Analysis and formal Verification of SDL'92 Specifications using Extended Petri Nets

Joachim Fischer, Evgeni Dimitrov and Udo Taubert

Humboldt-University Berlin, Department of Computer Science
Lindenstr. 54a, D-10099 Berlin

Abstract

In this paper extended Petri Nets, *SDL Time Nets*, are introduced and used as means to analyze and verify SDL'92 specifications. The extensions include an association of data structure with some net places and attachement of preconditions (guard functions) and time firing intervals to net transitions. With the help of these nets it is possible to describe effectively significant aspects of SDL specifications and to solve problems arisen from the modelling through ordinary Petri Nets. Additionally, data dependencies within a SDL specification can be taken into account. Thus, real SDL'92 specifications can be analyzed without any essential restrictions to the language constructs. The new extended Petri Net class is formally described.
The mapping of the SDL'92 language constructs into this net class is summarized as a set of rules. For analyzing and evaluating of these net models tool components are developed and implemented. The transformation of a SDL'92 specification into a corresponding net model as well as the transformation of the results back to the SDL-level takes place automatically. They remain unvisible to the user. The investigations presented here are done within the scope of the SDL Integrated Tools Environment *SITE*. The net tool is tested using two application examples, the modified *Inres* protocol and the *Sliding Window* protocol. A deadlock in the *Inres* protocol has been detected and is briefly described.

Keywords: *Extended Petri Net, Formal Specifications, SDL'92, SDL Integrated Tools Environment, Protocol Verification, Inres and Sliding Window Protocols*

Contents

1 Introduction **4**

2 Basic concepts for mapping SDL specifications into Petri Nets **9**
2.1 Petri Nets and their extensions. Net classification 9
2.2 Formal definition of Petri Nets 11
2.3 Transformation principles 13
2.4 Problems by mapping SDL specifications into PN 14
2.4.1 Data dependencies in PN 14
2.4.2 Representation of process input buffers and their management 15
2.4.3 Timer modelling 16

3 Representing SDL specifications using extended PN **18**
3.1 Theoretical aspects. Formal definitions 18
3.1.1 Time PN 18
3.1.2 Mapping rules. An example 22
3.1.3 SDL Time Net 26
3.1.4 Semantic description of important SDL mechanisms 31
3.2 Transformation SDL'92 $\longrightarrow$ SDL Time Net 34
3.2.1 Process Scheme 34
3.2.2 Specific transformation features. Resolution of some SDL constructs 37

4 Implementation aspects: the net tool **39**
4.1 Net tool components 39
4.2 Transformation and net generation components 41
4.3 Analysis and evaluation components 43

5 Applications. Protocol examples **44**
5.1 Inres protocol analysis 44
5.1.1 Description of the used *Inres* protocol 44
5.1.2 Verification results: detection of a deadlock 45
5.1.3 Modelling of local process variables as external data 46
5.2 Sliding Window protocol 48
5.2.1 Protocol description 48
5.2.2 Modelling of SDL signal parameters 50
5.2.3 Results 50

6 Conclusions **52**

A The SDL description of the used *Inres* protocol **58**
A.1 The starting specification 58
A.2 The modified process Initiator 66

B The SDL description of the used *Sliding Window* protocol **69**
B.1 The starting specification 69
B.2 Modelling of signal parameters for *Sliding Window* protocol 75

1 Introduction

Different Formal Description Techniques (FDTs) have been recently used for the specification of complex distributed systems.
A distributed system must achieve a common goal through a coordinated cooperation of distributed software components. They interact with each other through synchronization and information passing mechanisms. Coordination affords interactions in a special regular way so that interests of other components are to be taken into account.
The specification of an distributed system is a very complex task and it requires different FDTs. In particular the FDTs should satisfy certain critical requirements ([SAJKOWSKI87]) such as:

- *concurrency* means that in a system the elementary actions may overlap in time,
- *nondeterminism* as an impossibility to uniquely determine the future behaviour of the system knowing the future input events to the system,
- *time constrains* and *aspects* as timeouts, interrupts , etc.

When considering computer communication systems as a special kind of distributed systems the specifications of their protocols have to be correct, clear and verifying as far as possible. It is also important for the FDT's to be able to express specification ideas at many levels of detail as well as to manage the complexity by specification structuring. In order to be a suitable communication mean for the users the FDT's should support a textual and a graphical description and be simply acquirable. Most of the modern FDT's are based on mathematical founded calculi such as finite state machines, transition systems, set theory, predicate and temporal logic and that's why they satisfy the above critical requirements to a different degree. In particular object oriented specification formalisms have recently got a large application, because thereby company knowledge may be formalized in building blocks and it can be reused. Previously defined specification blocks being validated may be composed to specify new systems [BALDASSARI91].

Two important FDTs groups are:

- the Formal Specification Languages (SDL, Estelle, LOTOS) and
- the Petri nets (PN) and related topics.

The Formal Specification Languages being developed by international standardization institutes as ITU-TSS (International Telecommunication Union - formerly CCITT) and ISO (International Standardization Organization) are based on different formal semantics. Thereby an unambiguous interpretation of the specification is possible so that different analysis and evaluation methods can be based on simulation or mapping into other model classes (net models, queuing models).

One of the most widespread formal specification languages is SDL which was extended in its current object oriented version SDL'92 [ITU92]. In SDL a distributed system consists of different independent components, which communicate asynchronously by message exchange via channels. This communication may influence the current state of each component. Different abstraction levels allow the description of the structure of a complex system as well as the behaviour of such a system. A system is composed of a set of blocks connected by channels. Each block may contain subblocks or processes. A process is an extended finite state machine, the behaviour of which is specified by a state-transition graph. The execution of a process transition depends on the current state, the current input signal, and the values of local or imported variables. Several actions may be done during a transition: changing local data, sending signals, setting timers, creating new processes or calling procedures. The complete behaviour of a process can be divided into number of partial behaviors (services). The object-oriented extensions of SDL are based on strict distinction between types (classes) and instances (objects). The important extensions of SDL'92 are: inheritance concept (refinement/specialization), virtuality concept of structure and behaviour, remote procedure calls, generic parameters, language constructs for non-deterministic behaviour and signal multicasting.

Parallel to the work on specification languages experience was gained with the use of Petri net theory ([COURTIAT84], [PAULE85], [BILLINGTON88]). Petri nets (PN) are founded upon a notation of concurrency; they express non-determinism simple and support structuring and system description at different levels of abstraction. The description of information and the control flow take place at an abstract level and within the same model. Nets are usually presented in graphical form, but a textual description can be also derived. Extensions and modifications of PN make the representation of time aspects and the investigation of performance parameters of specified systems possible. The greatest advantages of PN are the solid mathematical foundation and the large number of analysing techniques. Modifications and extensions increase the expressive power of PN but make net models be equal to a Turing machine, where most significant properties are undecidable. Generally it is valid: the higher the expressive power of a net model the weaker its decision power. Because of that conflict any net class can be defined, which is exactly as suitable as a specification means as an analysis one.

In this paper extended Petri nets, *SDL Time nets*, are introduced and used as a means to analyze and verify SDL specifications. The term *verification* is interpreted here as a proof of correctness, completeness and consistence of SDL specifications. The extensions of SDL Time nets are to be found in the following:

- data structures may be attached to some net elements,
- a precondition (guard function) and a firing interval expressed by means of two deterministic values may be associated to transitions.

With the help of these extensions it is possible to describe effectively significant aspects of SDL specifications as well as to solve problems, arising from the modelling through ordinary PN. In particular following problems can be successfully solved:

- modelling of data and their influence upon the system behaviour,
- buffer management in processes and channels (input, save, priority input etc.),
- process instance management and
- representation of time aspects including timer modelling.

Our approach combines the advantages of the specification language SDL with those of Petri nets theory. It has following advantages and particularities:

- Mapping SDL specifications into analyzable net class;
- Transformation follows automatically and it is not visible to the user;
- Net model analysis corresponds to an *exhaustive simulation* ([ALGAYRES93]) by generating the reachability graph (RG);
- The results of the analized net model are transformed automatically back to the SDL-level and they can be evaluated with other suitable tools;
- Specifications can be analyzed without any essential restrictions of SDL'92 through embedding of the net analyze tool into SDL environment ([FISCHER93], [FISCHER ET AL.93]). Especially data dependencies can be taken into account.

Thereby inconsistencies and errors of SDL specifications can be revealed before the specified system is actually implemented.
With the means of reachability analysis two kinds of properties of distributed systems can be investigated. On the one side it can be stated if some wishable system activities are really possible. On the other hand all possible states and state transitions can be snapped through reachability analysis in order to locate sets of states; this can be for example:

- terminal states (deadlocks) are states, from which no other state can be reached,
- states which should be never reached and
- reproducible states.

Unlike a *random simulation* [ALGAYRES93], where only specification errors can be found, which occur with high probability, the reachability graph analysis of a net model gives an absolute evidence. By this kind of exhaustive simulation all alternatives of each state transition are analysed in a backtracking mode. Thereby the set of examined state transitions and system states grows enormously; but when no

error can be found when finishing the simulation it is proved that specification is correct. For the successful reachability graph analyzing a significant preposition is its finiteness.

The investigations presented in this paper are done in the scope of the development and implementation of an program environment for SDL'92 called SITE (SDL Integrated Tools Environment) [FISCHER93]. The environment consists of tools that support the whole process from designing through simulation and analysis to target code generation of SDL (fig.1). All tools are realized as standalone components,

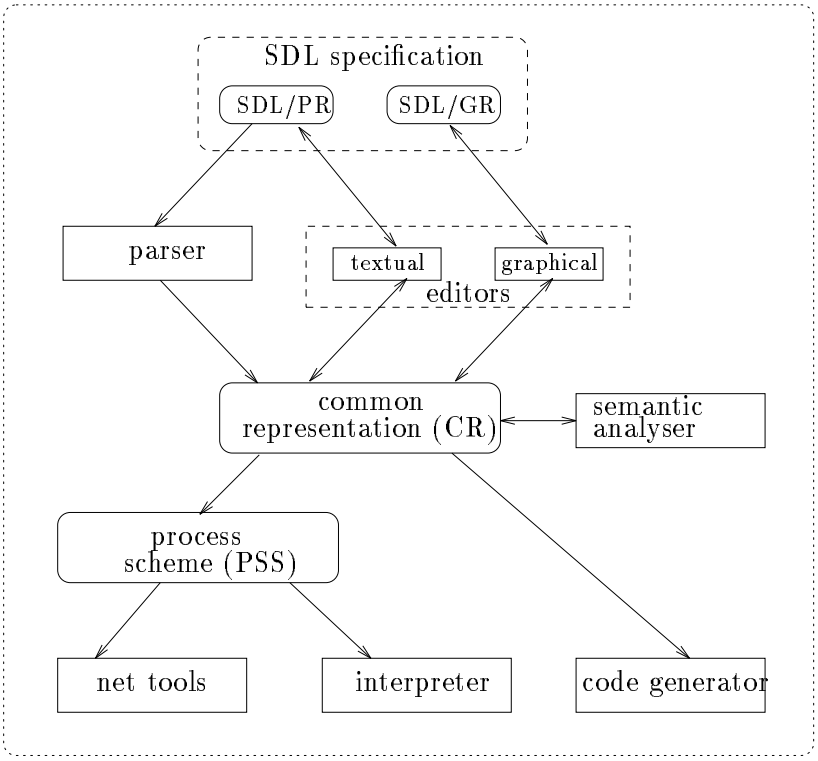

Figure 1: SDL Integrated Tools Environment (SITE)

which communicate via an uniform exchange format *common representation* (CR) [SCHADE94]. The separate tools can be grouped according to their tasks in:

- *front-end* tools, which transfer graphical or textual SDL specifications into CR and back and
- *back-end* tools, which do further processing of the CR.

To the front-end tools belong a *parser*, a *syntax oriented textual editor* for SDL/PR and a *graphical editor* for SDL/GR. Significant back-end tools are the *code generator*, the *interpreter* and the *net tool*. Between front-end and back-end tools the static

semantics analysis takes an exceptional position. The *semantics analyser* reads CR-Files and checks the semantics correctness of its content. All back-end tools don't need to testify the semantics correctness any more. The interpreter and the net tool do not get their input directly from CR. They are based on a more simplified, according to the requirements of analysis and interpretation, SDL intermediate format, the so called *process scheme* (PSS) [MÄTZEL92].

The paper is structured as follows.

First, the basic concepts of transformation of SDL specification into ordinary PN are outlined. Prepositions and transformation principles of the mapping SDL into PN are presented as well as problems and disadvantages of it are discussed.

Second, two extended Petri nets classes, Time Petri nets and SDL Time nets, are formally introduced. Mapping SDL basic concepts into SDL Time nets is described. A restricted version of SDL'92, SDL-PSS (further only PSS), serves as a starting point for the mapping. Transformation of specifically SDL'92 concepts in PSS is briefly sketched. The mapping of PSS into SDL Time net is summarized as set of rules. Further the semantics of significant SDL mechanisms is represented using SDL Time nets.

The next part of this paper considers the implemented net tool: transformation and net generating components, analysis and evaluating programs.

Finally, the net tool is tested using two application examples, the *Inres* protocol and the *Sliding Window* protocol. The effect of modelling of local process variables as well as the considering of large data dependences upon the analysis results are checked. A deadlock in the *Inres* protocol has been detected and is briefly described.

2 Basic concepts for mapping SDL specifications into Petri Nets

2.1 Petri Nets and their extensions. Net classification

Different variations and extensions of PN have been proposed. The early Condition/Event nets and their extension, Place/Transition nets (further only PN) serve as a basis for other higher level or more specialized PN models.

The modifications and extensions of PN concern separate net elements (places, transitions, arcs and tokens) as well as important net theory concepts such as marking, transition firing rules, transition firing strategy and conflicts. Thereby two trends of extension can be stated.
The first one enriches PN by an object structure, which is represented by individual tokens and it can be modified by transition firing. These nets have been summarized in the literature under the term of *High-Level Petri Nets* (HLPN) [JENSEN91].
The second trend line extends the PN with a time concepts; thereby time can be attached to different net elements: transitions, places, arcs or tokens. This class, here called *time-augmented PN*, can be divided in two groups:

- time-augmented PN, which have been arisen directly from PN only by adding time concept, and which have no other modifications (*Timed PN*),
- time-augmented PN, which have time concept as well as additional modifications, concerning net elements (*Modified Timed PN*).

In particular, the MTPN have a very high expressive power, but analyzing methods and theorems of PN-theory are not applicable without modifications. The kind of time delays is significant for time-augmented PN. Thus it can be distinguished between time-augmented PN with:

- *stochastic* time distribution, which are suitable to performance evaluation of distributed systems and
- *deterministic* time distribution (non-stochastic models), which are more suitable for specification and verification of time-critical systems.

The above classification (including important representatives) is given in fig. 2.

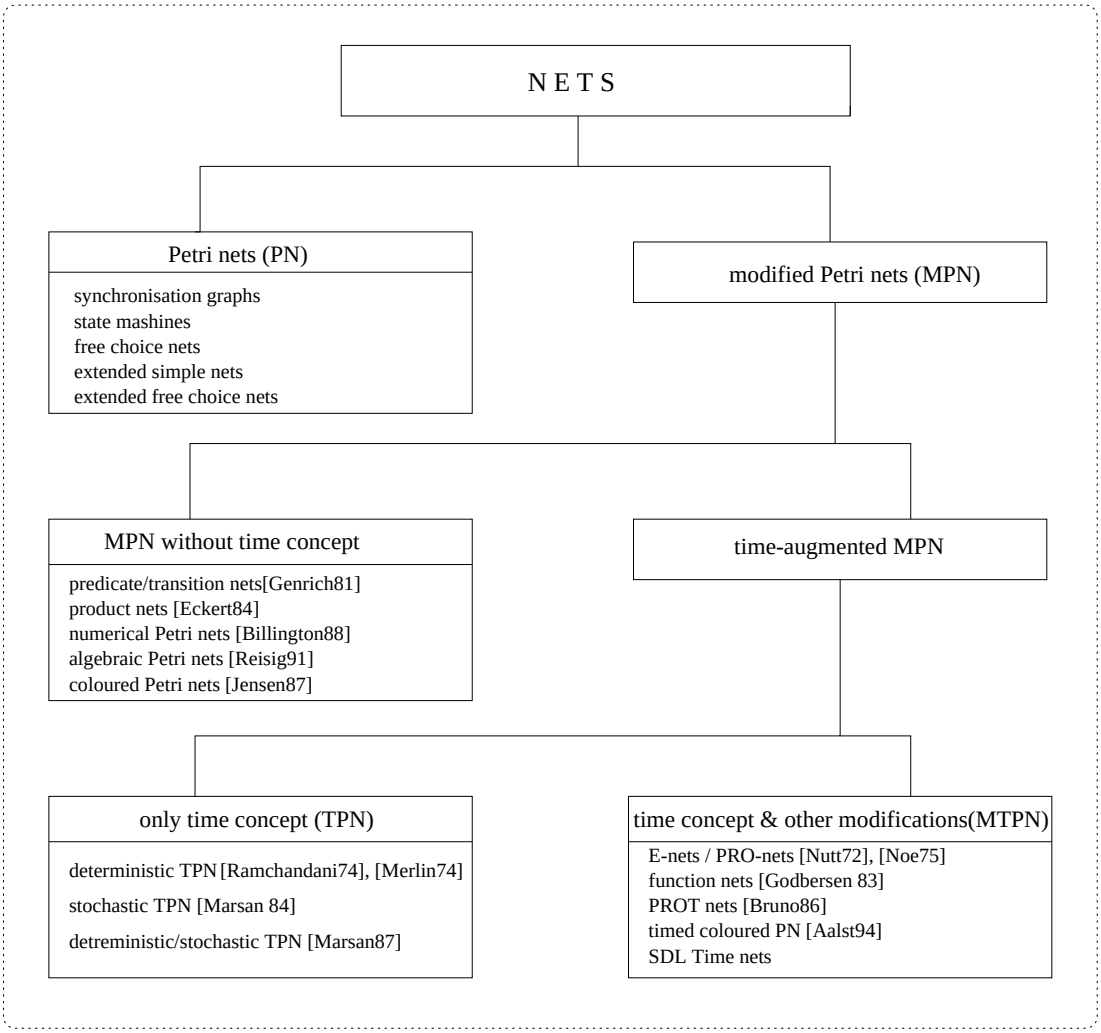

Figure 2: Classification of PN modifications

Several attempts of analyzing and verifying language specifications in SDL ([De Cindio84], [Comparin85], [Kettunen88],[Augustin89]) using PN and their extensions have been made in the past years. Some of them are based on ordinary PN which is resulting in very large and complicated net models, even in case of rather simple specifications. Important aspects such as time and data dependencies cannot be taken into account. Other approaches consider a subset of specification languages and they derive results, which are only restricted valid. In both cases practical applications can be restrictively investigated.

Hereafter important definitions and terms of PN theory have been formally introduced. They serve as a foundation for defining SDL Time nets in section 3.

2.2 Formal definition of Petri Nets

All definitions here follow [STARKE90].

Definition 2.1 (net)
The triple $N = [P, T, F]$ is called a net, iff

1. *P and T are finite non-empty sets with $P \cap T = \emptyset$,*
2. *$F \subseteq (P \times T) \cup (T \times P)$ is a binary relation, where:*
   $dom(F) \cup cod(F) = P \cup T$. Thereby $dom(F)$ (resp. $cod(F)$) is the set of all net nodes $x \in P \cup T$, which appear in a ordered pair of F on the first (resp. second) position.

Definition 2.2 (Petri Net)
The quintuple $\mathcal{N} = [P, T, F, V, m_0]$ is called Petri Net (place/transition net), iff

1. *$[P, T, F]$ is a net,*
2. *$V : F \longrightarrow \mathbf{N}^+$*
3. *$m_0 : P \longrightarrow \mathbf{N}$.*

The elements $p \in P$ are called *places*, $t \in T$ *transitions*, F is a *flow relation*, $V(f)$ is a *weight* of the arc f and m_0 is an *initial marking* of $\mathcal{N}$. For each place $p \in P$ the set $Fp := \{t | t \in T \wedge tFp\}$ respectively $pF := \{t | t \in T \wedge pFt\}$ is the set of the *pretransitions* respectively *posttransitions* of p. In analogy, the set Ft resp. tF is called the set of the *preplaces (inputplaces)* resp. *postplaces (outputplaces)* of t.

Definition 2.3 (marking of $\mathcal{N}$)
An injective mapping $m : P \longrightarrow \mathbf{N}$ is called a marking *of $\mathcal{N}$.*

Definition 2.4 (t^+, t^- and Δt)
Let $\mathcal{N} = [P, T, F, V, m_0]$ be a PN. Each transition $t \in T$ induces the markings t^+, t^- and Δt for each place $p \in P$, which are defined as follows:

$$t^+(p) := \begin{cases} V(t,p), & \text{iff } p \in tF, \\ 0, & \text{otherwise,} \end{cases}$$

$$t^-(p) := \begin{cases} V(p,t), & \text{iff } p \in Ft, \\ 0, & \text{otherwise,} \end{cases}$$

$$\Delta t := t^+(p) - t^-(p).$$

Definition 2.5 (transition enabling and transition firing)
Let $\mathcal{N} = [P, T, F, V, m_0]$ be a PN, $t \in T$, and m be a marking.

1. *The transition t is enabled (may fire, has concession) by m, iff for all inputplaces $p \in Ft$: $m(p) \geq V(p,t)$.*
2. *An enabled transition $t \in T$ by m can be fired and the successor marking m' is defined as:*

$$m'(p) := \begin{cases} m(p) - V(p,t) & +V(t,p), \quad \textit{iff } p \in Ft \wedge p \in tF, \\ m(p) - V(p,t), & \quad \textit{iff } p \in Ft \wedge p \notin tF, \\ m(p) & +V(t,p), \quad \textit{iff } p \notin Ft \wedge p \in tF, \\ m(p), & \quad \textit{otherwise.} \end{cases}$$

With definition 2.4 is also valid: $m' = m + \Delta t$.

On the following a reachability relation $\xrightarrow{\star}$ will be introduced. It serves for defining of reachability graph for PN. Before the term *word* is to be defined.

Definition 2.6 (word)
Each finite sequence of elements of a non-empty set A is called word over A, the empty word let be e. The set of all words over A is noted as $W(A)$.

Definition 2.7 (reachability relation)
Let $\mathcal{N} = [P, T, F, V, m_0]$ be a PN. For markings m, m' of P, transitions $t \in T$ and words $q \in W(T)$, $qt \in W(T)$ the relation $m \xrightarrow{q} m'$ is to be inductive defined:

1. $m \xrightarrow{e} m' \iff m = m'$ *and*
2. $m \xrightarrow{qt} m' \iff \exists m''(m \xrightarrow{q} m'' \wedge m'' \xrightarrow{t} m')$.

Thereafter the reachability relation can be defined:

$$m \xrightarrow{\star} m' \iff \exists q(\ q \in W(T) \ \wedge m \xrightarrow{q} m').$$

When $m \xrightarrow{\star} m'$ is in $\mathcal{N}$ valid, m' is reachable from m in $\mathcal{N}$.
$R_{\mathcal{N}}(m)$ denotes the set of all markings being reachable from m in $\mathcal{N}$:

$$R_{\mathcal{N}}(m) := \{m' | m \xrightarrow{\star} m'\}.$$

Definition 2.8 (boundedness of PN)
A PN $\mathcal{N} = [P, T, F, V, m_0]$ is bounded, if their reachability set $R_{\mathcal{N}}(m_0)$ is finite.

Definition 2.9 (reachability graph of PN)
Let $\mathcal{N} = [P, T, F, V, m_0]$ be a PN. The reachability graph of $\mathcal{N}$ is noted as the graph $EG(\mathcal{N}) := [R_{\mathcal{N}}(m_0), B_{\mathcal{N}}]$, whose nodes represent markings, being reachable in $\mathcal{N}$ and whose arcs are elements of the set $B_{\mathcal{N}}$ being subscribed with transitions. Thereby

$$B_{\mathcal{N}} := \{[m, t, m'] | m, m' \in R_{\mathcal{N}}(m_0) \wedge t \in T \wedge m \xrightarrow{t} m'\}.$$

The triple $[m, t, m']$ notes an arc from a node m to a node m'; this arc is with t subscribed.

2.3 Transformation principles

The transformation of SDL specifications into PN consists of two tasks. The first one is related to the transformation of the syntax structure. The second task is more complicated; it concerns the transformation of the SDL semantics. Thereby the initial net, which represents only the transformed formal SDL syntax, is explicitly enlarged by some net parts. That's why it is possible to plead for a structure and behaviour concept of transformation.

Generally two transformation principles are distinguished ([GRABOWSKI90]):

- elementwise transformation and
- generalized transformation.

The structure of SDL specification is thereby flattened, so that only processes and channels form the new structure.

Within the first method the transformation realizes elementwise per SDL structure symbol of the specification. In consequence the SDL states are represented by net places, SDL operations (actions) - by net transitions and signals - by net tokens.

The generalized transformation is an oversimplification of the elementwise transformation. Thereby whole SDL state transitions are mapped into net transitions. This method is more efficient regarding the growth and the time of generating of the RG.

A combination between an elementwise and a generalized transformation is almost always used. Such a combined transformation of SDL specification into PN could happen in the following way. All SDL states incl. *start-*, *stop-* states and *label*'s can be mapped into net places. The signal consumption (*input*) as well as the signal sending (*output*) are transformed into net transitions. Each possible branch of a *decision* is mapped into net transition. The input places of each net transition consist of actual state place, as well as 0, 1 or more signal output places. Being structural units, the SDL processes can be mapped into subnets. For each process instance there is a corresponding subnet, so that just one token is at one of the state places. This token is called a *control token*. In such a way the process is always in a strong defined state. Per process each possible input signal type is mapped into an input place and each output signal type is mapped into an output place. The *process input buffer* is modelled by the means of signal input places. Per state place additional transitions are applied for not consumed input signal types. These transitions represent implicit *discard* operations. The output place of these transitions is again the actual state place.

Without going into details of the transformation (section 3.2) the following questions and problems can be stated:

- How can data dependencies in PN be represented in order to map, e.g., the choice of state transitions at the *decision* 's and *enabling condition*' s?

- The presentation of the process input buffer by a single net place doesn't correspond to its FIFO semantics. How can operations, such as *save* and *priority input* be mapped?
- The time behaviour of *channels* and the *timer* semantics (with the related *set* and *reset* operations) cannot be (effectively) modelled.
- Adequate modelling of dynamically generated process instances and *create* operations is not possible.

Some of these problems and questions are answered below; proposals are briefly discussed concerning their advantages and disadvantages.

2.4 Problems by mapping SDL specifications into PN

2.4.1 Data dependencies in PN

A state transition in a SDL process graph depends on the input signal as well as on the local variables. When using ordinary PN local data is not taken into account when transforming. So net conflicts arise in situations, where in SDL the evaluation of variables controls the splitting of processing (*decision*, *enabling condition*). *Conflict* within a net means that more than one transition are enabled under one and the same input event (i.e. all transitions have one or more common input places), but only one of them may be fired. For example, a conflict arises between net transitions which represent different *decision*-branches. There isn't such a conflict in SDL. So all possible *decision*-branches are equally to be considered by generating RG.

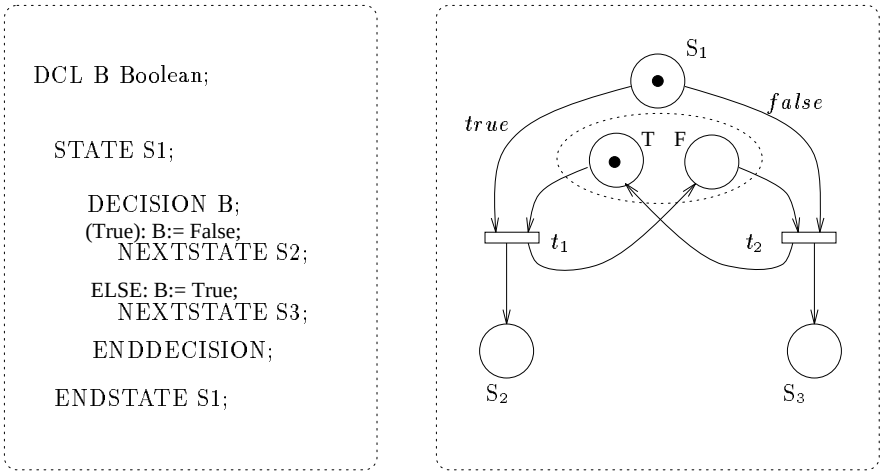

Figure 3: Conflict resolution by PN modelling of *decision* action

PN conflict decisions are possible through the addition of complementary places (the so called side conditions). The last one represents the necessary condition for the occurrence of an event. So a boolean variable can be modelled through a double place pair (fig. 3). Both possible values of boolean variable B are modelled through the place T (for TRUE) and the place F (for FALSE). Thereby one of the places is always marked with the complement of marking of the other place. The use of this approach to representing variables of another type (f.e. integer) leads to an enormous enlargement of the number of the net places . So data dependencies cannot be described effectively in PN using net elements only.

2.4.2 Representation of process input buffers and their management

The SDL signal input buffers are theoretically unbounded. For the purpose of real specifications it makes sense to study bounded signal input buffers. For this reason bounded buffers of finite capacity are to be modelled by PN. The maximal buffer length is determined experimentally.

The modelling of a bounded FIFO input buffer in a PN requires additional places and transitions. Figure 4 shows a simple net realization of a FIFO buffer with a length of 2 for two signal types S_1 and S_2 ([TAUBERT93]). The transition out_S_1 models the removal of signal S_1 from the buffer and the transition out_S_2 models the consumption of signal S_2. In analogy the transitions inp_S_1 and inp_S_2 model the reception of the signals into the buffer. Generally a FIFO buffer with a length n can be modelled

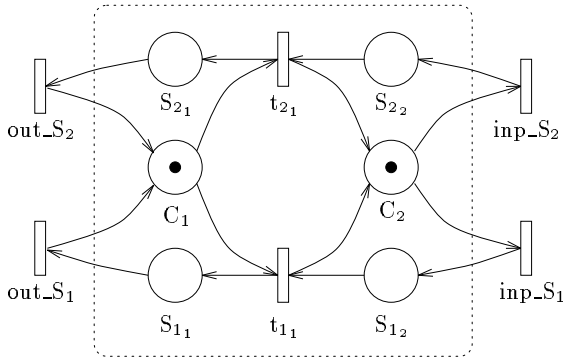

Figure 4: PN modelling a FIFO input buffer with a length 2 for two signal types S_1 and S_2.

in the following way. n places $C_1,\ldots,C_n$ are to be generated which initially have a token. A token within the place C_i means that the i-th position in the buffer is free. For each signal S_i (n-1) transitions $t_{i_1},\ldots,t_{i_{n-1}}$ and n places $S_{i_1},\ldots,S_{i_n}$, are generated. These places are initially not marked. Being a token on S_{i_j}, the signal S_i is on the j-th position in the buffer. The transition t_{i_j} models the movement of signal S_i to the j-th position in the buffer and it has the places $S_{i_{j+1}}$ and C_j as preplaces and

the places S_{i_j} and C_{j+1} as postplaces. All arcs have the weight 1. The transitions inp_S_i which 'bring' a signal S_i into the buffer, have S_{i_n} as a postplaces and C_n as a preplace. The transitions out_S_i which model the consumption of a signal S_i from the buffer, have S_{i_1} as a preplaces and C_1 as a postplace. One needs (1+s)*(n) places and s*(n-1) transitions for the modelling of a buffer with the length n for s signals. This construction leads to an essential enlargement of the net (a modelling of a FIFO buffer with length 5 needs for 5 signal types 30 places and 20 transitions). However many state transitions in the RG show only positional changes in the buffer. The result is an overexpansion of the RG. The SDL transitions *save* and *priority input* cannot be represented using the above technique.

2.4.3 Timer modelling

A timer is a stop watch which is set with an expiration time. Timers are controlled by *set* and *reset* operations. The setting of the expiration time takes place through a *set* operation. The expired timer is signaled to the process as an ordinary input. It's possible to stop the timer through *reset* before the end of expiration time. The modelling of timer is quite complicated because ordinary PN have no time concept. That why the *timeout* signals cannot be arranged in the signal queue since there is no watch resp. no tic in the net. A subnet, which interprets this semantics without details, has to be explicitly generated. Figure 5 makes the functioning of the timer subnet clear [LEHNERT93]. The subnet consists of 4 places and 5 transitions, where

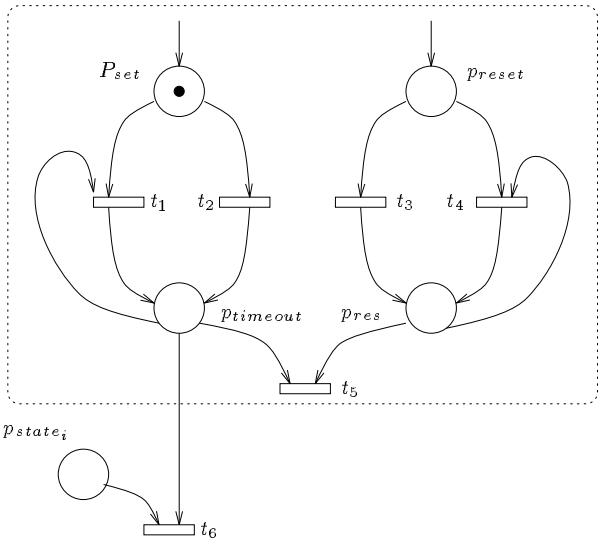

Figure 5: Modelling of SDL timer concept using PN.

place $p_{timeout}$ has a capacity of 1. The places p_{set} and p_{reset} are output places of those net transitions, which model SDL state transition consisting of the operations *set*

resp.*reset*. The execution of a *set* operation is represented by the firing of modelled transition and by the marking of place p_{set}. Now the transition t_2 can be fired and the token reaches the place $p_{timeout}$. In such a way a *timeout* signal can be received (transition t_6) in state i (place p_{state_i}). When in the meanwhile one more *set* operations are executed, the transition t_1 is fired, because the place $p_{timeout}$ has the capacity 1 and so the transition t_2 is blocked. Therefore the transition t_1 serves for 'swallowing up' of tokens which could be otherwise gathered on the place p_{set} and in such a way other transitions, which have the place p_{set} as a postplace, can be blocked. Analogous the *reset* operation is modelled (the right branch of the subnet in fig. 5). The *reset* operation itself takes place upon the transition t_5 through which the token from place $p_{timeout}$ is also 'swallowed up'.
So, an explicitly setting of timer with a given expiration time cannot be modelled in PN. An additional disadvantage is the enlargement of the size (each time with 4 places and 5 transitions per timer).

3 Representation of SDL specifications using extended PN

In order to solve successfully the problems discussed in section 2.4 extensions of PN are necessary in two directions:

- introduction of a time concept in PN and
- an attachment of data to some net elements.

Very suitable for the modelling of time aspects are the Time PN of [MERLIN74], in which two time values (the earlest and the latest firing time) are associated with each transition. The extension of the Time PN with data structures has lead to the SDL Time Nets. Both net classes are going to be formal defined below.

3.1 Theoretical aspects. Formal definitions

3.1.1 Time PN

The following definitions refers to [STARKE90].

Definition 3.1 (Time PN)
The triple $\mathcal{Z} = [\mathcal{N}, eft, lft]$ is called Time PN, iff $\mathcal{N} = [P, T, F, V, m_0]$ is a PN and eft, lft are mappings from T into $\mathbf{N}$ *so, that for all $t \in T$ it holds: eft(t) $\leq$ lft(t).*

The function eft means earlest firing time and the function lft - latest firing time of the transition t. The firing of transition t takes place at the earlest $eft(t)$ time units and at the latest - $lft(t)$ time units after enabling of t (so far as the enabling of t is not lost). If $eft(t) = lft(t) = 0$ the transition t must immediately fire after its enabling. Each transition t has a *watch*. The watch doesn't work (-1), if t is disabled at m. If t is enabled at m, the watch of t shows the elapsed time since t was enabled. After firing t resp. disabling of t the watch is off. A further preposition is the existence of a global time scale which synchronizes all watches. The state of time net depends consequently on the marking as well as on the watch position.

Definition 3.2 (state in a Time PN)
Let $\mathcal{Z} = [\mathcal{N}, eft, lft]$ be a Time PN.

1. *A mapping $u : T \longrightarrow \mathbf{Q}^+ \cup \{ -1\}$ is called watch function of $\mathcal{Z}$ at the marking m, if for all $t \in T$ it is valid:*

$$\begin{array}{lcl} u(t) = -1 & \Longleftrightarrow & t \text{ is disable at } m \text{ in } \mathcal{N}, \\ u(t) \neq -1 & \Longleftrightarrow & t \text{ is enable at } m \text{ in } \mathcal{Z} \\ & & \text{and } 0 \leq u(t) \leq lft(t). \end{array}$$

The watch function indicates the watch position of all transitions.

2. *A state of $\mathcal{Z}$ is a pair $z = [m, u]$, where m is a marking of P and u is a watch position of $\mathcal{Z}$.*

3. *The initial state z_0 of $\mathcal{Z}$ is a pair $z_0 := [m_0, u_0]$, thereby*

$$u_0(t) := \begin{cases} 0, & \textit{iff } t^- \leq m_0, \\ \textit{-1}, & \textit{otherwise,} \end{cases}$$

for all $t \in T$.

The so defined state of a Time PN can be altered in two ways: through an elapsed time and throug the firing of transition. A transition t can fire only if the thereby necessary number of tokens is present at the preplaces and its watch has passed $eft(t)$ time units.

Definition 3.3 (change of the state in a Time PN)
Let $z = [m, u]$ and $z' = [m', u']$ are two states in $\mathcal{Z}$.

1. *The transition t is enabled (may fire, has concession) in state z, iff*

$$t^- \leq m \textit{ and } u(t) \geq eft(t).$$

2. *The state z changes into the state z' by the time duration τ (notation: $z \xrightarrow{\tau} z'$), iff $m = m'$ and for all $t \in T$ it holds:*

$$u'(t) = \begin{cases} u(t) + \tau \leq lft(t), & \textit{iff } u(t) \neq \textit{ -1 }, \\ \textit{-1}, & \textit{otherwise.} \end{cases}$$

3. *The state z changes into the state z' by firing of transition t^* (notation : $z \xrightarrow{t^*} z'$), iff t^* is enabled in z, $m' = m + \Delta t^*$ and for all $t \in T$ it holds:*

$$u'(t) = \begin{cases} \textit{0}, & \textit{iff } t^- \leq m' \wedge [(t = t^*) \vee \neg(t^- \leq m) \\ & \vee (t^- \leq m \wedge Ft \cap Ft^* \neq \emptyset)], \\ u(t), & \textit{iff } t^- \leq m' \wedge t^- \leq m \\ & \wedge Ft \cap Ft^* = \emptyset \wedge t \neq t', \\ \textit{-1}, & \textit{otherwise.} \end{cases}$$

A state z is *reachable* in Time PN $\mathcal{Z}$ from the initial state z_0 if and only if there exist a sequence of rational numbers $\tau_0, \tau_1, \ldots, \tau_n \in \mathbf{Q}^+$, a sequence of transitions $t_1, \ldots, t_n$ and a sequence of states $z_0', \ldots, z_n', z_1, \ldots, z_n$ such that

$$z_0 \xrightarrow{\tau_0} z_0' \xrightarrow{t_1} z_1 \xrightarrow{\tau_1} z_1' \xrightarrow{t_2} \cdots \xrightarrow{t_n} z_n \xrightarrow{\tau_n} z_n' = z.$$

Definition 3.4 (boundedness of Time PN)
A Time PN $\mathcal{Z}$ is bounded, if only a finite number of markings appear in its reachable states.

In a Time PN infinite many states are reachable from the initial state already time duration. In consequence the corresponding RG is also infinite. On the following only these states of the Time PN are to be used whose watches have integer values.

Definition 3.5 (integer state)
A state $z = [m, u]$ is called integer, when u is a mapping into $\mathbf{N} \cup \{ -1\}$.

For Time PN it is valid ([STARKE90], [POPOVA91]) that each reachable state z in $\mathcal{Z}$ is integer-reachable in $\mathcal{Z}$, too. Under the preposition, that the Time PN $\mathcal{Z}$ is bounded, the set of all integer states of $\mathcal{Z}$ is finite and it can be calculated. Outgoing from z_0 for each reached state z the next state is constructed at a time lapse of $\tau = 1$ (iff possible) as well as all states, which arise from z through the firing of the enabled transitions. Because lft(t) has a finite value for each transition t, each marking can appear only in finite number of different watch positions of the transitions, the construction breaks up at a bounded net $\mathcal{Z}$.

Definition 3.6 (reachability graph of a Time PN)
Let $\mathcal{Z} = [\mathcal{N}, eft, lft]$ be a Time PN. The reachability graph of $\mathcal{Z}$ is noted as the graph $IG(\mathcal{Z}) := [IZ_{\mathcal{Z}}(z_0), BZ]$, whose nodes are elements of the set $IZ_{\mathcal{Z}}(z_0)$ of all integer states, being reachable from z_0 and whose arcs are elements of the set BZ of all triples $[z, t, z']$ with $z, z' \in IZ_{\mathcal{Z}}(z_0)$, $z \xrightarrow{t} z'$ and $t \in T$, as well as of triples $[z, 1, z']$ with $z, z' \in IZ_{\mathcal{Z}}(z_0)$ and $z \xrightarrow[1]{} z'$.

The Time PN is further extended by functions (*guard* functions), which can be attached to transitions.

Definition 3.7 (Guard Time PN)
The quadruple $\mathcal{W} = [\mathcal{Z}, W, w, w_0]$ is called Guard Time PN, when

1. *$\mathcal{Z} = [\mathcal{N}, eft, lft]$ is a Time PN with $\mathcal{N} = [P, T, F, V, m_0]$.*
2. *W is a finite set whose elements are called guard states.*
3. *for $\mathbf{B} := \{0, 1\}$ the guard function w is the following mapping:*

$$w : W \times T \rightarrow \mathbf{B} \times W.$$

4. *$w_0 \in W$ is an initial guard state.*

A state in a Guard Time PN consists of: the marking of the places, the position of all watches and the guard state.

Definition 3.8 (state in a Guard Time PN)
Let $\mathcal{W} = [\mathcal{Z}, W, w, w_0]$ be a Guard Time PN.

1. *A state of $\mathcal{W}$ is a triple $[m, u, d]$, whereby $[m, u]$ is an integer state of the time PN $\mathcal{Z}$ and d - an element of W.*

2. *The initial state z_0 of $\mathcal{W}$ is the triple $z_0 := [m_0, u_0, w_0]$, thereby $[m_0, u_0]$ is the initial state of the underlying Time PN $\mathcal{Z}$.*

A change of states can be initiated through an elapsed time and through the firing of transition. A transition t can fire only, if the thereby necessary number of tokens is present in its preplaces, the own watch has passed $eft(t)$ time units and the guard function w returns the boolean value 1.

Definition 3.9 (projection)
Let a set $D = D_1 \times \cdots \times D_n$ be a cross product of the sets $D_{i,(1 \leq i \leq n)}$. A mapping Π_i of D onto D_i is called projection, if for $d_i \in D_i$ it is valid :

$$\Pi_i([d_1, \ldots, d_n]) = d_i.$$

When the sets D_i are disjunct, so the projection onto the set D_i is signified with Π_{D_i}.

Definition 3.10 (change of the state in a Guard Time PN)
Let $z = [m, u, d]$ and $z' = [m', u', d']$ be states in the Guard Time PN $\mathcal{W} = [\mathcal{Z}, W, w, w_0]$.

1. *The transition t is enabled (may fire, has concession) in state z, iff:*

$$t^- \leq m \text{ and } u(t) \geq eft(t) \text{ and } \Pi_{\mathbf{B}}(w(d,t)) = 1.$$

2. *The state z changes into the state z' by the time duration τ (notation: $z \xrightarrow{\tau} z'$), iff $m = m'$, $d = d'$ and for all $t \in T$ it holds:*

$$u'(t) = \begin{cases} u(t) + \tau \leq lft(t), & \text{iff } u(t) \neq \text{ -1 }, \\ \text{ -1}, & \text{otherwise.} \end{cases}$$

3. *The state z changes into the state z' by firing of transition t^* (notation : $z \xrightarrow{t^*} z'$), iff t^* is enabled in $\mathcal{W}$ by z, $m' = m + \Delta t^*$ and for all $t \in T$ it holds:*

$$u'(t) = \begin{cases} 0, & \text{iff } t^- \leq m' \wedge [(t = t^*) \vee \neg(t^- \leq m) \\ & \vee (t^- \leq m \wedge Ft \cap Ft^* \neq \emptyset)], \\ u(t), & \text{iff } t^- \leq m' \wedge t^- \leq m \\ & \wedge Ft \cap Ft^* = \emptyset \wedge t \neq t', \\ \text{ -1}, & \text{otherwise} \end{cases}$$

and

$$d' = \Pi_W(w(d,t)).$$

The algorithm for generating of the integer RG works as for Time PN. Additionally the guard function is evaluated at each state transition and the new guard state is calculated. If W has finite many values and the underlying Time PN $\mathcal{Z}$ is bounded, so the Guard Time PN $\mathcal{W}$ has only finite many state, too; the algorithm finishes after a finite period of time.

Definition 3.11 (reachability graph of a Guard Time PN)
Let $\mathcal{W} = [\mathcal{Z}, W, w, w_0]$ be a Guard Time PN. The reachability graph of $\mathcal{W}$ is noted as the graph $IGW(\mathcal{W}) := [IZ_{\mathcal{W}}(z_0), BZ]$, whose nodes are elements of the set $IZ_{\mathcal{W}}(z_0)$ of all states that are integer-reachable from z_0, and whose arcs are elements of the set BZ of all triples $[z, t, z']$ with $z, z' \in IZ_{\mathcal{W}}(z_0)$, $z \xrightarrow{t} z'$ and $t \in T$, as well as of triples $[z, 1, z']$ with $z, z' \in IZ_{\mathcal{W}}(z_0)$ and $z \xrightarrow[1]{} z'$.

3.1.2 Mapping rules. An example

In order to make the following formal SDL Time Net definition comprehensible, the basic rules for mapping of SDL elements into net structures (Time PN) are summarized in this section. The mapping rules base on the prescriptions for PN (section 2.3) and they include further extensions and modifications.

Mapping rules:

1. For each SDL process (instance) a corresponding subnet (called *process net*) is generated. This process net consists of:
   - a *start* place pz_{start} and a *stop* place pz_{stop}
   - a place pz_i for each appearing state i,
   - a place psi for signal input buffer modelling,
   - a place pso_i for each output signal type i,
   - two places $ptset_i$ and $pttimeout_i$ for each timer i,
   - a place pc_i for each process i, which can be dynamically created by the underlying process.
2. A transition tz, which changes the state transition graph from state i to state j is mapped into a net transition t. Thereby the following cases can be observed (fig. 6):
   - When tz is a *input* transition, so gets t the places psi and pz_i as preplaces and pz_j as a postplace. When the input signal of tz is a *timeout* signal of timer i, so is $pttimeout_i$ instead of psi a preplace of t.
   - When tz is a *spontaneous* transition, so gets t the place pz_i as a preplace and the place pz_j as a postplace.
   - When tz is a *save* transition, so gets t the places psi and pz_i as well as preplaces and as postplaces.

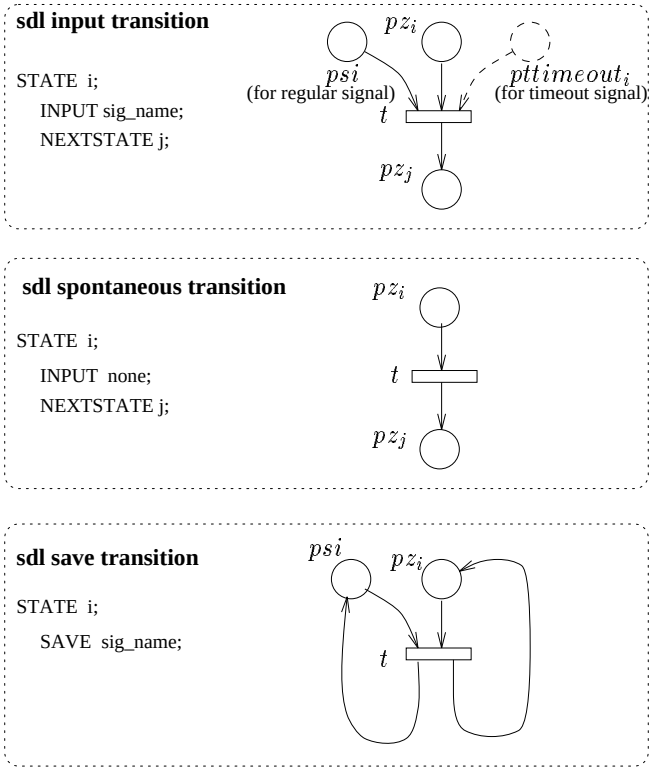

Figure 6: Modelling of important SDL transitions

3. The actions in the transition body of tz are modelled by the net transition t in the following way:

   - The output of a signal i to a process (action *output*) is represented so, that the transition t gets the place pso_i as a postplace.
   - The dynamic generation of the process i (action *create*) is modelled such that the transition t gets the place pc_i as a postplace.
   - The setting of a timer i (action *set*) is represented such that the transition t gets the place $ptset_i$ as a postplace; by resetting the timer i (action *reset*) transition t gets the place $ptset_i$ as a preplace.

4. Each timer is modelled by a subnet (fig. 7 a), consisting of two places $ptset_i$ and $pttimeout_i$ connected by a transition tt_i. The setting of a timer i (action *set*) is represented such, that the corresponding transition gets the place $ptset_i$ as a postplace; by resetting the timer (action *reset*) the transition gets the place $ptset_i$ as a preplace.

5. A channel between two processes j and k is represented by a transition $tg_{i,j,k}$ connecting the place pso_i of the process j with the place psi of the process k (fig. 7 b).

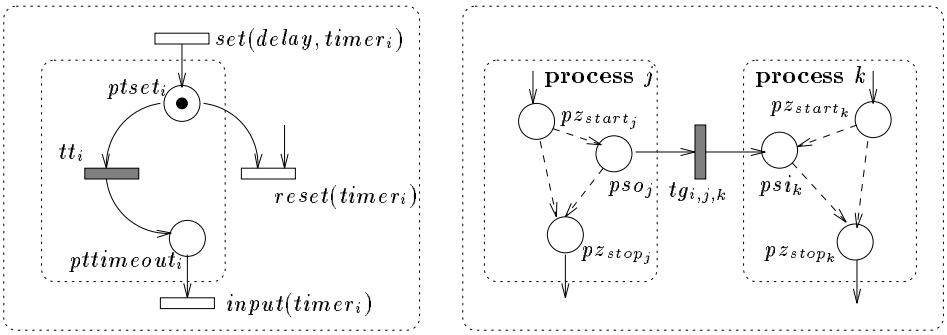

Figure 7: Modelling of: a) timer, b) signal sending between two processes

6. For each process type with dynamically created instances there is a subnet for instance management modelling. So, for the instance management of i instances of process type k a subnet with $2+i$ places is generated. The subnet includes the places $ready_k$,

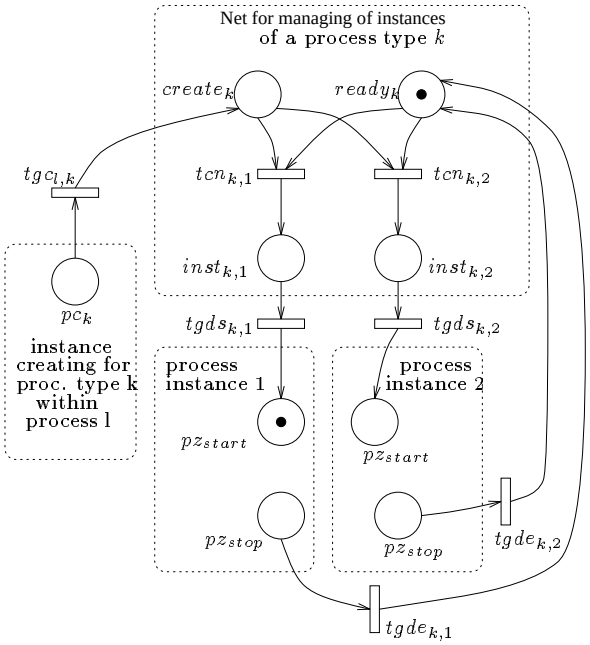

Figure 8: A net modelling two process instances (one passive instance at system start), which can be dynamically created; all transitions have the firing interval $[0, 0]$.

$create_k$ and $inst_{k,j,(1\leq j\leq i)}$. Further it has i transitions $tcn_{k,j,(1\leq j\leq i)}$, which get $ready_k$ and $create_k$ as preplaces and $inst_{k,j}$ as postplaces. The modelling of process instance creation of process type k through a *create* in the process l takes place by means of

a transition $tgc_{l,k}$, which has the place pc_k of process l as a preplace and the place $create_k$ - as a postplace. Moreover j transitions $tgds_{k,j,(1\leq j\leq i)}$ are to be generated. These transitions have $inst_{k,j}$ as preplaces and as a postplace - the place pz_{start}, which has been obtained for the start-state of the j-th process instance of process type k. Further j transitions $tgde_{k,j,(1\leq j\leq i)}$ are to be generated. They have $ready_k$ as a postplace and as a preplace - the place pz_{stop}, which has been obtained for the stop-state of the j-th process instance of the process type k. The place $ready_k$ is marked initially with so many tokens as much passive instances of the process type k exist at the system start. Figure 8 shows a subnet for managing two process instances, where one instance is active at system start.

```
PROCESS P1(1,1);

SIGNALSET S, S1;
TIMER t;

/* transition t0 (without signal input) */
START;
        NEXTSTATE ST1;

STATE ST1;
        /* input transition t1 */
        INPUT S1;
                SET(NOW+DELTA, t);
                OUTPUT S TO P2;
                NEXTSTATE ST2;
ENDSTATE ST1;

STATE ST2;
        /* input transition t2, consuming of a timeout signal */
        INPUT t;
                NEXTSTATE ST1;

        /* input transition t3 */
        INPUT S1;
                RESET t;
                NEXTSTATE ST3;
ENDSTATE ST2;

STATE ST3;
        /* save transition t4 */
        SAVE S1;
ENDSTATE ST3;

ENDPROCESS P1;
```

Figure 9: An example for a SDL process

The transformation of a SDL process into a Time PN is to be illustrated on one example. The process P1, being described in fig. 3.1.2, is to be translated into Time PN. The net from figure 10 has been automatically generated from this process.

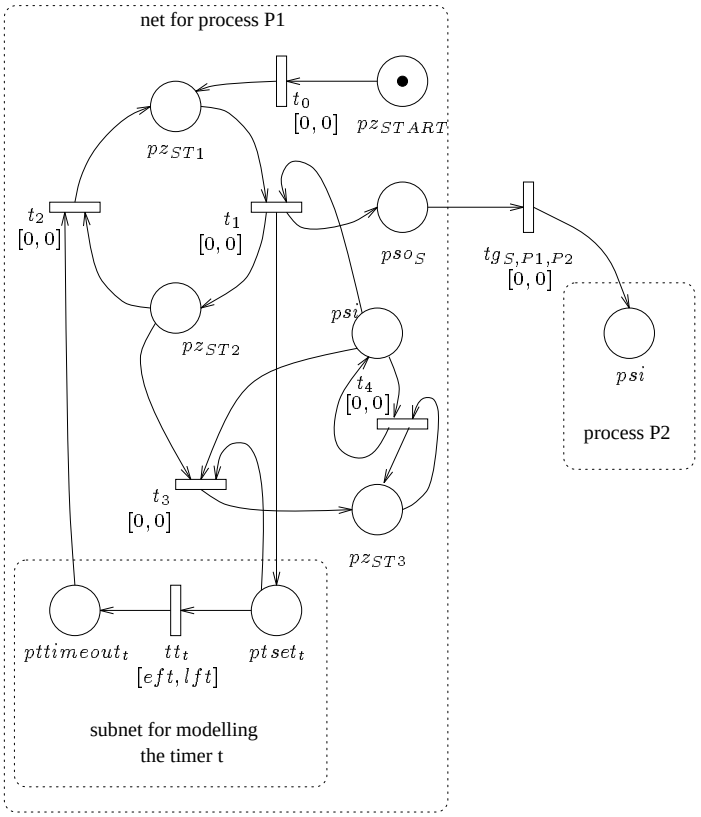

Figure 10: Automatically generated Time PN for process P1; the firing interval $[eft, lft]$ of transition tt_t (exceeding of the timer t) is given by the user

3.1.3 SDL Time Net

The SDL Time Net is established as a Guard Time PN $[\mathcal{Z}, W, w, w_0]$, i.e. the set W, the guard function w and the initial data w_0 have to be defined. These elements are to be defined according to the special features of the SDL semantics. In such a way these special interpreted Guard Time PN are called SDL Time Net.

In order to describe the signal input buffer of the processes the term *word* (definition 2.6, page 12) is used. A set of operations is introduced in order to make use of this term.

Definition 3.12 (word operations)
Let A be a non-empty set, $q = q_1 q_2 \ldots q_n$ and $p = p_1 p_2 \ldots p_k$ be words over A with $q_1, \ldots, q_n \in A$ and $p_1, \ldots, p_k \in A$ and e be the empty word.

1. $first(q) := q_1$ *resp.* $first(e) := e$,
2. $q \circ p := q_1 \ldots q_n p_1 \ldots p_k$ *and* $q \circ e := q$,
3. $length(q) := n$ *resp.* $length(e) := 0$,

4. *consume*$(q, m) := q_1 \ldots q_{m-1} q_{m+1} \ldots q_n$ *for* $q \neq e$ *and* $m \in \mathbf{N}$ *with* $1 \leq m \leq n$.

Each process (process instance) has been modelled by a subnet when transforming the SDL specification into a SDL Time Net according to the mapping rules (section 3.1.2). This subnet includes an input signal place which represents the input buffer. The number of the tokens on this input place corresponds to the number of the signals in the input buffer. The sequence and the type of the signals in the buffer are modelled with the help of the data defined below. The modelling of the FIFO behaviour of the input buffer takes place using a word whose length is given by the modeller.

Definition 3.13 (modelling of process input buffer)
Let the SDL system consist of p processes which are numbered from the beginning with 1. For each process j a FIFO buffer of the length l is modelled, $S_{j,(1 \leq j \leq p)}$ *let be the set of all possible signal types which can appear in the process j as an input. Let* $W_{j,(1 \leq j \leq p)} := S_j^{\leq l}$ *be the set of all words over* S_j *whose length is maximal l. Let the data set for modelling of the signal input buffer of a process j be*

$$D^{Signal}_{j,(1 \leq j \leq p)} := W_j \times \{1, \ldots, l\}.$$

Let the data set for the modelling of the signal input buffer of all process be

$$D^{Signal} := D_1^{Signal} \times \cdots \times D_p^{Signal}.$$

At the system start all input buffers are empty, the initial data d_0^{Signal} *of* D^{Signal} *is*

$$d_0^{Signal} := [[e, 1], \ldots, [e, 1]].$$

For each process type y, from which instances are dynamically created, a subnet for instance management (figure 8) has to be generated. In this subnet the token number of the place $ready_y$ gives the number of the passive process instances.

Definition 3.14 (modelling of the process instance management)
In the SDL specification d process types can be dynamically created. They are to be numbered with 1 from the beginning. For each of those process types y $(1 \leq y \leq d)$ *a FIFO buffer of the length* L_y *is modelled for the place* $ready_y$ *in the corresponding subnet for managing of process instances. Thereby* L_y *is equal to the maximal number of instances of the process type y.* $D^{Instance}_{y,(1 \leq y \leq d)}$ *is the set of all words over* $\{1, \ldots, L_y\}$*, where each element of* $\{1, \ldots, L_y\}$ *appears at most once in each word. Let be valid*

$$D^{Instance} := D_1^{Instance} \times \cdots \times D_d^{Instance}.$$

For each process type y the initial data $d_0^{Instance} \in D^{Instance}$ *consists of the order numbers of the process instances being passive at the system start. When* I_y *is the number of the passive instances of the process type y at system start, so* $p_{y,b,(1 \leq y \leq d),(1 \leq b \leq I_y)} \in \{1, \ldots, L_y\}$ *are the order numbers of these instances. Let be*

$$d_0^{Instance} := [[p_{1,1} \ldots p_{1,I_1}], \ldots, [p_{d,1} \ldots p_{d,I_d}]].$$

The two sets D^{Signal} and $D^{Instance}$ form the set of the 'internal data' in difference to the 'external data' which represent the data (signals, process variables) being defined by the modeller.

Definition 3.15 (internal data)
Let the internal data be

$$D^{intern} := D^{Signal} \times D^{Instance}.$$

the initial data $d_0^{intern} \in D^{intern}$ of the internal data be

$$d_0^{intern} := [d_0^{Signal}, d_0^{Instance}].$$

Definition 3.16 (external data)
Let $D^{extern}_{i,(1 \le i \le u)}$ be the set af all symbols of the type of the i-th external data elements. The set of all external data is to be defined as

$$D^{extern} = D_1^{extern} \times \cdots \times D_u^{extern}.$$

The initial data of the external data $d_0^{extern} \in D^{extern}$ is to be given by the modeller.

Some mappings and functions for an exact description of the following definitions have to be introduced.

- The mapping *tosdl* returns the corresponding transition of the SDL state transition graph for the transition t of the SDL Time Net, in case that there is a correspondence for t (otherwise -1 has to be returned) .
- The mapping *inputsignal* returns the signal type of the input signal of a transition belonging to the SDL state transition graph or - e, when the transition requires no inputs.
- The function *outputsignal* returns 1 for a transition t and data d, when the signal output, being realized by t, is possible at the actual internal data d. It means that for each process y, to which a signal output takes place, the length $length(\Pi_1(\Pi_{D_y^{Signal}}(d)))$ of the word, corresponding to the signal input buffer, must be not larger as the given buffer limit l minus the number of the output signals of the transition t to the process y. If this is not possible, the function returns 0.
- The mapping *prs* returns for a net transition t the order number of the process to which belongs the corresponding state transition $tosdl(t)$.

During the test whether a transition t can get a concession, one has to prove if the first element of the corresponding internal data is identical with the signal, that is consumed by the transition $tosdl(t)$ in the state transition graph of the process $prs(t)$. This takes place by the means of the below defined function w_{Signal}. Furthermore, the function w_{Signal} checks if the signal output, being realized by t, is possible at the actual internal data. With the removal or addition of a signal to the internal data a new data is generated according to the FIFO buffer behaviour.

Definition 3.17 (guard function for signal input buffer)
Let for the function $w_{Signal} : D^{Signal} \times T \rightarrow \mathbf{B} \times D^{Signal}$ *be valid for all* $t \in T$, $d = [[d_1, z_1], \ldots, [d_p, z_p]] \in D^{Signal}$ *with* $d_{j,(1 \leq j \leq p)} \in W_j$ *and* $z_{j,(1 \leq j \leq p)} \in \{1, \ldots, l\}$*:*

$$\Pi_{\mathbf{B}}(w_{Signal}(d,t)) := \begin{cases} 1, & \textit{iff } [(inputsignal(tosdl(t)) \\ & = first(\Pi_1(\Pi_{prs(t)}(d)))) \\ & \vee(inputsignal(tosdl(t)) = e)] \\ & \wedge[outputsignal(t,d) = 1], \\ 0, & otherwise. \end{cases}$$

$\Pi_{D^{Signal}}(w_{Signal}(d,t))$ *models the FIFO buffer of each process, i.e. with the removing or adding of a signal to* d*, a new content of the* d *has been generated according to the FIFO behaviour.*

$\Pi_{D^{Signal}}(w_{Signal}(d,t)) :=$

$$\begin{cases} subst(d, prs(t), [d_{prs(t)}, z_{prs(t)} + 1]), & \textit{iff } \Pi_{\mathbf{B}}(w_{Signal}(d,t)) = 1 \\ & \textit{and tosdl(t) is a save transition,} \\ subst(d, prs(t), [consume(d_{prs(t)}, z_{prs(t)}), 1]), & \textit{iff } \Pi_{\mathbf{B}}(w_{Signal}(d,t)) = 1 \\ & \textit{and tosdl(t) is a input transition,} \\ subst(d, prs(t), [d_{prs(t)}, 1]), & \textit{iff } \Pi_{\mathbf{B}}(w_{Signal}(d,t)) = 1 \\ & \textit{and tosdl(t) is} \\ & \textit{a spontaneous transition,} \\ subst(d, prs(t), [d_{prs(t)} \circ Timeout_i, z_{prs(t)}]), & \textit{iff } \Pi_{\mathbf{B}}(w_{Signal}(d,t)) = 1 \\ & \textit{and t is a timer transition,} \\ subst(d, to, [d_{to} \circ sig, z_{to}]), & \textit{iff } \Pi_{\mathbf{B}}(w_{Signal}(d,t)) = 1 \\ & \textit{and t is a channel transition,} \\ d, & otherwise. \end{cases}$$

Thereby the mapping subst is such defined: Let $A_{i,(1 \leq i \leq n)}$ *be sets,* $A = A_1 \times \cdots \times A_n$ *and for* $a = [a_1, \ldots, a_n] \in A$, $a_{j,1 \leq j \leq n} \in A_j$, $i \in \{1, \ldots, n\}$ *and* $a' \in A_i$ *be valid:*

$$subst(a, i, a') := [a_1, \ldots, a_{i-1}, a', a_{i+1}, \ldots, a_n].$$

The dynamical generation of processes is modelled with the below defined function $w_{Instance}$.

Definition 3.18 (guard function for process instances)
Let for the function $w_{Instance} : D^{Instance} \times T \rightarrow \mathbf{B} \times D^{Instance}$ *be valid for all* $t \in T$, $d = [d_1, \ldots, d_d] \in D^{Instance}$ *with* $d_{y,(1 \leq y \leq d)} \in L_y$ *and with the transition* $tcn_{y,l}$ *and* $tgde_{y,l}$ *for process instance management (introduced in section 3.1.2):*

$$\Pi_{\mathbf{B}}(w_{Instance}(d,t)) := \begin{cases} 0, & \textit{iff t is a transition } tcn_{y,l} \textit{ and} \\ & first(\Pi_y(d)) \neq l, \\ 1, & otherwise. \end{cases}$$

$\Pi_{D^{Instance}}(w_{Instance}(d,t))$ *adds the number* l *to* $\Pi_y(d)$ *at the firing of the transition* $tgde_{y,l}$. *When the transition* $tcn_{y,l}$ *fires, the element with the value* l *is removed from the word* $\Pi_y(d)$. *All other transitions have no influence over* d. *This goes to prove that:*

$$\Pi_{D^{Instance}}(w_{Instance}(d,t)) :=$$

$$\begin{cases} subst(d, y, d_y \circ l), & \text{iff } \Pi_{\mathbf{B}}(w_{Instance}(d,t)) = 1 \\ & \text{and } t \text{ is a transition } tgde_{y,l}, \\ subst(d, y, consume(d_y, 1)), & \text{iff } \Pi_{\mathbf{B}}(w_{Instance}(d,t)) = 1 \\ & \text{and } t \text{ is a transition } tcn_{y,l}, \\ d, & otherwise. \end{cases}$$

The management of the signal input buffer and of the passive process instances is modelled using the so called *internal guard* function.

Definition 3.19 (internal guard function)
The mapping $w_{intern} : D^{intern} \times T \rightarrow \mathbf{B} \times D^{intern}$ *is called internal guard function, when for all* $t \in T, d \in D^{intern}$ *is valid:*

$$\Pi_{\mathbf{B}}(w_{intern}(d,t)) := \begin{cases} 1, & \text{iff } \Pi_{\mathbf{B}}(w_{Signal}(\Pi_{D^{Signal}}(d),t)) = 1 \\ & \wedge \Pi_{\mathbf{B}}(w_{Instance}(\Pi_{D^{Instance}}(d),t)) = 1, \\ 0, & otherwise, \end{cases}$$

$$\Pi_{D^{intern}}(w_{intern}(d,t)) :=$$

$$\begin{cases} \Pi_{D^{Signal}}(w_{Signal}(\Pi_{D^{Signal}}(d),t)) \times \\ \Pi_{D^{Instance}}(w_{Instance}(\Pi_{D^{Instance}}(d),t)), & \text{iff } \Pi_{\mathbf{B}}(w_{intern}(d,t)) = 1, \\ d, & otherwise. \end{cases}$$

Whenever a net transition is able to fire in a given marking and given internal data, user defined *external* guard function w_{extern}, is called. If this function returns also 1, the transition can fire.

Definition 3.20 (external guard function)
The mapping $w_{extern} : D^{extern} \times T \rightarrow \mathbf{B} \times D^{extern}$ *is called external guard function.*

Definition 3.21 (SDL Time Net)
The Guard Time PN $\mathcal{W} = [\mathcal{Z}, W, w, w_0]$ *is called a SDL Time Net, when*

1. *the Time PN* $\mathcal{Z} = [\mathcal{N}, eft, lft]$ *with the underlying PN* $\mathcal{N} = [P, T, F, V, m_0]$ *is generated according to the mapping rules from section 3.1.2,*
2. $W := D^{intern} \times D^{extern}$,
3. *the guard function* w *is defined by the means of the functions* w_{intern} *and* w_{extern} *so that for all* $t \in T, d \in W$ *is valid:*

$$\Pi_{\mathbf{B}}(w(d,t)) := \begin{cases} 1, & \text{iff } \Pi_{\mathbf{B}}(w_{intern}(\Pi_{D^{intern}}(d),t)) = 1 \\ & \wedge \Pi_{\mathbf{B}}(w_{extern}(\Pi_{D^{extern}}(d),t)) = 1, \\ 0, & otherwise, \end{cases}$$

$$\Pi_W(w(d,t)) :=$$

$$\begin{cases} \Pi_{D^{intern}}(w_{intern}(\Pi_{D^{intern}}(d),t))\times & \\ \Pi_{D^{extern}}(w_{extern}(\Pi_{D^{extern}}(d),t)), & \textit{iff } \Pi_{\mathbf{B}}(w(d,t)) = 1, \\ d, & otherwise, \end{cases}$$

4. for the initial data w_0 is valid:

$$w_0 := [d_0^{intern}, d_0^{extern}].$$

The internal data set D^{intern} is always finite. If the user defined data set D^{extern} is also finite, then the set of states of the SDL Time Net $\mathcal{S}$ is finite, if the underlying Time PN $\mathcal{Z}$ is finite. In this case the corresponding reachability graph is also finite and the SDL Time Net model is analysable.

3.1.4 Semantic description of important SDL mechanisms

The mapping rules from section 3.1.2 have more pragmatical character. Below the semantical content of important SDL mechanisms are briefly sketched.

Signal input behaviour

For each process instance net there is exactly one signal input place. More tokens can stay in this place; they represent different signals. The tokens in a signal input place must also be identified and distinguished one from another. With the help of data structures, which represent the signals, and of suitable functions operating with these data structures, the FIFO regime of a signal input buffer is modelled, though in the net no distinguishable tokens can be found.
The concession of a transition t, whose preplace represents a signal input buffer, depends additionally on the value of the guard function w_{Signal}. With the help of this function is to be checked:

1. whether the first element of the corresponding internal data is identical with the signal, which is consumed from the transition $tosdl(t)$ in the state transition graph of the process $prs(t)$;
2. whether the internal data enable a signal output, i.e., whether the signal input buffer has enough free capacity to receive the output signals being output at $tosdl(t)$.

With the removal or addition of a signal to the internal data a new data is generated according to the FIFO buffer behaviour. Thereby different modifications of the input buffer content are undertaken in dependence of the modelled state transition (definition 3.17, page 29):

- a signal consumption takes place always at an *input* transition and so the first signal from the content of the input buffer has been deleted,
- at a *save* transition, the saved signal is retained in the input buffer and so the buffer content has been extended with the identifier of the saved signal,

- at a *spontaneous* transition a transition activation takes place without any signal reception and so the buffer content remains the same.

Modelling of dynamically created process instances

In the modelling of dynamically created process instances 'active' and 'passive' subnets have to be distinguished. If an instance has been executed, i.e. the (control) token is at the *stop* place, then this token is moved to the *ready* place. This subnet is now a passive one - it represents a passive process instance. Now this subnet can be again activated. By *create* the token is moved from *ready* place (fig. 8) to the init place ($inst_{k,j}$) and it makes the subnet active. The selection of the process instance and their initiation are controlled upon the external data structures. Such data structures have been attached at the *create* and *ready* places. So a receive buffer is attached at the *ready* place in analogy to the signal input buffer. In this buffer the subnet order numbers of the passive instances are processed according to the FIFO regime. The first element of the buffer determinates which of the initiating transitions ($tcn_{k,j}$) is to be fired. Thereby this first element is to be consumed. Such a transition removes all tokens from the initiating process net and it empties its signal buffer. The *ready* place includes as an initial marking so many tokens as passive instances of the process type exist at the system start. The dynamical process generating is controlled by the guard function $w_{Instance}$. At this managing a passive process instance is selected with the help of the data $d \in D^{Instance}$; at termination of dynamically generated process instances their order number has been put down back into d. Summarizing the management of the signal input buffers and of the dynamically generated processes can be represented by the internal guard function w_{intern}, which consists of $w_{Instance}$ and w_{Signal}.

Modelling of the SDL timer concept

In the SDL Time Net the timer modelling takes place upon a subnet and external data structures. The timer semantics itself is modelled through external data structures. Functions, which are attached to the corresponding net transition, model *set* and *reset* operations. At firing the transition, which models the operation *set*, the place $ptset_i$ (fig. 7 a) is marked. So it is modelled that the timer is active and that it is set to a given absolute time (integer delay). This value is equal to the firing interval $[eft, lft]$ of the timer-transition tt_i, given by the user. If a *reset* operation takes place within this interval the according net transition fires and it removes the token from the place $ptset_i$. So the transition tt_i looses its concession; the *timeout* signal cannot be received. The timer is not more active. When the transition tt_i fires at first, it corresponds to the expiration of the timer and to the following consumption of the timer signal.

Modelling of local process data and signal parameters

The modelling of local process data and signal parameters is possible to be made by the modeller. Here only the principle of the modelling of local process variables is

discussed. The proceeding of the modeller by the use of the net tool is described in more details in section 5. The modelling of local process data and signal parameters takes place according to an uniform scheme:

1. The local data (resp. signal parameters) has to be declared as C-variables (pointers to this variables).

2. For each modelled variables (resp. signal parameters) the following three user-defined *help functions* are to be given: *duplicate*, *compare* and *write*, which perform the corresponding operations with the modelled data;

3. Further user defined *application functions* can be given. They perform different actions with the modelled data (f.e. modification of variables, tests and s.o.).

```
PROCESS Initiator (1,1);

SIGNALSET ICONreq;

DCL counter Integer;
    MAX_Value Integer;
TIMER tc;

    START;
        TASK MAX_Value:= 4;
        NEXTSTATE Disconnect;

    STATE Disconect;
      INPUT ICONreq;
        TASK counter := 0;
        NEXTSTATE Wait;

    ENDSTATE Disconnect;

    STATE Wait;
      INPUT tc;
        DECISION counter< MAX_Value;
          (True): OUTPUT CR;
                  TASK counter := counter + 1;
                  NEXTSTATE - ;
           ELSE: OUTPUT IDISind;
                 NEXTSTATE Disconnect;
        ENDDECISION;

    ENDSTATE Wait;

ENDPROCESS Initiator;
```

Figure 11: A part of the SDL specification of process *Initiator* from *Inres* protocol

A part of the SDL specification of process *Initiator* from the *Inres* protocol example (section 5.1) is given in fig. 11. Figure 12 shows the corresponding SDL Time Net. The local process variable *counter* is modelled by the user. From figure 12 step 2 (the declaration of help functions *dup_counter*, *cmp_counter* and *write_counter*) and step 3 (the declaration of application functions *set_counter*, *test_counter* and *increase_counter*) are obvious. The attachment of the application functions to net transitions is to be given by the user.

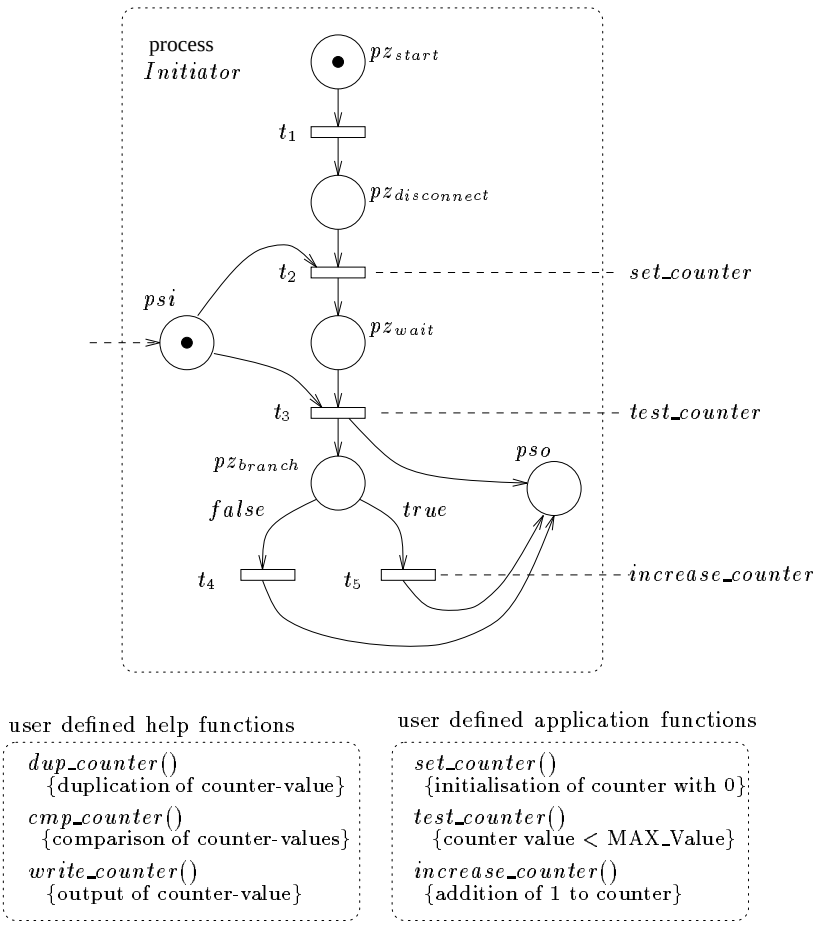

Figure 12: Subnet modelling of a part of the process *Initiator* from *Inres* protocol

3.2 Transformation SDL'92 $\longrightarrow$ SDL Time Net

3.2.1 Process Scheme

For the analysis of SDL'92 system specification processes and their communication structure are exclusively important. More abstract structural and logical levels, such

as SDL blocks, have no influence upon the behaviour of modelled system.

Our approach of SDL'92 specification analysis bases on the following three transformation steps (fig. 13). A SDL'92 system specification is first transformed into a

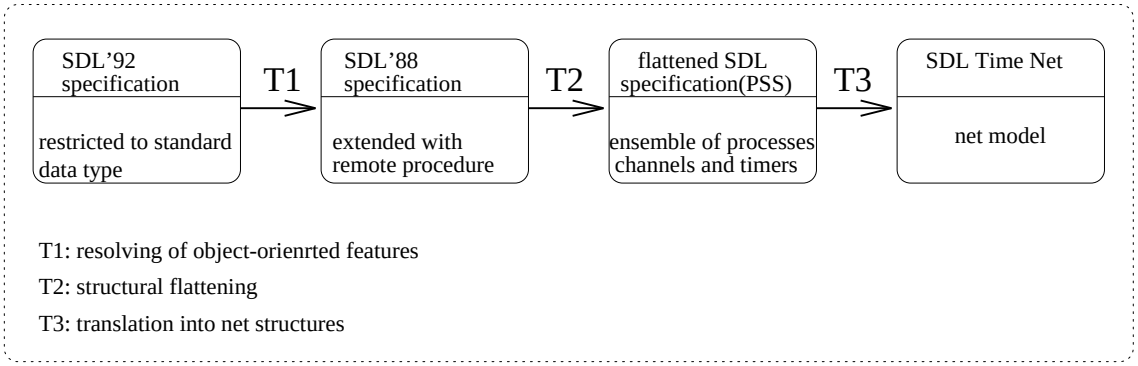

Figure 13: Transformation steps of SDL'92 into net structures

description according to SDL'88 [ITU88]. A second step translates this into a PSS description. The last one serves as a starting point for the mapping of the initial specification into a SDL Time Net (transformation step T3).

A PSS specification describes a system as a collection of processes and channels. The system interacts with its environment by sending and receiving signals. The environment is whatever outside the system.

A system in PSS has the following format:

```
system :        identifier
                declaration_list
                process_list
                channel_list
```

The further description uses a simplified *Backus-Naur Form*, where syntactic elements grouped by the using of curly brackets { and }, are optional.
The `declaration_list` consists of all signals and procedures which are defined on a SDL'92 system and block level. The `process_list` includes all process instance set types; the `channel_list` - the so called *pure channels*, i.e. channels between processes and between processes and the environment. In PSS all this language constructions are explicitly typewise defined, for which the SDL'92 offers the corresponding possibility. This is valid for processes, services, signals but also for procedures (in difference to SDL'92):

```
component:      identifier
                component_type
                {actual_context_parameter_list}
```

`Component` is here a substitute for processes, services and procedures. For processes a number of appearances has to be additionally given in order to guarantee the boundedness of the appropriate net. A signal can be specificated only by its name and type.

A type defined in a scope unit depends normally on the definitions within the same scope unit or in enclosing scope units. A type definition may be *parameterized* by so-called *formal context parameters*, so that it is partly or completely independent of definitions in the enclosing scope units. All formal context parameters must be bound by *actual context parameter*s. A (sub)type may be defined as a *specialisation* of another (super)type. A subtype inherits all the properties defined in the supertype definition; it may add properties and it may redefine *virtual types* and *virtual transition*. Only types and parameterized types can be used as supertypes. Inheritances are possible only from process, service, procedure and signal types.

`Component type` definitions for processes, services and procedures have some common elements:

```
component_type: selected
                formal_parameter_list
                formal_context_parameter_list
                declaration_list
                {specialization}
                {virtualities}
```

Additionally:

- process type definition consists of a service list,
- service type definition consists of two lists, for input and output channels, and of state transition graph,
- procedure type definition can include a state transition graph; this is optional.

The PSS processes contain always at least one service. Processes without services get one *main service* whose state transition graph is identical with the initial process transition graph. Component type definition for signals can consist of a list of formal parameters, of a list of actual contex parameters and eventually also of specializations.

PSS is to be distinguished by a simplified communication structure. While processes consist always of services, the communication structure is actually between services, and between services and their environment to be described. The signal-routes between services can be thereby ignored, because there's no semantic difference whether a signal is transferred within a process in different routes between two services. Processes are connected only through unidirectional channels; bidirectional communication structures must be accordingly splitted.

```
channel:        identifier
                delay
                connection
                signal_list
```

`Connections` in PSS represent an abstraction of signal routes. It has to discriminated between connections within processes and those between processes itself or between processes and channels. In consequence of the demand for unidirectionality there is also a difference between connections which lead to a process and those which lead from a process away. A channel can be defined as time-consuming by giving it an integer time delay. Channels in PSS can not be decomposed into channel substructures. Connections (as abstraction for signalroutes) are always non-delaying.

The important tasks of the first two transformation steps T1 and T2 are briefly summarized below.

In the transformation step SDL'92 $\longrightarrow$ SDL'88 the following tasks must be solved:

- resolution of referenced definitions and package-reference-clauses,
- replacement of constructs for instance building of system- and blocktype definitions with the corresponding resulting definitions,
- resolution of inheritances between type definitions as well as redefinition of virtual types,
- resolution of implicit signal routes and
- resolution of channel refinements.

All this tasks are implemented by the program component *semantic analyser* (in detail described in [SCHADE94]).

The transformation step SDL'88 $\longrightarrow$ PSS performs two main tasks:

- the first one concerns the resolution of block structures (also within channel substructures) and
- the second one concerns the simplification of the communication structure.

The further tasks hereby concern the realization of the correspondence between actual and formal context parameter as well as the treatment of inheritances and virtualities.

3.2.2 Specific transformation features. Resolution of some SDL constructs

The translation of a SDL system specification into a SDL Time Net takes place processwise, i.e. for each process a set of places has been generated and each transition of the process state graph has been modelled into one net transition. It is assumed that

the SDL processes consist of a common state transition graph. So a connected net graph arises in the course of the transformation. The last one bases on the mapping rules being introduced in section 3.1.2. Some SDL constructs have no direct correspondence in the net model. They are resolved at SDL level during the generation of internal net data structures.

Resolution of SDL transitions

The SDL *primitive* transitions: *input*, *save* and *spontaneous* are directly modelled into the net. The rest: *priority input*, *continuous signal* and *enabling condition* are to be resolved in the transformation and to be represented by primitives ones.

At an *enabling condition* in dependence with a boolean expression a signal consumption with a state transition or a *save* for this signal is undertaken. According to this an *enabling condition* has been expressed through an *input* transition and a *save* transition for the corresponding signal.

A *continuous signal* allows a transition to be initiated directly by a True value of a certain boolean expression. This construct is represented in the internal net data structures by a transition without signal input.

The modelling of the boolean expression takes place in both cases with the help of external data.

A *priority input* realizes a signal consumption with state transition undepend from the order of the signals in the input buffer. A resolution of this construct takes place according to [ITU92] through establishment of two new states for each state, in which a *priority input* is specified.

Resolution of SDL actions

The actions *output*, *create*, *set* and *reset* are directly interpreted into the net. They are represented only by additional input and output places of the corresponding net transition. The rest ones: *task*, *decision*, *procedure call*, *remote procedure call* and *label* are to be resolved.

The *task* serves primarily the manipulation of process variables (in contrast to its documentary function) and it is modelled with the help of external data.

For each branch of a *decision* a net transition has been created; data dependencies can be again introduced by external data.

When a *procedure* has been called in a process, the actual transition is finished and it has been transited into a new state Begin_Procedure. In a second new state Return_Procedure all other actions of this transition are executed. Each procedure, called within a process, is for this process only once translated. From state Begin_Procedure leads a transition to the start-state of the procedure, from each end-state of the procedure leads a transition to the state Return_Procedure.

In analogy a *remote procedure call* is resolved.

When a *label* appears in a transition a new state and a new transition have been created for it. In a transition, outcoming from this new state, all actions of the actual transition from the label on have been marked. The table 1 summarized once more

SDL construct	Modelling using
enabling condition	one input and one save transition, condition through external data
continuous signal	one transition without signal input, condition through external data
priority input	construction according to [ITU92]
task	external data
variables	external data
decision	one transition for each alternative
procedure call	a begin and an end state and the translated procedure body
remote procedure call	in analogy to procedure call
label	one state and one transition without signal input

Table 1: Resolution of some SDL constructs

the resolved SDL constructs.

4 Implementation aspects: the net tool

4.1 Net tool components

The transformation and the analysis of the SDL specification takes place with more net tool components and in more steps. In dependence on its tasks the components can be divided into two groups:

- transformation and generation components and
- analysis and evaluation components.

After the input of the SDL'92 specification by an editor and after checking of the syntax and semantics correctness by parser and semantics analyser the outgoing specification is represented in the form of CR syntax tree. For the creation of this CR representation the term processor *Kimwitu* ([EIJK90]) has been used. The transformation component *cr2ps* generates from CR the PSS format (fig. 15). The PSS syntax tree is read by the transformation component *ps2pn*. From the PSS syntax tree *ps2pn* generates the internal net representation for both model classes Time PN and SDL Time net as well as input files (references) for the transformation component *filter* (fig. 14).

The analysis of Time PN model takes place using the net tool INA (Integrated Net Analyser) [STARKE92]. Further analysis and evaluation tools for this net class are not available.

For the SDL Time net analysis first of all the RG is to be generated by means of the program component *bed*. The modelling of external data is to be made in a C-File *extern.c*. This file is translated and linked with the library *bed.a*. This gives the net analyser *bed*. It generates three important files. The file *.mrk* contents all states of RG; in *.egn* are included: the arcs of the RG, the maximal utilization of the FIFO buffers and the dead transitions; *.ext* consists of the external data, which have been kept with the user defined help functions. Further the RG is to be transformed by means of program component *filter* at SDL level (file *.all*). Thereby all deadlocks are determined (file *.vkl*). Additionally necessary informations are included in a file *.ana* in order to enable the search for states by giving patterns. The actual analysis and evaluation are overtaken by both tools: *path_ana* and *ext_all*. *Path_ana* serves to search for paths to deadlocks and for states by giving patterns. The external data can be added to the RG in each state with the tool *ext_all*.

4.2 Transformation and net generation components

The transformation and net generation components consist of: *cr2ps*, *ps2pn* and *filter*. The transformation component *cr2ps* doesn't belong directly to the net tool components. As they have still an important role within the scope of the automatic transformation of SDL'92 specification into SDL Time net, this component is briefly

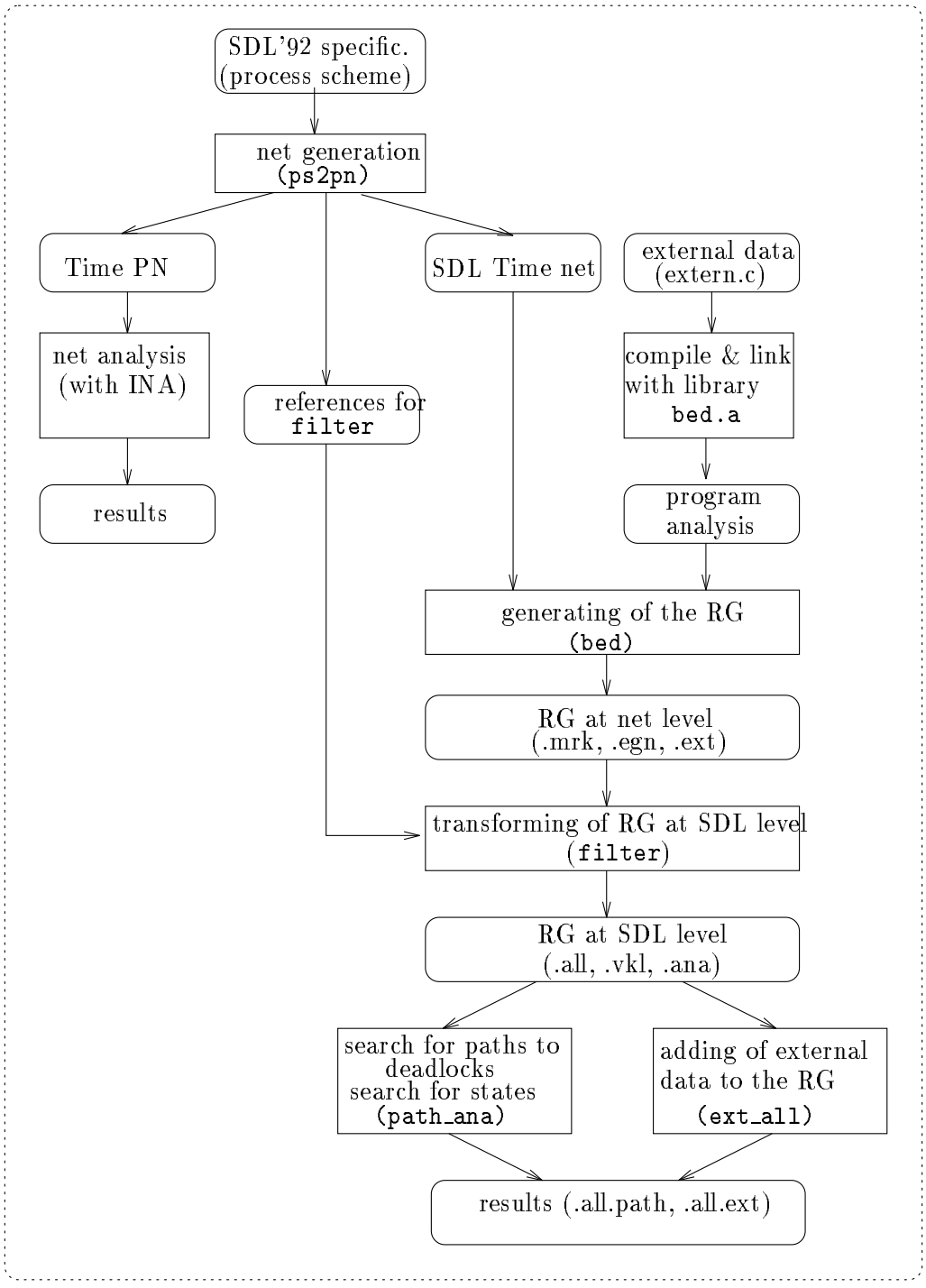

Figure 14: Structure and flow scheme of the net tool

sketched below.
As input for *cr2ps* serves the CR syntax tree of the SDL specification (fig.15). This CR syntax tree is reformed by *cr2ps*, according to the specific features of the PSS (section 3.2.1):

- resolution of block structures,
- simplification of the communication structures and
- treatment of inheritances and virtualities.

As a result the *cr2ps* produces an equivalent description of the starting SDL'92 specification in the form of a PSS syntax tree. The transformation component *ps2pn*

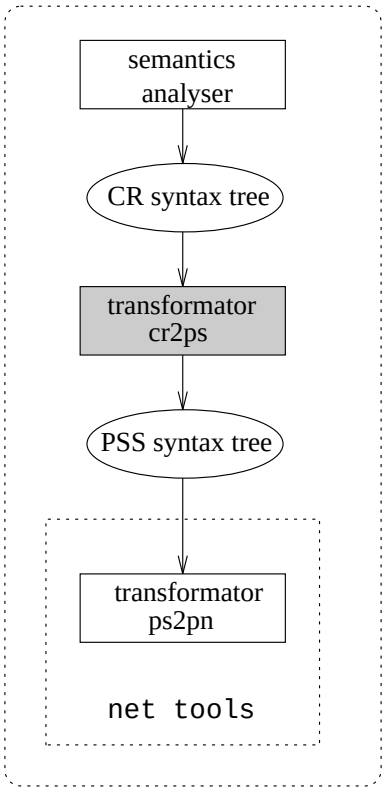

Figure 15: Transformation component *cr2ps*

generates from the PSS the internal net description for the both model classes, working in more steps:

1. Generating the internal data structure from the system description in PSS. The main work is done from the metatool *Kimwitu*. *Unparsing rules* and *Kimwitu functions* for transformation have been provided.

2. Resolving of the services of the particular processes.

3. Determination of the number of places, signals a.s.o.

4. Including of *discard transitions*.

5. Output of the internal net description for Time PN.

6. Generating the internal net description for SDL Time net and references for the program *filter*.

The internal data structures of *ps2pn* describe the SDL system as a set of processes. Each process consists of services, each service - of a set of transitions, which correspond to the according transitions of the SDL state transition graph. In the data structure of a transition among others the input signal, the old and the new state, the set of output signals, the timer actions and the dynamically created processes are entered.

It is necessary for the analysis to have a tool for the transformation of the RG at SDL level. This task has been undertaken by the program component *filter*. With the help of this program component it is possible to make a first evaluation of RG. The most important task of program *filter* have been summarized below:

- The RG can be presented in a tabular form (file *.all*) with all SDL identifiers; thereby not only all reachable states are visible but also the paths and branches of RG as well as all reachable deadlocks. The file *.all* is the basis for further analysis tools.

- All states, which represent a deadlock, have been described in a file *.vkl*.

- Furthermore with *filter* a first evaluation of the RG is possible:

  - the RG can be traversed step by step. In each state the concessed transitions are established. Selecting such a transition a new state is obtained, which is written out with all SDL identifiers. The paths are stored, a way back to earlier nodes is possible.
  - it can be tested if a state, being given by a pattern is reachable.

  These data are written in a file *.ana*.

4.3 Analysis and evaluation components

The tool *path_ana* is a shell program, which uses the UNIX tools *awk*, *sed* and *sort*. The RG at SDL level (file *.all* or *.all.ext*) is expected to be the input. With the help of the following steps a path to the state can be established.

1. The numbers of the states, which represented deadlocks, have been found out by the means of the awk-program *find_av.awk*. This state numbers are written in a temporary file.

2. Now the paths to all deadlocks or special states can be established. The output of the paths takes place with or without external data. Thereby the awk-program *make_av.awk* is called, which uses the shell program *path*. This establishes the path to a state by the following algorithm:

   (a) The lines in the input file are so re-ordered, that the last line is going to be the first one, the last but one line is going to be the second line and so on. This is done with the help of the shell program *umk*. At the beginning of each line the line number has been written, with *sort* the file is sorted in the reverse order and after that the line numbers at the line beginning have been removed again with *sed*.

   (b) With the help of the awk-program *ppf.awk* the path is established in the reverse order.

   (c) With the help of the *umk* the path is put into the right order according to 1 and it is written in a file *.pathNumber*, thereby *Number* is the order number of the target state.

The external data are added to the RG in each state using the shell program *ext_all*. Thereby two awk-programs are used, which:

- search the state order number in RG and
- find out and write out the external data for each found state number.

5 Applications. Protocol examples

5.1 Inres protocol analysis

5.1.1 Description of the used *Inres* protocol

The *Inres* protocol being described in [HOGREFE89] and [BELINA93] is used here as an investigation basis. It is connection-oriented and operates between two protocol instances *Initiator* and *Responder*. For the communication of both protocol instances an unreliable *Medium* service is used. The service is symmetrical and it operates in the connectionless mode. The data transmission is unreliable, and data can be either correctly transmitted or it can be lost. On the other hand no dates for itself can be generated; data cannot be corrupted or duplicated.
Two modifications have been made:

1. Data exchange between *Initiator* and *Responder* takes place directly and not using additional processes (*Codierer_Ini, Codierer_Resp, MSAP_I, MSAP_R*) as it has been described in [HOGREFE89].

2. In order to take into account the influence of the environment a block ENV with its two processes (User_Initiator and User_Responder) is introduced additionally (fig. 16).

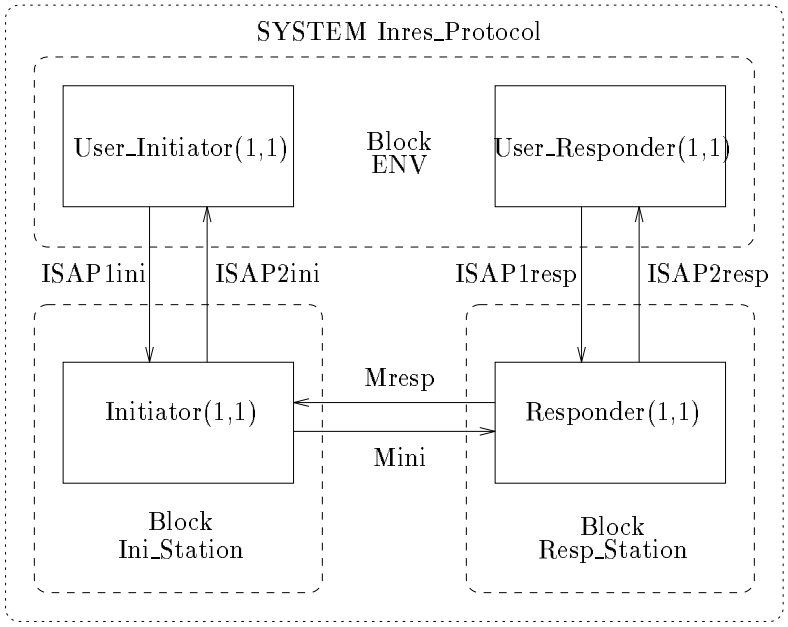

Figure 16: Structure of the used *Inres* protocol

Further bounded buffers of finite capacity are modelled within this example (cf. 2.4.2). The maximal buffer length is determined experimentally.

The SDL description of this *Inres* protocol is to be found in appendix A.1.

5.1.2 Verification results: detection of a deadlock

The *Inres* protocol has been first modelled through Time PN. An important disadvantage in this case is that FIFO regime cannot be modelled for the process input buffers. Thus, signals may overtake each other. Further, data dependencies cannot be taken into account resp. their modelling through an addition of complementary places leads to an enormous enlargement of the corresponding RG. The execution of the Time PN model by the net tool INA ([STARKE92]) has not indicated any deadlocks. It was not possible to determine and to isolate the states, which arise by signal overtaking. A realistic investigation was not possible using this net class, because the SDL semantics could not be adequately represented.

Next, *Inres* example has been investigated through SDL Time Net, at first without modelling of external data.
Two kinds of deadlock appeared during this analysis. The first one can be explained by the boundedness of the signal input buffer. In the net it means that the supposed maximal capacity of the corresponding input signal place has been exhausted so that it leads to blocking of transitions. The other deadlock arises from the following actions:

1. correct connection establishment, begin of data transfer,
2. *Initiator* recognizes the false acknowledgement(AK), it breaks up the connection, i.e. it sends a signal *IDISind* (indication of disconnection) to the *User_Initiator*,
3. *User_Responder* closes also the connection,
   now there are two *break-up-signals* for one connection,
4. *User_Initiator* consumes the first *break-up-signal* and initiates a second connection,
5. a new correct connection establishment, the second *break-up-signal* for the connection is still in the buffer of the *User_Initiator*,
6. *User_Initiator* consumes the first *break-up-signal* for the connection,
7. a signal for the occurred second establishment of the connection arrives at the *User_Initiator*, it is in state *wait* and it rejects this signal,
8. timer *tc* of *User_Initiator* exceeds,
9. *User_Initiator* initiates a third connection,

10. *Initiator* is still in the state *connected* from the second connection and it receives a signal for the new connection establishment, it rejects this signal,

11. now there isn't any more signal in the system, no timer is set; that means that no net transition can be fired any more , the consequence is a deadlock.

This deadlock arises because it was not possible to react correctly to the two *break-up-signals* for one and the same connection. This situation may happen only when the *Initiator* receives a signal AK with a false order number having been in the state *send*. All this may take place under unsuitable time relations in the *Inres* protocol: the *Initiator* sends data record many times to the receiver because it becomes no aknowledgement during the waiting time up to the exceeding of the timer; however the data records have been correctly transferred, the remaining acknowledgements arrive by sending the next data record and that's why they have a false sequence number.
On the following a possible elimination of this deadlock has been introduced. It takes place by adding a new timer *t_neu* within the process *Initiator*. The new timer has been set by each state transition into the state *connected*. In the state *connected* this timer has been reset when consuming the signals *IDATreq* (data from the *User_Initiator* to the provider) and *DR* (disconnection). In the case the *timeout*-signal is consumed by *t_neu*, the *Initiator* transfers to the state *interrupted*. For the time relations of the timers must be valid that the time delays of timer *t_d* (within the process *User_Initiator*) and of timer *t_c* (process *User_Initiator*) have to be less as the ones of the new timer *t_neu*.

In appendix A.2 (page 66) the so modified process *Initiator* can be found. During the analysis now there are only deadlocks resulting from the buffer boundedness; the previously described situation doesn't happen any more.

The deadlocks based on the buffer boundedness cannot be eliminated by the means of larger buffer lengths, too. The reason for it is the failing modelling of data dependences (process *Initiator* can send the signal DT any times) within the net model. In such a way the proposed buffer limit can be always exhausted.

5.1.3 Modelling of local process variables as external data

In order to eliminate deadlocks, resulting from the boundedness of input buffers, it is necessary to model the local variable counter of process *Initiator*. The modelling takes place according to the scheme in the section 3.1.4. Generating RG for each transition it is to be tested whether for it an user defined application function exists; then this function is called with the modelled signal parameters and its return value is evaluated. During the test, whether a state already exists, a user defined help function for comparison is carried out for all external data. At each call of an application function a copy of the local data is to be generated, using the user defined help function *duplicate*. The local data have been described in data structure *ext_data*.

An extract of the file *extern.c* has been introduced below.

```
/* modelling of the local variable counter */

int counter = 0;
DAT_P ext_data[] = {
     (DAT_P) & counter , NULL };

/* user defined help functions for counter */

/* duplicate */
DAT_P dup_counter( p_z1 )
DAT_P p_z1;
{    int *i;
     if ( ( i = (int *)(malloc(sizeof(int))) )
          == NULL ) {
          printf("\nMemory overflow in
          extern.c");exit(1);
     }
     *i = *(int *)p_z1;
     return ((DAT_P) i);
}

/* compare */
int cmp_counter (data1, data2)
DAT_P data1;
DAT_P data2;
{
     if ( *(int *)data1 == *(int *)data2 )
          return(1);
     else
          return(0);
}

/* write out */
void write_counter( ex_d, dat )
FILE *ex_d;
DAT_P dat;
{
     if ( dat == NULL )
          fprintf(ex_d, "counter:  dat ==
NULL");
     else
          fprintf(ex_d, "counter:  %2d", *(int
*) dat );
}

/* array for user defined help functions */
struct one_action action_ext_data[] = {
     { dup_counter, cmp_counter, write_counter
},
     { NULL, NULL, NULL }
};
```

```
/* user defined application functions */

/* counter initialization */
struct ret_sig *set_counter(sig_par)
DAT_P sig_par;
{    struct ret_sig *ret;
     /* memory for return value */
     if ( ( ret = NEW(ret_sig) ) == NULL ) {
          printf("\nMemory overflow in
          extern.c ");
          exit(1);
     }
     /* number of output signal parameters */
     ret→sig_c = 0;
     /* output signal parameter */
     ret→sig_v = NULL;
     /* return value */
     ret→wert = 1;
     /* initializes counter with 0. */
     *(int *)ext_data[0] = 0;
     return(ret);
}

/* increases counter with 1. */
struct ret_sig *increase_counter(sig_par)
DAT_P sig_par;
{    struct ret_sig *ret;
     . . . .
}

/* returns 1, if counter < MAX_Value, */
struct ret_sig *test_counter(sig_par)
DAT_P sig_par;
{    struct ret_sig *ret;
     . . . .
}

/* attachment of user defined functions to global
net transition order numbers */
struct n_fkt ext_fkt_feld[] = {
     { 25, set_counter },
     { 29, increase_counter },
     { 30, test_counter },
     { 34, set_counter },
     { 38, increase_counter },
     { 39, test_counter },
     { 43, increase_counter },
     { 44, test_counter },
     { -1, NULL }
};
```

The reached maximal buffer length is now dependent on the maximal value *MAX_Value* of the variable *counter*. The FIFO behaviour of the signal input buffer is modelled by a bounded buffer, whose given maximal length cannot be reached during the modelling of the *counter*. On the other hand, the modelling of the variable *counter* has brought an obvious reduction of the growth of the reachability graph. Table 2 shows the growth of the RG and the maximal buffer length by a given maximal value of the variable *counter*. On the contrary the growth of RG for *Inres* protocol without modelling of external data amounts by buffer limit of 2: 19140 states and 58818 arcs.

MAX_Value	states in RG	arcs in RG	max. buffer length
1	798	1588	3
2	1483	3292	4
3	2433	5873	5
4	3755	9633	6

Table 2: The growth of RG for *Inres* protocol with modelling of the variable *counter*.

So the proof of deadlock-freeness of the specification is given through generating the complete RG using net tool components. This proof is valid only for the given time intervals of the time-augmented transitions. The intervals act as parameters for the system and can be changed.

5.2 Sliding Window Protocol

5.2.1 Protocol description

The *Sliding Window protocol* ([SPRAGINS91] and [TURNER93]) is a real installed protocol for data transfer. It is connection-oriented and it operates between two protocol instances *Transmitter* and *Responder*. The protocol has no connection or disconnection procedures. It supports an unidirectional flow of data with a positive handshake on each transfer. An acknowledgement window is used for flow control. The protocol is designed to operate over an unreliable medium and so it employs sequence numbering and acknowledgements. The following cases can appear at the data transfer through the medium:

- a correct data transfer,
- loss of data,
- duplication of data,
- re-ordering of data,
- corruption of data.

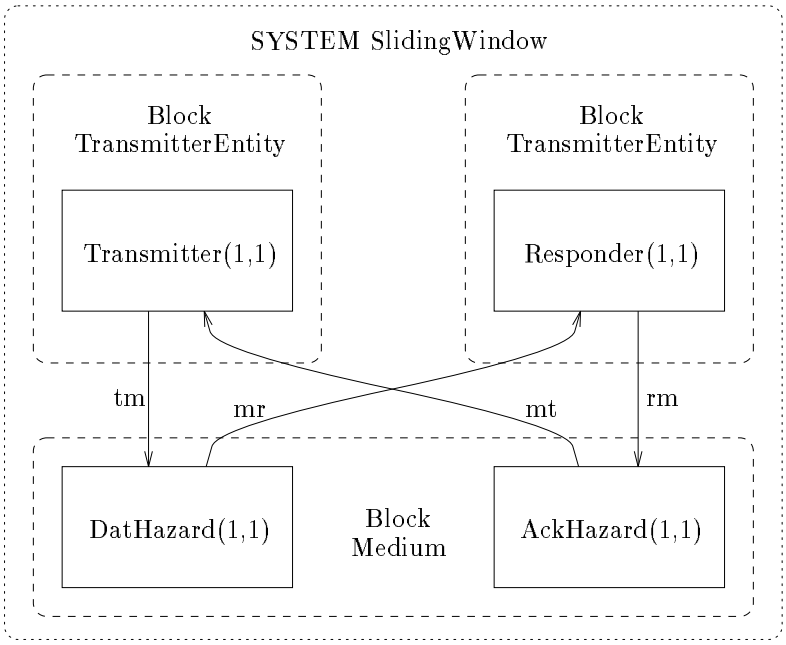

Figure 17: Structure of the used *Sliding Window* protocol.

The *Transmitter* sends a sequence number with each data record beginning with 1. A sequence number is unbounded and is incremented for each new data record. At the receive of a not corrupted message the *Responder* sends an acknowledgement including the number of the last correct received message. *Transmitter* and *Responder* maintain the sent resp. the received data sequence numbers in arrays (so called windows). For each sent message a timer has been started by the *Transmitter*. If the timeout signal of this timer is consumed, the according data record is sent once more. At the receive of acknowledgement, belonging to the data record, the timer is reset. In order to generate a finite RG it is necessary to have a finite number of data records, which has to be transmitted. Therefore in the used SDL specification (appendix B) only two data records are transmitted by the *Transmitter*; the window size is equal to 2. The data records consist only of the sequence numbers (1 or 2); two timer t1 and t2 are necessary in the *Transmitter*. The medium may loose, corrupt, duplicate, re-order or transmit the data records correct. While corruption of data can be stated, they are not modelled in the medium, but they are represented by a data loss. The states *second_out* have been introduced so that no transition in a given state can output two equal signals to one process. The tokens modelling the output signals reach first the according signal output place of the process in the SDL Time net. This place doesn't model a queue with FIFO behaviour. Thus, signals may overtake each other. Consequently the signal output place could not be any more given with a maximal capacity 1; an enlargement of the RG takes place.

5.2.2 Modelling of SDL signal parameters

For each signal the user can specify a list of any type signal parameters. They can be modelled by use of external data and functions. In analogy to the modelling of local process variables (section 5.1.3) the three help functions *duplicate*, *compare* and *write* are to be defined in array *actions* (file extern.c). Each external function gets as actual parameter the parameter of input signal. Its return value includes the parameter of the output signal being modelled by the user. In appendix B.2 (page 75) two external functions are given for modelling of the signal parameters in Sliding Window protocol.

5.2.3 Results

Table 3 shows the modelled local process variables as external data; table 4 - the modelled signal parameters.

process	modelled local variables
Transmitter	lu
Responder	dat1, dat2
DatHazard	datq, used, unused
AckHazard	datq, used, unused

Table 3: Process variables in Sliding Window Protocol modelled as external data.

signal	modelled signal parameters
DATr	sequence number
DATi	sequence number
AKr	sequence number
AKi	sequence number

Table 4: Signal parameters in Sliding Window Protocol modelled as external data.

For all transitions in table 5 the earlest and the latest firing time have an equal value. The rest transitions get the firing time 0.

In table 6 the growth of the RG is represented when the data are again to be sent at each receive of a timeout signal of the timer *t1* and *t2* of the Transmitter and the timers have been always set.

At all appearing deadlocks the *Transmitter* is in the state *trans_ok*; it means the data have been correctly transmitted and the acknowledgements about it are received by the *Transmitter*. Because it has to send only two data, no more signals

transition	firing time (eft = lft)
channel tm	1
channel mr	1
channel rm	1
channel mt	1
timer t1 of Transmitter	8
timer t2 of Transmitter	8

Table 5: Firing times of the transitions in net model for Sliding Window protocols.

max. used buffer limit	states in RG	arcs in RG
2	4045	7600

Table 6: Growth of the RG by Sliding Window protocol analysis with external data and permanent repeat of the data sending by the *Transmitter* and when no acknowledgement has been received.

are generated and in consequence a deadlock arises. When the data are going to be lost in the medium, the timers *t1* or *t2* of the *Transmitter* are expired and the data are sent again. The system goes into a state, which was taken at the previous data transmission. It is a cycle in the RG. When the data to be transmitted are sent only at the first expiration of the timers *t1* and *t2* of the *Transmitter*, e.i. the timers are not set once more, then the attained growth of the corresponding RG is shown in table 7. The maximal used buffer length is 2.

max. used buffer limit	states in RG	arcs in RG
2	4234	8248

Table 7: Growth of the RG by Sliding Window protocol analysis with external data and one-time-repeat of the data sending by the *Transmitter*.

6 Conclusions

In this paper, we have presented the definition of a new Petri net formalism called SDL Time Net, which supports the analysis and the verification of SDL'92 specifications. Different net tool components as a part of the SITE environment have been implemented. By means of these tools two application examples regarding to the Inres and Sliding Window protocols are analysed and validated.

Some advantages and disadvantages of the modelling means as well as of the implemented net tool are summarized and some modifications and improvements are proposed below.

Two important advantages are:

- No essential restrictions of the SDL'92 language have to be made through net tool embedding into the SDL program environment. So SDL specifications of real communication protocols can be analyzed;
- The automatic transformation and interpretation enable the application of this approach by users without detailed knowledge of the net theory.

As disadvantages can be stated:

- An enlargement of the net growth takes place by modelling of process type with a large number of dynamically created instances;
- Certain knowledge is necessary about the internal data structures of the analysis tools because of the internal 'not visible' net description, especially when defining and attaching guard functions to net transitions as well as when modelling process variables and signal parameters by the user;
- The integration of further net analysis algorithms is difficult. Knowledge is needed about the interfaces of net tools and also about the structure of the files, generated by these tools.

In connection with the above the following modifications and extensions are important. They regard to the modelling means:

- Through *folding* of all process-subnets of a process only one process net arises, which represents all instances of the process simultaneously. This folding is possible, while all process-subnets have the same structure and the same behaviour. The internal numbering of process instance nets is preserved in order to guarantee a differentiation between individual process instances when analysing. The assignment of the net elements to 'different process instance nets' is kept internal as an additional attribute. So the corresponding places and transitions of 'different process instance nets' are physically only once present. In such a way an essential increase of the effectiveness of the net description has been reached through reduction of the net growth.

- Introduction of *individual tokens*:
  - *control* tokens have e.g. the order number of the passive process instance and they serve so to the selection of the corresponding process instance net at *create*;
  - *data (signal)* tokens model SDL signals and their parameters. The modelling of signal parameters becomes more efficient and it takes place in a more natural way - now the signal parameters can be interpreted as token attributes and they are not more attached to the signal input places.

The *graphical visualization* of the generated net could mean an additional support of the user when user functions have been attached to transitions. So the above mentioned disadvantages can be essentially diminished.

References

[Aalst93] Aalst, v. d., W.M.P.: Interval Timed Coloured Petri Nets and their Analysis, Application and Theory of Petri Nets 1993, 14th International Conference, Chcago, USA, LNCS 691, Springer Verlag, Berlin, Heidelberg, pp.453-472, 1993.

[Algayres93] Algayres, B., Lejeune, Y., Hugonnet, F., Hantz, F.: The AVALON Project: A VALidatiON Environment For SDL/MSC Descriptions, Verilog S.A., Toulouse, 1993.

[Augustin89] Augustin, L.: Ein Beitrag zur Verifikation von Kommunikationsprotokollen mit Petri-Netzen, Dissertation, Ingenierhochschule Mittweida, 1989 (in German).

[Baldassari91] Baldassari, M., Bruno, G.: PROTOB: An Object Oriented Methodology for Developing Discrete Event Dinamic Systems, Computer Languages 16, vol. 1, 39-63, 1991.

[Barbeau91] Barbeau, M., Bochmann, G.v.: Formal Verification on Object-Oriented Specifications Using a Technique for Colored Petri Nets, Progress Report Document for CRIM/BNR Project, Centre de recherche informatique de Montreal, 1991.

[Barbeau93] Barbeau, M., Bochmann, G.v.: A Subset of Lotos with the Computational Power of Place/Transition-Nets, Application and Theory of Petri nets 1993, 14th International Conference, Chcago, USA, LNCS 691, Springer Verlag, Berlin , Heidelberg, pp.49-67, 1993.

[Belina93] Belina, F., Hogrefe, D., Sarma A.: SDL with Applications from Protocol Specification, Carl Hanser Verlag, 1991

[Billington88] Billington, J., Wheeler, G.R., Wilbur-Ham, M.C.: PROTEAN: A High-level Petri Net Tool for the Specification and Verification of Communication Protocols, IEEE Transactions on Software Engineering, Special Issue on Tools for Computer Communication Systems, SE-14, 3, 301-316, 1988.

[Bruno86] Bruno, G., Marchetto, G.: Process-translatable Petri Nets for Rapid Prototyping of Process-Control Systems, IEEE Trans. Software Eng., vol. SE-12, no.2, pp.346-357, 1986.

[Cerone93] Cerone, A.: A Net-based Approach for Specifying Real-Time Systems, Ph.D. Thesis: TD-16/93, Dipartimento di Informatica, Universita di Pisa, 1993.

[Comparin85] Comparin, G., Lanzarone, G.A. et al.: Analysis of SDL Specifications Using Petri Nets Techniques, Workshop of Second SDL-Users' Forum, Helsinki, 1985.

[COURTIAT84] Courtiat, J.P., Ayache, M., Algayres, B.: Petri net are good for protocols, in: ACM SIGCOMM'84 Symp. Comm. Architectures and Protocols, Monreal, Canada, pp.66-74, 1984.

[DE CINDIO84] De Cindio, F., Lanzarone, G., Torgano, A.: A Petri Net Model of SDL, Proc of Fifth European Workshop on Application and Theory of Petri Nets, Aarhus, Denmark, pp.272-289, 1984.

[ECKERT84] Eckert, H., Prinoth, R.: Produktnetze – Definition eines PROSIT-Beschreibungsmittels, GMD-Arbeitspapiere 92, GMD Darmstadt, 1984 (in German).

[EIJK90] Eijk, P., Belifante, A.: The Term Processor Kimwitu – Manual and Cookbook, University of Twente, Tele-Informatics and Open Systems Group, 1990.

[FISCHER93] Fischer, J.: An environment for SDL'92, SAMS, vol.13, pp.107-123, 1993.

[FISCHER93-1] Fischer, J., Holz, E.: Towards an Integrated Technology for Object-Oriented Specification and Implementation of Distributed Systems, SAMS, vol.13, pp.89-106, 1993.

[FISCHER ET AL.93] Fischer, J., Holz, E., v. Loewis, M., Witaszek, D.: A Run Time Library for the Simulation of SDL'92-Specifications, Proc. of the Sixth SDL Forum Darmstadt, Germany, Okt.1993 in Færgemand, O., Sarma, A. (ed.): SDL'93 Using Objects, North-Holland, Amsterdam, pp. 105-118, 1993.

[GENRICH81] Genrich, H.J., Lautenbach, K.: System Modeling with high-level Nets, Theoretical Computer Science, no. 13, pp. 109-136, 1981.

[GODBERSEN83] Godbersen, H.P.: Funktionsnetze – eine Modellierungskonzeption zur Entwurfs- und Entscheidungsunterstuetzung, Ladewig-Verlag, Birkach, 1983 (in German).

[GRABOWSKI90] Grabowski, J.: Statische und dynamische Analysen für SDL-Spezifikationen auf der Basis von Petri-Netzen und Sequence-Charts, Diplomarbeit, München, 1990 (in German).

[HOGREFE89] Hogrefe, D.: Estelle, LOTOS und SDL, Springer-Verlag, Berlin, Heidelberg, 1989.

[ITU88] CCITT:Recommendation Z.100: CCITT Spezification and Description Language (SDL'88), Genf, 1988.

[ITU92] CCITT:Recommendation Z.100: CCITT Spezification and Description Language (SDL'92), Genf, 1992.

[JENSEN91] Jensen, K., Rozenberg, G.: High-level Petri Nets, Springer-Verlag, Berlin, Heidelberg, 1991.

[JENSEN92] Jensen, K.: Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use, Volume 1, Springer-Verlag, Berlin, Heidelberg, 1992.

[KETTUNEN88] Kettunen, E., Montonen, E., Tuuliniemi, T.: An Interactive PrT-Net Tool for Verification of SDL-Specifications, Technical Report No.3, Helsinki University of Technology, Digital System Laboratory, Otaniemi, 1988.

[LEHNERT93] Lehnert, J.: Netzdarstellung und Petri-Netz-Analyse von SDL-Spezifikationen, Diplomarbeit, Humboldt–Universität zu Berlin, Fachbereich Informatik, 1993 (in German).

[LESZAK87] Leszak, M., Eggert, H:: Petri-Netz-Methoden und -Werkzeuge, Springer Verlag Berlin, Heidelberg, 1987 (in German).

[MÄTZEL92] Mätzel, K.: Das Prozeßschema – eine Darstellung von SDL'92 Spezifikationen, Jahresarbeit, Humboldt–Universität zu Berlin, Fachbereich Informatik, 1992 (in German).

[MARSAN85] Marsan, M.A., Balbo, G., Chiola, G. et al: On Petri Nets with Stochastic Timing, Proc. Int. Workshop on Timed Petri nets, Torino, Italy, 1985.

[MARSAN87] Marsan, M.A., Balbo, G., Chiola, G., Conte, G.: Generalized Stochastic Petri Nets revisited, Proc. Int. Workshop on Petri nets and performance models, Madison, 1987.

[MERLIN74] Merlin, P.M.: A Study of Recoverablility of Computing Systems, Ph. D. thesis, Department of Information and Computer Science, University of California at Irvine, USA, 1974.

[NOE75] Noe, J.D.: Pro-Nets: For Modelling Processes and Processors, Conf. an Petri Nets and Related Topics, MIT, USA, 1975.

[NUTT72] Nutt, G.J.: Evaluation Nets for Computer System Performance Analysis, AFIPS Fall Joint Comp. Conf., vol.41, pp.279-286, 1972.

[PAETZOLD90] Paetzold, P.: Erreichberkeitsanalyse in verteilten Algorithmen mit Petri-Netzen bei unterschiedlicher Vereinfachung des datenabhaengigen Steuerflusses, Institut fuer Informatik und Rechentechnik Berlin, Informatik Informationen–Reporte No.7, 1990 (in German).

[PAULE85] Paule, C., Eckert, H.: The Net Simulation SYstem NESSY, summary and example, GMD Arbeitspapiere 182, 1985.

[Popova91] Popova, L.: On Time Petri Nets, J. Inform. Process. Cybern. EIK 27, vol.4, pp. 227-244, 1991.

[Ramchandani74] Ramchandani, C.: Analysis of Asynchronous Concurrent Systems Using Petri Nets, Ph. D. thesis, Project MAC, MAC-TR 120, MIT, 1974.

[Reisig91] Reisig, W.: Petri Nets and Algebraic Specifications, Theoretical Computer Science no.80, pp.1-34, 1991.

[Sajkowski87] Sajkowski, M.: Protocol Verification using Discrete-Event Models, IIASA Conference Discrete Event Systems: Models and Applications, Hungary, 1987, in: Lecture Notes in Control and Information Sciences Nr.103, pp.100-114, 1987.

[Schade94] Schade, A.: Statistische Semantikanalyse von SDL'92 Spezifikationen, Diplomarbeit, Humboldt–Universität zu Berlin, Fachbereich Informatik, 1994 (in German).

[Spragins91] Spragins, J., Hammond, J., Pawlikowski, K.: Telecommunications – Protocols and Design, Addison-Wesley Publishing Company, 1991.

[Starke90] Starke, P.H.: Analyse von Petri-Netz-Modellen, B.G. Teubner, Stuttgart, 1990 (in German).

[Starke92] Starke, P.H.: Handbuch zum Integrated Net Analyzer, Humboldt–Universität zu Berlin, Fachbereich Informatik, 1992 (in German).

[Taubert93] Taubert, U.: Development of a tool for analysis of SDL'92 specifications using Extended Petri Nets, diplom thesis (in German), Humboldt-University Berlin, Depart. of Comp. Sci., 1993.

[Turner93] Turner, K.J.: Using Formal Description Techniques. An Introduction to Estelle, Lotos and SDL, John Wiley and Sons, Chichester, New York et. al., 1993.

A SDL description of the used *Inres* protocol

A.1 The starting specification

The SDL specifications of the processes *User_Initiator* and *User_Responder* are given below.

```
SYSTEM Inres_Protokoll;

    NEWTYPE ISDUtyp
            /* The ISDU-type is here given */
          STRUCT start Time;
    ENDNEWTYPE ISDUtyp;

    NEWTYPE IPDUtyp
            LITERALS cr, cc, dr, dt, ak;
    ENDNEWTYPE IPDUtyp;

    NEWTYPE MSDUtyp
            STRUCT id IPDUtyp;
            Num Natural;
            Daten ISDUtyp;
    ENDNEWTYPE MSDUtyp;

    SIGNAL ICONreq,
           IDATreq(ISDUtyp),
           ICONconf,
           ICONind,
           ICONresp,
           IDISreq,
           IDISind,
           IDATind(ISDUtyp),
           CR,
           CC,
           DT(Natural, ISDUtyp),
           AK(Natural),
           DR;


    CHANNEL ISAP1ini FROM ENV TO Ini_Station
            WITH ICONreq, IDATreq;
    ENDCHANNEL ISAP1ini;

    CHANNEL ISAP2ini FROM Ini_Station TO ENV
            WITH ICONconf, IDISind;
    ENDCHANNEL ISAP2ini;

    CHANNEL ISAP1empf FROM ENV TO Resp_Station
            WITH ICONresp, IDISreq;
    ENDCHANNEL ISAP1empf;
```

```
CHANNEL ISAP2empf FROM Resp_Station TO ENV
        WITH ICONind, IDATind;
ENDCHANNEL ISAP2empf;

CHANNEL Mresp FROM Resp_Station TO Ini_Station
        WITH DR, CC, AK;
ENDCHANNEL Mresp;

CHANNEL Mini FROM Ini_Station TO Resp_Station
        WITH CR, DT;
ENDCHANNEL Mini;

BLOCK ENV;

 PROCESS User_Initiator(1,1);
   SIGNALSET ICONconf, IDISind;

   DCL Pd Duration:= 10.0,
       Pc Duration:= 12.0,
       d ISDUtyp;
   TIMER tc, td;

     START;
       ConnectionEstablishment:
           OUTPUT ICONreq;
           NEXTSTATE Acknowledgment;

     STATE Acknowledgment;
       INPUT ICONconf;
           Dataphase:
           TASK d!start:= NOW;
           OUTPUT IDATreq(d);
           SET (NOW+Pd,td);
           NEXTSTATE Connection;
       INPUT IDISind;
           SET (NOW+Pc,tc);
           NEXTSTATE Wait;
     ENDSTATE Acknowledgment;

     STATE Wait;
       INPUT tc;
           JOIN ConnectionEstablishment;
     ENDSTATE Wait;
```

```
        STATE Connection;
          INPUT IDISind;
               RESET (td);
               JOIN ConnectionEstablishment;
          INPUT td;
               JOIN Dataphase;
        ENDSTATE Connection;

    ENDPROCESS User_Initiator;
    PROCESS User_Responder (1,1);
      SIGNALSET ICONind, IDATind;
      DCL d ISDUtyp;

        START;
            NEXTSTATE Connection;

        STATE Connection;
          INPUT ICONind;
              conn:
               DECISION True /*draw (0.5)*/;
               (True): OUTPUT ICONresp;
                         NEXTSTATE Receive;
               ELSE: OUTPUT IDISreq;
                         NEXTSTATE Connection;
               ENDDECISION;

       STATE Receive;
          INPUT ICONind;
               JOIN conn;
          INPUT IDATind(d);
               DECISION True /*draw (0.5)*/;
               (True): NEXTSTATE Receive;
               ELSE: OUTPUT IDISreq;
                         NEXTSTATE Connection;
               ENDDECISION;

    ENDPROCESS User_Responder;

  ENDBLOCK ENV;


  BLOCK Ini_Station;
    PROCESS Initiator REFERENCED /*%FILE init.sdl*/;
  ENDBLOCK Ini_Station;

  BLOCK Resp_Station;
    PROCESS Responder REFERENCED /*%FILE resp.sdl*/;
  ENDBLOCK Resp_Station;

ENDSYSTEM Inres_Protokoll;
```

The SDL specification of the process *Initiator* follows below:

```
PROCESS Initiator (1,1);

SIGNALSET ICONreq, DR, CC, IDATreq, AK;

DCL counter Integer,
    Z Real,
    d ISDUtyp,
    Num Natural,
    Nummer Natural,
    Loss Natural,
    P Duration,
    MAX_Value Integer;
TIMER tc, td;

    START;
        TASK MAX_Value := 4;
        NEXTSTATE Disconnect;

    STATE Disconnect;
      SAVE IDATreq;
      INPUT ICONreq;
            TASK counter:=1, Z:=1.0;
            OUTPUT CR;
            SET (NOW+P,tc);
            NEXTSTATE Wait;

      INPUT DR;
            DECISION Loss = 1;
            (True): NEXTSTATE -;
            ELSE :
                        OUTPUT IDISind;
                        NEXTSTATE -;
            ENDDECISION;
    ENDSTATE Disconnect;

    STATE Wait;
      SAVE IDATreq;
      INPUT CC;
            DECISION Loss = 1;
            (True): NEXTSTATE -;
            ELSE :
                        RESET(tc);
                        TASK Nummer:=1;
                        OUTPUT ICONconf;
                        NEXTSTATE Connected;
            ENDDECISION;
```

```
  INPUT tc;
       DECISION counter < MAX_Value;
       (True): OUTPUT CR;
                TASK counter:= counter+1,
                     Z:= Z+1.0;
                SET (NOW+P,tc);
                NEXTSTATE -;
       ELSE: OUTPUT IDISind;
                NEXTSTATE Disconnect;
       ENDDECISION;

  INPUT DR;
       DECISION Loss = 1;
       (True): NEXTSTATE -;
       ELSE :
                RESET(tc);
                OUTPUT IDISind;
                NEXTSTATE Disconnect;
       ENDDECISION;
 ENDSTATE Wait;

STATE Connected;
  INPUT IDATreq(d);
       OUTPUT DT(Nummer,d);
       TASK counter:=1, Z:= 1.0;
       SET (NOW+P,td);
       NEXTSTATE Send;

  INPUT DR;
       DECISION Loss = 1;
       (True): NEXTSTATE -;
       ELSE :
                OUTPUT IDISind;
                NEXTSTATE Disconnect;
       ENDDECISION;
ENDSTATE Connected;

STATE Send;
  SAVE IDATreq;
  INPUT AK(Num);
       DECISION Loss = 1;
       (True): NEXTSTATE -;
       ELSE :
               RESET(td);
               DECISION Num=Nummer;
               (True): TASK Nummer:= (Nummer+1) MOD 2;
                       NEXTSTATE Connected;
               ELSE:
                       l:
                       DECISION counter < MAX_Value;
                       (True): OUTPUT DT(Nummer,d);
```

```
                                    TASK counter:= counter+1,
                                    Z:= Z + 1.0;
                                    SET (NOW+P,td);
                                    NEXTSTATE Send;
                            ELSE: OUTPUT IDISind;
                                    NEXTSTATE Disconnect;
                            ENDDECISION;
                    ENDDECISION;
            ENDDECISION;

        INPUT td;
            JOIN l;

        INPUT DR;
            DECISION Loss = 1;
            (True): NEXTSTATE -;
            ELSE :
                    RESET(td);
                    OUTPUT IDISind;
                    NEXTSTATE Disconnect;
            ENDDECISION;
    ENDSTATE Send;

ENDPROCESS Initiator;
```

The SDL specification of the process *Responder* follows below:

```
PROCESS Responder (1,1);

SIGNALSET CR, ICONresp, DT, IDISreq;

DCL
    d ISDUtyp,
    Num Natural,
    Nummer Natural,
    Loss Natural;

    START;
        NEXTSTATE Disconnect;

    STATE Disconnect;
      INPUT CR;
          DECISION Loss = 1;
          (True): NEXTSTATE -;
          ELSE :
                    OUTPUT ICONind;
                    NEXTSTATE Wait;
          ENDDECISION;
      INPUT IDISreq;
          OUTPUT DR;
          NEXTSTATE Disconnect;
    ENDSTATE Disconnect;

    STATE Wait;
      INPUT CR;
          DECISION Loss = 1;
          (True): NEXTSTATE -;
          ELSE :
                    OUTPUT ICONind;
                    NEXTSTATE Wait;
          ENDDECISION;
      INPUT ICONresp;
          TASK Nummer:= 0;
          OUTPUT CC;
          NEXTSTATE Connected;
      INPUT IDISreq;
          OUTPUT DR;
          NEXTSTATE Disconnect;
    ENDSTATE Wait;


    STATE Connected;
      INPUT DT(Num,d);
          DECISION Loss = 1;
          (True): NEXTSTATE -;
          ELSE :
                    DECISION Num = (Nummer + 1) MOD 2;
```

```
                (True): OUTPUT IDATind(d);
                        OUTPUT AK(Num);
                        TASK Nummer:= (Nummer+1) MOD 2;
                        NEXTSTATE -;
                ELSE: OUTPUT AK(Num);
                        NEXTSTATE -;
                ENDDECISION;
        ENDDECISION;

    INPUT CR;
        DECISION Loss = 1;
        (True): NEXTSTATE -;
        ELSE :
                OUTPUT ICONind;
                NEXTSTATE Wait;
        ENDDECISION;

    INPUT IDISreq;
        OUTPUT DR;
        NEXTSTATE Disconnect;
ENDSTATE Connected;

ENDPROCESS Responder;
```

A.2 The modified process Initiator

The timer *t_neu* is additionally added to the process *Initiator*.

```
PROCESS Initiator (1,1);

SIGNALSET ICONreq, DR, CC, IDATreq, AK;

DCL counter Integer,
    Z Real,
    d ISDUtyp,
    Num Natural,
    Nummer Natural,
    Loss Natural,
    P Duration,
    P_neu Duration,
    MAX_Value Integer;
TIMER tc, td, t_neu;

    START;
        TASK MAX_Value:= 4;
        NEXTSTATE Disconnect;

    STATE Disconnect;
      SAVE IDATreq;
      INPUT ICONreq;
          TASK counter:=1, Z:=1.0;
          OUTPUT CR;
          SET (NOW+P,tc);
          NEXTSTATE Wait;

      INPUT DR;
          DECISION Loss = 1;
          (True): NEXTSTATE -;
          ELSE :
                  OUTPUT IDISind;
                  NEXTSTATE -;
          ENDDECISION;
    ENDSTATE Disconnect;

    STATE Wait;
      SAVE IDATreq;
      INPUT CC;
          DECISION Loss = 1;
          (True): NEXTSTATE -;
          ELSE :
                  RESET(tc);
                  TASK Nummer:=1;
                  OUTPUT ICONconf;
                  SET (NOW+P_neu,t_neu);
                  NEXTSTATE Connected;
          ENDDECISION;
```

```
  INPUT tc;
       DECISION counter < MAX_Value;
       (True): OUTPUT CR;
                TASK counter:= counter+1,
                     Z:= Z+1.0;
                SET (NOW+P,tc);
                NEXTSTATE -;
       ELSE : OUTPUT IDISind;
                NEXTSTATE Disconnect;
       ENDDECISION;

  INPUT DR;
       DECISION Loss = 1;
       (True): NEXTSTATE -;
       ELSE :
                RESET(tc);
                OUTPUT IDISind;
                NEXTSTATE Disconnect;
       ENDDECISION;
ENDSTATE Wait;

STATE Connected;
  INPUT t_neu;
       OUTPUT IDISind;
       NEXTSTATE Disconnect;
  INPUT IDATreq(d);
       RESET(t_neu);
       OUTPUT DT(Nummer,d);
       TASK counter:=1, Z:= 1.0;
       SET (NOW+P,td);
       NEXTSTATE Send;
  INPUT DR;
       DECISION Loss = 1;
       (True): NEXTSTATE -;
       ELSE :
                RESET(t_neu);
                OUTPUT IDISind;
                NEXTSTATE Disconnect;
       ENDDECISION;
ENDSTATE Connected;
STATE Send;
  SAVE IDATreq;

  INPUT AK(Num);
       DECISION Loss = 1;
       (True): NEXTSTATE -;
       ELSE :
                RESET(td);
                DECISION Num=Nummer;
                (True): TASK Nummer:= (Nummer+1) MOD 2;
```

```
                    SET (NOW+P_neu,t_neu);
                    NEXTSTATE Connected;
                ELSE :
                    l:
                     DECISION counter < MAX_Value;
                     (True): OUTPUT DT(Nummer,d);
                             TASK counter:= counter+1,
                             Z:= Z + 1.0;
                             SET (NOW+P,td);
                             NEXTSTATE Send;
                     ELSE : OUTPUT IDISind;
                             NEXTSTATE Disconnect;
                     ENDDECISION;
                ENDDECISION;
        ENDDECISION;

    INPUT td;
        JOIN l;

    INPUT DR;
        DECISION Loss = 1;
        (True): NEXTSTATE -;
        ELSE :
                RESET(td);
                OUTPUT IDISind;
                NEXTSTATE Disconnect;
        ENDDECISION;
  ENDSTATE Send;

ENDPROCESS Initiator;
```

B The SDL description of the used *Sliding Window* protocol

B.1 The starting specification

```
SYSTEM SlidingWindow;
/* window size 2, two data record are transmitted by the Transmiter. */

SYNONYM MedBuffer = 3;
SYNONYM Delta Duration = 5.0;

NEWTYPE MedData
    STRUCT
        seqno Natural;
ENDNEWTYPE MedData;

NEWTYPE MedDatQueue Dimarray(MedBuffer, MedData);
ENDNEWTYPE MedDatQueue;

SIGNAL
DATr(Natural), DATi(Natural), AKr(Natural), AKi(Natural);

CHANNEL tm FROM TransmitterEntity TO Medium WITH DATr;
ENDCHANNEL;
CHANNEL mr FROM Medium TO ResponderEntity WITH DATi;
ENDCHANNEL;
CHANNEL rm FROM ResponderEntity TO Medium WITH AKr;
ENDCHANNEL;
CHANNEL mt FROM Medium TO TransmitterEntity WITH AKi;
ENDCHANNEL;

BLOCK TransmitterEntity;

PROCESS Transmitter(1,1);
/* window size 2, two data record are transmitted by the Transmiter. */

DCL lu, seqno Natural;
TIMER t1, t2;

START;
    /* it transmits the first data record for the first time. */
    OUTPUT DATr(1);
    TASK lu := 1;
    NEXTSTATE second_out;

STATE second_out;
    INPUT none;
        /* it transmits the second data record for the first time . */
        OUTPUT DATr(2);
        /* setting of timer. */
```

```
        SET(NOW+Delta, t1);
        SET(NOW+Delta, t2);
        NEXTSTATE wait_retrans;
    SAVE *;
ENDSTATE second_out;

STATE wait_retrans;
    INPUT AKi(seqno);
        DECISION (seqno ≥ lu);
            (True): DECISION ( lu );
                        (1): RESET(t1);
                             TASK lu := 2;
                             DECISION seqno = 2;
                                    (True): RESET(t2);
                                            NEXTSTATE trans_ok;
                                    (False): NEXTSTATE -;
                             ENDDECISION;
                        (2): RESET(t2);
                             NEXTSTATE trans_ok;
                    ENDDECISION;
            (False): NEXTSTATE -;
        ENDDECISION;

    INPUT t1;
        OUTPUT DATr(1);
        SET(NOW+Delta, t1);
        NEXTSTATE -;

    INPUT t2;
        OUTPUT DATr(2);
        SET(NOW+Delta, t2);
        NEXTSTATE -;
ENDSTATE wait_retrans;

STATE trans_ok;
    INPUT AKi(seqno);
        NEXTSTATE -;

    INPUT t1;
        NEXTSTATE -;

    INPUT t2;
        NEXTSTATE -;
ENDSTATE trans_ok;

ENDPROCESS Transmitter;

ENDBLOCK TransmitterEntity;
```

```
BLOCK ResponderEntity;

PROCESS Responder(1,1);
/* it sends AKr only be receiving of a new data record. */
/* it remains forever in state send_ack. */

DCL seqno Natural,
    dat1, dat2 Boolean; /* are the data records 1 and 2 already received? */

START;
    TASK dat1 := False,
         dat2 := False;
    NEXTSTATE send_ack;

STATE send_ack;
    INPUT DATi(seqno);
        DECISION ( seqno );
            (1): /* the first data record is received */
                DECISION( dat1 );
                        (True): NEXTSTATE -;
                        (False): TASK dat1 := True;
                                 DECISION( dat2 );
                                     (True): OUTPUT AKr(2);
                                             NEXTSTATE -;
                                     (False): OUTPUT AKr(1);
                                             NEXTSTATE -;
                                 ENDDECISION;
                ENDDECISION;
            (2): /* the second data record is received */
                DECISION( dat1 );
                        (True): DECISION( dat2 );
                                     (TRUE): NEXTSTATE -;
                                     (FALSE): OUTPUT AKr(2);
                                              TASK dat2 := True;
                                              NEXTSTATE -;
                                 ENDDECISION;
                        (False): TASK dat2 := True;
                                 NEXTSTATE -;
                 ENDDECISION;
        ENDDECISION;
ENDSTATE send_ack;

ENDPROCESS Responder;

ENDBLOCK ResponderEntity;
```

```
BLOCK Medium;
/* two processes are necessary */
/* (one for the data and one for acknowledgements). */

PROCESS DatHazard(1,1);

DCL datq MedDatQueue,
    seqno Natural,
    error Natural,
    unused, used Natural;
TIMER t;

START;
    TASK unused := 0, used := 0, error := 0;
    SET(NOW+Delta, t);
    NEXTSTATE transfer;

STATE transfer;
    INPUT DATr(seqno);
        TASK datq(unused)!seqno := seqno,
             unused := (unused+1) MOD (MedBuffer);
        DECISION( used = unused );
            (True): /* medium is too slow */
                    OUTPUT DATi(datq(used)!seqno);
                    TASK used := (used+1) MOD (MedBuffer);
             ELSE: TASK ";
        ENDDECISION;
        NEXTSTATE -;

    INPUT t;
        DECISION (used = unused); /* data in buffer? */
            (True):TASK "; /* no data */
            ELSE: DECISION( error );
                    (0): /* correct data transmition */
                        TASK seqno := datq(used)!seqno,
                             used := (used + 1) MOD (MedBuffer);
                        OUTPUT DATi( seqno );
                        SET(NOW+Delta, t);
                        NEXTSTATE -;

                    (1): /* loss of data */
                        TASK used := (used + 1) MOD (MedBuffer);
                        SET(NOW+Delta, t);
                        NEXTSTATE -;


                    (2): /* duplication of data */
                        TASK seqno := datq(used)!seqno;
                        OUTPUT DATi( seqno );
                        NEXTSTATE second_out;
```

```
                    (3): /* re-ordering of data */
                        TASK datq(unused)!seqno := datq(used)!seqno,
                            used := (used + 1) MOD (MedBuffer),
                            unused := (unused + 1) MOD (MedBuffer);
                        SET(NOW+Delta, t);
                        NEXTSTATE -;
                ENDDECISION;
        ENDDECISION;
ENDSTATE transfer;

STATE second_out;
    INPUT none;
        TASK seqno:= datq(used)!seqno,
            used := (used + 1) MOD (MedBuffer);
        OUTPUT DATi( seqno );
        SET(NOW+Delta, t);
        NEXTSTATE transfer;

    SAVE *;
ENDSTATE second_out;

ENDPROCESS DatHazard;


PROCESS AckHazard(1,1);

DCL datq MedDatQueue,
    seqno Natural,
    error Natural,
    unused, used Natural;
TIMER t;

START;
    TASK unused := 0, used := 0, error := 0;
    SET(NOW+Delta, t);
    NEXTSTATE transfer;

STATE transfer;
    INPUT AKr(seqno);
        TASK datq(unused)!seqno := seqno,
            unused := (unused+1) MOD (MedBuffer);
        DECISION( used = unused );
            (True): /* medium too slow */
                OUTPUT AKi(datq(used)!seqno);
                TASK used := (used+1) MOD (MedBuffer);
            ELSE: TASK ";
        ENDDECISION;
        NEXTSTATE -;

    INPUT t;
        DECISION (used = unused); /* data in buffer? */
```

```
            (True):TASK "; /* no data */
            ELSE: DECISION( error );
                    (0): /* correct data transfer */
                        TASK seqno := datq(used)!seqno,
                                used := (used + 1) MOD (MedBuffer);
                        OUTPUT AKi( seqno );
                        SET(NOW+Delta, t);
                        NEXTSTATE -;

                    (1): /* loss of data */
                        TASK used := (used + 1) MOD (MedBuffer);
                        SET(NOW+Delta, t);
                        NEXTSTATE -;

                    (2): /* duplication of data */
                        TASK seqno := datq(used)!seqno;
                        OUTPUT AKi( seqno );
                        NEXTSTATE second_out;

                    (3): /* re-ordering of data */
                        TASK datq(unused)!seqno := datq(used)!seqno,
                                used := (used + 1) MOD (MedBuffer),
                                unused := (unused + 1) MOD (MedBuffer);
                        SET(NOW+Delta, t);
                        NEXTSTATE -;
                ENDDECISION;
        ENDDECISION;
ENDSTATE transfer;

STATE second_out;
    INPUT none;
        TASK seqno := datq(used)!seqno,
                used := (used + 1) MOD (MedBuffer);
        OUTPUT AKi( seqno );
        SET(NOW+Delta, t);
        NEXTSTATE transfer;

    SAVE *;
ENDSTATE second_out;

ENDPROCESS AckHazard;

ENDBLOCK Medium;


ENDSYSTEM SlidingWindow;
```

B.2 Modelling of signal parameters for *Sliding Window* protocol

```
/* user defined help functions for managing of the signal parameters */
/* duplication of an integer value */
DAT_P dup_int( p_z1 )
DAT_P p_z1;
{
    int *i;

    if ( ( i = (int * )(malloc(sizeof(int))) ) == NULL ) {
        printf("\nmemory overflow by duplicate of integer");
        exit(1);
    }
    *i = *(int * )p_z1;
    return ((DAT_P) i);
}

/* comparison of two poiters to integer */
int cmp_int (data1, data2)
DAT_P data1;
DAT_P data2;
{
    if ( *(int * )data1 == *(int * )data2 )
        return(1);
    else
        return(0);
}

/* output of an integer value */
void write_int( ex_d, dat )
FILE *ex_d;
DAT_P dat;
{
    if ( dat == NULL )
        fprintf(ex_d, " dat == NULL");
    else
        fprintf(ex_d, " %2d ", *(int * ) dat );
}

/* output of signal parameter DATr */
void write_Num_DATr( ex_d, dat )
FILE *ex_d;
DAT_P dat;
{
    fprintf(ex_d, " DATr");
    write_int( ex_d, dat );
}
```

```
/* output of signal parameter AKr */
void write_Num_AKr( ex_d, dat )
FILE *ex_d;
DAT_P dat;
{
    fprintf(ex_d, " AKr");
    write_int( ex_d, dat );
}

/* array for user defined help functions, */
/* the functions are attached to the signal numbers. */
struct action_sig actions[] = {
    { 32 /* DATr */, {
        dup_int, cmp_int, write_Num_DATr }
    },
    { 34 /* AKr */, {
        dup_int, cmp_int, write_Num_AKr }
    },
    { -1, {
        NULL, NULL, NULL }
    }
};

/* attachment of the user defuned functions to the transition order numbers. */
struct n_fkt ext_fkt_feld[] = {
    { 1, trans_start_out_d1 },
    { 16, resp_nr1_dat1_0_dat2_0_out1 },
    { -1, NULL }
};

/* the function writes out the first data record DATr with */
/* modelling parameter (sequence number 1) in state */
/* START of the process Transmitter. */
struct ret_sig *trans_start_out_d1(sig_par)
DAT_P sig_par;
{
    struct sig_par *sig_par_v;
    struct ret_sig *ret = NULL;

    /* memory for return structure */
    if ( ( ret = NEW(ret_sig) ) == NULL ) {
        printf("\nmemory overflow in extern.c");
        exit(1);
    }

    /* memory for one output signal parameter */
    if ( ( sig_par_v = NEW(sig_par) ) == NULL ) {
        printf("\nmemory overflow in extern.c");
        exit(1);
    }
```

```
    /* memory for one output signal parameter */
    if ( ( sig_par_v→sig_par_p = (int * )malloc(sizeof(int)) ) == NULL ) {
        printf("\nmemory overflow in extern.c");
        exit(1);
    }

    /* number of the output signals (DATr), whose */
    /* parameter are to be modelled. */
    sig_par_v→sig_typ = 32;
    /* value of the signal parameter */
    *(int * )(sig_par_v→sig_par_p) = 1;

    /* return value of transition */
    ret→wert = 1;
    ret→sig_c = 1;
    ret→sig_v = sig_par_v;

    /* initialize of the variable lu */
    *(int * )ext_data[0] = 1;

    return(ret);
}

/* the function returns 1, when it has been received the data record */
/* DATi with sequence number 1 as Parameter and */
/* when no data records have been received previosly. */
struct ret_sig *resp_nr1_dat1_0_dat2_0_out1(sig_par)
DAT_P sig_par;
{
    struct sig_par *sig_par_v;
    struct ret_sig *ret = NULL;

    /* memory for return structure */
    if ( ( ret = NEW(ret_sig) ) == NULL ) {
        printf("\nmemory overflow in extern.c trans_start");
        exit(1);
    }

    /* net transition is to be fired only if the receiving */
    /* signal parameter is 1 and dat1 == 0 and dat2 == 0. */
    if ( ( *(int * ) sig_par == 1) && ( *(int * )ext_data[1] == 0 )
        && (*(int * )ext_data[2] == 0) ) {

        /* memory for output signal parameter structure */
        if ( ( sig_par_v = NEW(struct sig_par ) ) == NULL ) {
            printf("\nmemory overflow in extern.c");
            exit(1);
        }
```

```
        /* memory for one output signal parameter */
        if ( ( sig_par_v→sig_par_p = (int * )malloc(sizeof(int)) ){
             == NULL )
             printf("\nmemory overflow in extern.c ");
             exit(1);
        }

        /* number of output signal (AKr), whose */
        /* parameter has to be modelled. */
        sig_par_v→sig_typ = 34;
        /* output of signal parameter 1. */
        *(int * )(sig_par_v→sig_par_p) = 1;

        /* return value of transition */
        ret→sig_c = 1;
        ret→sig_v = sig_par_v;
        ret→wert = 1;

        /* dat1 is to be set to 1, i.e. the data record 1 is received */
        *(int * )ext_data[1] = 1;
    } else
     {
        /* return value of transition */
        ret→sig_c = 0;
        ret→sig_v = NULL;
        ret→wert = 0;
    }

    return(ret);
}
```