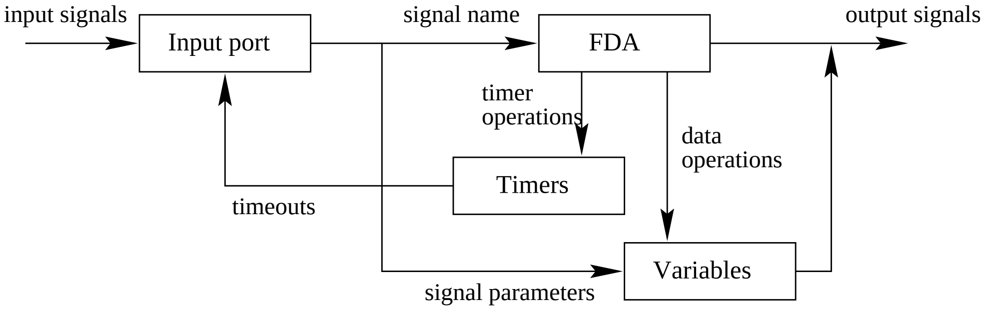

Figure 2: The components of an SDL process [2].

it is in an SDL state, not while it is executing an SDL transition. An *SDL transition* is a sequence of *SDL statements* starting with an INPUT statement which defines the message required for the execution of the SDL transition. In an SDL transition messages can be output with the OUTPUT statement, the values of variables can be manipulated and timer operations can be performed. All execution paths through an SDL transition are terminated with a *terminal statement*, NEXTSTATE, RETURN or STOP. A NEXTSTATE statement causes the automaton to move into the SDL state specified in the statement. A RETURN statement causes the automaton to return from a subprogram. In SDL, subprograms are called procedures. A STOP statement terminates the process.

The input port contains a *queue* of incoming messages and can always accept new messages. While the length of the queue is unbounded according to the standard, in practice it is of course finite due to finite memory resources and therefore queue overflows are an issue in the verification of SDL programs. An incoming message is held in the queue until the automaton is ready to consume it. The messages are normally consumed according to the first-in-first-out principle, but the order can be modified by using the SAVE statement, which delays the consumption of the saved message to the next state of the automaton. If the queue is empty, the automaton will remain waiting for a message to appear. For each consumed message, the automaton executes the corresponding SDL transition of the current SDL state. If the consumed message has no SDL transition in the current SDL state, it is just discarded. This is called *implicit consumption* and differs from the formal definition of finite deterministic automata, as does the SAVE statement. Communication in SDL is *asynchronous*, meaning that a message can be sent at any time, it is not relevant if the receiver is ready to receive a message. In synchronous communication a message can be sent only when the receiver is ready to receive it.

Messages are discrete and they carry their name, the process identifier of the sender and possibly parameters. A parameter is an instance of a certain data type defined either in the specification of the SDL language or in the SDL program. All messages that have the same name will carry the same set of parameters meaning that they have the same number of instances of all data types.

All processes have some predefined variables of the process identifier (`pid`) type. The SELF variable contains the `pid` of the process itself. The OFFSPRING variable contains the `pid` of the process most recently created by this process. The PARENT variable contains the `pid` of the process that created this process. The SENDER variable contains the `pid` of the process that sent the message which triggered the execution of the current SDL transition.

3.2 Other Basic Concepts of SDL

In SDL, the *hierarchy* of a system is composed from *processes*, *procedures* and *blocks*. Process behaviors can be defined using sub-behaviors defined in procedures. Blocks are composed from processes and other blocks. Finally a *system* is composed from blocks. Blocks are a useful aid when trying to write modular SDL programs, but they have no meaning for EMMA. In EMMA, the process is the largest unit of interest.

Some of the processes are created implicitly at program startup. These processes are called *master processes*. Exactly one instance of each master process always exists. All other processes are *hand processes* and instances of them are created dynamically with the CREATE statement which also determines the actual values of parameters needed in the created process. The number of process instances of a certain hand process can be limited. A hand process instance can terminate itself with the STOP statement.

Procedures are pieces of SDL programs that can be called from the enclosing process or procedure. Procedure declarations can be nested. They can have variables and SDL states of their own. A procedure requires a certain set of formal parameters. A procedure is called with the CALL statement which also determines the actual values of the parameters. The parameters can be transferred either by value or by reference. The RETURN statement gives the control back to the caller. All procedures are defined in some process or inside some procedure which is nested arbitrarily deep in a process.

Timed events can be created with *timers*, which can be declared in the processes. The *timeout message* sent by a timer will be received by the process which activated the timer. The structure of a message sent by a timer is identical to a message sent by a process. The name of a message generated by a timer is the same as the name of the timer. When a timer is activated, the time to the emission of the message and the parameters of the message are given. The parameters of the message implicitly define an *instance* of the timer. After the activation of a timer instance the time to the emission of the message starts to elapse. At any time, an arbitrary number of instances of the same timer can be active. When the time period associated with a timer instance runs out, the instance sends the message defined when the instance was activated to the process that activated it. In short, a timer can be used to send a timeout message after a specified amount of time to the same process that activated the timer. A timer is activated with the SET statement, and can be made inactive with the RESET statement. If a timer instance which already has sent the message is canceled with the RESET statement, the message is removed from the queue. The possible removal of a timer message is the only difference between timer messages and other messages. If a timer instance with message parameters identical to the message parameters of an active timer instance of the same timer is activated, the instance activated earlier is reset automatically and replaced with the new one.

Normally one process contains one automaton. An automaton has one special SDL *start transition* which can be executed after a process instance containing the automaton is created. The execution of an SDL start transition does not require a message and thus an SDL start transition contains no INPUT statement. The SDL start transition of an automaton is always executed before any other SDL transition of the automaton.

In an SDL process there can be several automata which share the same message queue. Because only one automaton can receive a particular message and the choice of the receiving automaton must be deterministic, the message sets of the automata have to be *disjoint*. In other words, the receiving automaton is based directly on the message name. Processes containing more than one automaton have a separate *internal message queue* for messages sent from one automaton of the process to another. Messages in an internal queue are always consumed before messages in the normal queue. The use of several automata in a process is appropriate, when a process contains some functions which must be executed independently, but still are closely related. Using this construction to bring down the number of processes in a system can raise efficiency, too. The automata are not fully concurrent. This means that only one SDL transition at a time can be executed in the process. This guarantees error-free handling of the process variables which are shared by the automata. On the other hand, the reduced concurrency also reduces the number of possible executions. This should be remembered, when planning to use many automatons in a process. The automata of an SDL process having more than one automaton can be called *services*.

The parameters carried by a received message will be stored into the local variables of the receiving process instance. The INPUT statement determines into which variables the parameters are stored. The OUTPUT statement naturally determines the parameter values of a message. The target process of a message can be chosen either dynamically or statically. The target is chosen dynamically by providing the process identifier of the receiver in the OUTPUT statement. Static targeting is used by providing a unique receiver for all messages that have the given name. The TASK statement is used for assignment. Branching according to the values of internal variables is implemented with the DECISION statement.

3.3 Differences between SDL and TNSDL

The TeleNokia SDL language (TNSDL) is based on the SDL standard published in 1988, the CCITT SDL-88 [7], but there are some differences. Here the main differences are listed.

In TNSDL, there is a construct called *module*. A module contains procedures which are *shared* among all processes. Module procedures can be implemented with any language. Also the variables declared in the module procedures are shared and thus in fact *global*. In TNSDL, a timer can be declared also in a module procedure, but the timer instances will be local to the procedure.

In addition to implicit consumption messages can also be *saved implicitly* in a procedure. Implicit saving is performed when the control of a process is in an TNSDL state of a procedure ready to read the next message and the message is not handled anywhere in the procedure. If the message is handled by the enclosing process it would be dangerous to implicitly consume it, because then the process never has any chance to react to the message. Therefore the message is implicitly saved and it can be accessed when the control has returned to the process. On the other hand, if the message is handled in some SDL state of the procedure, it is consumed implicitly instead. Implicit saving was present in the SDL version of year 1984.

In TNSDL, there are no channels, signal routes or signal lists. These constructs have been replaced with other mechanisms, such as static and dynamic selection of the target process of a message. Message refinement has not been implemented. There are no continuous messages in TNSDL.

The method for creation of new data types is implemented differently, it greatly resembles creation of data types in the C language. TNSDL transitions can have local variables. The MACRO feature of SDL is omitted. The block structure of SDL has been modified for TNSDL.

Services are called *subautomata* in TNSDL and internal signals of a subautomaton construction can not be saved. Only one subautomaton can have SAVE statements.

TNSDL has a statement called WHILE, which allows looping without JOIN statement. A WHILE loop can be terminated with the OUT statement. If none of the branches of a DECISION statement is true, TNSDL does nothing, but in SDL this is an error.

The amount of data in timers is limited. The structure of the data is not limited, however. A time quota can be given to a hand process instance to restrict the lifetime of it. Time quotas are implemented with implicit expiration timers. If a hand process terminates due to the time quota, the process that created the hand process will receive a special time quota message.

3.4 Verification of SDL Systems

The following text is based on [12]. Although testing of an SDL specification resembles testing of a concurrent program, it is often called *verification* instead of testing. This distinction is made, because during verification an SDL system usually is executed on an *abstract machine*. For example a Pr/T net model of an SDL specification is an abstract machine. The different parts of the verified system can be specified with different tools which complicates the verification process. Usually SDL specifications do not exist in isolation, they require an *environment*. However, currently in EMMA the verified system must be specified fully with SDL. It is also required that the environment is explicitly specified with SDL and integrated into the system. In general, the verification process consists of the following steps [6], [27]:

(1) Identify the scope of verification
(2) Choose implementation options. In EMMA, only SDL is used.
(3) Make the system complete. This is made by specifying the environment of the system with SDL.
(4) Optimize the model in terms of state space.

The third and the fourth step are discussed further in Section 5.

Analysis of a system is often divided into *static* and *dynamic* analysis. Static analysis covers the techniques that can be used *without* execution of the model, while dynamic analysis covers everything else. Because the dialect of SDL used in the EMMA project can be compiled into an executable program, static analysis is readily available. Therefore EMMA has only dynamic analysis capabilities. Some advanced static analyzing techniques might, however, be useful. They could be used to optimize the generation of the Pr/T net model of an SDL program.

Dynamic analysis can be further divided into two classes. The first covers the dynamic *semantics of the system*, whether the functions defined in the model correspond to the system requirements. This is accomplished with model-specific assertions and invariants. Currently EMMA lacks automatic verification of model-specific qualities, and checks involving them must be done manually by inspecting execution paths. The second class includes the *general properties* that can be expressed in terms of assertions common to broad classes of systems. For example freedom of deadlocks and freedom of implicit message consumption are such properties. EMMA allows automatic detection of some general properties.

Distinction should be made between *exhaustive* verification methodologies and *non-exhaustive* ones. The first contains terminating algorithms that perform complete analysis of a system when executed. Non-exhaustive algorithms may not terminate and therefore they can only be used for error searching, not for verification of properties like deadlock freedom. They can nevertheless be quite practical, because the verified system often is too large for exhaustive methods and non-exhaustive methods can still find errors efficiently. The idea is to go through a random or manually defined subset of the system state space.

In the EMMA project, non-exhaustive verification methodology is not yet implemented. *Reachability analysis* of Pr/T nets is used as exhaustive verification methodology. In the future of EMMA project, much work will be put in the creation of non-exhaustive verification methods which can be guided by the user.

3.5 TNSDL and Predicate/Transition nets

TNSDL is a well-defined language which has a strict syntax. This makes it plausible to use a formalism such as Pr/T nets to model TNSDL programs. In order to model TNSDL programs a formalism is of course obligatory. While it might be argued that TNSDL itself is formal enough for analysis it does rely on an underlying compiler and is therefore not a complete, stand-alone formalism. The model must be executable which suggests that a *meta-machine* working on top of the actual computer is needed. The Pr/T net formalism allows the construction of such meta-machine. With Pr/T nets it is also possible to generate the complete state space of the modeled system which is essential in verification. State space generation of Pr/T net models has also been extensively studied which allows fairly efficient state space generation to be implemented. This all makes Pr/T nets a logical choice. Still translation of TNSDL programs into models could be much easier if there was a formalism made exclusively for verification of TNSDL programs. There actually is a formal definition for SDL-88 [23], [24], but it is extremely long and complex. Furthermore, since there are differences between TNSDL and SDL-88 it is not clear which parts of the definition could be applied to TNSDL. The formal Pr/T net model used in EMMA relies completely on internal documentation provided by Nokia Telecommunications.

4 The EMMA Analyzer

The EMMA analyzer consists of four parts, namely the EMMA *net model generator*, the Pr/T net *reachability analysis tool* PROD, the reachability graph *query program* `probe` and the *result interpreter* program `filter` (Figure 3). The EMMA net model generator translates the TNSDL system into a Pr/T net. The reachability analysis tool PROD constructs the reachability graph. The resulting graph can be inspected with the query program `probe` which can search interesting nodes and paths in the graph and display paths found with on-the-fly methods. Finally, with the result interpreter program `filter` the nodes and paths can be translated back to a format that resembles TNSDL with additional information about states of processes and their queues and variables. During the EMMA project the EMMA net model generator and the result interpreter were built. Most of the work was spent constructing the EMMA net model generator. The result interpreter is a much simpler program, but very useful. PROD and `probe` were already available. Some improvements to the usability of `probe` were made by Keijo Heljanko.

The EMMA net model generator does not take the textual TNSDL program as input. Instead, it uses two files produced by the *TNSDL translator* made by Nokia Telecommunications. The TNSDL translator translates TNSDL into C language, and produces two files as side products. These files are TNSDL External Symbol Table Presentation (TESTP), containing the *symbol table* of the TNSDL program, and Flow Control File (FCF), which contains the *control flow* of the TNSDL program. They both are represented as hierarchical lists. The lists are transformed into lists of C++ class instances by the two parsers, `parser1` and `parser2`. The EMMA net model generator uses the symbol table and the control flow to produce a Pr/T net model, which is described with PRODL, the language accepted by PROD. The user can give the EMMA net model generator some options which will adjust the state space of the model or add on-the-fly verification (Section 4.6) aspects to it. Thereafter PROD will produce the reachability graph. The graph is looked into with `probe` and interesting paths can be saved and transformed into a more readable form with the result interpreter program.

Basic ideas behind EMMA can be found in [14]. More information is available in [15] and [16]. Also, there is another report written about EMMA [18]. EMMA will be summarized in the forthcoming technical report [17]. Integration of EMMA net model generator and `filter` is looked into in [20]. Integration was also implemented.

There are some terms used in the following chapters that need clarifying. The word *transition* is used to represent two different concepts. A *net transition* is a transition of the Pr/T net model of the SDL program. An *SDL transition* means a whole transition from an SDL state in an SDL program. It

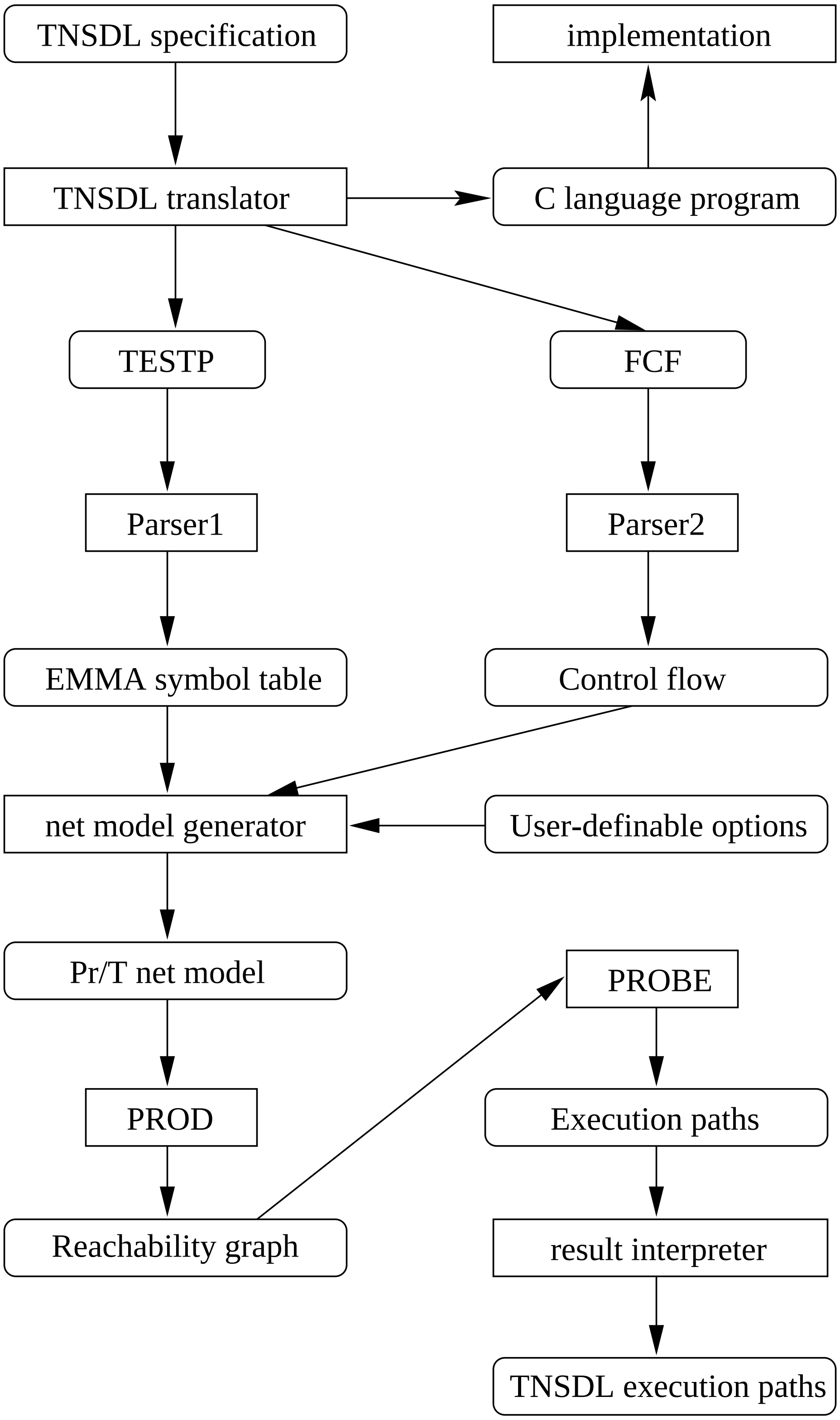

Figure 3: The structure of EMMA.

contains statements that are executed when the message needed to trigger the SDL transition is received. The first statement is always an INPUT statement which receives the message. The names of SDL statements are written with upper case letters when they refer to actual SDL statements. When referring to a net transition corresponding to an SDL statement, the name of the net transition is capitalized. For example "INPUT" refers to an actual SDL statement while "Input" refers to a net transition. Also the word *state* is used in two different ways. The words *SDL state* refer to a state of a process of an SDL program. A SDL state is a point in the SDL program in which messages are read from the queue. Such a state is defined to accept certain messages. When an acceptable message is received, the corresponding SDL transition, which is also defined in the state, is executed. The most important use of the single word *state* refers to a state of the reachability graph of a system model. However, the word also has several other meanings that should become clear from the context.

We will now introduce a modeling technique used frequently in EMMA. A problem with P/T nets is to check if there is a token at a place. Therefore in a Pr/T net it is often useful to store data in the net in a constant form. Constant form means that the tokens containing the data will always have the same arity and they will *always* be available at a certain place. Only the contents of the tokens will change. A collection of such tokens in a place is called a *frame*. The token existence problem can be solved with frames when using high level nets. In net models generated by EMMA frames are used to represent for example variables, timer instances and message queues. In short the use of frames is the natural way to represent a fixed amount of information that always exists.

4.1 Modeling SDL Processes

The basic idea of converting a TNSDL program into a Pr/T net is very simple. Usually one TNSDL statement is converted into one *net transition* and one *control place*. Various control structures require a bit more complex conversion and possibly more than one place. In some cases several *mutually exclusive* net transitions are created to cover all situations that may arise in the execution of an TNSDL statement. Mutual exclusion means, that at most one of the mutually exclusive net transitions can be enabled at any given time. Each process instance has one *control token* moving in the net. The control token is a pair $\langle pid, state \rangle$ where `pid` is the process identifier and `state` refers to the current SDL state of the process. The control token marks the point of execution of the process instance. A control token is always at some control place. A control place represents the point of execution between two TNSDL statements. A net transition corresponding to a TNSDL statement can be enabled if the control token of a process instance is at the control place that precedes the net transition in the control flow. The control token is in the control place if the net transition corresponding to a predecessor of that TNSDL statement has been fired. If an enabled net transition is fired, the control token of the process instance is moved to the control place denoting the next state of the control flow of the process and the side effects originally defined by the corresponding TNSDL statement are carried out (Figure 4). The control token is now available for the successors of this net transition. In the other figures arcs without annotations always carry a control token.

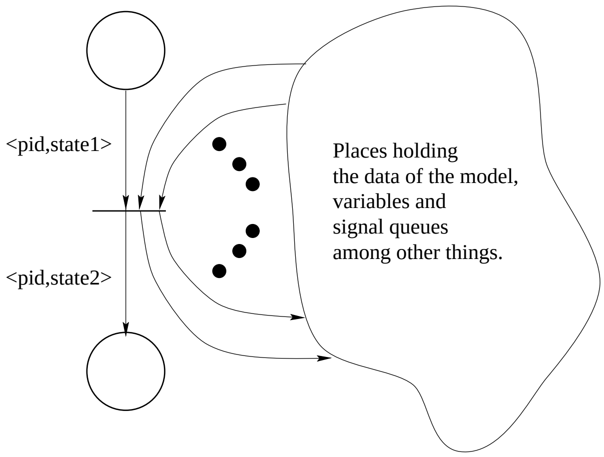

Figure 4: The principle of modeling an SDL statement in EMMA.

4.2 Modeling Message Queues

Currently the length of the queues is defined as a constant in the net model generator. It probably will be user-definable later, maybe separately for each process. An infinite queue is not sensible, because message removal from such a queue can not be implemented with one net transition. A message queue is implemented as three net places, the `queue` place, the `free` place and the `cursor` place. The `queue` place holds the actual contents of the queue. The number of tokens in the place is always the same as the length of the queue. Each token contains the `pid` of the process owning the queue, the process identifier `spid` of the sender process, the number of the queue slot which the token models, the name of the message and the unique identifier of the message (Figure 5). The message names are mapped onto natural numbers starting from one. The unique identifier, an integer, is needed to connect the possible parameters with the message. A new identifier is acquired by incrementing the last identifier by one. In a token modeling an empty queue slot only the `pid` of the process and the number of the slot can differ from zero. A token of an internal queue [18] found in the sub-automaton constructions lacks the `spid`. The sender field is not needed, because an internal queue is only used to buffer messages sent from one subautomaton to another subautomaton in the same process.

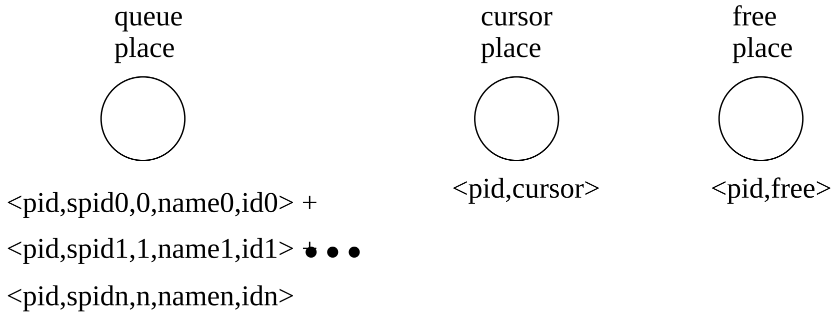

Figure 5: A queue of length `n`.

The `free` place holds the number of the first free queue slot. An incoming message is put into this slot. The `cursor` place holds the number of the current slot. All input, save or consumption operations operate on the message in the current slot.

There is no separate queue for the saved messages like in some implementations, instead the number of the current slot is incremented by a Save transition. Thus the saved message is skipped and will be inspected again in the next SDL state. There is no benefit in using a separate queue for saved messages, because it does not matter whether we have a queue of certain length or two queues which together have the same length, because the saved messages can just as well be at the beginning of the only queue. In addition, it is easier to let the messages be in one queue than to move them back and forth between two queues.

4.2.1 Variables and Message Parameters

Each SDL variable `X` has its own place `V$X` in the net. An SDL variable has a certain *type* and therefore any value of the variable is an instance of that type. The place of a variable contains the current value of the type instance representing the variable. Each *atomic value* of the type instance is represented by one token. An atomic value is an indivisible data object, for example an integer. Hence a two dimensional array with dimensions `m` and `n` has $m \times n$ atomic values. In Figure 6 there is an example of how a type is represented by tokens in a variable place. The type is a structure with two fields. One field is an atomic value `v1` and the other is an array of three atomic values, `v2`, `v3` and `v4`. The type is depicted as a type tree whose leaves represent the atomic values. The marking of the variable place is thus $\langle pid, f1, v1\rangle + \langle pid, f2, i0, v2\rangle + \langle pid, f2, i1, v3\rangle + \langle pid, f2, i2, v4\rangle$.

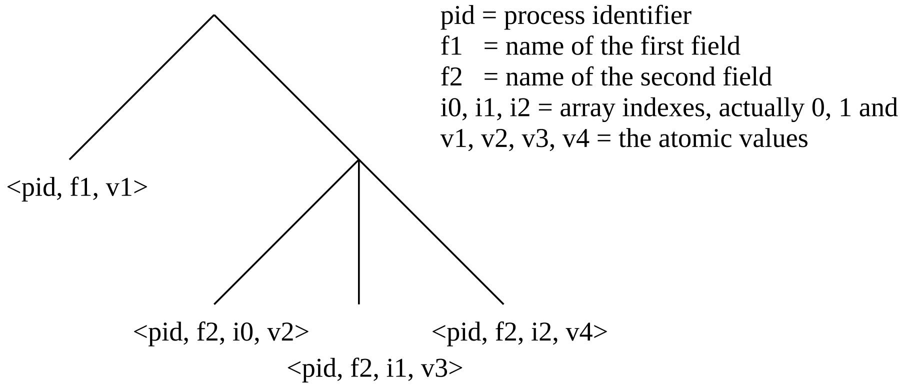

Figure 6: A type represented by tokens.

A message can have any number of parameters and each of them can be of any type. When a message is output, all the parameter values are put into a global `parameter` place. This place is used to store all message parameters of all messages sent by any process. As for variables each atomic value of a parameter is represented by one token in the `parameter` place, but now all the tokens are tuples of equal length. This is due to the fact that the structural information of the type of a message parameter can not be stored in the `parameter` place. The structural information is replaced by information about the order of the atomic values in the type tree. All atomic value tokens of a message have the same unique identifier, the parameter number, the parameter atomic value number and the value itself. Thus the tuples have the form $\langle identifier, parameternumber, atomicvaluenumber, value\rangle$. The atomic value numbers are generated in depth-first order, the first encountered atomic value will have one as the atomic value number. When the message is input by the receiver, the reverse operation is performed by building the type instances from the atomic values. The same depth-first order as before is used, thus the atomic values will represent the correct leaves in the target type tree. It is necessary to abolish the structural information of the source type, because the form of the tuples that will represent the transmitted type instance in the receiving process is not yet known. This arises from the fact that the transmitted type instance can replace a subtree of a type instance in the target process if the subtree represents an instance of the same type as the transmitted type instance.

For example an instance of the type in Figure 6 could have tuples $\langle 1, 2, 1, v1\rangle$, $\langle 1, 2, 2, v2\rangle$, $\langle 1, 2, 3, v3\rangle$ and $\langle 1, 2, 4, v4\rangle$ in the `parameter` place assuming that the unique identifier is 1 and the second parameter of the message has the right type. The order of the parameters is inherited from the message declaration in the SDL source code. The atomic value number ranges from one to four, because the type has four atomic values. Now suppose there is a type that has the type in Figure 6 as a subtype. Let's say this type is a structure with two fields. The first field is an atomic value and the second field is of the type in Figure 6. When the second field in an instance of the type is replaced with a transmitted instance of the type in Figure 6, the actual tokens representing the atomic values of the instance as a part of the receiving type instance will be different from the tokens used to represent them in the transmitting process. The tokens will now also contain the name of the second field of the receiving type. The tuples will be $\langle pid, field2, f1, v1\rangle$, $\langle pid, field2, f2, i0, v2\rangle$, $\langle pid, field2, f2, i1, v3\rangle$ and $\langle pid, field2, f2, i2, v4\rangle$, where `field2` denotes the second field of the receiving type. The token that represents the atomic value of the first field of the receiving type instance is left unchanged.

4.2.2 Reorganizing the Queue

When a message is removed from a queue it must be reorganized to fill the gap left by the removed message (Figure 7). Nothing needs to be done for the parameters of the shifted messages, because they all reside in the same place and they are connected to the corresponding message in the queue through the unique message identifier. Thus shifting the messages of a queue does not change this connection in any way. The shifting is always made from the queue slot whose number is in the `cursor` place. Currently this is done by creating one net transition for each possible `cursor` place value.

It is possible to construct only one net transition which can shift the queue at any slot. This would reduce the number of net transitions in the net and the time needed for creating the graph generator executable. It can be achieved by defining all the cases separately in a single net transition or by using suitable conditional expressions in the net transition with the tokens corresponding to queue slots. The first method requires the use of conditional expressions, one for each possible `cursor` place value. Exactly one of the conditions is true. The code executed when such a condition is true would explicitly rearrange the messages in the queue.

The latter method may be more straightforward but still requires adding some new tuples to the net transition, because a message in the queue has three possible roles while shifting the queue. It can be one of the messages untouched by the shift, it can be the removed message or it can be one of the messages that need to be moved one queue slot towards the head of the queue. The method is depicted in Figure 8. The net transition contains a condition testing the name of the message and for this reason the name of the removed message must be in a certain variable. Thus for each slot there must be a tuple in the net transition that binds the name of the message to the variable. This tuple is

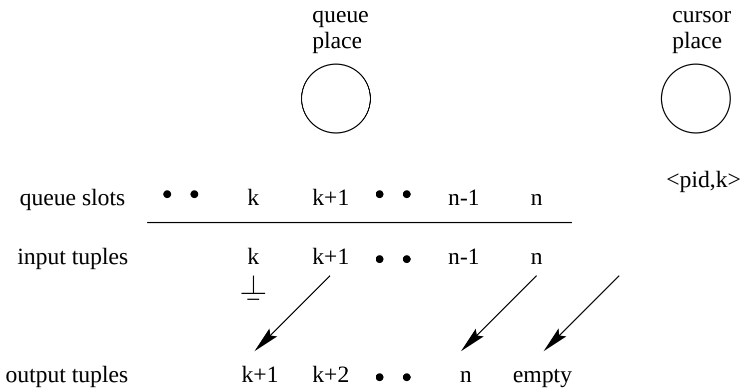

Figure 7: Reorganizing a queue of length `n` at slot `k`.

activated by the corresponding condition if the removed message occupies this slot. In the figure, `k(r)` marks the tuple that is activated when the message in slot `k` is removed. In addition for each slot there must also be a tuple in the net transition that binds the name of the message to another variable, which must be different for each slot, because each slot can contain a message with different name. This tuple is activated when the slot contains one of the messages that need to be shifted. The first queue slot does not need this kind of tuple, because the message occupying it can never be shifted, only removed. In the figure, `k` marks the tuple that is activated when the message in slot `k` is shifted. Thus there are two tuples for each token of a queue. Each token represents a queue slot. Because a token can be removed only once, all these tuples must be conditional. The token corresponding to a conditional tuple is removed from the `queue` place only if the condition of the tuple is true. The first kind of the tuple of a slot has a condition that requires the value of the `cursor` place to be the same as the number of the slot. The second kind of the tuple of a slot has a condition that requires the value of the `cursor` place to be smaller than the number of the slot. If the value is bigger, the message in the slot is one of those that are not moved.

The received message and its parameters are removed from the net in the net transition that terminates the SDL transition which was triggered by the message. In other words they are removed in a Nextstate transition, in a Stop transition or in a Return transition. A message and its parameters can not be removed at the beginning of the SDL transition, because they must be available if the message is forwarded. *Forwarding* means sending the message and its parameters to another process. This would not be possible if the parameters were removed from the net model already in the Input transition that started the current SDL transition. In TNSDL the received message is removed from the queue at the start of the SDL transition. Therefore the length of a queue in the Pr/T net model does not necessarily reflect the

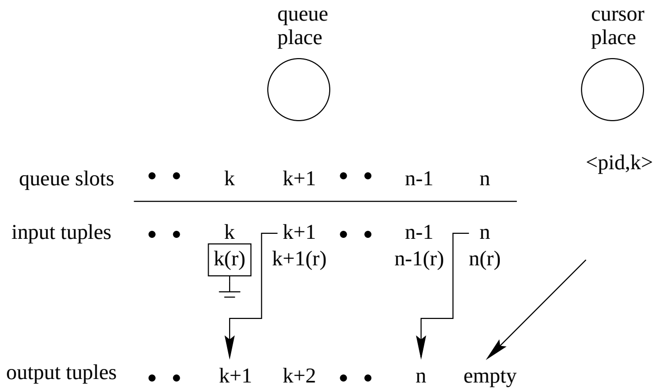

Figure 8: Reorganizing a queue of length `n` at any slot `k` in one net transition.

length of the queue in the real TNSDL environment, it can be longer. This can be significant when procedures containing SDL states are called, because the message read at the start of the calling SDL transition is still in the queue when execution of the procedure starts. This called procedure can call another procedure and so on. In this case the queue would grow accordingly and could get full. This might even cause a deadlock that would not exist in a model in which a received message is removed from the queue when the SDL transition is started. When procedures with states are not called the difference between the actual queue length and the queue length of the model can not be bigger than one, because the deletion of the message that triggered the execution of the SDL transition is only delayed to the end of the SDL transition and no other SDL transition can be executed in the process before the execution of the current one is finished. The difference between queue lengths could be eliminated by moving the consumed message to an additional stack in the Input transition. Then the queue lengths would be equal and the message would still be available. The terminal transition of an SDL transition would remove the message from the stack. Furthermore storing of the consumed message could be done only if the SDL transition contains such an OUTPUT statement which forwards the consumed message.

Currently each SDL state has a set of net transitions that handle implicit consumption. The number of net transitions in each set equals the queue length used in the net model. Each net transition handles the case where the implicitly consumed message is in a certain queue slot. With the idea in Figure 8 the number of net transitions used for implicit consumption could be reduced to one per SDL state. It is possible to use only one net transition per process. However, the net transition must be able to refer to all the SDL states of the process, but the places representing SDL states are empty if no process instance is in them. Thus either the places must always contain at least one token or one place could represent all SDL states of a process. Because all SDL states have different names in the net model, one place could even represent all SDL states of the analyzed system. If all messages queues are also represented with one place only, one net transition could handle implicit consumption in the system.

When a message is implicitly consumed the parameters should also be removed, otherwise they will be left in the net as useless data that even could make the reachability graph bigger. The net transitions handling implicit consumption in a certain SDL state must be able to consume any message for which there is no SDL transition in the SDL state. Therefore it contains conditional parameter tuples for each known message. The conditions of the tuples are names of the known messages. It follows that when a message is implicitly consumed, exactly those parameter tuples that are needed to remove the tokens modeling the parameters of the message are activated. An Input transition that accepts any message is a very similar case. Thus the removal of the parameters of a message received in such an Input transition is dealt with in the same way.

Implicit saving is handled with an additional queue for implicitly saved messages. The accumulation of messages to the queue is started when a procedure is called from a process. It ends when the return statement is executed in the procedure. Procedure calls in the procedure have no effect on implicit saving, the implicitly saved messages are accumulated until the procedure called from the process returns. At that time the messages are copied into the main message queue. The implicitly saved messages will appear between the messages that were saved in the SDL state before the SDL transition that called the procedure was started and the messages that were saved normally in the procedures. The current implementation needs many net transitions, because there can be any number of implicitly saved messages and their required location in the main message queue also varies. This implementation can be improved by using one net transition to shift a queue at any slot as described earlier (Figure 8). In this case, `n` net transitions for a queue of length `n` would still be necessary without additional improvements, because the length of the shift must vary to make room for the implicitly saved messages.

4.3 Modeling Expressions

Modeling of expressions is a bit tricky, because the net model generator must be able to model any arbitrary expression as a net transition. An expression can have any number of normal variables, array references and structure references. Furthermore an SDL statement can have many expressions.

A *net variable* means a variable used in a net transition. An *SDL variable* means a variable used in a source SDL program. As explained earlier, an SDL variable is of certain type. All instances of a type contain the same number of atomic values. These values are represented by tokens (Figure 6).

When transforming an expression into a net transition, the most nested subexpressions of the expression are handled first. The SDL variable references of a handled subexpression are replaced with net variables and the incoming and outgoing arcs of the formed net transition are created accordingly. If an SDL variable reference points to a type instance with many atomic values, the reference is handled recursively and an arc is created for each atomic value. If the handled subexpression represents an array index, it will be added to the firing condition of the formed net transition to allow dynamical computation of array indexes. When all subexpressions are handled, the expression will no more contain any SDL variable references, only net variables. Thus the expression can now be added to the formed net transition for dynamical computation of the expression value. If an SDL statement has more than one expression, all expressions are handled in the same way and all resulting arcs will be part of the formed net transition.

A token can only be removed once, thus the encountered references to atomic values must be stored in order to know if an atomic value is referred to more than once. This enables multiple references to the same atomic value. The token modeling the atomic value is removed only once and all references will point to the net variable that holds the atomic value found from the token. There is, however, a problem concerning array references. If an index expression of an array reference contains an SDL variable reference, it is statically not known what array slot is actually referred to. Thus it might be that dynamically two different index expressions evaluate to the same number. If the index expressions are used to refer to the same array, it follows that the same array slot is referred to two times and two arcs will try to remove the token representing the atomic value of that slot. A deadlock results. This could be solved by searching from the expressions of an SDL statement all sets of array references that can dynamically refer to the same array slot. Then all combinations of references of each set must be explicitly handled by comparing the values of net variables holding the array indexes. Furthermore each group of combinations that requires a certain set of arcs must be handled in separate net transition. Thus a set of mutually exclusive net transitions is needed.

The rather tedious solution presented above can be avoided by direct source SDL program manipulation. The array references of the expressions of a statement that can dynamically refer to same array slot are assigned to temporary SDL variables in separate ASSIGN statements. The statements can then be rewritten using these temporary SDL variables. Now all problematic array references are in different statements. This of course increases the work of users. Another solution would be to use a formalism that enables reading a data item without implicitly deleting it.

Figure 9 contains an example of an ASSIGN statement and the corresponding net transition. The net transition does not include places and arcs relating to control flow.

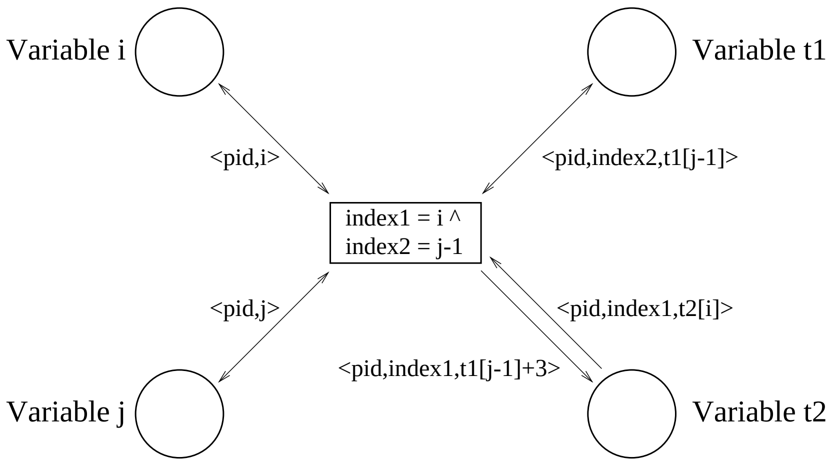

Figure 9: The net transition corresponding to the ASSIGN statement $t2(i) = t1(j-1)+3$.

4.4 Modeling Timers

There are two main reasons why implementation of timers is problematic. First, there is no concept of time in Pr/T nets. Second, the naming of timers in TNSDL is dynamic. While there is no real time model in EMMA, it still contains a partial solution to the timer problem. Expirations of a timer can be restricted to certain parts of the control flow of the process with the so called *timer expiration windows* (Section 5.3.2). This means that a timer can only expire when the point of execution of the process that activated the timer is contained in some user-defined interval. Without this mechanism a timer instance can expire at any time after it has been activated, thereby expanding the reachability graph.

A timer declared in an TNSDL program can have an unbounded number of active *instances*. Timer instances are separated from each other by the timer parameters given when a timer instance is activated. Thus the parameters are a *part of the name* of a timer instance. This feature is here called *dynamic naming*. Dynamic naming is a problem due to the general nature of the SET statement. A SET statement will activate an inactive timer instance, but it will also set an previously active timer instance by resetting it first. Because these cases must be handled separately, they have their own net transitions which must be mutually exclusive. However, it is impossible to write a general Set transition handling the case of previously inactive timer instance, because it should check that the dynamic timer name is not already in use, which in turn would mean dynamic comparison of all the active timer instance names with the name of the new timer instance. While the number of active timer instances could be kept dynamically available, it is still not possible to make a net transition with a variable number of active arcs without an upper bound, because all the arcs of the net transition are determined statically. For the same reason an unbounded queue is not possible.

Thus it was necessary to give an upper bound for the number of active timer instances a process instance can have per declared timer. There is always a token for a timer instance whether it is active or not. This way it is easy to check if a timer instance is in use, because the number of arcs in the net transition needed for comparison of all the active timer instances of a timer is available. Theoretically the fact whether a timer instance is active or not could be dealt with using a negation place which would contain all timer instances that are inactive. Unfortunately in this case the number of all possible timer instances is too great due to dynamic naming.

Each timer instance, active or not, is represented by exactly one token in the `timer` place. This token is a tuple $\langle pid, name, id, state \rangle$ containing the `pid` of the process that activated the timer instance, the *name* of the timer which unambiguously determines from which timer the instance was instantiated, a unique message identifier *id* and the *state* of the timer instance. When a token represents an inactive timer instance, only the `name` field has a nonzero value. When a timer instance is activated, its parameters are put into the `parameter` place. The message token connected to the parameter tokens via the message identifier is put into the queue when the timer instance expires. This is equivalent to sending a normal message, the parameter tokens are just created earlier. A token reserved for a timer instance can be in three states. It can represent an inactive timer instance, it can represent an active timer instance and finally it can represent an expired timer instance. When the message sent by an expiring timer is read from the queue, the token representing the timer instance is replaced with a token representing an inactive timer instance.

Even an expired timer instance can be reset if the message sent by the instance has not been read from the queue. In that case the message must be removed from the queue. In the current implementation several net transitions are needed due to the fact that the message to be removed can be in any slot. The situation is analogous to the removal of a message in other contexts. Again, the net transition of Figure 8 could be used to improve the net model. It follows that the case presented above where an already active timer instance is reset before activating it again actually branches to several net transitions which are again mutually exclusive. One of those net transitions handles the case where the reset timer instance has not expired. The other ones handle the case where the timer instance has expired and the message sent is in some slot. The mutually exclusive net transitions that together model a SET statement can be seen in Figure 10. The leftmost net transition represents the case where the referred timer instance is inactive and it is activated. The net transition in the middle is used when the referred timer instance is active, but has not expired. The active timer instance is replaced with the new one. The rightmost net transition represents the case where the referred timer instance is active and has expired, but the generated message has not been read. The active timer instance is replaced and the message is removed from the queue. In the current implementation this net transition is actually represented by `n` net transitions where `n` is the length of the queue.

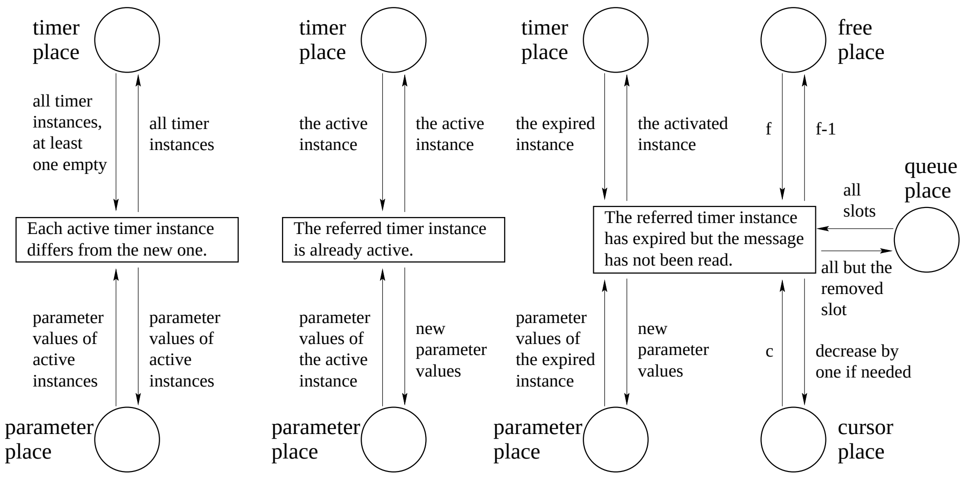

Figure 10: The net transitions used to model a SET statement.

The RESET statement is modeled just like the reset aspects of the SET statement. Currently, if one tries to reset a timer instance that is not active, the model deadlocks, because the instance does not exist. It is trivial to add a net transition that will result in the abortion of the state space generation on-the-fly if a Reset transition tries to reset an inactive timer instance. This net transition would be very similar to the Set transition that activates an inactive timer instance. A tester with reject state is needed.

The comparison whether two timer instances are the same or not means comparing all atomic values of their parameters. This can be done, because each instance of the same timer has the same number of parameters instantiated from the same types. Thus the tokens used to represent them in the `parameter` place are identical in all other respects except the atomic values. The token representing a timer instance contains the message identifier and therefore fetching the parameter tokens of the message from the `parameter` place for comparison is easy.

4.5 Interpreting the Results

It is necessary to translate the results from the Pr/T net domain into a form that can be easily related to the source language which in this case is TNSDL. Otherwise the interpretation of the results would be very slow and frustrating. The `filter` program is a tool that translates the paths found by `probe` into a more readable form. This is possible because the names of the net transitions in a net constructed by the EMMA net model generator contain enough information for finding the C++ objects corresponding to the net transitions. The C++ objects represent TNSDL statements of the source TNSDL program. To gain access to the correct C++ objects the `filter` uses the same parsers as the EMMA net model generator to build them according to symbol and control flow information generated by the TNSDL translator. Symbol information is called `TESTP` and control information is called `FCF` (Figure 3). When the C++ object is found it is easy to produce output almost identical to the TNSDL statement represented by the C++ object. In the same way, the names of the places containing variable values can be translated into variable names of the source program. In addition, data concerning a certain process instance is collected before printing. Otherwise the output of the `filter` resembles the output of `probe`, it still is structured as nodes and arrows. The `filter` has options that make it possible to acquire compact or long and informative prints. In Figure 11 there is an excerpt of a path, one Assign transition and one node. The data not involved in the operation of the `filter` has been removed. When the output of `probe` is given to the `filter`, the output as seen in Figure 12 results.

4.6 The Reachability Analyzer PROD

PROD is a Pr/T net based *reachability analysis tool* that supports on-the-fly verification. Some partial order methods designed to make it possible to avoid generation of the whole state space while verifying a property can be used, too. The analyzed net must be described with the net description language of PROD. The language is the C preprocessor language extended with net description directives. The syntax of the language can be found in [26]. A net description is compiled into an executable *reachability graph generator* program which in turn generates the actual graph. In this section, all transitions are net transitions. A *node* in the reachability graph corresponds to a reachable marking of the net.

On-the-fly verification means that properties are verified during the generation instead of after the generation of the whole reachability graph. As soon as it is known that some property does not hold, the generation can be stopped. There are two approaches to on-the-fly verification in PROD, testers and temporal formulas. When the generation is stopped by the on-the-fly verification,

```
Arrow 0: transition Automaton$170$Trans0$1$Assign3, precedence class 0
  pid = 2
  .
  .
Node 12
  AvailablePids: <.3.>
  Process$1$V$Parent: <.2,0.>
  Process$1$V$Offspring: <.0,2.> + <.1,0.> + <.2,0.>
  .
  .
  Process$113$Queue: <.0,0,0,0,0.> + <.0,0,1,0,0.> + <.0,0,2,0,0.>
  Process$113$Cursor: <.0,0.>
  Process$113$Free: <.0,0.>
  .
  .
  Process$113$V$Self: <.0,0.>
  Process$113$V$Sender: <.0,0.>
  state$181: <.0,181.>
  .
  .
  Process$117$Queue: <.2,0,0,0,0.> + <.2,0,1,0,0.> + <.2,0,2,0,0.>
  Process$117$Cursor: <.2,0.>
  Process$117$Free: <.2,0.>
  .
  .
  Process$117$V$Self: <.2,2.>
  Process$117$V$Sender: <.2,0.>
  V$168: <.2,0.>
  V$169: <.2,3.>
  Automaton$170$Trans0$1$Assign3: <.2,0.>
  .
  .
  Process$125$Queue: <.1,0,0,0,0.> + <.1,0,1,0,0.> + <.1,0,2,0,0.>
  Process$125$Cursor: <.1,0.>
  Process$125$Free: <.1,0.>
  .
  .
  Process$125$V$Self: <.1,1.>
  Process$125$V$Sender: <.1,0.>
  Automaton$222TransBegin0: <.1,0.>
  V$221: <.1,0.>
  .
  .
```

Figure 11: A fired net transition and following node as output by `probe`.

```
------------------------------------------------------------

TRANSITION:     (Arrow 0)
Process       : client_slave_2   (pid = 2)
Automaton     : G1G  (in START transition)
Element No    : 1
 -----
TASK count :=  3

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

NODE 12:
 ---
Available Pids: 3
 ---
 ---
Process clientp:  (pid = 0)
State: clientp_idle
Queue : - - -
Cursor: 0
Free  : 0
*Specials:
Offspring = 2     (at top level)
Self = 0
Sender = 0
*Variables:
 ---
Process client_slave_2:  (pid = 2)
State: in START transition       (Automaton: G1G)
Queue : - - -
Cursor: 0
Free  : 0
*Specials:
Parent = 0     (at top level)
Offspring = 0     (at top level)
Self = 2
Sender = 0
*Variables:
server_party = 0
count = 3
 ---
Process serverp:  (pid = 1)
State: in START transition       (Automaton: G4G)
Queue : - - -
Cursor: 0
Free  : 0
*Specials:
Offspring = 0     (at top level)
Self = 1
Sender = 0
*Variables:
client_party = 0

------------------------------------------------------------
```

Figure 12: And here is the same data transformed by the `filter`.

the illegal node found from the graph is provided. With `probe` it is easy to search the path that leads to the illegal node.

A *tester* has always a place which contains the *state* of the tester as an unary tuple. A tester is defined by the place and all the arcs that are connected to the place. The state of a tester is changed by normal transitions just like the marking of any place. The states of a tester are defined by enumerating the monitor states separately for deadlocks, livelocks and infinite paths. A tester stops graph generation if a deadlock, livelock or infinite path is encountered and the current state of the tester is a monitor state for the found anomaly. A transition instance is *visible* with respect to a tester if at least one of its arcs is connected to the tester place, otherwise it is *invisible*. A *deadlock* means that no transition is enabled at the current marking of the net. A *livelock* means a loop that contains only invisible transition instances. An *infinite path* means a loop that has at least one visible transition instance. The generation is stopped at such a node in the infinite path in which the state of the tester allows stopping and the next transition instance is visible. The state of the tester needs to be an infinite path monitor state only at the node where the generation is stopped. Thus also the state of the tester can loop. In addition, a subset of states of a tester can be *reject* states. The graph generation is stopped whenever a tester enters a reject state. For example on-the-fly detection of deadlocks can be achieved by creating a tester which only has one state. The only state is defined to allow stopping if a deadlock is found, it is a deadlock monitor state.

Linear temporal logic formulas [5] can be verified on-the-fly. A tester with infinite path monitor states corresponding to the formula is constructed automatically. Two special transitions are also used, one is fired before any other transition and performs certain initializations, the other eliminates terminating firing sequences by being enabled when no other transition is enabled. All infinite paths starting from the initial node have to satisfy the formula. Rules of satisfaction can be found in [26]. The graph generation is stopped if an infinite path violating the formula is found.

Many times firing of transitions is defined in such a way that any subset of enabled transitions can be fired simultaneously. This property is called *step* semantics. If all enabled transitions are always fired simultaneously, *maximal step* semantics is in use. *Interleaving* semantics means that two transitions can not be fired simultaneously, one is always fired before the other. It should be noted that only interleaving semantics of transitions can be used in PROD.

The net of Figure 1 is defined as PRODL in Figure 13. When generating the reachability graph with plain depth-first search, the 40 megabytes of memory reserved for the graph is filled in a few minutes. Generation while using the harddisk to store the graph is too slow to be sensible and therefore the generation was stopped. The graph contained 671,000 nodes of which almost all were

```
#place NotAdded lo(<.4.>) hi(<.18.>) mk(3<.18.> + 5<.17.> + 4<.14.> +
                                        4<.5.> + 3<.6.> + 3<.8.>)
#place NotVisited lo(<.0.>) hi(<.21.>) mk(<.0..21.>)
#place Value lo(<.0.>) hi(<.21.>) mk(<.0.>)
#place Counter lo(<.0.>) hi(<.22.>) mk(<.0.>)

#trans add
  in {
    NotAdded:   <.nro.>;
    NotVisited: <.(x+nro) % 22.>;
    Value:      <.x.>;
    Counter:    <.c.>;
  }
  out {
    Value:      <.(x+nro) % 22.>;
    Counter:    <.c+1.>;
  }
#endtr

#tester Counter reject(<.22.>)
```

Figure 13: The modulo 22 problem defined with PRODL.

completely processed. 173,000 terminal nodes were found. A simple search through the nodes revealed that the solution node was among the terminal nodes. With depth-first search many paths to the solution node can be found. These paths are solutions to the problem. Unfortunately searching for paths in a large graph is quite slow. When using breadth-first search 527,000 nodes were found, only 133,000 were completely processed. No terminal nodes were found. This shows that the graph is very wide, the nodes have a big branching factor. Thus breadth-first search must go through hundreds of thousands of nodes before finding even one terminal node, while the depth-first search quickly founds thousands of terminal nodes. When using breadth-first search there are more nodes that are not completely processed, because breadth-first search follows many paths at the same time while depth-first search always completes searching one path before starting to search another path. Of course, if the generation would proceed for several days, it might eventually terminate even though use of storage is very slow.

When using an optimized stubborn set method, after the graph generator had been running about 19 hours 1.6 million nodes were generated. The selected method used depth-first search. Almost all nodes were completely processed meaning all enabled transitions in the stubborn set for that node had been fired. 390,000 terminal nodes were found. Roughly thousand solution paths were found in the partial reachability graph. Finding these paths was quite slow and the harddisk was in continuous use. The size of the graph was 80 megabytes.

In this case unfolding (Section 5.4) is not critical, it is done very quickly, although there are many instances of the only transition in the unfolded net, because the tokens taken from `NotAdded`, `Value` and `Counter` places are chosen non-deterministically and the number of transition instances is the product of the numbers of the possible tokens in these three places. The token taken from the `NotVisited` place is a function of other tokens and thus its selection is deterministic. The range of possible values for each place is about twenty which means there are around 20^3 or eight thousand transitions in the unfolded P/T net. If there were more places connected to the transition with a number of possible tokens in the range of twenty the unfolding would be possible with 64 megabytes of memory. Thus when unfolding is needed, it might be useful to divide one transition with a great number of possible instances into several transitions. The number of instances might decrease. If unfolding is not needed, the number of transition instances is not critical. In fact it might be more efficient to minimize the number of transitions rather than the number of transition instances. However, even if the unfolding is successful, actual generation of the reachability graph is much slower for a P/T net. We do not know whether the stubborn sets method pays off, because the generation was never finished regardless of the method used. Because the stubborn sets method preserves all deadlocks and should reduce the number of nodes, some heuristic information about efficiency can be acquired by comparing the ratios between the number of nodes and the number of terminal nodes. If the ratio is considerably smaller in the partial graph generated using the stubborn sets method, the method has been useful.

If we only want to find a solution, one can be found very quickly with on-the-fly analysis. On-the-fly analysis always uses depth-first search. A tester is attached to the `Counter` place. The tester state where the `Counter` place holds the token $\langle 22 \rangle$ is defined as reject state. The tester can be very easily added to the PRODL net as seen in Figure 13. When the value of the unary tuple in the `Counter` place is incremented from 21 to 22, the tester enters the reject state and graph generation is stopped. On-the-fly analysis works well, because there are many solution paths to the solution node. However, even if there would be only one path to the solution node, on-the-fly analysis would still be twice as fast as normal analysis on the average. The only solution node is expected to be found when half of the graph is generated.

4.7 The Query Language Program PROBE

A generated reachability graph can be inspected with the interactive tool `probe`. An *arrow* corresponds to a fired transition instance. An introduction of the use of `probe` can be found in [11]. Also in this section all transitions are net transitions.

`Probe` is an interactive, command-line based program which supports macros. With `probe` it is possible to move freely in the graph and search for paths that satisfy certain formulas. Paths beginning from some node can be searched by first moving to that node. It is also possible to evaluate a formula in every node of the graph. Both breadth-first and depth-first searches are possible. When searching for paths, formulas can be used to determine the end nodes of a path, the nodes preceding either the end node of a path or the entrance node of the loop of an infinite path, and the node that starts the loop of an infinite path. Of course formulas can also be used to test the contents of nodes locally. PROD contains macro definations which provide CTL-operators [4] implemented with native formulas of `probe`.

The syntax and function of the statements and definitions of expressions, formulas and so forth can be found in [26]. In the following example the `probe` has been used to inspect the modulo 22 problem.

```
prod mod22.init      ; Creates the graph generator executable.
mod22                ; Creates the reachability graph.
                     ; There are a great number of options that
                     ; can be used for example to select a
                     ; variation of the stubborn sets method.

; Suppose the tester of the net is enabled. Then the generation will be
; stopped with following notification.

Reject state reached
Node 5954
For more information, start "probe mod22"
and look at the set %1

probe mod22          ; Starts the query program for the graph.

0#statistics         ; Prints information about the graph.
Number of nodes: 5955
Number of arrows: 6139
Number of terminal nodes: 1760
Number of nodes that have been completely processed: 5926
Strongly connected components have not been computed

0#query volatile dspan(true,true) %1
                     ; Searches the path that allowed the solution
                     ; node to be found.
   0  [5>    6  [5>   12  [5>   18  [3>   22  [3>   26  [2>   29  [2>   32  [1>
2791  [0> 3330  [0> 4897  [0> 5668  [1> 5893  [1> 5925  [0> 5934  [0> 5941  [0>
5944  [0> 5945  [0> 5946  [0> 5949  [0> 5952  [0> 5953  [0> 5954
1 paths
------------------------------------------------
```

```
; The plain numbers are node numbers and numbers surrounded with [ >
; are numbers of arrows used to get from one node to the next.
; Much more detailed printing of paths is possible.

0#goto 5954          ; Chooses node 5954 as current node.
5954#look            ; Displays the marking of the node.
Node 5954
  Value: <.15.>
  Counter: <.22.>
------------------------------------------------
; Because places NotAdded and NotVisited are not shown, they are empty.
; Thus indeed in the solution node all operations have been used and all
; numbers of modulo 22 arithmetic have been visited.

; Now suppose we use the plain depth-first search to generate a
; partial reachability graph. The option -D forces depth-first search.
; The other options optimize the performance of the graph generator.
mod22 -Db16 -m40000000
probe -G mod22       ; The option -G will load the whole graph to the memory.

0#statistics
Number of nodes: 679724
Number of arrows: 791990
Number of terminal nodes: 174854
Number of nodes that have been completely processed: 679695
Strongly connected components have not been computed

; The solution node can be found by searching for a node that has
; the number 22 in the first field of some tuple in the {\tt Counter} place.

0#query node ((Counter) : (field[0] == 22)) != empty
5954
1 paths
Built set %1
------------------------------------------------

; Let's search all the solution paths.

0#query volatile dpath(true,true) %1
.
.
5954,        0  [5>    6  [5>   12  [5>   18  [3>   22  [3>   26  [1>   28  [2>
16807  [1> 16809  [3> 20030  [1> 20032  [1> 20209  [2> 20314  [0> 20315  [0>
20333  [0> 20340  [0> 20341  [0> 20342  [0> 20343  [0> 20344  [1> 20346  [0>
9542  [0> 5954,        0  [5>    6  [5>   12  [5>   18  [3>   22  [3>   26  [2>
  29  [1>   31  [2> 8055  [2> 9275  [1> 9277  [1> 9425  [1> 9503  [0> 9504  [0>
9522  [0> 9529  [0> 9530  [0> 9538  [0> 9539  [0> 9540  [0> 9541  [0> 9542  [0>
5954,        0  [5>    6  [5>   12  [5>   18  [3>   22  [3>   26  [2>   29  [2>
  32  [1> 2791  [0> 3330  [0> 4897  [0> 5668  [1> 5893  [1> 5925  [0> 5934  [0>
5941  [0> 5944  [0> 5945  [0> 5946  [0> 5949  [0> 5952  [0> 5953  [0> 5954
408 paths
------------------------------------------------
```

```
; Just one path can be found with the following command.

0#query volatile dspan(true,true) %1
   0  [5>    6  [5>   12  [4>   17  [4> 629147  [3> 629151  [2> 663933  [2>
663936  [0> 668573  [0> 670676  [0> 671366  [0> 671367  [1> 671577  [2> 671580
 [1> 671582  [0> 671583  [1> 671592  [0> 671593  [1> 229901  [0> 90970  [0>
90971  [0> 88158  [0> 5954
1 paths
-----------------------------------------------

; The next command will return one of the shortest paths to the node.
; The property comes form the fact that breadth-first search is used.
; In this case the command is not very useful, because all paths
; leading to the solution node are of equal length.

0#query volatile bspan(true) %1

; The search for a shortest path is quite heavy and it turned out that
; it could not be done with the current implementation in the machine
; which I am using.
```

5 Analysis Methodology

Typically the state space of a concurrent system is so large that space requirements of reachability analysis are too great. Furthermore space requirements might grow even exponentially with respect to the size of a system. While the generation of state space also can take a very long time, space requirements usually dominate. Thus it is reasonable to reduce the space requirements even if it means stricter time requirements.

There are two ways to reduce the space requirements. The number of states in the state space can be made smaller. This can be done by optimizing the net model, with general state space reduction algorithms in the reachability graph generator or by modifying the SDL specification. The other way is to reduce the storage requirements of a single state. This can be achieved by either packing the information contained in the states more tightely or leaving out insignificant information from the net model and thus also from states.

5.1 About the State Space Explosion Problem

The most challenging problem in reachability analysis is the *state space explosion problem*. The state space explosion is closely related to concurrency, the large number of states in concurrent systems is in fact the very reason why the errors are so hard to find manually. Therefore there is no way to dismiss the problem entirely, but it can be made less severe.

The basis of the state space explosion is the fact that the number of states in a concurrent system might be as large as the product of the number of states of the subsystems of the system. In other words if a system has `n` processes which have `s` states each, the number of states in the system can be as large as s^n. Thus the state space might grow exponentially with respect to the number of processes. The worst case is reached exactly when the processes are completely independent. In practice there is less states because all the processes do not usually exist at the same time and there is some synchronising communication between processes. Still the number of states often grows very fast when the number of processes or the complexity of processes is increased. A trivial example can be seen in Figure 14. The processes in the figure are completely independent and the use of interleaving semantics is assumed. The edges that would be present if the step semantics was used are drawn with dotted lines.

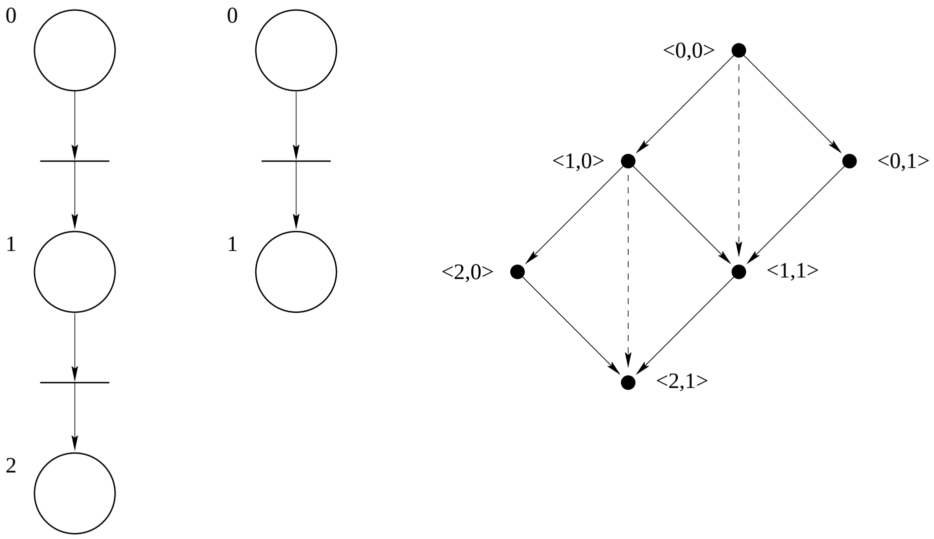

Figure 14: A simple example of two processes and their state space.

5.2 Atomic Sections in EMMA

Here an *atomic section* means a sequence of net transitions corresponding to consecutive SDL statements of an SDL transition of an SDL process. When executing the model, net transitions belonging to such a sequence are fired one after another while other processes are disabled, which means that no net transition belonging to any other process can become enabled before the last net transition of the sequence has been fired. Atomic sections greatly reduce the number of possible interleavings of net transitions belonging to different processes and hence also the number of states of the net model.

TNSDL transitions are not atomic by default, the executed process can change in the middle of a TNSDL transition. This means that a TNSDL transition is not necessarily completely executed before execution of another TNSDL transition is started. Therefore atomic TNSDL transition models exclude a huge number of possible execution orders of TNSDL statements. Use of atomic sections is still reasonable, because most TNSDL statements in a process can not have any effect on the state of other processes. They only affect local data of the process. The exceptions are OUTPUT statements and such ASSIGN and INPUT statements that alter data viewable from other processes. It might be worthwhile to model TNSDL transitions as atomic, because many errors can still be found. Moreover, TNSDL does have a mode that makes TNSDL transitions atomic. This mode reduces the memory and processor time requirements of the implementations of TNSDL specifications. However, the reduced concurrency of processes diminish the service providing ability of processes. Thus the mode is a tradeoff between resource requirements and efficiency of services provided by processes of the system. If the mode is not used, modeling TNSDL transitions atomically does leave out some possibly essential executions of implementation of the modeled system. More on this later. In the following text execution normally refers to the execution of the net model generated by EMMA.

The simplest implementation of atomic sections makes all SDL transitions atomic. For this only one *resource token* is needed in the place reserved for this purpose, the `execution resource distributor` place. If the resource token is in the place, execution of an SDL transition can be started. If the place is empty, one SDL transition is being executed and the execution of another SDL transition can not begin. When the execution of an SDL transition is started the resource token is removed from the place and when the execution is finished the resource token is returned to the place. From this point on, the resource token is called *execution resource.* There is always an INPUT statement at the beginning of an SDL transition, consequently Input transitions can only be fired when the execution resource is available. Nextstate, Stop and Return transitions are the only ones that can and always will terminate an SDL transition, therefore they return the execution resource. In addition, a Call transition returns the execution resource and the net transition that is fired when the control is returned from the called procedure back to the process claims the execution resource. This kind of net transition is fired after firing the Return transition of the procedure. These rules cover the simplest implementation, but there is an exception. The executions of procedures not containing any SDL states are merged with the atomic sections of the net transition which called the procedure. This is done, because the only reason why a Call transition normally returns the execution resource, is the fact that a procedure can have SDL states which means it can have more than one SDL transition each needing an own atomic section. An example of a model of an SDL transition with reduced concurrence can be seen in Figure 15.

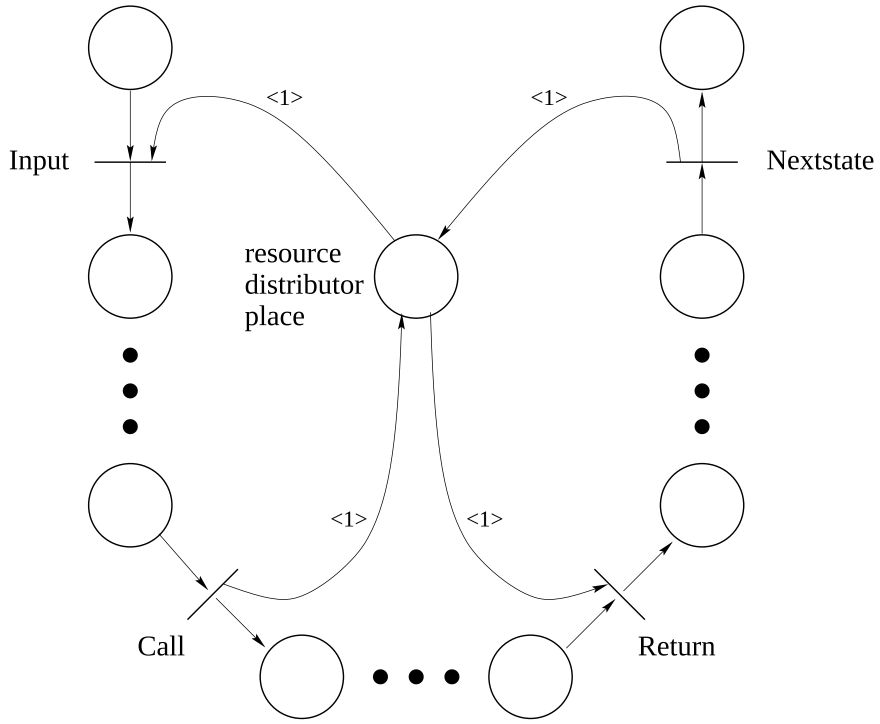

Figure 15: A model of an SDL transition containing a call to a procedure which has at least one SDL state.

5.2.1 Problems in Using Atomic SDL Transitions

Using atomic TNSDL transitions clearly excludes some interesting behaviours, because even when a net transition is executed which directly changes the state of another process the execution of the TNSDL transition must be completed before the affected process can react. Output transitions and Assign transitions that write to a global variable directly affect other processes. An Input transition can also write to a global variable. Also net transitions that model an TNSDL statement which reads from a global variable are problematic, because two read operations in the same TNSDL transition from the same global variable can get different values if some other process changes the value of that variable in between. In the following text the term *critical transition* will refer to a problematic net transition falling into one of the groups described above. If two read operations referring to the same global variable are in the same atomic section, another process can not change the global variable value between them. Thus it is easy to see that the handling of global variables requires added concurrency compared to completely atomic TNSDL transitions, because changing the value of a global variable has immediate effect on shared data of the system. It might be useful to notify the user if a model produces different execution paths due to the handling of global variables, because this is usually an error. It is, however, not straightforward how to detect such situations dynamically.

Added concurrency is needed even if the communication between processes is only made asynchronically with Output transitions. For example we have three processes P_1, P_2 and P_3, and the process P_2 sends two messages to P_3 in the same SDL transition. The process P_1 sends one message to P_3. If SDL transitions are atomic, the two messages sent by P_2 will always appear one immediately after another in the queue of P_3. If concurrence is not restricted P_1 can send its message in such a way that it will appear in the queue of P_3 between the two messages sent by P_2. This example is depicted in Figure 16. The thick vertical lines represent processes, the horizontal arrows represent sending of messages, the dotted horizontal arrows represent the change of control from one process to another and the vertical arrows represent executions of SDL transitions. The message sending order of the right picture is not possible if atomic SDL transitions are used, because it requires that the execution of the SDL transition of the process P_2 is interrupted. Always when a process sends at least two messages in the same SDL transition, there may be an execution in the fully concurrent model in which some message queue has a such sequence of messages which can not appear in any execution of the net model using completely atomic SDL transitions.

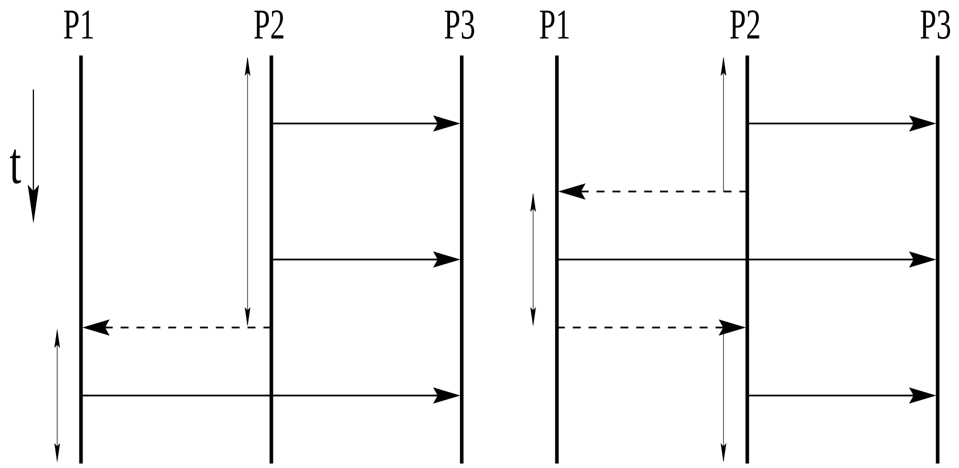

Figure 16: An example illustrating a simple situation where the use of atomic SDL transitions leaves out interesting executions.

5.2.2 Dynamic Atomic Sections

The problems of atomic SDL transitions can be avoided by allowing the temporary release of the execution resource while executing an SDL transition.

Of course the net transition following the net transition temporarily releasing the resource must reclaim the execution resource. Unfortunately, it can not be statically known whether a net transition should claim the execution resource or not. This follows from the fact that a net transition can have several possible predecessors some of which release the resource while the others do not. If a net transition releasing the resource is executed, the next net transition must reclaim the resource. On the other hand, if a net transition that does not release the resource is executed, the next net transition must not try to claim the resource. These situations can arise with Decision, While and Join transitions. This means that for each process the knowledge whether the next net transition should claim the resource or not must be stored dynamically. For every existing process one additional place is created that holds this knowledge.

To implement the temporary release of the execution resource, in every net transition that corresponds to some SDL statement both the knowledge about execution resource availability and the knowledge whether the execution resource is required or not is taken in, possibly modified and put back. Unfortunately, if the execution resource is unavailable the resource distributor place will be empty, and any net transition that tries to remove a token from it is disabled. This problem can be circumvented by redefining the execution resource in such a way that the resource distributor place always contains exactly one token. A certain tuple in the place means that the execution resource is available and another tuple is used to denote that execution resource is unavailable. Thus now the execution resource availability is expressed with a frame. Now it is easy to check whether the execution resource is available or not. It is clearly true that only those net transitions that have more than one possible predecessor in the process instance need the dynamical execution resource reclaiming mechanism. Currently all net transitions corresponding to some SDL statement use the mechanism because the knowledge about predecessors of net transitions is not easily obtainable. The extra arcs can not be fully dismissed, but they can do little harm for the performance of the analysator.

The mechanism to dynamically decide whether to claim the execution resource in a certain net transition or not is seen in Figure 17. Transition 3 has two predecessors, 1 and 2. Transition 2 releases the execution resource while transition 1 does not. If transition 1 is executed, the `resource distributor place` that holds the resource frame and the `claim execution resource` place both contain zero, which means that the process instance already has the execution resource and the next net transition of the process instance should not try to claim it. The execution of transition 3 does not cause any changes. On the other hand, if transition 2 is executed instead of transition 1, the execution resource is made available and the next net transition is forced to reclaim the execution resource by changing the `claim resource` value to one. Now the process instance can immediately reclaim the resource or any other process instance could claim it. In any case, transition 3 can be executed only when the execution resource is available, because the `claim resource` value of the process instance is one. Explicit claiming and release of the redefined execution resource can be seen in the SDL transition model of Figure 18.

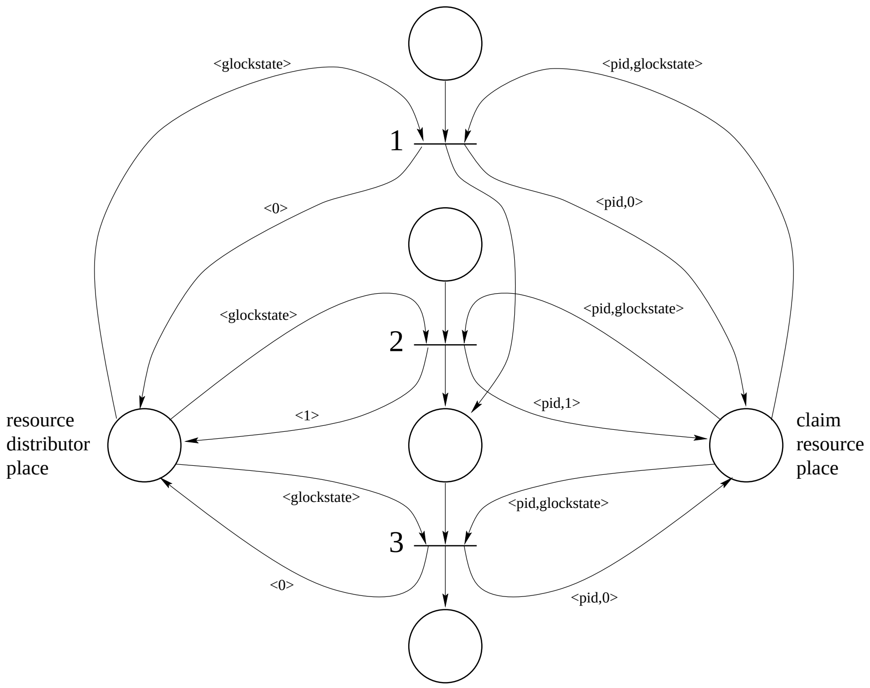

Figure 17: An illustration of dynamical claiming of the execution resource.

Another way to implement the execution resource reclaiming would be to explicitly reclaim the execution resource in the net transition following a critical transition by introducing a new net transition for each critical transition. This way the net transition reclaiming the resource would always have exactly one predecessor. This would be required only after such a net transition whose successor has more than one predecessor, but this optimization would again require knowledge about possible predecessors of a net transition. While this method would be simpler in a way, it would introduce unnecessary net transitions. This solution would not require any dynamic knowledge about execution resource reclaiming, but storing this knowledge in the net model is much more harmless than adding new net transitions. Yet another way would be to duplicate parts of the control flow in the processes in order to ensure that each net transition has only one predecessor. In other words, the control flow of an SDL transition would be a tree. This is hardly a sensible alternative.

The user can currently choose from three execution resource handling schemes.

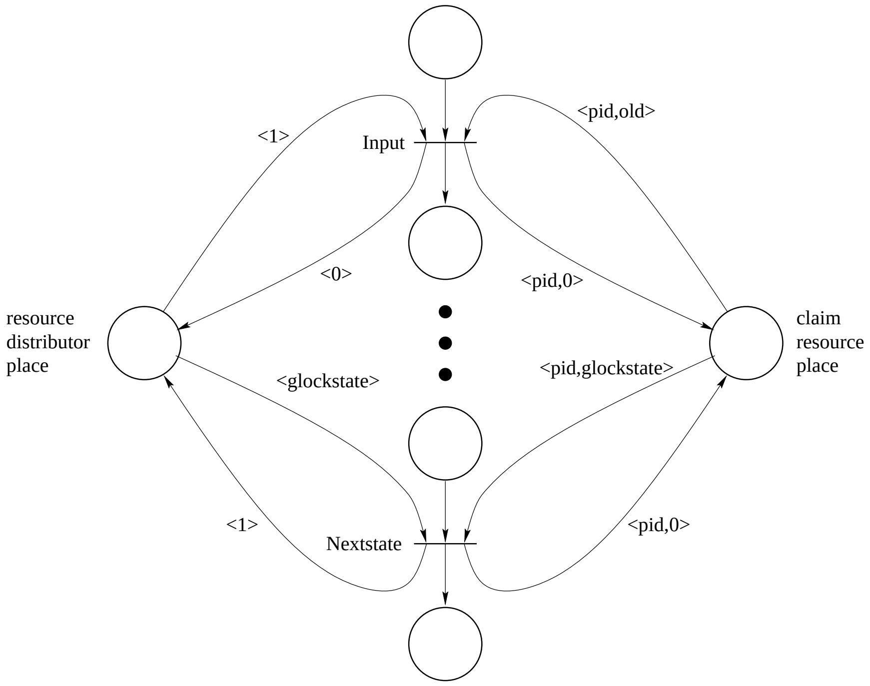

Figure 18: An SDL transition modeled using the more versatile concurrency reduction scheme.

The first forces atomic TNSDL transitions, the second releases the execution resource after all Output transitions and the third also releases the resource after Assign transitions altering the value of a global variable. It would be better if atomicity options could be selected separately which would enable the global variables related execution resource releases only. An Input transition changing the value of a global variable and net transitions that read a global variable are not yet detected. Currently the dynamic execution resource management is used even with atomic TNSDL transitions which is unnecessary but in the end has negligible impact on time and space requirements and no impact on results.

It is often the case that using a execution resource handling scheme that does not make all SDL transitions atomic drastically increases the space and time requirements of the state space generation when compared to a net model with atomic SDL transitions. Therefore the user has the alternative to create manual execution resource release points in places where additional concurrency is required. This makes it possible to raise the concurrency to a reasonable level at the interesting points in the net model. The manual release points are implemented in the same way as other release points in the middle of SDL transitions. It might be a good idea to enable the use of automatic execution resource handling schemes per process or even per SDL state basis, because then the user would have no need to customize the execution resource handling of the whole program when he only wants to customize a small part of it. It might also be useful to be able to remove unnecessary execution resource release points created by an automatic scheme.

5.2.3 Managing Message Queue Overflows

If in an SDL transition an Output should be executed but the target queue is full, the whole execution of the model will terminate, because the execution resource is not returned which will cause a deadlock. This is avoided by adding for each Output transition a new parallel net transition named QueueFull that will make the execution resource available if the target queue is full and the execution resource is not already available (Figure 19). In the figure, `L` means the length of the queue. Indexes of a queue start from zero, therefore queue slot `L` is the lowest numbered non-existent slot. It can be presumed that the `claim resource` value is zero, because if it were one, the execution resource would already have been released by the process. This is actually needed, because otherwise a QueueFull transition would remain enabled after it was fired. Thus it could be fired again immediately which would completely mess up the execution resource handling.

In essence the QueueFull net transition will create a gap in the atomic section if the target queue happens to be full at the time the Output transition should be fired. The target process of the Output transition can then consume some message from the queue to make room for a new message. Then the Output transition can be fired. Note that it will implicitly reclaim the execution resource. It might be valuable to know, if a queue of a certain length can overflow. It is easy to replace the described net transition with one that intercepts any queue overflow situations on-the-fly. If no overflow situation is detected, making the queues longer will have no effect on the reachability graph of the system, because there is always room for a message. On the other hand, if overflow situations are detected, there are some global system states that will not be reached with the current queue lengths. These states might or might not be relevant for the analysis.

5.2.4 Optimal Dynamic Atomic Sections

It is enough to guarantee that in every atomic section there is at most one critical transition. This means that a single critical transition in an SDL transition model should not cause an additional execution resource release or more generally, an execution path of an SDL transition model containing `n` critical transitions should contain `n-1` execution resource release points in such a way that every atomic section has one critical transition in it.

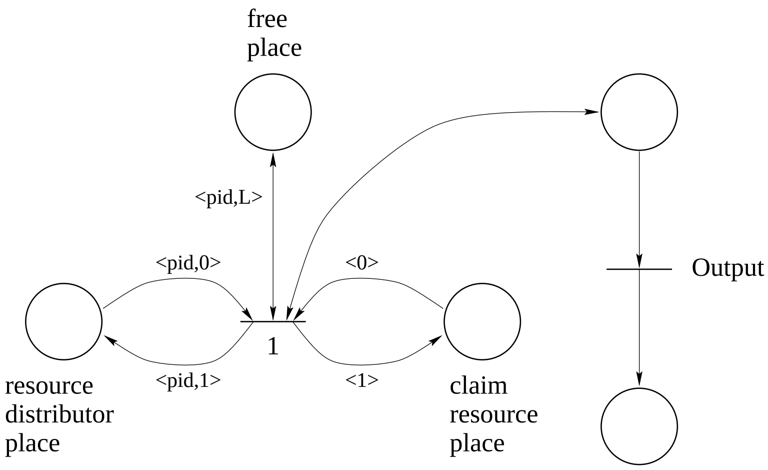

Figure 19: A QueueFull net transition that releases the execution resource if the target queue is full.

The idea behind the plan stated above is, that if there is at most one critical transition in an atomic section, it follows that there is at most one indivisible action in an atomic section that can either influence or depend on some action made in another process. All non-critical transitions of a process are independent of other processes. Thus a whole atomic section of a process can be seen as an atomic action which can depend on another process and the interleavings of atomic sections will contain all possible interleavings of actions which can depend on each other.

This contains the assumption that a critical transition is an inherently atomic unit capable of reading or changing a global variable or affecting the state of another process. Here inherently atomic means that execution of such a unit can not be interrupted by context switching while executing *an implementation* of a TNSDL specification in *the real target environment*. Clearly, this is a very low-level matter and depends on the implementation. In EMMA, the smallest execution unit is of course a net transition. The smallest part of a TNSDL source that a net transition generated by EMMA can correspond to is a statement. Before a program is executed in a computer, it is eventually compiled into machine code. If context switching is not restricted with some mechanism, the granularity of context switching is very fine. The current implementations of TNSDL specifications do not have such restriction mechanisms. Some operations provided by the operating system are atomic, however.

Reading a global variable is problematic, because the same global variable can be read multiple times in a single TNSDL statement. For example in assignment $j = i + i$ the variable i is read two times. In an actual implementation the two read operations could return different values. In fact changing more than one global atomic value in one statement is also problematic, because a context switch can happen after only a part of the atomic values are updated. This situation arises when changing the value of a non-atomic type instance in an ASSIGN statement. The source TNSDL program can be manipulated so that a statement refers to at most one global atomic value. However, this manipulation can be quite tedious when complex data types are used. Automatic breakdown of references to global type instances in the net models would be very useful.

There can not be any problems with Output transitions, because an OUTPUT statement can not contain any additional operations and obviously in any implementation an OUTPUT statement either is executed before a context switch or is not. When a message queue is inspected by the process that owns the queue, a message either is or is not in the message queue. In the implementations of TNSDL specifications OUTPUT and INPUT statements among others are implemented in the operating system which provides the required level of atomicity.

5.2.5 Implementing Optimal Dynamic Atomic Sections

When using dynamic atomic sections in the current implementation, the execution resource is released too often. Some dynamic atomic sections which do not contain a critical transition could be joined to another atomic section. The superfluous execution resource releases can be left out, if the execution resource is released just before a critical transition is fired, not within the critical transition. One way to achieve this is to release the resource in every possible predecessor of a critical transition. However, the resource should not be released before the first critical transition encountered on the chosen execution path, because there would be no critical transition in the resulting atomic section. Thus in every process instance model it must be known whether a critical transition has already been fired in the chosen execution path. This information is used to make the releases of the resource conditional. The place which holds the information is of course reset whenever execution of an SDL transition terminates. This way the execution resource is released exactly once between every two successive critical transitions on any chosen execution path. It follows that every atomic section has exactly one critical transition in it.

An example can be found in Figure 20. The top picture in the figure represents an SDL transition model which has two critical transitions. The pictures in the middle and at the bottom represent the two possible execution paths through the SDL transition. In the execution path in the middle both critical transitions are encountered, and thus the execution resource is released in the predecessor of the second critical transition. In the execution path at the bottom only one critical transition is encountered, and therefore the execution resource is not released in the middle of the SDL transition. In the special case where two critical transitions immediately follow each other the first critical transition will always release the execution resource. In this case the execution resource will be released as a side effect of the predecessors of a critical transition and no additional net transitions need to be introduced. Because the predecessors of a critical transition must be known, the arcs connected to the `resource distributor place` are only needed in the critical transitions, their predecessors and Output transitions. They are needed in Output transitions due to QueueFull transitions which can release the execution resource. However, the current implementation does not support detection and modification of predecessors of a net transition. This can certainly be overcome, but an alternative method is presented anyway.

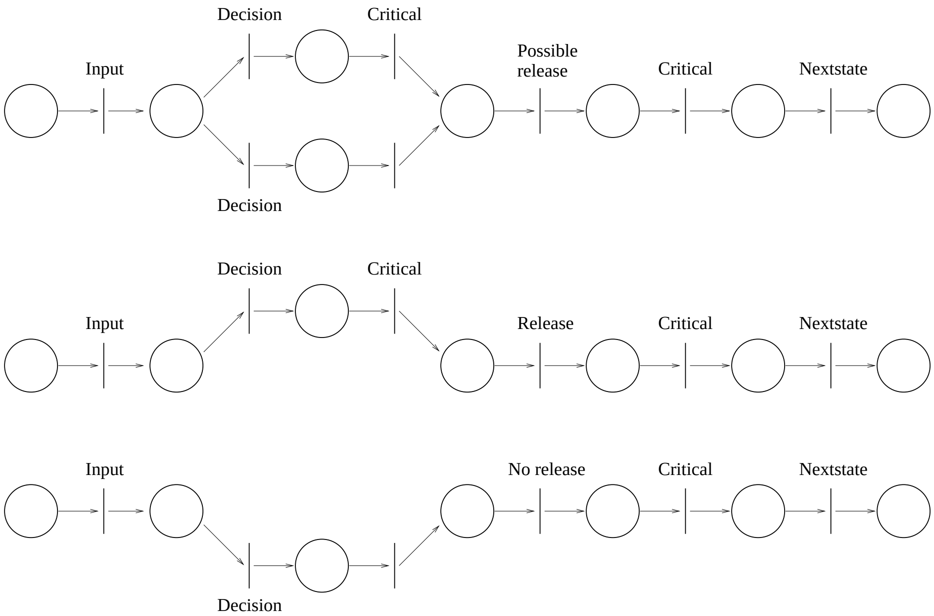

Figure 20: Dynamic release of the execution resource before critical transitions.

The release of the execution resource can be made just before critical transitions also using *additional net transitions*. In that case every critical transition will have a mutually exclusive net transition which will release the resource if necessary. More accurately the release transition will release the resource if the critical transition is not the first occurrence of a critical transition in the chosen execution path and the release transition has not already released the resource during the current execution of the SDL transition. The execution resource release will not change the point of execution in the SDL transition, thus the critical transition can be fired if it is the first one in the chosen execution path or the execution resource has already been released. The arcs connected to the `resource distributor place` are only needed in the critical transitions, release transitions and Output transitions.

The mutually exclusive transitions can be seen in Figure 21. The upper net transition is the critical transition which has some side effects relating to the modeled SDL program, which are not shown in the figure. The lower net transition is only used to release the execution resource. It will not affect the execution point of the process. The Figure 20 naturally relates closely to this method also. The only difference is that in the execution path in the middle the execution resource is released by the release transition instead of the predecessors of the critical transition. While this method is easier to implement, it has the drawback that every critical transition has an additional net transition reserved for it. These net transitions will only make the atomic sections a bit longer, they will not cause any exponential growth of the reachability graph. If n is the number of net transitions in the model of an SDL transition that has only one possible execution path and c is the number of critical transitions contained in the n net transitions, every execution of the SDL transition would be represented by $n + c - 1$ nodes in the reachability graph instead of n nodes. Thus the growth of the reachability graph is just linear with respect to the increase in length of the atomic sections.

5.2.6 Boundaries of Atomic Sections

The biggest possible atomic section is currently a model corresponding to an SDL transition. It would of course be possible to execute more than one SDL transition model within a single atomic section if the queue of the executed process model happens to hold the messages needed. The Nextstate transitions could keep or release the execution resource depending on the contents of the queue. Especially when short message queues are used, the improvement might not reduce the size of the reachability graph considerably. Also there is no reason why the execution resource always should be released between a Call transition and the start SDL transition of the procedure. The same applies to a Return transition and the corresponding net transition of the caller. The main rule is that there always should be at most one critical transition in an atomic section.

Whatever the size of the atomic sections, it can always be that many of them are actually independent, in which case they will generate a great amount of interleavings which are completely useless at least when trying to find deadlocks. Thus the partial order methods used in the graph generator can always be useful, since they can get rid of useless interleavings of independent actions (Section 5.4.1). Suppose we create n instances of the same process which will make certain local initializations before it starts to communicate with other

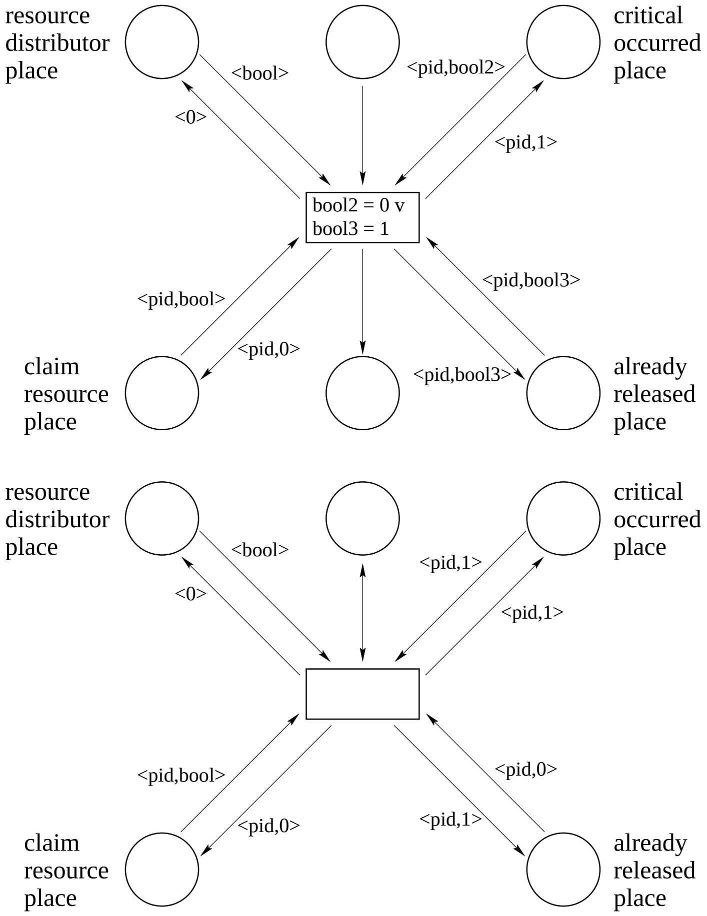

Figure 21: Mutually exclusive net transitions for release of the execution resource.

process instances. The created process instances can make these initializations in any order, but in the end they all are initialized and the chosen order does not have any impact on this state. Any single initialization order in the reachability graph could be used and the other $n! - 1$ orders could be left out. The partial order methods try to achieve just that.

5.2.7 Merging Net Transitions in an Atomic Section

It would be optimal if an atomic section could be modeled with a *single* net transition. This would further reduce the number of states in the model. While it is not straightforward to merge all the net transitions of an atomic section, some mergings can be done easily. A net transition is said to *touch* a place if there is at least one arc between the place and the net transition. The direction of the arc does not matter. The most trivial rule is that two successive net transitions of an atomic section can be merged together if the sets of places they touch are disjoint not counting the control place between the two merged net transitions. When the merged net transition is fired the control token will be moved from the control place preceding the first merged net transition straight to the control place following the second merged net transition. The resulting net transition will have all the arcs of the two merged net transitions except the two arcs that are left out due to removal of the control place between the two net transitions. The names of the net variables of the merged net transitions must be made disjoint. The firing condition of the resulting net transition is acquired by concatenating the firing conditions of the merged net transitions with the boolean `and` operator. The computations used to calculate the values of the output variables of the net transitions are joined to acquire the computations for the merged net transition. If there are any additional acceptance conditions in the two merged net transitions, they are concatenated with the boolean `and` operator to obtain an acceptance condition for the merged net transition.

The condition of merging can easily be weakened. The merged net transitions can touch the same place if the first leaves its tokens unmodified. The first merged net transition can not modify the tokens of the same place, because the second net transition should then use the updated value, which it can not do, if it is merged to the net transition that changes the value. An obvious improvement would be to detect the overlappings of the two merged net transitions based on tokens rather than on places. The merged net transitions can read and modify the tokens of the same place in any way as long as they do not touch the same tokens. They can of course read the same token and the second net transition can even modify a token read by the first net transition. Via manipulation of annotations of the arcs and the firing condition of the second net transition the condition could be weakened further. The first transition could modify a token refered to by the second transition.

In Figure 22 there is an example of merging. On the left there is a chain of two net transitions, 1 and 2. The control goes through control places `A`, `B` and `C` respectively. Transition 1 refers to variables X and Y and changes the value of Y. Transition 2 refers to variables X and Z and changes the value of X. Transition 1 can change the value of Y, because transition 2 does not refer to Y. Transition 2 can change the value of X even though transition 1 refers to it, because transition 2 is the second merged net transition. Transitions 1 and 2 can be merged and the result can be seen to the right in the figure. Now control passes straight from the control place `A` to control place `C`.

When actually reducing an atomic section, the merging condition is used recursively. When reduction of an atomic section begins, the condition is applied to the first two net transitions of the atomic section. If they can be merged, next the condition is applied to the merged net transition and its successor.

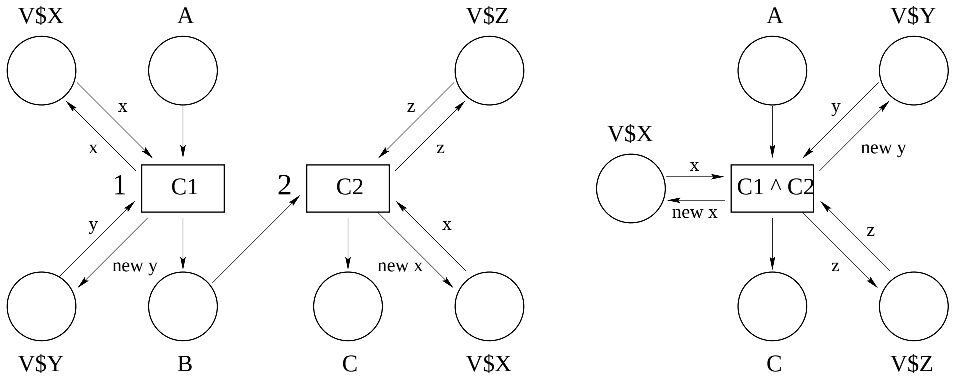

Figure 22: Merging of two successive net transitions.

If the condition can be applied to all net transition pairs, the atomic section collapses to one net transition. Whenever the condition can not be applied, the first net transition of the inspected pair is left as is and the second net transition becomes the first net transition in the new pair to be inspected. Net transitions which might belong to different atomic sections must not be merged. Therefore each critical transition must stop merging even though it does not necessarily force creation of a dynamic atomic section. With additional analysis this condition can be weakened. For example if an SDL transition only has one critical transition, it can not cause creation of an dynamic atomic section, thus it can be ignored. Branching control flow due to Decision, While and Join transitions creates a minor complication. When a Decision transition is encountered, merging must be stopped just before branching, each branch should be merged separately and net transitions following the conditional branches can also be merged separately. The other option is to try to merge the net transitions following the conditional branches to each branch. A While transition is handled in the same way, the net transitions inside the loop are merged separately. A Join transition just aborts merging as does any control place which can be jumped into with a Join transition. Generally statements modeled with mutually exclusive net transitions interrupt merging, because their models contain branching. Such statements include, in addition to the already mentioned statements, SET, RESET, OUTPUT and NEXTSTATE statements. OUTPUT statements are quite common, but they can be modeled with only one net transition, if all message queues are contained in the same place. Obviously also NEXTSTATE statements are common, and modeling them with one net transition is unfortunately hard, maybe even impossible. Merging should also be stopped after merging a Call transition, except if the called procedure has no SDL states. The implementation of merging does not have very high priority, because it only offers a linear reduction of the size of the reachability graph. It only reduces the length of atomic sections which are already mutually exclusive. It would, however, still be a worthwile improvement.

5.3 Other Optimizations of the Net Model

By modifying the net model both the size and the number of states in the state space can be reduced. Currently only reductions of the number of states are implemented. Creating atomic sections is an example of such reduction.

5.3.1 Separation of Control and Data

This approach addresses the size of states and also has indirect impact on the number of states. An SDL program can contain a great amount of data which does not have any effect on the execution paths through the program. This data can be left out of the net model to make states smaller. The number of states will also decrease. This comes from the fact that modification of such data creates unnecessary states, because the difference in the data can be the only difference between two states. Thus without the data the two states would be the same.

A variable can affect the control flow if a conditional part of the program sends a message and the variable is used in the entry condition for the part. A variable can also indirectly affect the control flow by influencing the value of a variable that is used in an entry condition of a conditional program part sending messages. A variable influences the value of another variable if it appears on the right hand side of an assignment affecting the second variable or if it replaces the value of the second variable by being a message parameter. Automatic analysis on these relations is certainly possible although not necessarily straightforward. Ignoring all data might have some use also resulting in a non-deterministic model. However, the model should be changed considerably to achieve that and the usefulness is not obvious.

5.3.2 User-defined Constraints in EMMA

A couple of methods for leaving some parts of the state space out of the reachability graph are implemented in EMMA. These are net transition firing counters and timer expiration windows. Because the constraints generated by these methods are embedded in the net model, the graph generator still generates the whole graph, but the graph is smaller due to the constraints.

A *firing counter* gives an upper bound for the number of times a net transition can be fired in any path in the graph. Let's call the reachability graph generated without the constraint the unconstrained graph. Those nodes of the unconstrained graph that can be reached *only* by violating the upper bound are left out from the constrained graph. A counter is implemented with a counter place and two additional arcs that are connected to the constrained net transition. The place contains an unary counter token. Each time the constrained net transition is fired, the additional arcs increment the value of the unary token (Figure 23). When the upper bound of the counter place has been reached, the net transition can no longer become enabled. With firing counters it is easy for example to say how many times a certain SDL transition can be executed by placing the constraint on the Input transition.

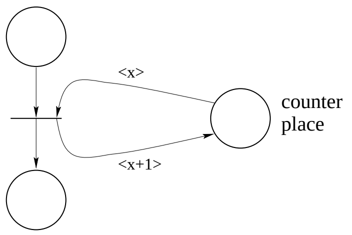

Figure 23: Limiting the number of firings of a net transition.

Timer expiration windows are used to tell in which parts of a process execution a timer can expire. First the timer is given the permission to expire as a side effect of a net transition, and later the permission is withdrawn in the same way at some net transition (Figure 24). A timer expiration transition of course connects to many places, but only the `timer lock place` is shown in the figure. If atomic sections are used, expiration of timers can only happen when the execution resource is available. This is enough, because Output transitions are critical transitions and therefore there is at most one Output transition in a atomic section, and thus there is no such state of the queue that includes a timer generated message and is not reachable when expirations require the execution resource. With timer expiration windows possible expiration points can be further reduced. However, reducing the number of possible expiration points with respect to the control flow of the process that activated the timer does not directly reduce concurrency of the system, because it does not constraint the interleaving of the processes of the system. Due to synchronization between processes the timer expiration windows can still have significant impact on the number of global states in which a timer can expire.

Expiration windows are implemented with locking tokens in the global `timer lock place`, each process instance has one token for each timer declared in the process. Timer instance based constrainting is not possible. There is always a token in the `timer lock place` for a timer instead of leaving the place empty when the lock of the timer is closed. Thus the frame idea is used once more. This way it is not necessary that a lock closing always follows a lock opening and vice versa in every possible execution path, an open lock can

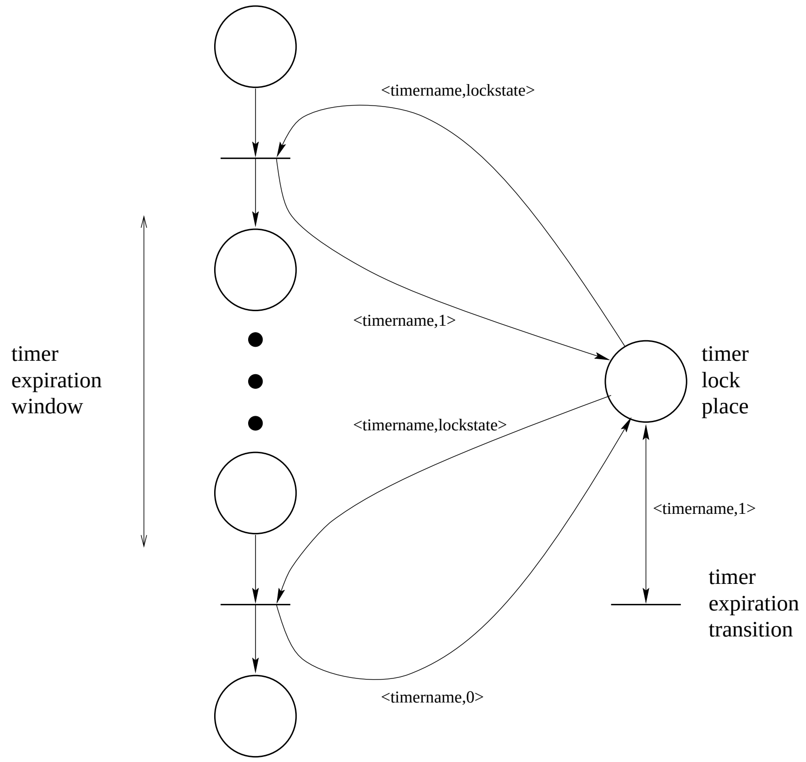

Figure 24: The timer expiration mechanism.

be opened and a closed lock can be closed. Of course if in some execution path a timer expiration window is opened but never closed, unwanted state space explosion might result. It should be noted that in the current implementation a timer instance is not forced to expire in any window. It follows that when using timer expiration windows, there may be such finite paths with end states containing unexpired timer instances that would be present neither in a model without expiration windows nor in the actual system. The forced expiration is hard to implement, because due to many possible execution paths it can not be said what window will be the last one. One possibility would be to allow only one expiration window per timer. Then a lock closing would force expiration of the related timer.

While the idea behind timer expiration windows is simplistic and has nothing to do with real time concept, it is still a good compromise. Inclusion of real time or some approximation of it would require some kind of counter which would considerably expand the state space of a system model. Furthermore, there are no rules for distribution of the execution resource between processes, thus the next executed process is selected non-deterministically. This reduces the number of fixed partial orders between timers of different processes. Thus also the usefulness of real time is reduced.

In TNSDL the expiration time of a timer instance is given by adding an amount of time to the time of activation of the instance. Without real time modeling a fixed order between two timer instances occurs only if in all possible execution paths the activation of the timer instance with smaller or equal additional expiration time always precedes the activation of the other instance. Finding such instance pairs requires an extensive analysis. A fixed order between two timer instances can be modeled by chaining the corresponding timer expiration windows. This means that the timer expiration window of the latter timer instance can not open before the timer expiration window of the first timer instance has been closed. In this implementation closing a timer expiration window should also force the expiration of the correponding timer instance. With real time it would be possible to model the timer expirations accurately. Still it is not sensible to include real time in the models due to state space explosion. Real time should at least be an internal part of the used formalism to be practical.

Many new constraints can easily be introduced. For example the maximum number of hand process instances and accepted range of variable values are good candidates. It would be practical to implement constraints based on testers of PROD. This would include implementing a *cut* monitor state which would resemble a reject monitor state, but instead of stopping the graph generation when entering a cut monitor state the state would be deleted. Thus the states reachable only through cut monitor states would be left out of the constrained reachability graph. However, there is a problem. The state of a tester is defined to be an unary tuple in PROD. Because variable values are not stored in unary tuples, a tester can not be used to test the value of a variable directly from the variable place. It might be non-trivial to generalize the definition of a tester state, and thus it is better to implement constraints with firing conditions when a tester can not be used directly. Of course the state of a tester can be modified explicitly in the net transitions, but then testers would not have any advantage over firing conditions. If a contrained variable value is modified by an outgoing arc of a net transition, the expression annotating the outgoing arc must be copied to the firing condition, or else the old variable value is tested instead.

For example the number of hand process instances is not available in the current models. Thus it must be added. As long as this new information can be computed from the information already available in the net model, it is not harmful (Section 6.1). The number of hand process instances is not harmful, because each hand process instance is already represented by the data in the model.

In addition to these graph reducing constraints a couple of on-the-fly constraints is available. These are deadlock monitoring and timer consumption monitoring. Deadlock monitoring means that on-the-fly searching of deadlocks is used when generating the graph. The on-the-fly searching is activated easily by adding a trivial tester to the net. Timer consumption monitoring means that if a message generated by a timer is consumed implicitly, the graph generation is stopped. This is implemented with a tester that has a reject state. The net transitions that take care of implicit consumption of messages will cause the tester to enter the reject state if the consumed message is a timer message. It would be easy to implement on-the-fly searching of deadlocks which can be restricted to certain areas in the control flow. A tester with two states is needed. The areas would be defined in the same way as timer expiration windows. Also livelock monitoring can be implemented, user options would determine the number of states of a livelock tester.

All these constraints are defined in the option file. Linear temporal logic formulas [5] to be verified can also be given in the option file, from which they are directly copied into the net file. It might be worthwile to enable use of names of the source SDL program in a formula given in an option file. Then it would be possible for example to refer to a SDL variable with its real name instead of the name of the place containing the variable value in the net model.

5.4 Verification Based on Partial State Spaces

There are some algorithms in PROD that make it possible to avoid generating the whole state space of the system for example when searching for deadlocks.

Unfortunately, in order to use these advanced methods the net must be *unfolded*. Unfolding means transforming the high-level Pr/T net into the corresponding P/T net. Each transition instance of the Pr/T net will have a corresponding transition in the P/T net and each possible marking of a place of the Pr/T net will have a corresponding place in the P/T net. The unfolded net can be extremely large, in fact it seems probable that unfolding can not be used in EMMA for any non-trivial model. This means that state space reduction currently only can be achieved via constraints in the Pr/T net model, by using an algorithm that does not require unfolding or via direct manipulation of the source SDL code. Currently in EMMA some model based constraints are in use.

5.4.1 Partial Order Methods

It was discussed earlier how models of SDL transitions can be made atomic. Net transitions in atomic SDL transitions are always executed in a sequence. While concurrency certainly is greatly reduced this way, the basic problem of unnecessary interleavings is not solved, it is just moved to a higher level.

Now complete SDL transition models play the role of single net transitions. Atomization of SDL transition models has no effect on the number of SDL transition model interleavings.

The idea of partial-order methods [10] [30] is to remove interleavings that are redundant. In other words, if there are many interleavings of independent actions, all leading to the same state, it does not matter which interleaving is used. The goal of these methods is to select one arbitrary interleaving from each set of independent interleavings reducing the number of states considerably. Thus these methods could solve the problem of redundant interleavings of SDL transition models. However, it is not clear whether they solve the problem in practice, because it is hard to say what kind of independencies a partial order method can exploit. In general, partial-order methods usually only preserve deadlocks, but by guiding the reduction with additional constraints also other properties can be preserved.

These methods require unfolding, which is a problem. It might be possible to develop a method that can manage with only local unfolding. This would allow partial-order reduction without an enormous unfolded net. However, there is no guarantee that such a method can be invented.

It might be useful if the user could explicitly define independence between net transitions. Usually independencies are computed automatically during state space generation, which can be very time consuming.

5.4.2 On-the-fly Methods

If properties are verified using an *on-the-fly method*, the generation of the state space is stopped as soon as it is known that a property does not hold. There are many kinds of properties, some are local while others are global. Given a global state of a system, it can be determined whether a *local* property holds in the state or not without using any information which is not available in that state. The so called *safety properties* are local properties. These include for example limiting the values of a variable. When verifying *global* properties paths must be inspected instead of states. Therefore an automaton is required. For example properties stating that something *eventually* happens are global. Verification of certain properties even requires automata that read infinite paths. However, such an infinite path has a finite number of states and thus it must contain a loop. An important class of automata which read infinite paths is called *Büchi automata* [25]. For example verification of properties stating that something *always* eventually happens requires an automaton which reads infinite paths.

Local properties can be verified on-the-fly by simply checking that the property holds for all global system states found during the generation of the state space. If a state violating the property is found, the generation is stopped. If a global property should be verified, the property is first *negated* and an automaton representing the result is contructed. This automaton is called *property automaton*. Then this automaton is executed in parallel with the reachability graph of the system, which also can be seen as an automaton. The combination of these automata is called *product automaton*. Execution in parallel simply means that a state transition is only possible in the product automaton if it is possible in both automata. Thus only such states of the system automaton which can be reached in the product automaton are needed. The same applies to the property automaton. The system automaton does not have any acceptance states, thus the product automaton inherits its acceptance states from the property automaton. Because the property automaton represents the negation of the verified property, any path that is accepted by the product automaton violates the property to be verified.

There is a couple of ways to verify global properties on-the-fly. If the property automaton is fully constructed before verification, but the system automaton is constructed during verification, it can be said that verification is done on-the-fly. This approach has the drawback that even the property automaton can be quite large. Thus it is better to construct also the states of the property automaton only when they are needed [9]. It follows that both the system automaton and the property automaton may be only partly constructed when the property to be verified is found to be false.

Properties can of course be defined in many ways, usually using some temporal logic. A model checking algorithm is used to verify a property represented by a temporal formula. Such an algorithm can for example transform the negation of the formula into a Büchi automaton and check whether the product automaton accepts some path. Stating a property as a temporal formula is not always straightforward, more intuitive means would be useful. There might not be a way to present a property that is both formal and intuitive. Sometimes it may be even easier to give an automaton for a property rather than a temporal formula. One possibility is to define an interface which can transform a fairly simple language into temporal formulas. Such an interface would be a useful addition to EMMA, because PROD already can check linear temporal logic formulas on-the-fly.

5.5 Modification of SDL Specifications

All state space reductions can not be done automatically. Direct manipulation of the modeled SDL specification is often necessary to make the size of the state space manageable. Direct manipulation enables simplification of uninteresting parts of the system. By using proper environments this might even be done automatically (Section 6.2). With direct manipulation it is also possible to remove insignificant interleavings of SDL transitions. Atomic sections do not have any effect on these.

Simplification of system parts includes removal of insignificant behaviour and abstraction of system parts to create models of manageable size. Removal of insignificant behaviour of a process obviously is a very loose concept and depends both on the analyzed system and the system aspects which are important for the current analysis. Removal of data that does not have any effect on the behaviour of the system can be seen to belong to this category also. In practice it is easier to use a constant value than to remove the data. Abstraction of system parts is used when only a part of a system is analyzed. The other parts can be represented by an environment (Section 6.2).

If it is known that the execution order of some SDL transitions has no importance, some order can be fixed in the SDL specification. This reduces the number of possible interleavings of the SDL transitions and thus the size of the reachability graph. This situation can typically arise with instances of the same process. There are some symmetries between the behaviour of process instances of the same process, if all the instances have exactly the same role in the system. In this case the order in which the instances execute some operation of the process can be fixed. For example if some local initializations are made in the process, the order in which the process instances make them is probably unimportant. Superfluous interleavings can be removed by fixing the order of initializations in the SDL specification. Exploiting the symmetries might not be trivial, and therefore it is always a good idea to create only a few process instances of one process, if possible. Of course there can be some errors which can only be detected if there are a certain number of instances of a process.

5.6 Packing of States in the Reachability Graph Generator

It is vital to reduce the size of the states. However, with this method space can roughly be saved by a constant factor only, while partial-order reduction methods might give much more drastic reductions through state omissions. Nevertheless, the constant can be quite large in some cases and thus these methods are useful.

A safe option is to use one of the traditional packing algorithms that do not lose data [13]. This way the only difference is, that packed states are stored instead of whole states.

The other option is to use so called probabilistic verification [28] [29]. In these methods states are packed much more tightly, but some data is lost. Because data is lost, two different states might be identical after packing. This situation is called a collision and it results in omission of such states that can only be reached through the state that collided. The collided state is omitted also, because the collision makes it seem that it is already stored. Thus in probabilistic verification the main issue is the trade-off between size of a state and probability of state collisions.

Usually states are packed by hashing. Of course selection of the hash function is very essential, because a good hash function minimizes the collision probability. It is also very common to store the packed states in a hash table. Selection of the hash functions that determine the locations of the states in the table is maybe even more important than the selection of the hash function used for packing. Through careful selection of hash functions even identical packed states might reside in the same hash table without collisions. This requires that the functions take the whole state as input instead of the packed state. Good hashing functions reduce the state collision probability. Because state omission probabilities are usually estimates, better estimates provide tighter upper bounds for the state omission probabilities. Better estimates of course do not reduce the real probabilities.

A modification of the method described above is called bitstate hashing. The method replaces the normal hash-table with a large bitvector. Each bit corresponds to one state. This correspondence is again determined with a hash function. Because each state is represented by a boolean only, the hash table implemented with a bitvector should always be sparsely filled to guarantee low enough collision probability. Collision probability can be reduced by assigning more than one bit for each state. This is accomplished by using more than one hash function, each assigning one bit from the bitvector to the packed state.

6 Some Problems and Improvements of EMMA

There are still a few minor TNSDL language constructs which are modeled unsatisfactorily. For example, the modeling of the JOIN statement is not yet complete, because the received message is not removed until the end of the SDL transition is reached and a JOIN statement can jump to another SDL transition. Also the modeling of procedure invocations in expressions needs some additional work.

In addition there are some issues concerning the technical details of the current EMMA implementation. The net model generator is implemented with C++. Unfortunately, some properties of the C++ classes restrict the development of EMMA. This could have been avoided with better planning of programming details. In EMMA, every net transition corresponding to a TNSDL statement is a class instance. This is sensible, but the control flow is only represented by lists that are located in the father class instances of the leaf class instances. Thus the inspection of the control flow is hard especially for branches. There is enough information to generate the net, but an abstraction is needed for the convenient inspection of the control flow. In general the class instances are too isolated and thus retrieval of the needed information can be difficult.

Currently it is not possible to directly refer to the SDL source code in the user interface of EMMA. Therefore the use of options that require references to SDL statements can be a bit awkward.

An *interactive generation* could be implemented in order to avoid the generation of uninteresting paths and states. A primitive example is an interactive breadth-first-search. After generation of states that can be reached within a certain number of net transitions from the initial state, the user could prune the leaves of the current search tree by using constraints and manual selection. Then the new states reachable within a certain number of net transitions are generated starting from the leaves which were not eliminated by the user and so on. Of course there would be quite many leaves to prune and it would be difficult to decide how often the search is interrupted.

With the `filter` program EMMA can present the paths of the reachability graph as sequences of SDL statements. This is a rather detailed way to express paths and thus is not necessarily very easy to read in all cases. It would be useful to have different more efficient ways to visualize paths. The user could then choose the visualization method best suited for the current needs. For example, visualization of control flow only could be useful.

The reduction of net places in the model could enable reduction of net transitions. Thus the model and the corresponding graph generator executable would become more compact. For example all the message queues could be located in the same place. This would enable modeling of dynamic OUTPUT statements with only one net transition. Reduction in the number of net transitions used for implicit consumption would also be possible. In the current implementation there is one net place for each SDL state which is not really necessary.

6.1 Superfluous Data in the Models

In this section superfluous data in the current net models will be discussed. Without this data the models produced by EMMA would be more well-defined. A goal of the model generation in EMMA is to remove all concurrency that can be removed according to the independence of certain TNSDL statements from other processes and to collapse all atomic sections to a single net transition. The previously introduced optimal execution resource management scheme addresses the first requirement. The granularity of the net models is currently too coarse in certain situations as discussed earlier. Recursive length reduction of atomic sections addresses the second requirement. However, the presented length reduction scheme rarely allows collapsing of an atomic section to a single net transition.

Superfluous data could be defined in the following way. If a certain global state of a system model contains data that can not be accessed by any process of the system and the value of the data is neither constant nor redundant, the data should be considered superfluous. In short non-constant data of a global state that can not be computed from the system-related data of the state is superfluous. Data that directly models the system is system-related. If the only differences between two nodes of the reachability graph of a system model are related to superfluous data, these nodes should be collapsed. Thus the superfluous data is harmful, because it can make the reachability graph bigger.

There are two kinds of superfluous data in a model, data that exists only for bookkeeping inside the model and has nothing to do with the modeled system and data that models aspects of the system but can exist in a global system state where it can not be seen by any process. There are at least two items of data of the first kind, the unique message identifier and the `pid` that will be given to the next created process instance. Of these the unique message identifier is the bigger problem. The unique message identifier is incremented, every time a message is sent. Therefore any loop in the behaviour of the system that contains even one OUTPUT statement will not be a loop in the reachability graph generated from the system model created by EMMA, because the unique message identifier will not be the same after one round of the supposed loop. This is probably the most severe defect in the models created by EMMA.

What makes this defect very harmful is the fact that any loop in a system specified with SDL is bound to have at least one OUTPUT statement. Thus any loop will become an almost infinite sequence. In EMMA systems are currently analyzed with such environments that generate only finite execution paths, thus the defect is not yet crippling. The environments are created with SDL, and parts of an environment are normal processes. However, when trying to analyze infinite behaviours, the defect becomes very serious indeed. The information about the `pid` of the next created process is much less harmful, but will prevent finding loops that contain process creation and termination.

Obviously the use of a monotonously increasing counter to produce unique message identifiers and `pids` is a problem in detecting loops, because the identifiers will never get the same values again. The unique message identifiers of the messages in the queues should be acquired from the information of the current state of the modeled system. One way is to redefine the unique message identifier of a message as tuple $\langle pid, queueslot\rangle$, where `pid` is the process identifier owning the queue which contains the message and `queueslot` is the number of the queue slot containing the message. However, in this case all the atomic value tuples of the message parameters must be updated when the queue is reorganized. This is difficult, because the queue can contain any messages and different messages can have a different number of atomic values. This can be solved by changing the model so that all messages carry the same number of atomic value tuples. When a message does not need all the tuples, the unused ones are ignored, but they still exist in the `parameter` place. Because there is a constant amount of atomic value tuples for all messages, it is easy to update them. Unfortunately, much data can be moved with messages and therefore the constant must often be quite large. This enlarges the markings of the reachability graph nodes, because many messages probably carry only little or no data.

Another method is to provide a place containing a frame which holds as many tokens as there can be messages in the queues of the model. Each token is a tuple $\langle number, boolean \rangle$, where *number* is the running number of the token and *boolean* determines whether the *number* of the token is currently used as the identifier of a message. When a message is sent, the smallest number not currently used as the identifier of a message becomes the identifier of the sent message. When a message is consumed, the number used as the identifier will become available. The choosing of the smallest available number can be implemented by creating one mutually exclusive net transition for every number. Such a net transition will be enabled exactly when the number is available and all the smaller numbers are not available. Alternatively there could be an own set of identifier numbers for each process instance. When using this method the message identifiers can not be acquired from the information of the current state of the modeled system. Thus the method might generate a bit more states than the first method, but loops are still found. Both correction methods can be used also for the `pid` of the next created process.

Superfluous data of the second kind is represented by variables that are either not used any more or currently not readable. These variables include variables of terminated hand processes and variables of non-active procedures, meaning procedures that are not currently under execution. The tokens representing the local variables of a process should all be removed when the process is terminated. The removal is easy, because it is statically known which kind of process each Stop transition will terminate and what variables it has. The tokens modeling parameters of procedures are already created when a procedure is called and removed when the control is returned from the procedure. Tokens representing local variables of procedures could be handled in the same way. Again, this is fairly straightforward, because it is statically known which procedure each Call transition activates and what variables the procedure has. The same kind of argument applies to Return transitions that terminate procedures. At least tokens representing the local variables of a procedure should be removed when the process containing the procedure is terminated. However, this solution would only partly eliminate the problem. If the tokens are not removed, values of the variables will be part of the global state of the system model even after the process or procedure is terminated. This clearly can make the reachability graph bigger.

6.2 Environment Models

Environments are not really part of the modeled system, they are needed only to close the modeled system. A *closed* system has all parts it needs to become fully functional. In SDL this usually means that the system has the possibility to receive all the messages it needs to proceed. The environment will provide the messages that are not generated anywhere else in the system. Environments can represent the external parts of the system which are not implemented within the system model. Typical examples of external parts would be the user or users of an interactive system. When only a subsystem is modeled, an environment is used to represent other subsystems of the real system. This may be necessary if the whole system is too complex to be analyzed in one model. The environment will provide very crude models of those other subsystems, essentially the messages that the models of the other subsystems would generate if they were part of the analyzed model. These messages are here called *external messages*. Of course modeling one subsystem at a time obviously does not produce exactly the same execution paths that would occur in the model of the whole system. It is a complicated issue how to obtain executions of the whole system from the executions of the subsystems. Clearly modeled subsystems should be as autonomous as possible, which means that only a few messages must be generated by the environment. This way the loss of modeling accuracy can be minimized.

Currently the environments are given as SDL programs, and they do not differ from the SDL source that defines the system. Thus the environments are manually written simple SDL processes that provide external messages for the modeled system or subsystem and might contain some interaction with the system model if necessary. Very often the environment of a system is supposed to generate some requests which can come in any order. This can be achieved by making one process for each message the environment should generate. The sole purpose of such a process is to generate instances of the request. It may also receive the possible response to each request before sending another request. This way all the possible interleavings of the requests are tested. Because in practice it is rarely possible to model the whole behaviour of a system, it is best to constraint the number of generated requests. In many cases it is also useful to test some particular request orderings, that are known to be either rare and problematic or essential in some other way. In this case it is enough to create one process that models the wanted request ordering.

If the environment is such that it should respond to certain messages by making some computations and generating messages, in other words it has non-trivial behaviour, it should be viewed as a process in the system instead of environment. If a process is made simpler to enable a more simple model, it may be a matter of opinion whether it is called a simplified process or an environment modeling the process. This follows from the fact that there is no difference in implementation between the processes of the modeled system and the processes that define the environment.

The way environments currently are defined has some problems. If an environment only sends messages it can send them at any time which will make the size of the reachability graph very large. Especially if an environment process is used to generate an arbitrary amount of messages, it will probably generate some infinite paths, because when the target process consumes a message implicitly, the environment can always send a new message which can again be implicitly consumed and so on. Without the unique message identifier this kind of behaviour is merely a livelock, but with it a vast number of states are generated. If the environment receives some response from the system model before sending another message, much less state space expansion results. However, this approach introduces a new problem. If messages sent by the environment are implicitly consumed, no response is sent by the system model to the environment. It follows that the environment never sends another message. This could be circumvented by making such arrangements that an implicit consumption of a message generated by an environment is interpreted as a response to the message. This in turn would again enable the endless cycle containing message sending and implicit consumption.

The so called *cooperative environment* mentioned in [1] is a better alternative. The idea of the cooperative environment is that an external message is always implicitly available in any SDL state of a process model where an SDL transition for that message exists. Thus external messages are created *on demand* only. This environment realizes the assumption that a process can receive any external message at any time. This assumption is also true if the environment is composed of SDL processes which are dedicated to sending external messages to the system model. However, the messages sent by the SDL processes will appear in queues. This greatly expands the reachability graph. A cooperative environment does the same implicitly, external messages never need to be in queues.

A cooperative environment is a simple concept and of course it has some problems. But any environment is bound to have some problems if it is not designed especially for the system it is used with. This follows from the fact that a common environment model is always a crude abstraction of the behaviour of the system parts it represents. Simple cooperative environments for example can not take into consideration any order the external messages are supposed to follow. A cooperative environment also induces a substantial growth of the reachability graph due to the very assumption that there always are external messages available. Because external messages are created on demand only, the cooperative environment model does not reveal whether the external messages can be lost due to implicit consumption.

In a cooperative environment an Input transition that is used to receive external messages is enabled exactly when the control is in the SDL state of the Input transition. If a normal Input transition is enabled in an SDL state, any Inputs used to receive external messages are also enabled and the executed SDL transition model is selected non-deterministically. This situation differs from the principles of SDL, because in SDL at most one of the SDL transitions of an SDL state can be enabled. Still exactly one SDL transition of an SDL state is chosen to be executed, therefore the introduced non-determinism does not create any problems.

A modification of a cooperative environment could be that not all external symbols are supposed to be available, instead the user can define the order in which they become available. The next step is to adjust the availability of external messages per process. This approach is pretty close to those SDL environments, where message sequences are explicitly given, but queue related state space expansion is still avoided. Responses to external messages could be taken into account also. After a certain external message is consumed, next one could become available after a response is send to the environment. Also the number of available external messages could be controlled by the responses. Initially a certain number of an external message is defined to be available. When such external message is consumed, the number is decremented and when the response arrives, the number is incremented.

7 Conclusions

During the EMMA project Pr/T nets were found to be quite suitable for modeling TNSDL programs although there is no support for complex data types in the PROD analyzer. The EMMA analyzer can model and analyze the *complete* TNSDL language. Only the size of the generated reachability graph limits its application.

Fortunately there are many techniques which can substantially reduce the size of the reachability graph. The current net model optimizations are a good start, but they must be improved and new optimizations should be introduced. However, inventing and implementing efficient optimizations is not easy. The actual generation of the reachability graphs should be optimized also, but many state space reduction algorithms are defined for the P/T net domain and their implementation for Pr/T nets is not trivial. However, promising results have been acquired recently with packing of states and graph based state space models, which together can raise the efficiency of the state space representation with several orders of magnitude [13]. Unfortunately the size of a state space can be too large for any exhaustive verification methods. Therefore development of efficient non-exhaustive analysis methods is very important even though actual verification of properties is not possible with them. For example constraint-based, interactive and even random non-exhaustive methods can be developed. Some constraints are already implemented in EMMA. Even though exhaustive methods can not always be used, the reachability of system states is still a very sensible basis for an analysis tool [31].

In many analysis methods and tools the verification model is created more or less manually, and therefore the correspondence between the model and the actual implementation can be questionable. In the EMMA analyzer both the model and the implementation are generated from the same symbol table and control flow information. Thus the differences between the implementation and its model are minimal. The symbol table and the control flow information are extracted directly from the TNSDL specification by the TNSDL translator.

The user interface of the current version of EMMA is still very primitive. However, the results of analysis are already translated back to TNSDL which is an extremely important feature [3]. The notations and names are identical to those used in the TNSDL specifications. A user interface for definition of properties to be verified should be implemented. Currently the properties are defined solely in terms of the net models. When the user can define properties based on the names of the TNSDL specification, the Pr/T net part of the EMMA analyzer can be completely hidden.

EMMA has a great potential to develop into a user friendly, efficient system closely integrated into the programming environment of TNSDL. The main decisions made in the design of EMMA seem to have been successful and future work can be directed at improving the analysis methods and the user interface.

References

[1] Algayres, B., Lejeune, Y., Hugonnet, F., Hantz, F.: *The AVALON Project: A VALidatiON Environment for SDL/MSC Descriptions.* 6th SDL Forum, Darmstadt, Deutchland, October 1993.

[2] Braek, R.: *SDL Basics.* SDL'95 Forum Tutorial, 1995, pp. 11–29.

[3] Christensen, S., Jørgensen, J. B., Kristensen, L. M.: *Design/CPN - A Computer Tool for Coloured Petri Nets.* In Brinksma, E. (ed.): Proceedings of the third Tools and Algorithms for the Construction and Analysis of Systems (TACAS) workshop, Enschede, the Netherlands, April 1997.

[4] Clarke, E. M., Emerson, E. A., and Sistla, A. P.: *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.* ACM Transactions on Programming Languages and Systems 8 (1986) 2, pp. 244–263.

[5] Emerson, E. A.: *Automated Temporal Reasoning about Reactive Systems.* In Möller, F., Birtwistle, G. (eds): Logics for Concurrency - Structure versus Automata, Lecture Notes in Computer Science 1043, Springer-Verlag, Berlin, 1996, pp. 41–101.

[6] *ETSI: Use of SDL in ETSs.* Draft European Telecommunication Starndard 300414, 1995.

[7] *Functional Specification and Description Language (SDL).* Recommendations Z.100 and annexes A, B, C and E. CCITT Blue Book 1988.

[8] Genrich, H. J.: *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 1987, Springer-Verlag, pp. 207–247.

[9] Gerth, R., Peled, D., Vardi, M. Y., Wolper, P.: *Simple On-the-Fly Automatic Verification of Linear Temporal Logic.* In proceedings of the 15th IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, IFIP series, Chapman & Hall, Warsaw, 1995.

[10] Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem.* Ph.D. thesis, 1995.

[11] Grönberg, P., Tiusanen, M., and Varpaaniemi, K.: *PROD—A Pr/T-Net Reachability Analysis Tool.* Digital Systems Laboratory, Helsinki University of Technology, Technical Reports, No. 11, Otaniemi, 1993.

[12] Hogrefe, D.: *Validation of SDL Systems.* SDL'95 Forum Tutorial, 1995, pp. 95–105.

[13] Holzmann, G. J.: *State Compression in SPIN: Recursive Indexing and Compression Training Runs.* Proceedings of the Third Spin Workshop, University of Twente, the Netherlands, April 1997.

[14] Husberg, N., Ojala, L., Penttinen, O-M. and Vainonen, A.: *Report of a preliminary investigation: EMMA - an Extendible Multi Method Analyzer*, 1995, (not published).

[15] Husberg N.: *SDL Modeling with High Level Petri Nets.* Proceedings of the Workshop on Concurrency, Specification & Programming, Informatik-Bericht Nr. 69, Humboldt University of Berlin, September 1996.

[16] Husberg N.: *Formal Models for Real Programs.* Presented at the 8th Nordic Workshop on Programming Theory, 1996.

[17] Husberg, N., Malmqvist, M., Jyrinki, T.: *The Emma Analyzer.* Technical report, Digital Systems Laboratory, Helsinki University of Technology, (to be published).

[18] Jyrinki, T.: *Dynamical Analysis of SDL Programs with Predicate/Transition Nets.* Technical Report ???, Digital Systems Laboratory, Helsinki University of Technology, December 1997.

[19] Lewis, H. R., Papadimitriou, C. H.: *Elements of the Theory of Computation.* Prentice-Hall International, 1981.

[20] Manner, T.: *Integration of EMMA Programs fcf and filter to emma.* Special assignment, Digital Systems Laboratory, Helsinki University of Technology, March 1997.

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

[22] Smith, E.: *A Primer on High-Level Petri-Net Theory.* Humboldt University of Berlin, 1993.

[23] *Specification and Description Language (SDL) - SDL formal definition: Static Semantics.* Technical Report Z.100 F2 (1994), ITU-T, 1994.

[24] *Specification and Description Language (SDL) - SDL formal definition: Dynamic Semantics.* Technical Report Z.100 F3 (1993), ITU-T, April 1994.

[25] Thomas, W.: *Automata on Infinite Objects.* In Leeuwen, J. (ed.): Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics, Elsevier, Amsterdam, 1990, pp. 133–139.

[26] Varpaaniemi, K., Halme, J., Hiekkanen, K., and Pyssysalo, T.: *PROD Reference Manual.* Digital Systems Laboratory, Helsinki University of Technology, Technical Reports, No. 13, Otaniemi, 1995.

[27] Verilog: *GEODE technical presentation*, 1994.

[28] Wolper, P., Leroy, D.: *Reliable hashing without collision detection.* In Computer Aided Verification, 5th International Conference, pp. 59–70.

[29] Stern, U., Dill, D.: *A New Scheme for Memory-Efficient Probabilistic Verification.* In Gotzhein, R., Bredereke, J. (ed.): Formal Description Techniques IX: Theory, application and tools, Chapman & Hall, London, 1996.

[30] Valmari, A.: *A Stubborn Attack on State Explosion.* Formal Methods in System Design 1, 1992, pp. 297–322.

[31] Wolper, P.: *Where could SPIN go next? A unifying approach to exploring infinite state spaces.* Invited talk at the Third Spin Workshop, University of Twente, the Netherlands, April 1997.